MELDSW/D

REV 1

JANUARY 1997

MOTOROLA

EMBEDDED LINK EDITOR (MELD) Version 2.0 USER'S MANUAL

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Trademarks

This document includes these trademarks:

Motorola and the Motorola logo are registered trademarks of Motorola, Inc.
IBM, and PowerPC are trademarks of International Business Machines Corporation.

Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 RUNNING THE LINKER

CHAPTER 3 COMMAND LINE INTERFACE

CHAPTER 3 COMMAND LINE INTERFACE (Continued)

CHAPTER 4 LINKER DEFINITION FILE

CHAPTER 5 LIST FILE

APPENDIX B MOTOROLA ARCHIVER

APPENDIX C MOTOROLA S-RECORD GENERATOR

FIGURES

FIGURES (Continued)

TABLES

CHAPTER 1

INTRODUCTION

The Motorola Embedded Link eDitor (MELD) is a linker that combines object files and library file members (in object-file format) into an executable object file. Motorola's MSREC S-record generator can convert this executable object file to S-records. Motorola's MEDB debugger can load this executable object file into memory.

Using its knowledge of memory layout, the linker assigns physical addresses to code and data. It assigns address values to symbols; based on these address values, the linker changes code and data that reference the symbols.

MELD operates on object files and library files generated by these software tools: the MEPROJ development software, the MECC compiler, the MEAS assembler, and the MAR archiver. These tools support the PowerPC™ instruction set and conform to the *PowerPC Embedded Application Binary Interface* standard.

MELD accepts input object files and library members that are in ELF object-file format; MELD accepts input library files that are in archive-file format. For information on these formats, see the *System V Application Binary Interface* standard or its *PowerPC Processor* supplement.

NOTE

If you use the Motorola Embedded Project (MEPROJ) to develop your code, MEPROJ gives you direct access to the link editor, the archiver, and the S-record generator.

1.1 USER AND SYSTEM REQUIREMENTS

This manual is a guide for engineers and programmers who develop code for embedded PowerPC applications. To get the most from this manual, you should understand the difference between a high-level programming language and assembly code, as well as what distinguishes code for embedded applications. You should be familiar with what is in an object file, what a linker does, and what typically is in startup (system boot) code.

For computer and operating-system requirements, see the software release guide.

1.2 MANUAL CONVENTIONS

Table 1-1 lists the syntax and typographical conventions of this manual.

Table 1-1. Manual Conventions

Symbol, Typeface	Significance
`Courier`	Command syntax and examples.
`Courier italic`	Syntax indicator to be replaced by an actual value.
`[ ]`	Indicates entire value is optional.
`{ }`	Indicates portion of string is optional.
`...`	Indicates repetition of values is permissible.

The remaining chapters and appendixes of this manual cover these topics:

- Chapter 2: Examples of using the linker.
- Chapter 3: Command-line interface syntax and semantics.
- Chapter 4: Linker definition file syntax and semantics.
- Chapter 5: List file contents.
- Appendix A: Linker error messages.
- Appendix B: Motorola Archiver, which creates library (archive) files.
- Appendix C: Motorola S-Record Generator, which generates S-records from object files.

1.3 REFERENCES

System V Application Binary Interface, Third Edition, UNIX System Laboratories, 1994 (ISBN 0-13-100439-5).

System V Application Binary Interface PowerPC Processor Supplement, SunSoft and IBM, 1995 (SunSoft Part No: 802-3334-10).

PowerPC Embedded Application Binary Interface, Motorola and IBM, 1995 (Motorola EABI/D).

CHAPTER 2

USING THE LINKER

This chapter presents introductory examples of using the MELD linker. If you already are knowledgeable about linkers, reading these examples may yield enough information for you to use MELD with your application.

2.1 LINKER OPERATION

The linker produces an executable object file by combining object files and object members. (*Object members* are library file members in object-file format.) Object files and object members contain symbol definitions, references to symbols, and sections. A *section* is a named area that contains code, data, or information to be used by linkers, loaders, or debuggers. Paragraphs 3.1.1 and 3.1.4 give more about object files and object members.

First, MELD processes the input object files you specify. For each input object file, the linker records sizes and other section information, symbol definitions, and references to symbols the file does not define.

After the linker processes the input object files, it may have references to symbols not defined in any of the object files. If so, MELD searches the library files you specify, looking for the definitions.

object members that define those symbols. If a member defines a needed symbol, the linker extracts it from its library file and includes it in the link. The linker records that member's section information, symbol definitions, and which symbols it references but does not define. The linker repeatedly searches the libraries until all referenced symbols are defined, or until a complete pass through all the libraries fails to define any undefined symbols.

MELD then assigns the sections in input object files and library members to output file *segments*: areas of an executable object file containing code or data that is loadable into physical memory on your target system. If the linker assigns sections with the same name from two or more files or members to a segment, then in the segment the linker combines the sections into one section with that name. During assignment, the linker determines the physical addresses of all sections and all symbols within all sections.

You can describe segments to MELD in a *linker definition file* (LDF). An LDF lets you define the start address and size of each segment and declare which sections the linker should place in each segment. In addition, an LDF lets you define symbols and assign them values. Your code can refer to LDF-defined symbols to provide information such as the size of a segment or the address of a memory-mapped device. (See *Chapter 4, Linker Definition File*.) If you do not supply an LDF, or there are no segments defined in it, the linker defines a default set of segments, as described in *Section 4.3, Default Segments*.

Next, MELD defines the output object file's entry point (address of the first PowerPC instruction your application executes). The default entry point is the value of the global symbol **`__start`** (two leading underscore characters).

MELD generates the base addresses for the small data areas. Each small data area contains data items, and every byte in the area can be addressed as a 16-bit signed offset from that small data area's base address. If you load the base address into a general purpose register (GPR) at the start of your application, your code can take advantage of the several PowerPC processor instructions that load or store data using a 16-bit signed offset from the value in a GPR. (See *Section 4.1.2, Small Data Areas*.)

The linker copies the code and data from all input file and library member sections to the output file. Sometimes the value of code or data depends on the value (address) that the linker assigns to a symbol. In that case, the code or data is said to reference or refer to the symbol. For each reference, the input file or member contains a relocation instruction. A *relocation instruction* tells the linker how to change code or data copied to the output file based on the value (address) assigned to the referenced symbol. The linker also copies the relocation instructions themselves to the output file. For information about relocation instructions, see *PowerPC Embedded Application Binary Interface*, *System V Application Binary Interface PowerPC Processor Supplement*, and *System V Application Binary Interface*.

Optionally, after the linker writes the output object file, the linker generates a list file. The list file can display information about segments, sections, and symbols.

2.2 COMMAND LINE EXAMPLES

MELD is tailored to meet embedded needs. It lets you customize the assignment of code and data to the memory partitions. You can define symbols, with values set by the linker, that your code can use. You can put both customizations and symbol definitions in a linker definition file (LDF) that the linker processes. The examples in this section start with a simple link of object files and progress through using library files, an LDF, and a command file.

Example 2.1: Simple link

To link two object files type:

```
meld startup.o timer.o
```

The linker links the object files **startup.o** and **timer.o**. It produces an executable object file named **a.out** in the current directory.

Usually, when an application begins, it executes startup code that sets processor operating modes, copies read/write initialized data from ROM into RAM and zeros out uninitialized data. The startup code also sets up a stack, initializes registers, and branches to the beginning label of the application (usually **main**). For this example, the startup code is in **startup.o**.

Somewhere in the object files, the startup label **__start** must be defined. This label will usually be at the beginning of the startup code. The linker records the location of this symbol in the output file. The Motorola Embedded Debugger uses this address to locate the beginning point of execution.

If you want your application to boot at the startup address on the target hardware, you need to locate the startup code at the boot address for the processor. You can accomplish this with a linker definition file (LDF). See *Chapter 4, Linker Definition File*.

Example 2.2: Specifying an output file using the -o switch

This example is just like Example 2.1 above, except that you specify the name of the output object file. Type:

```
meld -o prog startup.o timer.o
```

The linker links the object files **startup.o** and **timer.o** to produce an executable object file named **prog** in the current directory.

Example 2.3: Specifying a library file using the -l switch

To link object files with a specific library file, use the **-l** switch. For example, type:

```
meld -o prog startup.o counter.o -lppc
```

MELD uses the **-l** switch parameter '**ppc**' to construct a file name. The parameter must follow '**-l**' with no intervening spaces. MELD prepends '**lib**' to the constructor '**ppc'** and appends '**.a**', to form the file name **libppc.a**. The linker expects **libppc.a** to be a library file in archive file format or a valid relocatable ELF object file (usually you use this switch to specify a library file).

As this example did not use the **-L** switch (see next example) to specify directories in which to search for constructed file names, MELD looks for a file named **libppc.a** only in the current directory.

MELD links **startup.o** and **counter.o**. In this example, assume **libppc.a** is a library file in the current directory, and **counter.o** references some functions and variables that neither **startup.o** nor **counter.o** defines. After linking the object files, MELD searches library **libppc.a** to find object members that define those referenced functions and variables. It links those members from **libppc.a** with the contents of **startup.o** and **counter.o**.

Example 2.4: Specifying a library path list using the -L switch

Consider this example:

```
meld -o prog ../project/startup.o counter.o average.o
     ../project/libmath.a -L ../project/lib -lppc
```

MELD links the object files **../project/startup.o**, **counter.o** and **average.o**. The linker finds that file **../project/libmath.a** is a library file, so the linker does not initially include it in the link.

As in Example 2.3, MELD constructs the file name **libppc.a** from the **-l** switch. MELD searches for **libppc.a** in the paths (directories) in the library path list. The library path list is initially empty, and each **-L** switch processed adds a path to the end of the list. In this example, the library path list for **libppc.a** is directory **../project/lib**. If the linker does not find **libppc.a** in that directory, then it looks in the current directory. If the linker cannot find a properly-formatted library or object file named **libppc.a**, MELD issues an error. In this example, assume the linker finds a library file named **libppc.a** in directory **../project/lib**.

The linker only searches in library path list directories for file names constructed using the **-l** switches. The linker determines whether the files it finds are object files or library files. If they are object files, the linker links them in before it searches any libraries. If they are libraries, the linker searches them for object members that resolve symbols that are referenced by not defined.

After linking the three object files, MELD discovers that there are symbols that are referenced but undefined. So it searches library **../project/libmath.a** and then library **../project/lib/libppc.a** for members that define any needed symbols. If needed symbols are defined in an object member, the linker extracts the member and links it in. If a linked member references other undefined symbols, the linker searches the libraries again. The linker repetitively searches the libraries until: 1) there are no more referenced-but-undefined symbols, or 2) a complete pass through the libraries fails to yield any additional needed symbol definitions.

The **-L** switch adds a directory (path) to the library path list. You use a separate **-L** switch for each directory you want to add. The linker starts with an empty list and adds each new directory specified with a **-L** switch to the end of the current list. After the linker constructs a file name using a **-l** switch, the linker searches for a file with that name in the current library path list. The current list is the list specified up to the point in the command line or a command file where the **-l** option occurs. If the linker does not find the file using the current library path list, the linker checks the current directory.

If we had typed **`-L ../project/lib`** after **`-lppc`**, there would have been an empty library path list when the linker processed **`-lppc`**. The linker would have searched only in the current directory. Make sure that you add the necessary directory to the library path list before you specify the files that you want the linker to find there.

Example 2.5: Using a linker definition file

MELD allows you to define segments that describe the major memory partitions of your application. You do this in a linker definition file (LDF). This file is also where you declare the values of absolute symbols. You can base the values on the addresses and sizes of the partitions, and your application's startup code can use them to initialize your application. For information about the contents of a linker definition file, see *Chapter 4, Linker Definition File*.

In this example, we have a linker definition file named **`project.ldf`**. The following is the contents of that file:

```
####
# Copyright Motorola 1994-1995 - All rights reserved.
#
# This linker definition file is specific to the PowerPC MPC505
# processor. It is meant to be used with the Motorola MPC505EVB
####
####
####
# Before the MPC505 can access code and data in off-chip memory, chip
# select registers must be set to define the off-chip memory. The
# segments chips0 and chips1_5 contain these chip selects. These
# Segments are the first segments defined, so that any loader that
# downloads segments in the order they appear in the executable file,
# will first set chip selects, and then download the code and data.
####
segment chips0          start=0x8007FDEC (chips0);
segment chips1_5        start=0x8007FDC0 (chips1_5);
####
# Place read-only (ROMable) code and data in addresses 0x2000 through
# 0x7fff, and read-write data in addresses 0x8000 and above.
#
# The number of bytes from the start of .data to the start of .sbss is
# forced to be a multiple of 8.
####
```

```
segment .text    start=0x20000(.text rodata sdata2) align=8;
segment .idata   type=romcopy(.data);
segment .data    start=0x40000(.data sdata) align=8;
segment .bss     type=reserved (.sbss bss) align=8;
segment .stack   type=reserved size=Ox40000 align=8;
####
# Define values to be used by the startup code before main( )
# is called
####
# Size in 4-byte words of initialized data values which will
# be copied from ROM (plus any padding between .data and .sdata;
# rounded to a multiple of 4)
symbol _startup.idata_words = (segsize(.data) + 3) / 4;
# Start address - 4 of where initialized data values in ROM
symbol _startup.idata_values = segstart(.idata) - 4;
# Start address - 4 of where initialized data will be copied into
# RAM
symbol _startup.idata = segstart(.data) - 4;
# Size in words of uninitialized data to be zeroed before main( )
# is called
symbol _startup.udata_words = (segsize(.bss) + 3) / 4;
# Start address - 4 of uninitialized data
symbol _startup.udata = segstart(.bss) - 4;
# The stack is 64K bytes, 8 byte aligned, starting after
# uninitialized data
symbol _startup.stack = segafter(.stack) - 8;
```

The above definition file defines seven segments (memory partitions).

Segments **chips0** and **chips1_5** contain the values for MPC505 processor chip select registers as described in the MPC500 family System Integration Unit reference manual. Chip select registers define the characteristics of off-chip memory.

The **.text** segment contains code and read-only data. The **.idata** segment follows the text segment. It contains the initialized data that is stored in ROM, and is copied into RAM at system boot time.

The **.idata** segment is a romcopy segment for the **.data** segment. That means the **.idata** segment contains the initial values for variables in the **.data** segment. The **.idata** segment is located in ROM, and the **.data** segment is in RAM. The startup code must copy the image of the **.data** segment, which is located in the **.idata** segment, to the **.data** segment. The **.data** segment is the location where your code accesses the initialized data during execution. For more information about romcopy segments, see *Section 4.9, Use of Romcopy Segments.*

The **.bss** segment reserves space for uninitialized data. It follows the **.data** segment.

Finally there is the **.stack** segment, which reserves space for the stack.

The symbol definitions in the LDF that start with **__startup** define absolute symbols based on the sizes or locations of segments.

An application's startup code copies the initialized data values in the **.idata** segment to the read/write **.data** segment. It also set the bytes in the **.bss** segment to 0. It initializes the first stack frame and sets the stack frame pointer to the current top of stack. The startup code uses symbols defined in the LDF (such as **__startup.idata_words**) to provide the start addresses and sizes of these various segments.

As the linker sets the values of those symbols, when the segments grow or change position you will not have to change your startup code.

You can only provide one LDF for any given link. To link the application in the previous example using the above LDF, type:

```
meld -o prog -def ../project/project.ldf
     ../project/startup.o counter.o average.o
     ../project/libmath.a -L ../project/lib -lppc
```

MELD links the object files **../project/startup.o**, **counter.o**, and **average.o**. The file **../project/libmath.a** is a library file. The linker constructs the file name **libppc.a** from the **-l** switch and finds this file in the directory **../project/lib** (a library path specified by the **-L** switch). It is also a library file. The linker searches the two libraries for object members that define symbols that are referenced elsewhere. It extracts those members so that the symbols become defined. This process is described in more detail in Example 2.4.

In the previous examples, the linker used the default memory segments. In this example, the user supplies a memory layout description in the linker definition file **../project/project.ldf**. You provide the name of this file to the linker using the -**def** switch.

Example 2.6: Using a command file

If you link repeatedly, you may discover that you type certain options and paths over and over again. It is laborious to retype the library paths, startup code file name and LDF file name every time. Instead, you can place these file names and options with their parameters in a command file. Build a linker command file named **project.cmd** that contains the following:

```
-def ../project/project.ldf ../project/startup.o
../project/libmath.a -L ../project/lib -lppc
```

Now type:

```
meld -f project.cmd -o prog counter.o average.o
```

This is the same as typing:

```
meld -def ../project/project.ldf ../project/startup.o
     ../project/libmath.a -L ../project/lib -lppc
     -o prog counter.o average.o
```

If you define the environment variable **MELDRC** to name the file **project.cmd**, the linker opens and uses the command file before processing the command line. So the effect is the same as though you typed **-f project.cmd** as the first item on the command line.

Type:

```
set  MELDRC=project.cmd
meld -o prog counter.o average.o
```

This is the same as typing (leaving the **MELDRC** environment variable undefined):

```
meld -f project.cmd -o prog counter.o average.o
```

CHAPTER 3

COMMAND LINE INTERFACE

3.1 INTRODUCTION

You may specify the names of files and switches, with their parameters, on the command line. Anything on the command line beginning with '**-**' is interpreted to be a switch. Parameters are either numeric values, symbol names, file names or file name constructors. These follow after the switch. Either a colon (**:**) or a space must separate the switch from its parameter, except for the **-l** switch.

Switches are used to set various options, provide names of output files, name the linker definition file or command files, set the starting address of segments and set the entry address.

Any other item on the command line is taken to be the name of an object file or library (archive) file. You can also specify object files or library files using the **-l** switch, which takes a file constructor, and searches a set of directories for the file name constructed. For more information see *Section 3.3.8,-lLibKey.*

The linker determines whether each file specified directly or using the **-l** switch is a library file or object file. An object file is included in the link; that is, its sections, symbol definitions and symbolic references become part of the output object file. The members of a library file are only extracted and included in the link if they are in object members (in object file format) and define symbols which are referenced but undefined by the set of object files and members already included in the link. The linker repeatedly searches through the libraries until all of the referenced symbols are defined, or a complete pass through the libraries fails to find a definition for any symbol that is referenced but not yet defined.

Some switches affect the way that object files are processed. These switches are accumulated as the linker encounters them on the command line from the leftmost switch to the rightmost one. As object files or library files are encountered on the command line, either directly or constructed via the **-l** switch, only the switches in effect up to that point in the command line affect the object processing. The switches that behave this way are called *order-dependent* switches.

Other switches have an effect over the entire link in some way. These switches are called *order-independent* switches. For more information, see *Section 3.2.2, Switch Processing Order.*

The linker lets you put command line options into a file, called a command file, which can then be read by the linker. Each file name or switch is processed from left to right and then from top to bottom as though it were typed on the command line. The linker checks for the environment variable **`MELDRC`** at startup. If it is defined, and if it names a file that can be opened and read, the linker begins its command processing from this file. After the end of the file, the command processing continues after the point where the command file was called. For more information, see *Section 3.4, Command Files* and *Section 3.3.7, -f CommandFile.*

You can define your own memory segments, which define the memory architecture of your system. You can also define symbolic values. This is done in the linker definition file. For more information see *Chapter 4, Linker Definition File* and *Section 3. 2.4.2, -def DefFile.*

The linker outputs an ELF object file in executable form. It optionally outputs a list file showing the locations and sizes of segments, composite sections, and the values of symbols. The list file also shows what sections reference and define which symbols.

3.1.1 Input Object and Library Files

The linker accepts input object files, and object members of libraries, that are in ELF object file format, are marked as for the PowerPC architecture, and are in relocatable form. These files and members contain initialized code and data, instructions for reserving uninitialized space, definitions of symbols, and instructions for changing code and data that references symbols when the addresses of those symbols change.

An ELF object file flag indicates whether the object file or object member is for embedded applications. The linker accepts input object files that are either embedded or not embedded. However, all input object files must be consistent with respect to this flag, and the linker will only include library members in the link whose flag is consistent the flags in input object files. The MEPROJ project tool, the MECC compiler, and the MEAS assembler produce embedded ELF object files.

The linker accepts library files which are in archive file format. The Motorola Archiver (MAR) can create a library file in archive file format from any set of files, but usually the members of a library file will be object files. See *Appendix B, Motorola Archiver*.

For information about ELF object file format and archive file format, see *System V Application Binary Interface*, *System V Application Binary Interface PowerPC Processor Supplement*, and *PowerPC Embedded Application Binary Interface.*

3.1.2 Linker Definition File

The linker definition file (LDF) is used to define the memory regions for your application. Each region in memory is called a segment, and segments are defined in this file.

You can also define symbols in this file. This allows you to define a symbolic address for a memory-mapped device, or provide symbolic data that can change at link time. Symbols can be defined in terms of the start addresses or sizes of segments or sections. For information about the syntax and meaning of declarations in the LDF, see *Chapter 4, Linker Definition File*.

3.1.3 Command Files

The linker allows you to create a file that contains the names of the object files, linker switches, parameters, and the names of library files. This linker can take command line information taken from the file. This is very convenient when there is a long command string to type and you need to repeatedly invoke the linker with the same command string.

To get the linker to accept input from a file, use the **`-f`** switch. For more information about this switch, see *Section 3.3.7, -f CommandFile.* The linker accepts input from the file as though the contents of the file had been typed on the command line instead of the **`-f`** switch. You may also nest command files; that is, have a command file call another command file.

The linker looks for the environment variable **`MELDRC`**. If this variable is defined, the linker attempts to open the file named by this variable and use this file as a startup command file.

For more information on command files, see *Section 3.4, Command Files.*

3.1.4 Output Object File

The linker takes the specified object files and those extracted from the libraries, locates the sections in the object files to some addresses, and outputs these sections to the output object file. This output file is an ELF object file in executable form. For more information about this file format, see *System V Application Binary Interface.*

The output file has the PowerPC architecture flag set. If all of the processed objects are embedded, the embedded flag is set; otherwise the embedded flag is not set.

In some cases you may want to create an output object that can be used as input to subsequent links. You can do this by using the **`-r`** switch; see *Section 3.3.17, -r*. In this case, the output file will be in relocatable format.

The linker optionally outputs a list file, if requested. This file shows the segments defined, the sections assigned to them, their addresses, sizes and types. It also lists what sections define what symbols, and which symbols they reference. The list file also shows the symbols defined, their values, scopes, and types. These symbols can be listed in alphabetic or numeric order. For more information, see *Chapter 5, List File*.

3.2 COMMAND SWITCH PROCESSING

The linker provides several switches that allow you to set address values, control symbol resolution, name output files, request a listing, etc. The syntax for these switches is the same whether they are typed on the command line or are part of a command file. See *Section 3.4, Command Files.*

The following is the command line with the various valid switches:

```
meld [-caps] [-def DefFile] [-dup] [-D DataStart]
     [-ent EntryLabel] [-error ErrCnt] [-f CommandFile]
     [-lLibKey] [-L Directory] [-LIST ListFile]
     [-nocaps] [-nodup] [-noent{ry}] [-o OutFile]
     [-pad PadChar][-q] [-r] [-seg] [-sym] [-symn] [-syma]
     [-T TextStart] [-warn WrnCnt] [-weak] [-xref]
```

3.2.1 Case Sensitivity

Switches always start with a **-**. Most switches are case-insensitive. That is, **-CAPS**, **-caps**, **-Caps**, or **-CaPs** are all accepted as the **-caps** switch. There are three exceptions, the **-l**, **-L** and **-LIST** switches. The **-l** and **-L** switches are case-sensitive. The **-LIST** switch must start with the uppercase **-L**. After that, it is case-insensitive. **-List**, **-LIST**, or **-LiST** are all equivalent.

3.2.2 Switch Processing Order

Switches are processed from left to right on the command line, starting right after the linker executable name, **meld**. When the commands appear in a command file, they are processed from left to right, and then from top to bottom. See *Section 3.4, Command Files.*

Switches can appear interspersed among object file names, library file names, and the **-l** switch, which is used to specify a library or object file. Some switches affect the manner in which an object file or object member of a library is processed. Other switches affect some attribute of the entire link process, and do not affect object file processing.

Switches that affect the attributes of the entire link are said to be *order-independent*. The order or position of these switches with respect to the object files, library files, or **-l** switches does not matter. Sometimes these switches can contradict one another. When this is the case, the linker simply uses the latest (rightmost) information. It does not issue an error or warning.

You may want to provide a default switch setting in a command file that is automatically set when you invoke the linker, see *Section 3.4, Command Files*. You might want to override some of the settings in this file without disabling the others. This can be done as the order-independent switches can be overridden without error.

```
meld -D 0x200 startup.o vector.o -D 0x1000
```

The first **-D** switch directs the linker to start the default **.data** segment at 0x200 (hexadecimal). This is overridden by the second **-D** switch, so the linker places the **.data** segment at 0x1000 instead.

Switches that affect the processing of an object file or object member of a library can be turned on or off. The state of each switch in effect at the time a file appears on the command line is the state used for processing that file. These switches are referred to as *order-dependent* switches.

The **-l** switch is a shorthand form of specifying a library file. It can also be used to specify an object file. It has the same effect as typing the full file name on the command line at the point where the **-l** file appears. Therefore, the *order-dependent* switches in effect at the point of the **-l** switch are those that affect the object processing, whether in a file or extracted from a library.

Object files are processed by the linker, then object members of libraries, so the processing order of files may not be the same as the command line order in which they appear. The switch state, accumulated from left to right on the command line (or in a command file) is remembered when the file is processed later.

```
meld -caps startup.o vector.o -nocaps factor.o -lc
```

When the linker encounters **startup.o** and **vector.o** on the command line, the **-caps** switch is turned on. This causes all of the symbols defined or referenced in these files to be converted to uppercase.

The object file **factor.o** and object library **libc.a** (specified by the **-lc** switch) is encountered after the **-nocaps** switch. For **factor.o** and all of the object members in **libc.a**, the symbols are retained in their original case.

3.2.3 Switch Parameters

Some of the switches require parameters. They must follow the switch with an intervening space or colon (:). Care must be taken to supply the parameter, especially for switches that name output files. If you forget, the linker may take the name of the next object file and use it as the parameter for the switch. This could cause your object file to be destroyed.

`meld -LIST startup.o vector.o -lc`

The **`-LIST`** switch is given, but the user forgot to give the name of the list file. The file **`startup.o`** is an input object file. The linker does not detect the error. Instead, it attempts to list to a file named **`startup.o`**, which overwrites your object file.

The **`-l`** switch is a special case. It requires a parameter, but the convention is to place it immediately against the switch (without intervening spaces). For this switch only, white space is not allowed.

Some switches supply the name of a file as a parameter. Where this is the case, the file name may be given without a path, in which case the linker assumes that the file exists in, or is to be created in, the current directory. If a path is given with the file name, it may be specified as an absolute path name or relative to the current directory.

Case 1: **`meld -LIST project.lst startup.o vector.o`**

Case 2: **`meld -LIST ../dave/project.lst startup.o vector.o`**

Case 3: **`meld -LIST C:/usr/home/dave/project.lst startup.o vector.o`**

Case 1 specifies that the list file is to be created in the current directory.

Case 2 specifies that the list file is to be created in directory **`dave`** contained in the parent directory of the current directory.

Case 3 specifies that the list file is to be created in directory **`C:/usr/home/dave`**.

Some of the switches require a numeric value as the parameter. Wherever this is the case, the value must be a simple numeric. It may not be an expression. The unary **`+`** and **`-`** operators are permitted.

Number strings that begin with '**`0x`**' or '**`0X`**', and continue with digits **`0`**-**`9`** and letters **`A`**-**`F`** or **`a`**-**`f`** are interpreted as hexadecimal values (base 16). Number strings that begin with 0, and continue with the digits **`0`**-**`7`** are interpreted as octal values (base 8). Number strings that begin with the digits **`1`**-**`9`** and continue with digits **`0`**-**`9`** are interpreted as decimal values (base 10).

These numeric values are always interpreted as 32-bit unsigned integers, so '**`-1`**' is a shorthand way of specifying 0xFFFFFFFF.

3.3 COMMAND SWITCHES

The following paragraphs document the specific switches supported by the linker. For each of the switches, we state whether the switch is order-dependent or order-independent. We also state the initial state or default value.

3.3.1 -caps

This switch causes the linker to convert all symbol names defined or referenced in object files or library object members to upper case. This has the effect of rendering the linker case insensitive with respect to symbols defined and referenced when the switch is in effect. It also converts the symbol to uppercase when it is written to the output object file. Remember that the references are converted also. If the object file or member that defines these references is not also converted, then the symbol is not resolved and an error occurs. Symbols which are defined in the linker definition file (LDF) are not converted.

Order Dependent: Yes.

Default State/Value: Turned off. Default is **-nocaps**.

3.3.2 -def *DefFile*

Description: This switch lets you specify the name of the linker definition file (LDF). The LDF allows you to define memory segments and absolute symbols. For more information on the contents of an LDF, see *Chapter 4, Linker Definition File*.

Unlike other order-independent switches, this switch is allowed to appear only once per linker invocation. You are not required to specify an LDF but if you do, you may have only one.

Order Dependent: No. Only one **`-def`** switch allowed.

Default State/Value: No default file, but there is a default set of segments defined when this file is not supplied or does not define any segments, see *Chapter 4, Linker Definition File*.

3.3.3 -dup

Description: This switch tells the linker to allow duplicate global symbol definitions. Normally, if a symbol is globally defined by some object file or library object member in the linker definition file, and is defined again later in another file or member, an error occurs.

When processing object files with the **`-dup`** switch, subsequent definitions do not cause a conflict. Duplicate definitions of any symbol are used for references within the same file or member that defined them. Any files or members that reference the symbol but do not define it uses the first definition that was globally defined. This switch is very useful when you need to pull in a member of an object library that globally defines a symbol that you also have globally defined. In the case where you cannot change the library and do not want to rename your symbol, you can use the **`-dup`** switch.

The linker processes object files first, in command line order, and then library files. Library files are processed in command line order, and their members are processed in file order. This processing order determines which of the duplicate symbols comes first. The linker keeps the first definition and converts subsequent ones to local visibility. The linker issues a warning message for the duplicate symbols, just to let you know that they are present. The ideal solution is to eliminate the duplicate. If you have your warning count set low enough, the warning could cause the linker to terminate prior to completion of the link.

Order Dependent: Yes.

Default State/Value: Turned off. Default is **`-nodup`**.

3.3.4 -D *DataStart*

Description: Allows you to set the start address of the default **`.data`** segment.

The linker must have segments defined in order to complete a link. You can define the segments in the linker definition file (LDF). If you do not, the linker creates default segments for you (for more information, see *Section 4.3, Default Segments*). One of these default segments is named **`.data`**. By default, the **`.data`** segment begins after the **`.rodata`** segment. If you want the **`.data`** segment to begin at some specific address, use the **`-D`** switch. This allows you to set the starting address of this **`.data`** segment to the value given in ***`DataStart`***.

This switch is only used to specify the start address of the **`.data`** segment created by default. If you define a segment named **`.data`** in your LDF, you must use the start parameter as part of the segment definition to locate the segment. If you define any segments, there is no default **`.data`** segment, and use of this switch causes a linker error.

As the default segments **`.bss`** and **`.other`** follow the end of the **`.data`** segment, changing the location of the **`.data`** segment moves these segments also.

The start address value ***`DataStart`*** must be a numeric value in hex, octal or decimal. Refer to *Section 3.2.3, Switch Parameters.* You may override any previous **`-D`** switches on the command line or in a command file by providing another **`-D`** switch to the right of the previous one.

Order Dependent: No.

Default State/Value: **`.data`** segment is located after the **`.rodata`** segment by default.

```
meld -D 0x10000 startup.o vector.o -lc
```

The default segment map is used. Because of the use of the **`-D`** switch, the **`.data`** segment begins at 0x10000 (hexadecimal). The **`.bss`** and **`.other`** segments are affected by this as well.

3.3.5 -ent *EntryLabel*

Description: The entry label is used to mark the start address for code execution. The Motorola Embedded Debugger (MEDB) uses this to determine the starting address for execution. The linker records this address in the output object file.

The default address for the start of code execution is at the entry label **`__start`** (two underscores). This label must be a global label. If the linker cannot find the entry label, it issues an error and terminates without completing the link. You can tell the linker to look for a different label by using the **-ent** switch. The parameter for the switch is any legal symbol name (see *Chapter 4, Linker Definition File*). If you supply an entry label, it must be a global symbol.

You can override the entry symbol given by a previous switch. If more than one **`-ent`** switch appears in the command line or file, the linker uses the last (rightmost) symbol given.

The startup symbol must exist in an object file or the linker definition file (LDF). The linker looks for the startup symbol at the end of the link, not the beginning. Therefore, if the startup symbol is not defined in any object file or library member that is already included as part of the link, it does not exist. The startup symbol is not a reference to that symbol, and does not cause the member that defines it to be extracted from a library and included in the link. If you do not want the linker to search for the startup label, use the **`-noentry`** switch.

Order Dependent: No.

Default State/Value: The default entry label is **`__start.`**

3.3.6 -error *ErrCnt*

Description: This switch sets the error message limit. If there are errors during the link process, the linker issues error messages. As some single error, such as omission of an object file, can generate many error messages, the linker sets a limit. When the linker reaches its error limit, it prints out the last error message and then terminates.

`ErrCnt` specifies the desired error limit. It must be a numeric value in hex, octal or decimal. For more information, refer to *Section 3.2.3, Switch Parameters*. The default error limit is 20.

If you supply an error limit value of 0, this means that there is no error limit, not that only 0 errors are allowed. At certain stages, the linker terminates if there are any errors. It may terminate at one of these points, even though the error limit has not been reached.

You can issue the error switch many times. The linker uses the last (rightmost) error limit given.

Order Dependent: No.

Default State/Value: Error limit of 20.

3.3.7 **-f** *CommandFile*

Description: All of the switches and files that you can specify on the command line can also be placed in an ASCII file. This file can then be specified with the **`-f`** switch. At the point where the switch is encountered, the linker takes all of the commands in the command file and starts processing them from left to right, and then top to bottom.

This has the same effect as though all of the commands and file names in the command file were inserted on the command line in place of the **`-f`** switch and its parameter. After the last switch or file name is encountered in the command file, the linker resumes processing back on the command line, after the switch.

The linker uses a command file, if any, specified by the startup environment variable **`MELDRC`**. If this variable is defined to be the name of a file that can be opened and read, the linker uses this as the command file as though it were specified by the first switch on the command line. For more information on the command file, see *Section 3.4, Command Files*.

Order Dependent: Yes. It is processed when encountered.

Default State/Value: Default is the file name that is specified by the environment variable **`MELDRC`**, if defined. Default command file is processed at the beginning before any other command line switches or file names.

3.3.8 **-l** *LibKey*

Description: This switch allows the user to specify a library file or an object file to the linker. The parameter supplied with the switch must be placed directly against the switch. That is, without an intervening space or delimiter. This parameter is not the file name, but rather a key that is used to construct the file name. Traditionally, this switch is used to specify a library.

The ***`LibKey`*** parameter is simply a string of characters. The linker appends this string to the prefix '**`lib`**'. It then takes the resultant string and appends '**`.a`**' which becomes the file name. For example, if ***`LibKey`*** is '**`c`**', the file name is **`libc.a.`**

The linker searches for this file in the library path list. The linker maintains a list of directories that is searched for the file constructed using ***LibKey***. Each directory is searched, in order, until the file is found in one of them. You can add to this list of directories. For more information, see *Section 3.3.9, -L Directory*. If the file is not found using the library path list, the current directory is searched. Once the linker finds the file, it opens it to see if it should be treated as an object file or library file. Typically this switch is used for specifying libraries. If the file cannot be found or opened, or exists but is not a correct format, the linker issues an error message.

This switch is used to specify a file, so it is not an *order-dependent* switch, but rather, is the target object of order-dependent switches. These other switches determine the linker's mode of processing the file that the **-l** switch specifies.

Order Dependent: No, but it specifies a file, and so is the target object of *order-dependent* switches.

Default State/Value: None.

Assume the current library path list is:

/usr/projecta/lib/

(which contains: **libutil.a**)

Type:

```
meld startup.o vector.o -lutil
```

The linker uses the **-l** switch to construct the file name **libutil.a.** The directory **/usr/projecta/lib/** is searched for this file. The file is found and is a library file. This file is searched for object members that define unresolved but referenced symbols.

Assume the current library path list is:

/usr/projecta/lib/

(which contains: **libutil.a**, **libstartup.a**)

Type:

```
meld -lstartup vector.o -lutil
```

The linker uses the **-l** switch to construct the file name **libstartup.a.** The directory **/usr/projecta/lib/** is searched for this file. The file is found, and is an object file, not a library. It is included in the link, whether any symbols defined in it are referenced or not. This example shows one method of referencing a generic startup file or member from a common project directory.

The linker constructs the file name **libutil.a** which is found in **/lib/projecta/lib/**. It is found to be a library file and is treated as such.

Assume the current library path list is:

/usr/projecta/lib/

(which contains: **libutil.a**, **libstartup.a**)

and assume file **libdsp.a** is in the current directory.

Type:

meld -lstartup vector.o -ldsp -lutil

The linker uses the **-l** switch to construct the file name **libstartup.a.** The directory **/usr/projecta/lib/** is searched for this file. The file is found, and is an object file, not a library. It is included in the link. The file **vector.o** is included in the link.

The linker uses the **-ldsp** switch to construct the file name **libdsp.a.** The directory **/usr/projecta/lib** is searched. The file is not found. Next, the current directory is searched. The file is found, is a library file, and is treated as such.

Finally, the linker uses the **-lutil** switch to construct the file name **libutil.a**, which is found in **/lib/projecta/lib**. It is a library file, and is treated as such.

3.3.9 -L *Directory*

Description: This switch is used to add directories (paths) to the library path list. This list is only used to search various directories for object or library files specified using the **-l** switch.

The library path list starts out empty. Each time the **-L** switch is encountered, the linker adds the directory specified by ***Directory*** to the end of the library path list.

Whenever the linker encounters the **-l** switch, it constructs the desired file name, and then searches for the file in each of the directories (paths) currently in the library path list. If the file is not found in any of these directories, the current directory is checked.

When the linker processes the **-l** switch, only the directories added to the library path list by previous **-L** switches are considered. Library path specifiers which lie to the right of the **-l** switch are not yet added to the library path list. Therefore, the **-L** switch is an order-dependent switch.

The ***Directory*** parameter switch may be any legal absolute or relative path name, such as **../dave/lib**. You can end the path name with '**/**' or not, as you prefer. For example, **C:/usr/lib/** or **C:/usr/lib** are both valid. Do not use a file name for this parameter, such as **C:/usr/lib/libc.a**.

Order Dependent: Yes. Only the paths in effect before a library **-l** switch is encountered are searched.

Default State/Value: The library path list is initially empty, so only the current working directory is searched .

```
meld -lc startup.o vectors -L /usr/lib -lutil -L /usr/dave/mylib
     -ltrans
```

The first **-l** switch specifies the file **libc.a**. The linker searches the current directory.

The second **-l** switch specifies the file **libutil.a**. The linker searches **/usr/lib/** and then the current directory.

The third **-l** switch specifies the file **libtrans.a**. The linker searches **/usr/lib/**, then **/usr/dave/mylib/** and then the current directory.

3.3.10 -LIST *ListFile*

Description: This switch specifies the name of the list file output by the linker. If no other list specific options are given, it also causes the linker to turn on certain default listing options.

The linker has several options which causes information to be output to the list file. The linker creates a list whose default name is **./linker.lst**. You can use the **-LIST** switch to change the name of the list file. The list file name is specified as parameter ***ListFile***.

This switch is *order-independent.* That is, it does not affect the way input objects are processed. If more than one of these switches appear on the command line, the linker uses the name given by the last (rightmost) switch.

The **-LIST** switch also turns on default listing content switches. The listing content switches are **-seg**, **-sym**, **-symn**, **-syma**, and **-xref**. If none of these switches appear on the command line or an included command file, then the **-LIST** switch also turns on certain default listing contents. The default contents is the same as though **-seg**, **-symn**, and **-xref** were typed on the command line.

As the list file is an output file, you need to take care that you supply a file name with the **-LIST** switch. If you do not, one of your input object files or libraries could be mistaken for a list output file. The linker would then erase this object file.

Order Dependent: No.

Default State/Value: Default list file name is **linker.lst**. In the default state, no list information is output.

3.3.11 -nocaps

Description: Turns off the effect of the **-caps** switch.

The linker provides a **-caps** switch that causes all of the symbols defined or referenced in included objects to be converted to uppercase on input. This switch is *order-dependent*, so only the object files or members or objects extracted from library files that are named on the command line after the **-caps** switch are converted. If you want to turn the conversion off for subsequent objects or libraries on the command line, use the **-nocaps** switch.

For more information on the **-caps** switch, see *Section 3.3.1, -caps.*

Order Dependent: Yes.

Default State/Value: Mixed case symbol input (-nocaps mode) is the default.

```
meld startup.o -caps vector.o -lc -nocaps dsp.o -lutil
```

None of the symbols defined or referenced in file **startup.o** are converted to upper case. After the **-caps** switch, object file **vector.o** and any of the object members extracted from library file **libc.a** (the **-lc** switch) have all of their defined or referenced symbols converted to uppercase. After the **-nocaps** switch turns off case conversion, none of the defined or referenced symbols from file **dsp.o** or from extracted members in library **libutil.a** are converted.

Note that if symbol **main** is defined in object **vector.o** it is converted to **MAIN**. If **startup.o** references this, it needs to call **MAIN**, not **main**. Also if object **startup.o** defines symbol **init**, it is not converted. There will not be a way for **vector.o** to reference **init**. Any reference to **init** in **vector.o** is converted to **INIT**.

Unless you carefully manage the case of what is called and what is referenced in each of the files or libraries, it is best to leave case conversion turned off for all of the files and libraries.

3.3.12 -nodup

Description: Directs the linker to disallow duplicate global symbols from object files or objects included from library files that appear on the command line after this switch. The linker permits duplicate global symbol definitions when the **-dup** switch is on. This switch is *order-dependent*, so object files and library files that occur after the **-dup** switch on the command line are affected. For more information on the **-dup** switch, see *Section 3.3.3, -dup.* The **-nodup** switch turns off the duplicate global symbols allowed mode for all subsequent object files or members from library files that are specified on the command line after the switch.

Order Dependent: Yes.

Default State/Value: At the start of command line processing. the linker does not allow duplicate global symbols.

```
meld startup.o -dup vector.o -lc -nodup dsp.o -lutil
```

All of the global symbols defined in file **startup.o** must not conflict with any global symbols already defined. In this case there are none, as this is the first member and there is not a linker definition file to define absolute symbols. After this, the **-dup** switch causes the linker to allow duplicate global symbols. If file **vector.o** defines any global symbols which conflict with global symbols that are already defined, you receive a warning. The same is the case for object members extracted from **libc.a** (the **-lc** switch).

The **-nodup** switch turns off allowing duplicates. Object file **dsp.o** and object members extracted from **libutil.a** may not define global symbols that are duplicates of global symbols already defined, even if they were defined by **vector.o** when duplicates were allowed.

3.3.13 -noent{ry}

Description: The **-noentry** switch directs the linker not to look for an entry point symbol. The entry address field of the output object receives the value 0. No error message is issued if the entry label does not exist, as the linker does not look for it.

This switch is not *order-dependent*. The **-ent** switch specifies the entry label. It does not specify that an entry label search is to be done. So if the **-noentry** switch and the **-ent** switch are both on the command line in any order, the linker does not look for the entry label specified by the **-ent** switch.

For the **-noent{ry}** switch to be recognized by the linker, you must type **-noent** and you may optionally type any leading part of the string shown in the braces (**{ry}**). Do not type the braces **{}**.

Order Dependent: No.

Default State/Value: The linker looks for an entry point, and the default entry label is **__start.**

```
meld -noentry -ent EntryLabel vector.o -lc
```

The linker does not look for any entry label. The entry address field in the output object is set to 0. If the **-noentry** switch is removed, the linker searches for the entry label **EntryLabel**.

3.3.14 -o *OutFile*

Description: This switch specifies the name of the output object file generated by the linker. The linker outputs the linked object files or members into an output file in ELF format. The default name for this file is `./a.out`. As you can see, it is created in the current directory.

If you want the file name to be something other than the default, use this switch. The parameter to the switch is the desired name of the output file.

This switch is not *order-dependent*. You can put several of these on the command line or in a command file, and the linker uses the last (rightmost) switch to determine the output file name.

Order Dependent: No.

Default State/Value: The default output file name is `./a.out`.

3.3.15 -pad *PadChar*

Description: The linker gathers sections of code or data from input object files or from extracted members from library files. As these input sections are located in the output segments, the alignment requirements of the sections must be taken into account. If the alignment criteria are not naturally met, padding must be inserted between the end of the last section already placed in the segment and the current section being placed. The linker fills these small padding holes with the pad byte, whose default value is 0. If you want some other pad value you can use this switch.

The pad byte value ***`PadChar`*** must be a simple numeric value between 0 and 255, specified in hex, octal or decimal. For more information refer to *Section 3.2.3, Switch Parameters*. You may override any previous **`-pad`** switches on the command line or in a command file by providing another **`-pad`** switch to the right of the previous one.

Order Dependent: No.

Default State/Value: The default value of the pad byte is 0.

3.3.16 -q

Description: This switch causes suppression of the linker banner message. This message provides the name of the linker, the version number, and copyright information. This switch is primarily intended for use by other tools, such as the compiler, that call the linker as part of their processing. Motorola recommends that you do not suppress the linker banner, so that version information is available if you should need to call for technical support.

Order Dependent: No.

Default State/Value: Default is to issue the banner message.

3.3.17 -r

Description: Changes the ELF form of the object file output by the linker to be relocatable, which is suitable as input to a subsequent link. Also allows unresolved symbols to be referenced. In effect, this causes the linker to do a partial link. The linker accepts only ELF relocatable form object files or object members of libraries as input for the link. It normally outputs the resultant output file in ELF executable form. This form cannot be re-input into the linker. If you direct the linker to produce a relocatable output object using the **`-r`** switch, then this object can be input to the linker in a subsequent link. The effect of this two-step process is to gather as many sections together from input objects as you can and prelink them into one larger object file or member. Later, you can link this with other files or members not included in the first partial link.

If a first partial link references symbols that are not defined in the link, the linker permits this. It outputs the symbols as referenced but not defined. Partial links are not in a form suitable for S-record extraction.

Order Dependent: No.

Default State/Value: Linker outputs an executable ELF file by default, not a relocatable one.

```
step 1: meld -r -o partial.o startup.o vector.o -lc
```

The linker combines the sections in **`startup.o`** and **`vector.o`** into the output object, **`./partial.o.`** If there are any unresolved symbolic references, the linker searches **`libc.a`** for them, and, if any are defined, it extracts the defining members and includes them in the output object files. The output object file is in relocatable form. Any symbols which are referenced but still not defined are output to the relocatable output object as a reference to an undefined symbol.

```
step 2: meld -o final.o partial.o dsp.o -lc
```

Later, the user has developed **`dsp.o`**, which contains the final code necessary to complete the application. The linker takes the combined sections from the previous link, which are in **`partial.o`**, and combines them with those from **`dsp.o`**. If there are any unresolved references, the linker searches the library **`libc.a`** for any object members which define them. The linker outputs an executable form object file for this link. There must not be any unresolved symbolic references after searching the library.

3.3.18 -seg

Description: This switch directs the linker to provide a listing of the segments and sections in the output object file.

For each segment the linker lists the name, start address, size, and type. The total size of the segments is also given.

For each section the linker shows the segment that contains the section, the section start address, size, and type. The linker lists for each output section the name of all of the input sections that comprise the output section and the name of the input file that each one came from.

If the **-seg** switch is given with the **-LIST** switch, it causes the **-LIST** switch to not turn on the default listing contents. See *Section 3.3.10, -LIST ListFile.*

Segments are listed in definition order.

Order Dependent: No.

Default State/Value: No listing is produced.

3.3.19 -sym, -symn, -syma

Description: There are several switches which cause the linker to list symbol information. They all produce a similar result with minor differences, so they are described together. When the linker lists symbol information, it outputs the symbol's name, value, size, type, and visibility, and the section where the symbol is defined. For more information, see *Chapter 5, List File*.

The **-sym** switch is just a short form of the **-symn** switch. Both switches cause the symbol listing to be output and be sorted by name using ASCII values in ascending order. The **-syma** switch causes the symbols to be output and sorted by their value in ascending order. If the **-sym**, **-symn** or **-syma** switch is given with the **-LIST** switch, then the **-LIST** switch does not turn on the default listing contents. See *Section 3.3.10, -LIST ListFile.*

If **-syma** and **-symn** (or **-sym**) are listed, two symbols listings are output, one sorted by each method.

Order Dependent: No.

Default State/Value: No listing is produced.

3.3.20 -T *TextStart*

Description: Allows you to set the start address of the default **.text** segment.

The linker must have segments defined to complete a link. You can define the segments in the linker definition file (LDF). If you do not, the linker creates default segments for you. For more information see *Section 4.3, Default Segments*. One of these default segments is named **.text**. The default start address for the **.text** segment is 0x200 (hexadecimal). If you want the **.text** segment to begin at some other specific address, use the **-T** switch. This allows you to set the starting address of this **.text** segment to the value given in ***TextStart***.

This switch is only used to specify the default start address of the **.text** segment. If you define a segment named **.text** in your LDF, you may use the start parameter as part of the segment definition to locate the segment. If you define any segments, there is not a default **.text** segment, and use of this switch causes a link error.

As the default segments **.rodata**, **.data**, **.bss**, and **.other** follow the end of the **.text** segment, changing the location of the **.text** segment moves these segments also.

The start address value ***TextStart*** must be a numeric value in hex, octal or decimal. Refer to *Section 3.2.3, Switch Parameters.* You may override any previous **-T** switches on the command line or in a command file by providing another **-T** switch to the right of the previous one.

Order Dependent: No.

Default State/Value: Default **.text** segment is located at 0x200 (hexadecimal).

```
meld -T 0x1000 startup.o vector.o -lc
```

The default segment map is used. Because of the use of the **-T** switch, the text segment begins at 0x1000. The **.rodata**, **.data**, **.bss**, and **.other** segments are affected by this as well.

3.3.21 -warn *WrnCnt*

Description: This switch sets the warning message limit. If there are warnings during the link process, the linker issues warning messages. The linker allows you to set a warning limit. When the linker reaches its warning limit, it prints out the last warning message and then terminates.

WrnCnt specifies the desired warning limit. It must be a numeric value in hex, octal or decimal. For more information, refer to *Section 3.2.3, Switch Parameters.* A warning limit of 0 means that there is no warning limit, not that 0 warnings are allowed. The default is no warning limit.

You can specify the warning switch many times; the linker uses the last (rightmost) warning limit given.

Order Dependent: No.

Default State/Value: The default is no limit on warning messages.

3.3.22 -weak

Description: This switch causes the linker to convert all global symbols to weak symbols before output. This can be used to convert global symbols in a needed object file or library member to weak so that they do not conflict with other symbols in other objects. Weak symbols have global visibility, but if they conflict with other globals, they are converted to local symbols. All external references to converted weak symbols that are in the same file or member where the converted symbol was defined still uses the now local definition.

This switch is not *order-dependent*, as it affects symbols as they are output, not as they are input from the included object files or extracted library objects. All of the global symbols that are included in the link and are output are converted to weak.

This switch is intended to be used in conjunction with the **`-r`** switch. If there is an object file that is needed but defines a duplicate symbol, you can process it through the linker with the **`-r`** and **`-weak`** switches. The file is output as a relocatable ELF object file but all of the global symbols are converted to weak.

If the object file or member with the conflicting symbol is a member of an object library, extract it first, then convert it, then replace it with the converted version. When the converted object is now used, the conflicting global symbol is of type weak.

Order Dependent: No.

Default State/Value: The default is not to convert global symbols to weak.

Assume object file **`startup.o`** defines a global symbol **`convert`**. Object file **`vector.o`** externally references symbol **`filter`** and **`convert`**. Object file **`dsp.o`** globally defines symbol **`filter`** and both references and globally defines symbol **`convert`**.

`step 1: meld -r -o dspl.o -weak dsp.o`

This creates a file **`dsp1.o`** that is like **`dsp.o`** except that all of the global symbols are weak.

`step 2: meld -o project startup.o vector.o dsp1.o`

Now the reference to symbol **`convert`** in file **`vector.o`** is resolved by the definition in **`startup.o`**. The reference to symbol **`filter`** in file **`vector.o`** is resolved by the definition in file **`dsp.o`**. The definition of symbol **`convert`** in file **`dsp1.o`** is converted to local, and does not conflict with the definition in file **`startup`**. Further, the reference to symbol convert in file **`dsp1.o`** uses its own definition as intended.

3.3.23 -xref

Description: This switch directs the linker to provide a cross-reference listing of the sections showing what symbols they define and what symbols they reference. As these are broken out by section, the linker outputs a section listing showing the composite input sections as well.

For each section, the linker lists symbols in two groups: symbols defined and symbols referenced. For each group, the linker lists the names and values of the symbols. If either group (define or referenced symbols) is empty then it is not listed.

If the **-xref** switch is given with the **-LIST** switch, it causes the **-LIST** switch to not turn on the default listing contents. See *Section 3.3.10, -LIST ListFile.*

Order Dependent: No.

Default State/Value: No listing is produced.

3.4 COMMAND FILES

A command file is an ASCII file that contains linker commands. The linker processes the commands in this file as though they were typed on the command line. The linker replaces end-of-lines (carriage returns) in this file with a space so that a multi-line file looks like a long single line command. Any command can be used in the file including one that starts processing (calls) other command files.

If the line ends with the character '\', the linker does not replace the end-of-line character with a space. Instead, it concatenates the next line to the first. For example, if the first line ends with **filename.** and the second line begins with **obj**, the linker sees the command input as **filename.obj**.

The linker processes switches, object files, and library files from left to right and then top to bottom order in the command file. This order determines what switches are in effect for the processing of object files or object members of library files. This also determines the inclusion order of object files and object members of libraries.

```
meld -f file.cmd
```

Assume the contents of **file.cmd** are:

```
startup.o
vector.o
-lc
```

The above example is the same as typing the following on the command line:

```
meld startup.o vector.o -lc
```

```
meld -f file.cmd
```

Assume the contents of **file.cmd** are:

```
start\
up.o
vector.o
-lc
```

The above example is the same as typing the following on the command line:

```
meld startup.o vector.o -lc
```

When processing the command file, the linker processes all of the files and switches as though they were inserted in the command line at the point of the 'call' to the command file (where the **-f** switch appears). The command file is then processed left to right and from top to bottom. After all of the command file is processed, additional command processing picks up after the point of call.

If the call is from the command line, after processing the command file processing resumes with the command line. If the call is from another command file, processing resumes from the point in the previous command file after the call.

```
meld startup.o -f file1.cmd -lc
```

Assume the contents of **file1.cmd** are:

```
vector.o -f file2.cmd
dsp.o
```

and the contents of **file2.cmd** are:

```
vectrtn.o
```

The above example is the same as typing the following on the command line:

```
meld startup.o vector.o vectrtn.o dsp.o -lc
```

NOTE

If environment variables, history substitution, globing (***** or **?** wild cards), sets (**[ ]**), command substitution or any other shell replacement feature appears in the command file, the linker **will not** provide the needed replacement. These shell-specific features are not supported in linker command file processing.

The linker checks for the definition of an environment variable by the name **MELDRC**. If this variable is defined, it is interpreted by the linker to be the file name of a command file. The linker attempts to open this file and treat it as a command file before any of the command line is processed. After the file is processed, command processing begins at the start of the command line. The effect of this is the same as though the file were named at the start of the command line using the **-f** switch.

MELDRC usage:

meld vector.o

MELDRC is defined to be **C:/usr/project/standard.cmd**

This is the same as though the following command line were typed, and **MELDRC** were not defined:

```
meld -f C:/usr/project/standard.cmd vector.o
```

If the environment variable is not set, or it is set to some string that is not a legal file name, or the file it names cannot be found or opened for read access, the linker does not issue an error. In this case, the linker continues without automatically opening any command file at startup. If you define **MELDRC** to name a read-access file but it is not a command file, you get many command line error messages.

The startup command file is useful when certain object files, library files, option switches or a linker definition file (LDF) is used for every link. Instead of retyping this information you can specify it in a command file and define **MELDRC** to point to this file. For example, the project may have a common memory map defined in an LDF. The **MELDRC** environment variable can be set for everyone on the project to the command file that tells the linker to use this LDF.

Remember, dependent switches are processed in the order in which they are encountered even within nested command files.

meld vector.o

MELDRC is defined to be **C:/usr/project/standard.cmd**

The contents of file **C:/usr/project/standard.cmd** are:

```
startup.o
-def C:/usr/project/memory.ldf
-lc
```

This is the same as though the following command line were typed and **MELDRC** were not defined:

```
meld -f C:/usr/project/standard.cmd vector.o
```

As **C:/usr/project/standard.cmd** is a command file, this expands as though the following were typed:

```
meld startup.o -def C:/usr/project/memory.ldf -lc
     vector.o
```

CHAPTER 4

LINKER DEFINITION FILE

A linker definition file (LDF) is an ASCII text file that serves three main purposes:

- It defines segments, which are the major groups of code or data that comprise the application being linked.
- It defines absolute symbols, which your code can reference as address or data constants.
- It can turn on checking operations to be performed on the segments you define, and the sections that the linker assigns to these segments.

This chapter discusses segments and sections, how the linker processes input sections, and the default segments the linker creates when you do not specify an LDF. The rest of this chapter discusses the syntax, semantics, and use of an LDF.

4.1 SEGMENTS AND SECTIONS

A segment is an area of an executable object file containing code or data that is loadable into physical memory on your target system. A segment contains sections from the input object files and object members (library file members in object file format) included in the link. Each segment represents an area of memory that a debugger will load or an S-record generator will convert to S-records. The linker creates segments in the output object that reflect the memory layout of the target hardware platform.

Each segment has a unique name. The Motorola S-Record Generator uses segment names to let you generate S-records for a specific segment or a set of segments. See *Appendix C, Motorola S-Record Generator*.

A section is a named area in an object file or object member that may contain code or data. When you write assembly language code, you can control which sections the assembler creates, and what code or data goes into them. If your code source is a higher level language, such as C, then the C compiler creates sections with standard names for you.

The linker does not create sections or section names; it includes them from input object files and library members. The linker uses a section’s name, along with other specifications, to determine which segment will contain the section. Within a segment, the linker will collect together all of the sections with the same name and create one larger section. The linker will not split a section within or across segments.

4.1.1 Reserved Sections

The linker accepts any valid section name, but the Motorola Embedded Assembler (MEAS) is more restrictive. Section names starting with '**.**' are reserved names in the assembler. The assembler does not let you arbitrarily name sections that begin with '**.**'. The name of a section that begins with '**.**' must be one of the reserved section names:

.text	**.data**	**.bss**
.PPC.EMB.sdata0	**.sdata**	**.sdata2**
.PPC.EMB.sbss0	**.sbss**	**.sbss2**

The assembler lets you create sections with arbitrary names if the names do not begin with '**.**'. Reserved sections have special meaning. Some of the meanings are more strictly observed than others. For example, **.text** sections are reserved for executable machine instruction words, while **.data** sections are reserved for data storage. However, none of the Motorola software tools restrict usage of these sections, to code or data. Sections with '**bss**' in their names reserve space for uninitialized data.

4.1.2 Small Data Areas

Sections whose names begin with **.s** or **.PPC.EMB.s** are reserved sections used for small data areas (SDAs). The linker supports the three SDAs that the *PowerPC Embedded Application Binary Interface* standard defines. (The user's manuals for the Motorola Embedded C Compiler and the Motorola Embedded Assembler also contain information about small data areas.)

The three SDAs are:

- SDA — which consists of the sections **.sdata** and **.sbss**. The SDA base register is GPR 13.
- SDA2 — which consists of the sections **.sdata2** and **.sbss2**. The SDA2 base register is GPR 2.
- SDA0 — which consists of the sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0**. The base address of SDA0 always is address 0. Although GPR 0 is the SDA0 base register, this may be misleading. PowerPC processor load and store instructions that specify offsets from GPR 0 always ignore the value in GPR 0. Instead, such instructions calculate the offsets from address 0, regardless of GPR 0's value.

Each small data area may contain no more than 65536 bytes. Your code can access data stored in an SDA via a single load or store instruction, using a signed 16-bit offset from the value in the SDA's base register to the data. Similarly, a single instruction can produce the address of data stored in an SDA. This can reduce the size of your code and increase its execution speed.

The linker assigns the base address for SDA to symbol **_SDA_BASE_**; the linker assigns the base address for SDA2 to symbol **_SDA2_BASE_**. Accordingly, your startup code must assign the values of these symbols to GPR 13 and GPR 2, respectively, before your application accesses data in SDA or SDA2. See *Section 4.9, Use of Romcopy Segments.*

4.2 ASSIGNING SECTIONS TO SEGMENTS

When the linker assigns sections from input object files and library members to each segment in the output files, it imposes an ordering within the segment.

First, the linker arranges sections with certain names or types within the segment in this order:

1. sections named **.PPC.EMB.sdata0**
2. sections containing initialized data that are not small data area sections
3. sections named **.sdata**
4. sections named **.sdata2**
5. sections named **.PPC.EMB.sbss0**
6. sections named **.sbss**
7. sections named **.sbss2**
8. sections containing uninitialized data that are not small data area sections

After the linker arranges sections in this manner, groups 2 and 8, which are the non-small data areas, are arranged so that all sections with the same name are adjacent to one another. Then the linker converts all adjacent sections with the same name into one larger section with that name.

The output object file format (ELF) specifies that uninitialized sections are placed at the end of a segment. If both uninitialized and initialized sections with the same name are in the same segment, the linker cannot create one larger section. Instead, the linker creates two sections with that name: one consisting of all of the initialized sections grouped together, and the other consisting of the uninitialized sections grouped together.

4.3 DEFAULT SEGMENTS

If you do not define any segments in the linker definition file (LDF), the linker defines a set of default segments. As this segment description likely will not be ideal for your target memory layout, you should become familiar with the syntax for describing segments, and define your own.

If you do define any segments yourself, the linker does not define any default segments.

The default segment map is useful when getting started. You can write your application code and link it together using the default map. This helps you verify that all the symbolic references are defined somewhere. The linker can produce a listing file that helps determine the size requirements of your code and data. At this point, you know approximately how much memory your application requires.

The system architect can then design the memory map as segments in the LDF. The linker can then use the LDF in subsequent links.

The default segment map is the same as if you supplied the LDF in Figure 4-1 to the linker.

```
# This segment is used to hold executable code. Note that alignment
# must be 4, and the start address may be different if the user
# supplied a start address using the -T switch.
  segment .text start=0x200 align=4 priority=1 protect=r,x
          section=(.text);

# Segment contains read-only data. This segment, and the .text segment
# above are assumed to reside in ROM. Note that changing the starting
# location of the .text segment will move this segment also.
  segment .rodata start=segafter(.text) align=4 priority=1 protect=r
          section=(.rodata,.sdata2);

# Segment contains read-write data, assumed to be in RAM. Usually
# this segment follows the end of the .rodata segment, but if you use
# the -D switch you can start the segment where you wish.
  segment .data start=segafter(.rodata) align=4 priority=1
          section=(.data,.sdata,.sbss);

# Segment contains uninitialized data. Note that this segment follows
# the .data segment, so when .data moves, this segment moves.
  segment .bss start=segafter(.data) align=4 priority=1
          section=(.bss,.sbss2);

# Segment contains all other sections that otherwise do not fit above.
# This is a catch-all segment. Note that it comes last, so that
# preceding segments will have first grabs at the sections that
# belong there. Also note: this segment follows the .bss segment, so
# when .bss moves, this segment moves.
  segment .other start=segafter(.bss) align=4 priority=1
          protect=r,w,x section=(*);
```

Figure 4-1. Default Segments

You will understand more about the syntax used in the default LDF after you read *Section 4.4, General Linker Definition File Syntax*, *Section 4.5, Segment Declarations*, and *Section 4.6, Symbol Declarations*.

4.4 GENERAL LINKER DEFINITION FILE SYNTAX

4.4.1 Declarations and Directives

A linker definition file (LDF) contains a set of segment and symbol declarations, and directives to the linker. The LDF syntax is free form: You may insert spaces and newlines (carriage-returns, linefeeds) freely between keywords, numbers, operators and functions. Every declaration or directive ends with a semi-colon (**;**). A declaration or directive may span as many lines as you desire, but you may not put a declaration or directive inside another declaration or directive.

Blank lines are permitted. Anywhere the pound sign (#) appears on a line, the rest of that line is ignored as a comment.

The LDF may contain as many as six types of declarations and directives, each starting with a keyword. These declarations and directives are:

- **seg{ment}** Declares and defines a segment in the memory layout.
- **symbol{define}** Declares an absolute symbol and its value. Same as **sym{define}**.
- **sym{define}** Declares an absolute symbol and its value. Same as **symbol{define}**.
- **checkov{erlap}** Directs linker to check whether segments that lie between this directive and the next **nocheckov{erlap}** directive overlap each other.
- **nocheckov{erlap}** Directs the linker to discontinue overlap checking for the segments that follow this directive.
- **single{section}** Directs the linker to check that every section of any given name exists in only one segment.

When the syntax shows part of a keyword in braces **{}**, that part of the keyword is optional. You must type all of the keyword before the braces, followed by any number of characters from the optional part. For example, for the **segment** (**seg{ment}**) keyword, **seg**, **segm**, **segme**, **segmen**, and **segment** are all valid.

4.4.2 Names

No two segments declared in the LDF may have the same name, and no two symbols declared may have the same name. However, a segment may have the same name as a symbol or section, and a symbol may have the same name as a segment or section. (The linker does not declare sections and give them names: their names and contents are defined in input files and libraries.)

A segment name, symbol name, or section name in the LDF must start with one of the characters in **A**—**Z**, **a**—**z**, **$**, **.** (period), and _ (underscore). Characters after the first may be any of the start characters or **0**—**9**.

The linker is case sensitive when processing segment, symbol, and section names. For example, the segment name '**Rom**' is not the same as '**rom**'.

4.4.3 Literal Numbers

Within declarations you can specify integer values, such as the size of a segment or the value to assign to a symbol. The simplest way to specify a value is to type a literal number.

A *literal number* is an integer value specified as a decimal (base 10), hexadecimal (base 16), or octal (base 8) number. A decimal number consists of a string of one or more characters from **0**—**9**, where the first character is not **0**. A hexadecimal number consists of **0x** or **0X** followed by a string of one or more characters from **0**—**9**, **A**—**F**, and **a**—**f**. An octal number consists of **0** followed by a string of one or more characters from **0**—**7**.

4.4.4 Segment and Symbol Expressions

Several specifiers in a segment declaration let you type a value as a segment expression. A *segment expression*, shown in the syntax as ***SegExpr***, may consist of:

- literal numbers.
- the unary operators **()**, **+** (unary plus), and **-** (unary minus).
- the binary operators **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), and **%** (modulo).
- the segment functions **segstart()**, **segend()**, **segafter()**, **segsize()**, and **segisize()**. (See *Section 4.4.5, Segment Functions*.)

A **symboldefine** or **symdefine** declaration lets you assign to a symbol the value of a symbol expression. A *symbol expression*, shown in the syntax as ***SymExpr***, may consist of:

- literal numbers.
- the unary operators, the binary operators, and the segment functions of a segment expression.
- the symbol functions **sectstart()**, **sectend()**, **sectafter()**, **sectsize()**, and **addrof()**. (See *Section 4.4.6, Symbol Functions*.)

The unary operators work the same as in the C language; e.g., they are right associative. In a segment expression, the operand of a unary operator is any segment expression; in a symbol expression, the operand is any symbol expression.

The binary operators work the same as in the C language; e.g., they are left associative. In a segment expression, the two operands of a binary operator are any segment expressions; in a symbol expression, the operands are any symbol expressions.

The precedence of operators is the same as in the C language. From highest to lowest precedence:

1. **()** operator
2. **+** and **-** unary operators
3. *****, **/**, and **%** operators
4. **+** and **-** binary operators

The result of a segment expression or a symbol expression is *always* an unsigned 32-bit integer. The linker treats values as unsigned because they generally represent physical addresses. The linker behaves as though every operator produces a result that must first be assigned to an unsigned 32-bit integer in the C language. The linker truncates every expression result before using it as the operator of the next expression. Therefore, a binary operator always operates on two 32-bit unsigned integers and produces a truncated 32-bit unsigned integer result.

Example 4.1:

You enter the expression: **3/(-1)**

You expect the result -3 (0xFFFFFFFD).

Instead, you get 0.

Why? Because the unary operator **-** operating on **1** produces the unsigned result 0xFFFFFFFF. Then 0x3 divided by 0xFFFFFFFF produces a number much less than 1, which the linker truncates to 0.

Example 4.2:

`5-3-1` results in 1. Because **`-`** is left associative, this is the same as **`(5-3)-1`**.

`5-(3-1)` results in 3.

Remember that the result of any expression is an unsigned 32 bit integer, so typing **`-1`** is a shorthand way of typing **`0xFFFFFFFF`**. Also, the linker does not allow use of unary operators **`+`** or **`-`** adjacent to the binary operators **`+`** or **`-`**.

Example 4.3:

`0xff8 + -1` is not allowed.

`10 - +3` is not allowed.

4.4.5 Segment Functions

A *segment function* returns information about the size, start address or end address of the segment whose name is passed as an argument to the function. Segment functions return unsigned 32-bit integers. The segment functions are:

- **`segstart(`*`SegmentName`*`)`** — This function returns the starting byte address of the segment named ***`SegmentName`***.
- **`segend(`*`SegmentName`*`)`** — This function returns the byte address of the last byte contained within the segment named ***`SegmentName`***. Note that this function is the same as **`(segafter(`*`SegmentName`*`) - 1)`**.
- **`segafter(`*`SegmentName`*`)`** — This function returns the byte address of the first byte beyond the end of the segment named ***`SegmentName`***. Note that this function is the same as **`(segend(`*`SegmentName`*`) + 1)`**.
- **`segsize(`*`SegmentName`*`)`** — This function returns the total size in bytes of the segment named ***`SegmentName`***. This is the size of both the initialized and uninitialized parts.
- **`segisize(`*`SegmentName`*`)`** — This function returns the size in bytes of the initialized portion of the segment named ***`SegmentName`***. The linker places all of the initialized data at the start of the segment. This function returns the size of that part of the segment.

Example 4.4:

`segafter(.text)` evaluates to the first byte address after the end of segment **`.text`**.

Be careful not to build descriptions that have cyclic expressions. The linker will not be able to evaluate such expressions and will terminate with an error.

Example 4.5:

```
segment .text start=segafter(.data) (.text);
segment .data start=segafter(.text) (.data);
```

This produces a cycle. The linker must know the start address of segment **.data** to evaluate the start address of segment **.text**. The start address of segment **.data**, however, depends on the start address of segment **.text**.

For more information about cycles and preventing them, see *Section 4.10.1, Cyclic Constraints.*

Example 4.6:

```
segment ramspace start=segafter(romspace) + 0x100 ...;
```

This defines a segment named **ramspace** that begins 0x100 (hexadecimal) bytes after the end of the **romspace** segment.

4.4.6 Symbol Functions

A *symbol function* returns information about a section or the value of a symbol. The name of the section or symbol is passed as an argument to the function. Symbol functions return unsigned 32-bit integers.

Most of the added functions return information about sections. The sections are referenced by name. It is possible that several sections may exist with the same name. If this is the case, the linker will return information about the first section (lowest address) found in the first segment defined that contains a section with the given name. Note that depending on the definition order and start address of the segments, this may not result in the lowest address section with the specified name.

The symbol functions are:

- **sectstart(*SectionName*)** — This function returns the byte address of the start of the first section found with the name ***SectionName***.
- **sectend(*SectionName*)** — This function returns the byte address of the last byte contained within the first section found with the name ***SectionName***. Note that this is the same as **(sectafter(*SectionName*) - 1)**.
- **sectafter(*SectionName*)** — This function returns the byte address of the first byte after the end of the first section found with the name ***SectionName***. Note that this is the same as **(sectend(*SectionName*) + 1)**.

- **sectsize(*SectionName*)** — This function returns the size of the first section found with the name ***SectionName***.
- **addr{of}(*SymbolName*)** — This function returns the value of the symbol named ***SymbolName***. If the symbol is a label, the return value is an address. If it is an absolute symbol, the return value is the value of the symbol. The function name is specified as **addr{of}**, which means that **addr**, **addro**, or **addrof** will be recognized as the keyword for this function.

4.5 SEGMENT DECLARATIONS

Segment declarations define segments that the linker creates. The segments describe major portions of the memory layout. Motorola recommends that you create a separate segment for each of the following:

- Each non-contiguous area of memory.
- Each differing technology type, such as RAM vs. ROM.
- Special purpose areas, such as regions of RAM that your code needs to initialize at system boot time.

Most segments will contain code or data that the linker gathers from input sections. The linker assigns sections from input object files and library members to the segments that you declare. Which sections go into which segments is under your control.

The syntax of a segment declaration is:

```
seg{ment} SegmentName [typ{e}=TypeSpec]
          [start=SegExpr] [size=SegExpr] [end=SegExpr]
          [min{size}=SegExpr] [max{size}=SegExpr]
          [align{ment}=AlignNum] [pri{ority}=PriorityNum]
          [pro{tect}=ProtectSpec,ProtectSpec,...]
          [[sec{tion}=](NameTemplate,NameTemplate,...)] ;
```

Each segment declaration must start with the key word **segment**, followed by the name of the segment, followed by any desired or required specifiers, and ending in a semicolon (**;**). Specifiers (keywords followed by **=** followed by parameters) may be in any order. The declaration may span several lines before you end it with a semicolon.

In the syntax, **=** is the equals operator. It represents that either the equals sign (**=**), a colon (**:**) or a colon with an equal sign (**:=**) is required.

When the syntax shows part of a keyword in braces **{}**, that part of the keyword is optional. That is, you must type all of the keyword in front of the braces, followed by any number of characters from the optional part.

If the keyword or equals operator is in brackets **[]**, typing the keyword or equals operator is optional. Many specifiers have default values when they are unspecified.

Example 4.7:

```
seg     .text start=0x100 ...;
segme   .text start:=0x100 ...;
segment .text start:0x100 ...;
```

Each of the above segment declarations uses a different style equals operator and version of the **segment** keyword, but they all declare that segment **.text** begins at 0x100 (hexadecimal).

4.5.1 Segment Name

No two segments declared in the LDF may have the same segment name (***SegmentName***). The rules for naming segments are in *Section 4.4.2, Names*.

4.5.2 Segment Type

typ{e}=stand{ard}

typ{e}=res{erved}

typ{e}=rom{copy}(*SegmentName*)

There are three types of segments. The most common is the *standard* type (**type=standard**), which is the default type. It describes a region of memory in which the linker is to place sections. The segment declaration for a standard segment must have a **section** specifier.

A *reserved* segment (**type=reserved**) may or may not have a **section** specifier. A reserved segment with no **section** specifier will simply reserve space in memory. If a reserved segment has a **section** specifier, it will have sections assigned to it. The linker writes those sections to the output object file, but the output file does not have segment information for the reserved segment. So a loader, debugger or S-record generator will not extract a reserved segment.

A *romcopy* segment (**type=romcopy(*SegmentName*)**) does not have sections assigned to it. Instead, it reserves space for sections assigned to the segment named ***SegmentName***. ***SegmentName*** is usually a segment located in RAM that contains initialized read/write data. ***SegmentName*** is the final run-time address of the data, but initial values for the data are kept in the romcopy segment, which is in ROM.

If you declare a romcopy segment, the linker makes the romcopy segment the same size as the copied segment. All of the symbols defined for the data are located within the RAM segment. At load time, or S-record extract time, the initial values for the data are copied from the romcopy segment to the RAM segment.

You should supply system boot code that will copy this segment into the RAM segment. See *Section 4.9, Use of Romcopy Segments,* for more information on the use of romcopy segments.

4.5.3 Segment Start Address

`start=`*`SegExpr`*

The **`start`** specifier declares the starting byte address for the segment. Every segment must have a start address, either explicitly declared or by default. The start address is a segment expression (***`SegExpr`***). See *Section 4.4.4, Segment and Symbol Expressions.*

If a segment declaration does not have a **`start`** specifier, the default is that the segment begins at the next byte after the last byte within the segment declared immediately preceding. If the first segment declared does not have a **`start`** specifier, its start address is 0.

4.5.4 Segment Size

`size=`*`SegExpr`*

`end=`*`SegExpr`*

`min{size}=`*`SegExpr`*

`max{size}=`*`SegExpr`*

There are several ways to specify the size or extent of a segment. The simplest way is to not specify it. In that case, the linker will make the segment large enough to contain all sections assigned to it, including the padding needed for alignment. Thus, the default size of a segment is *unconstrained*.

Each of the size-constraining specifiers in this section requires a segment expression (***`SegExpr`***). See *Section 4.4.4, Segment and Symbol Expressions.*

The **`size`** specifier sets the segment's size in bytes to the value given by ***`SegExpr`***. Only one **`size`** specifier is allowed in a segment declaration.

Example 4.8:

```
segment .text size=0x100 ...;
```

This sets the segment size to 0x100 (hexadecimal) bytes.

Example 4.9:

```
segment .rom size=segsize(.data) ...;
```

This sets the segment size to be the same as the size of the **.data** segment.

The **end** specifier's ***SegExpr*** segment expression specifies the segment's end address (the address of the last byte with the segment). The end address determines the segment's size, since size = end - start + 1. You might use this specifier when you know the end address of a memory region. Only one **end** specifier is allowed in a segment declaration.

If you use the **end** specifier when the segment's start address depends on information about other segments, the link may fail. For information about why, see *Section 4.10.1, Cyclic Constraints.*

The **minsize** specifier lets you set the minimum size of the segment to the value of the segment expression ***SegExpr***. After the linker places sections and adds alignment padding, if the segment size is less than that given by the **minsize** specifier, the linker increases the segment size to the value of ***SegExpr***. The default is no minimum size for a segment. Only one **minsize** specifier is allowed in a segment declaration.

The **maxsize** specifier lets you set the maximum size of the segment to the value of the segment expression ***SegExpr***. The linker will place sections in the segment until the placement of any section would cause the segment to exceed the maximum size. Only one **maxsize** specifier is allowed in a segment declaration.

The **size** and **end** specifiers specify the size of the segment. The **minsize** and **maxsize** specifiers do not set the size, but put constraints on the size. It does not make sense to constrain the size of something that has a size declared, so you cannot use a **minsize** or **maxsize** specifier with an **end** or **size** specifier. Also, since the **end** and **size** specifiers do the same thing, you cannot use them both in the same segment declaration. You can use **minsize** and **maxsize** in the same segment declaration.

Romcopy segments may not have any specifiers that declare or constrain their sizes. The size of a romcopy segment is always the size of the segment that it copies.

The linker does not assign a section to a segment until the linker has determined the size constraints of all segments. Then it places each section into the highest priority segment in which the section fits (see *Section 4.5.7, Segment Priority*). Therefore, before assigning any sections, the linker must reduce each ***SegExpr*** in a **size**, **end**, **minsize**, or **maxsize** specifier to a numeric value.

If a ***SegExpr*** includes a function that references a segment, such as **segsize()**, then the linker must have enough information about the function's segment so that the function can return a numeric value. Unless the function's segment has a **size** or **end** specifier, the linker may not be able to determine the value of the function. Careless use of functions that reference segments may keep the linker from completing the link. For more information, see *Section 4.10.1, Cyclic Constraints*.

4.5.5 Segment Alignment

align{ment}=*AlignNum*

The **alignment** specifier lets you specify a minimum alignment criterion for the segment start address. ***AlignNum*** is a literal number (see *Section 4.4.3, Literal Numbers*) that specifies the alignment in bytes. A value of 0 or 1 specifies no alignment (or byte alignment). The default for a segment is no alignment. Only one **alignment** specifier is allowed per segment declaration.

A segment's start address will be a multiple of ***AlignNum*** unless a section placed in the segment requires an alignment stricter (greater) than ***AlignNum***. In that case, the start address will be a multiple of the strictest alignment required by any section in the segment.

Example 4.10:

```
segment .data start=0x404 ...;
```

The **.data** segment originally has a start address of 0x404. Suppose the strictest alignment of any section placed in the segment is 8. The linker will adjust the start address to be 0x408.

Example 4.11:

```
segment .text start=0x100 align=4 ...;
```

The **.text** segment originally has a start address of 0x100. Its alignment constraint is 4. Suppose the strictest alignment of any section placed in the segment is 8. The start address 0x100 is already aligned by 8, so no adjustment is necessary.

Example 4.12:

```
segment .other align=8 ...;
```

The **.other** segment begins after the last segment before it in the LDF. Suppose the previous segment one ends at 0x42, so the **.other** segment could begin at 0x43. If no section placed in the **.other** segment has an alignment requirement greater than 8 bytes, then the linker will adjust the start address to 0x48, a multiple of the alignment constraint specified.

4.5.6 Segment Protection

pro{tect}=*ProtectSpec*,*ProtectSpec*,...

The **protect** specifier lets you specify the segment's read, write, and execute access protections when it is loaded into memory. By default, a standard segment is readable and writable, a romcopy segment is readable, and a reserved segment has no protection.

A reserved segment cannot have a **protect** specifier; otherwise, only one **protect** specifier is allowed per segment declaration.

NOTE

The Motorola Embedded Debugger does not make use of segment protection information.

The protect specifier lets you specify a comma-separated list of ***ProtectSpec*** protections. The allowed protections are:

- **r{ead}** — the segment should have read access when loaded.
- **w{rite}** — the segment should have write access when loaded.
- **ex{ecute}** or **x{ecute}** — the segment should be executable when loaded.

For the **r{ead}**, **w{rite}**, and **ex{ecute}** or **x{ecute}** keywords, the braces **{}** indicate that you must type the part of the keyword before the braces, and you may also type any or all of the part inside the braces.

If a segment has write access or execute access, the linker will automatically give it read access.

Example 4.13:

```
segment .text protect = w,r,x ...;
segment .data protect = write, read ...;
segment vector protect = exec ...;
segment table protect = wri ...;
segment .other ...;
```

The **.text** segment is writable, readable, and executable. The **.data** segment is writable and readable. The **vector** and **table** segments are declared to be executable and writable, respectively, but the linker also makes them readable. Finally, the **.other** segment gets the default protections for a standard segment: readable and writable.

4.5.7 Segment Priority

pri{ority}=*PriorityNum*

The **priority** specifier lets you specify a segment's priority. ***PriorityNum*** is a literal number (see *Section 4.4.3, Literal Numbers*) between 1 and 65535 that specifies the priority. The default priority is 1.

When the linker assigns sections from input objects to the segments, it looks for segments that have enough space for the section and a matching section specifier (see *Section 4.5.8, Segment Sections*). If the linker can place the section in more than one segment, the linker uses a priority number to decide in which segment to place it. Among several candidate segments, the segment with the highest priority is the one where the linker places the section. The highest priority has the lowest numeric value. The highest priority is 1, and the lowest is 65535.

Several segments in which the linker can assign the section may tie for highest priority. In that case, the linker picks the first segment in the order that you defined the segments in the linker definition file.

A romcopy segment cannot have a **`priority`** specifier, since sections are not placed directly in the segment; otherwise, only one **`priority`** specifier is allowed per segment declaration

4.5.8 Segment Sections

`{sec{tion}=}(`*`NameTemplate`*`,`*`NameTemplate`*`,...)`

The **`section`** specifier determines which input sections the linker can place in the segment. The linker processes each section in turn and attempts to match the section's name, and sometimes the name of the input object file or library member containing the section, with each segment's **`section`** specifiers. Only matching sections can be assigned to a segment. The linker assigns each section to the highest priority segment that has a matching **`section`** specifier and enough space for that section. See *Section 4.5.7, Segment Priority*.

A segment, other than a romcopy segment, may have more than one **`section`** specifier. A standard segment must have at least one **`section`** specifier. A reserved segment may have **`section`** specifiers, but is not required to, since sections assigned to it will not be loaded into memory. A romcopy segment may not have a **`section`** specifier.

A **`section`** specifier may start with the keyword **`section`** followed by an equals operator. A **`section`** specifier always has a comma-separated list of one or more ***`NameTemplate`*** parameters, enclosed in parentheses **`()`**.

A ***`NameTemplate`*** parameter is either a section name template or a module name template. A *section name template* is a string that can match the name of one or more input sections. A *module name template* is a string that can match the name of one or more input object files or library members. There must be at least one section name template in every **`section`** specifier; use of module name templates is optional.

You enclose a module name template in angle brackets **`<>`**, which distinguishes it from a section name template. The angle brackets are a delimiter, so you can run module name templates together without intervening commas, such as **`<file.o><member.o>`**, and you can run a module name template up against a section name template.

For each input section, the linker looks at all of the segment's **section** specifiers to match the section. The linker tries to match the section's name with a section name template. If the section's name matches, and there are no module name templates, the match succeeds. If there are module name templates, the match succeeds if the name of the file or library member containing the section matches a module name template.

Matching is case-sensitive. For example, a section named **.data** does not match a section name template of **.Data**.

Example 4.14:

```
segment .text start=0x100 section=(.text);
segment .data start=0x200 priority=2 section=(.data);
segment .data2 (.data,<vector.o>) (my_data);
```

The segment **.text** accepts any sections named **.text** from any object file or library member. The segment **.data** will accept any sections named **.data**, but if they come from a file or member named **vector.o**, they would go in segment **.data2**, since it has the higher priority. (The default segment priority is 1.) The segment **.data2** also accepts sections named **my_data** from any file or member.

Example 4.15:

```
segment .data start=0x100 (.data,<vector.o>) (.data,<coeff.o>)
              (.data);
```

The segment **.data** accepts only sections named **.data**, and it would seem only from modules named **vector.o** or **coeff.o**. However, the third **section** specifier matches any section named **.data**, without regard to the module name. This renders the first two **section** specifiers superfluous.

To assist in matching the names of sections, object files, and library members, the linker lets you use wildcard characters in a section name template or a module name template. A template containing a wildcard character can match more than one name, as opposed to matching several sections, files, or members that have the same name.

There are two wildcard characters. The asterisk (*****) character matches 0 or more characters. That is, where ***** appears in a parameter string, any number of characters in a name being matched against the template string will match the asterisk.. Therefore, this character matches any arbitrary substring in the section, object file, or library member name.

The second wildcard is the question mark (**?**). This is matches any one character in a name being matched against the template string.

Example 4.16:

```
segment special start=0x100 (vect*,tab?data);
```

The first section name template, **vect***, allows placement in the segment of any section whose name starts with the string '**vect**'. This would include **vector**, **vectOR**, **vector1**, **vect**, etc. It would not match **VECT**. The second section name template, **tab?data**, matches any section name that starts with the string '**tab**', ends with the string '**data**', and has any one character in between. This would include **tab1data**, **tabxdata**, **tab_data**, etc. It would not match **tabdata**.

If you have sections that you want assigned to a particular output segment, consider naming the sections according to some convention, such as, beginning all such section names with the string '**vect**'. You can then direct them to the intended segment by using a section name template of **vect***. Wildcard matching makes it easy to select out several different names that all follow a pattern.

You do not always have control over the names of the sections. For example, if your code is in the C language, then the section names are compiler generated. In such cases, you can isolate the code or data that you want to treat specially into a source file of its own. After compilation, this code or data will be in an object file by itself. You can then use a module name template to direct this code or data to a particular segment. In Example 4.14, the linker places a section named **.data** from a file or member named **vector.o** in a different segment from sections named **.data** from other files or members.

The linker does not consider the directory or drive containing an object file to be part of the object file's name when matching against a module name template. The linker does not consider the name of the library file to be part of the library member's name when matching against a module name template. You can use the Motorola Archiver to find out the names of a library file's members, as well as which global symbols are defined in which members. See *Appendix B, Motorola Archiver*.

4.6 SYMBOL DECLARATIONS

sym{define} ***SymbolName*** **=** ***SymExpr*** **[***Visibility***];**

symbol{define} ***SymbolName*** **=** ***SymExpr*** **[***Visibility***];**

The symbol declaration begins with either the **symdefine** or **symboldefine** keyword. The linker defines an absolute symbol named ***SymbolName*** and assigns to it the value of symbol expression ***SymExpr***. The visibility of the symbol, described below, is specified by ***Visibility***.

The linker definition file lets you define symbols that are not defined by included input object files and object members. The linker marks these symbols in the ELF output object file as *absolute* symbols. Absolute symbols associated with the code or data in any input sections; they provide symbolic constants.

There are several motivations for defining your own symbol, including:

- They let you complete a link when you have not yet defined some of the symbols. You may want to temporarily define a symbol referenced by an input section before you write the code that provides the definition.
- Hardware devices may be memory mapped at locations known at link time. A symbolic constant allows your code to treat a memory-mapped device as a global variable.
- You can record the sizes and locations of romcopy segments and uninitialized segments, so that your system boot code has a symbolic address and range to reference for boot time initialization.
- You might want to override the linker's choice of the small data area base addresses, **_SDA_BASE_** and **_SDA2_BASE_**. See *Section 4.1.2, Small Data Areas*.
- You can establish a symbolic base address of your own private small data area, and then load the base address in the register of your choice.

In general, the linker considers symbols defined in the linker definition file (LDF) to be defined at the beginning of the link process. Depending on the visibility of the symbol, another symbol with the same name as one defined in the LDF might conflict. In this case, the linker will issue an error message.

4.6.1 Symbol Name

No two symbols declared in the LDF may have the same symbol name (***SymbolName***). The rules for naming segments are in *Section 4.4.2, Names*.

4.6.2 Symbol Expression

The value assigned to the symbol is the value of the symbol expression ***SymExpr***. See *Section 4.4.4, Segment and Symbol Expressions.*

4.6.3 Symbol Visibility

The visibility of the symbol is given by the optional ***Visibility*** parameter, which can be either **global** (the symbol is a global symbol) or **weak** (the symbol is a weak symbol). If you do not specify visibility, the default is **global**. For a description of global and weak symbols, see *Section 5.3.4, Detailed Look at the Symbol Listing*.

If the symbol is a global symbol, it cannot have the same name as any global symbol defined in any input object file or library member included in the link.

Normally, when a weak symbol has the same name as a global symbol, the linker still uses the weak symbol to resolve references within the file or member that defines the symbol. For a weak symbol defined in the LDF, a global symbol by the same name in any included input object file or member redefines the symbol for all references to it.

If two weak symbols conflict, the linker converts the second one found to a local symbol, resolving references only in the file or member that defines the second symbol.. The linker considers symbols declared in the LDF to be defined before it reads any input object files or members. So a weak LDF symbol causes the linker to convert to local weak symbols with the same name defined in input object files and members.

Unless you declare symbols with the same names in the LDF, the linker creates the symbols **_SDA_BASE_** and **_SDA2_BASE_**. If you declare these symbols in the LDF, the linker uses your definitions. Relocation instructions that reference an address relative to these two symbols must still be in range (fit in the data or instruction field that has the reference). See *Section 4.1.2, Small Data Areas*.

Example 4.17:

```
# Set SDA bases to the values I have hard-coded in my boot code
symbol _SDA_BASE_ = 0x100;
symbol _SDA2_BASE_ = 0x200;

# Change a constant I use in my code
symbol WiperMotorDelay = 0x5;

# Set the address of a memory-mapped wiper motor module
symbol WiperModule = 0x400;
```

4.7 SINGLE SECTION DIRECTIVE

`single{section};`

The linker never splits input sections when it places them in output file segments. Usually, the linker the linker combines into a single section all input sections with the same name that the linker places in a single segment. However, the output object file format (ELF) requires the linker put uninitialized sections at the end of the segment containing them. So if the linker puts initialized and uninitialized input sections of the same name in a single segment, the linker creates two sections with that name. Also, you can declare segments in the LDF such that the linker places sections with the same name in more than one segment.

If it is important that all sections of any given name are together, and therefore merged into one section, then use the **singlesection** directive. Only one **singlesection** directive is necessary, and it may be anywhere in the LDF.

If there is a **singlesection** directive, the linker checks that no section name appears in more than one segment, or more than once in a single segment. In either case, the linker issues a warning message and continues with the link, unless the warning exceeds the warning limit.

The default is that the linker does not perform single section checking.

4.8 SEGMENT OVERLAP CHECKING DIRECTIVES

```
checkov{erlap};
nocheckov{erlap};
```

If you define your own segments, you are free to design them so that they overlap. You may decide to do this on purpose. Overlapping segments can help you implement a memory overlay scheme. The default is that the linker does not check whether segments overlap.

Embedded applications often have ROM areas and RAM areas that cannot overlap. It is often convenient initially to define one segment for each area. You can determine how much ROM and RAM you actually are using by leaving the segment sizes unconstrained and generating a list file with segment information in it. If you want the linker to alert you when segments overlap, use the **checkoverlap** and **nocheckoverlap** directives.

The **checkoverlap** directive turns on overlap checking among the segments declared after the **checkoverlap** directive and before the next **nocheckoverlap** directive, or the end of the LDF. The linker checks each segment between the directives against all, and only, the other segments between the directives. You can use several **checkoverlap** and **nocheckoverlap** directives in the LDF to check for overlaps among various sets of segments

The linker starts processing the LDF with overlap checking turned off (**nocheckoverlap**).

Example 4.18:

```
# group A
segment .text1 ...;
segment .text2 ...;

# group B
checkoverlap;
segment .data1 ...;
segment .data2 ...;
```

```
# group C
nocheckoverlap;
segment .bss ...;

# group D
checkoverlap;
segment .rodata1 ...;
segment .rodata2 ...;
```

The linker does not check segments in group A for overlap, because the linker turns off overlap checking by default at the start. The linker checks segments in group B for overlap. Segments **.data1** and **.data2** lie in between overlap being turned on and overlap being turned off. So the linker checks **.data1** against **.data2**.

The linker does not check the segment in group C. The segments in group D have overlap checking turned on again, so the linker checks segment **.rodata1** against segment **.rodata2**.

4.9 USE OF ROMCOPY SEGMENTS

Embedded applications often contain no non-volatile read/write storage devices, and have only volatile memory (RAM) and read-only memory (ROM) as storage. Code and read-only data can be stored ROM, and will be available at power-up (system boot). However, what about read/write data that has an initial value? Since this data is read/write, it must reside in RAM, but since it is in RAM, it will not contain a deterministic value when you boot the system.

You could write boot code that sets each data item in turn, but this is very inefficient in execution time and ROM space. A more efficient approach is to place all the initial values for initialized read/write data in a block of ROM, and, at system boot, copy this block of values into the RAM area. This approach takes only the space needed to store the initialized values, plus a few instructions of code to do the block copy.

In this approach, initialized read/write data has two distinct memory areas associated with it. The initial values are kept in an area in ROM, but your code read and writes the data in the RAM area. All references to symbols associated with the data, such as read/write variable names, must resolve to a RAM address.

The linker provides a mechanism to handle initialized read/write data. It is easiest to explain by describing the steps to be taken:

1. Place all of your read/write initialized data in a segment, or set of segments.

   Do not place any other data in this segment, unless you want to copy it as well. These segments should have the start addresses in RAM that you desire at run time; that is, after you copy initial values from ROM.

2. For each RAM segment defined in step 1, define a segment in ROM. In the segment's **`type`** specifier, make the type **`romcopy`** and reference one of the corresponding RAM segments defined in step 1.

3. Define symbols in the LDF: three for each romcopy segment.

   One symbol will contain the start address of the ROM segment, the second, the start address of the RAM segment that is copied by the ROM segment, and the third, the size of the ROM segment.

4. Write your startup code, which your application executes before your code references any of the copied data items.

   The startup code will use the values of the symbols defined in step 3 to copy the block of data in each ROM segment to its corresponding RAM segment. You can use a common subroutine to do the block copy if you like.

5. Proceed with the link. Extract segments using the Motorola S-record extraction tool, or load them using the Motorola Embedded Debugger.

   The S-record generator extracts, and the debugger loads, romcopy segments and standard segments. Reserved segments and the segments copied by romcopy segments are not extracted or loaded.

When you declare a romcopy segment, the linker reserves space in memory at the address of the romcopy segment. The amount of space reserved is the same as the segment it copies. The linker does not assign sections directly to a romcopy segment. Instead, the linker assigns sections to the RAM segment it copies.

The linker resolves any symbols defined in the initialized read/write data regions to their run-time addresses in RAM. The linker records information in the output object file about the romcopy segments, their start addresses, and the segments they copy.

In most cases, you will only have one romcopy segment and one initialized data segment that it copies. You can place initialized data that is read-only in another ROM segment and leave it there. It does not need to move into RAM, or be part of a romcopy segment.

Example 4.19:

```
# ROM area for code
segment .text start=0x100 ...;
# romcopy segment, contains image of initialized data
segment .irom type=romcopy(.idata);

# RAM area for data
segment .idata start=0x1000 ...;
# uninitialized data area
segment .bss type=reserved ...;

# symbols used for romcopy boot code
symbol StartRomcopy = segstart(.irom);
symbol StartRamcopy = segstart(.idata);
symbol SizeRomcopy = segsize(.irom);

# symbols used for initializing .bss segment to 0
symbol StartBSS = segstart(.bss);
symbol SizeBSS = segsize(.bss);
```

The **.idata** segment contains all of the read/write data sections. The **.irom** segment is a romcopy segment that copies segment **.idata**. Notice that there is also a segment named **.bss** to hold uninitialized data. It is often convenient to assume that all of the uninitialized data space is set to 0 at the start of the program (system boot-up). This does not require a copy segment, but this uninitialized data should be in a segment of its own to make it easy to initialize.

The **.bss** segment is type **reserved** to make sure that any extraction tools, such as an S-record generator or debugger, will not attempt to zero out this area. Your startup will zero it at application run time.

The symbols let your code reference the start address of the romcopy segment, start address of the segment it copies, and the uninitialized segment. Other symbols contain the size of the romcopy segment and the size of the uninitialized segment. You can use the symbols to copy data from ROM to RAM and to initialize the **.bss** segment. If your memory layout changes in the LDF, your boot code will get the new values when you re-link.

4.10 TYPICAL LINKER DEFINITION FILE PROBLEMS

The linker provides a great deal of flexibility when defining segments. It allows you to place segments one after the other, while letting the linker determine the start addresses. It also lets you create segments that mirror the size of other segments, or have size constraints that you do not completely specify. The **`start`**, **`end`** and **`size`** specifiers in segment declarations let you use expressions containing functions that return segment size and address information.

This puts a great deal of power in your hands, but can also result in some problems. This section deals with the problems you may encounter and suggests some remedies. It does this through a series of examples.

4.10.1 Cyclic Constraints

The linker places sections in segments in a two-step process:

1. The linker assigns sections to a segment; that is, the linker selects one segment to contain each section. At the end of this process, the linker knows what segment a section is in, but it does not know the section's address. Because a section's address affects how much padding the linker adds, the linker also does not know the size of any segment at the end of step 1.

   A section can sometimes match by name any one of several segments. Because the linker uses size as one of the constraints for assignment, the linker must know any upper-bound size constraints on any segment before it can assign any sections.

   So if the linker can assign a section to a segment, according to the **`section`** specifiers, then the segment's **`size`**, **`end`** or **`maxsize`** specifiers must be reducible. Reducible means that the linker can determine an integer value before step 1 is complete. If there is an **`end`** specifier for a segment, the **`start`** specifier must also be reducible in order for the linker to determine the size.

2. The linker places sections at absolute locations within each segment. The start address of at least one segment that contains sections must be reducible. The linker processes this segment. After processing the segment, the linker knows all of the attributes of that segment, including its size and start address.

   The linker continues to process in turn each segment in which the linker places sections. If any of the segments depend on each other's attributes in any cycle, direct or indirect, this step cannot complete.

Example 4.20: `size` specifier problem

```
segment .text start=0x100 size=segsize(.data) (.text);
segment .data start=0x200 (.data);
```

During step 1, if a section named **`.text`** is processed, the linker will match it to the **`.text`** segment. The size of the **`.text`** segment cannot be determined, since it depends on the size of the **`.data`** segment. The linker prints the error message:

```
unable to reduce end, max or size expression for segment .text
```

Example 4.21: Fix to Example 4.20

```
segment .text start=0x100 size=segsize(.data) (.text);
segment .data start=0x200 size=0x50 (.data);
```

When the linker processes a section named **`.text`**, the section will match the segment **`.text`**. The size of segment **`.text`** depends on the size of segment **`.data`**. Usually, the size of segment **`.data`** would not be known until step 2. However, in this case, the size of segment **`.data`** is absolutely defined, so the size of segment **`.text`** is reducible.

Example 4.22: `end` specifier problem

```
segment .text start=0x100 (.text);
segment .data end=0x500 (.data);
```

When the linker processes a section named **`.data`**, the section will match the segment **`.data`**. Now the linker must determine if there is enough room. The segment has an absolute end given, so its size depends on its start address. Since no explicit start address is given, the start address is the first byte after the end of segment **`.text`**. So the linker must know the size of segment **`.text`** to determine the size of segment **`.data`**. The start address of the segment **`.text`** is declared, but its size is unconstrained and depends on the placement of sections. So the size is not known until the end of step 2 processing. The linker issues the error message:

```
unable to reduce start expression to determine size of segment
      .data
```

Example 4.23: Fix to Example 4.22

```
segment .text start=0x100 size=0x100 (.text);
segment .data end=0x500 (.data);
```

The size of segment **`.data`** depends on the start address of the segment. This depends on the start address and size of segment **`.text`**, since segment **`.data`** follows segment **`.text`**. The start and size of segment **`.text`** are absolutely specified, so the linker can reduce the size constraint for segment **`.data`**.

Example 4.24: Start address-size dependency cycle problem

```
segment .text start=0x0 (.text);
segment .romcopy type=romcopy(.idata);
segment .idata (.data);
```

This is a tricky example. In step 2, the linker assigns addresses to sections within the segment. In order to process segment **.idata**, the linker must know the start address of the segment. Segment **.idata** follows segment **.romcopy**, so the start address of the segment depends on the size and start address of segment **.romcopy**.

The size of segment **.romcopy** depends on the size of segment **.idata**, but the linker cannot know that size until segment **.idata** is processed in step 2. Thus, there is a cycle. The linker issues the error message:

```
start address of segment .idata cannot be reduced
```

Fortunately, it is unlikely that you will declare a segment that holds initialized data (in RAM) with a start address to follow the segment that copies it (the romcopy segment that is in ROM). Since these are two separate devices, they will likely start on fixed device addresses.

Example 4.25: Fix to Example 4.24

```
segment .text start=0x0 (.text);
segment .romcopy type=romcopy(.idata);
segment .idata start=0x1000 (.data);
```

The linker can process the **.data** segment in step 2, since its start address is reduced. The same is true for segment **.text**. After the linker processes segment **.text**, it knows the size of the segment, so the start address for segment **.romcopy** is also be reducible.

4.10.2 No Matching Segment

The linker needs to assign each of the sections in the input object files and library to one of the output segments. The linker can assign a section to a given segment only if the section matches a **section** specifier for a segment.

If the section does not match any of the **section** specifiers for the defined segments, then the linker will issue an error message like this one:

```
unable to match section .text, file vector.o to any segment,
check segment defs
```

If you did not define any segments and are using the default segment map, you will not get this message. The linker defines the default segment map such that any conceivable section has a segment.

If a section does not match any segment, look at your segment declarations to see to which segment you expected the linker to assign the section. Then look at the name templates to see why that section did not match any of the **`section`** specifiers.

If a section matches the **`section`** specifiers of a segment, but the linker cannot place the section there because of the lack of space in the segment, you will get a different error message. See *Section 4.10.3, No Room in Segment.*

Example 4.26: no matching segment problem

```
segment .text start=0x100 (.text,<file1.o>);
segment .data (.data);
```

Assume the linker attempts to assign section **`.text`** from **`vector.o`**.

This section does not match any of the segments. It cannot go into the **`.text`** segment, because that segment accepts only sections named **`.text`** from object files or members named **`file1.o`**. The linker issues the error message:

```
unable to match section .text, file vector.o to any segment,
check segment defs
```

Example 4.27: Fix to Example 4.26

```
segment .text start=0x100 (.text<file1.o><vector.o>);
segment .data (.data);
```

Assume the linker attempts to assign section **`.text`** from **`vector.o`**.

The linker will match the **`.text`** section to the **`.text`** segment.

4.10.3 No Room in Segment

As the linker assigns sections to segments, a section may match the **`section`** specifier of a size-constrained segment. If there is not enough space left in the segment, the linker will not assign the section, even if the segment is the highest priority matching segment or the only matching segment.

In order to distinguish this case from not being able to find a matching segment, the linker issues a separate error message when there is not enough room. This message is similar to the following:

```
matching segments have no room for section .text, file factor.o
```

If you ask for a segment listing, the linker will list the segments and the sections assigned to them. The linker will also list the sections that it did not assign a segment because there was no match, or because of a lack of room. For each section that the linker did not assign due to a lack of space, look in the list file information for the each segment that it matches. Using the start address and size of the last segment assigned to each matching segment, you can calculate the space left in the segment, which should be smaller than the size of the section you are attempting to assign.

In some cases, it may appear that there is enough room, yet the linker says there is not. Remember that the linker must add padding space to meet alignment constraints. This padding is the reason that the section will not fit. In order to meet alignment criteria, the linker must consider both the size of the section and the padding that must precede its assignment.

MELD assigns input sections to output segments in a two-phase process. In the first phase, the linker assigns each section to some segment. The linker ignores the segment's start address, even if it is know, and it computes the worst-case padding it needs to add to the segment in order to align the sections assign to it. This phase is called the assignment phase.

In the second phase, the linker determines each segment's start address from the segment's **`start`** specifier or computed from another segment's location and size. The linker places each section assigned to a segment in the segment, in the order that it assigned sections to the segment. The linker adds padding in front of each section to meet its alignment requirement and computes the address of each section.

Before assigning a section to a segment in the assignment phase, the linker makes sure that there is room for both the section and the worst-case number of bytes of padding needed to align the start of the section.

The linker makes sure that the segment's start address is a multiple of the strictest (highest) alignment of any section in the segment. So the linker needs no padding bytes before the first assigned section. The linker determines the worst-case padding needed for sections assigned after the first section by using the segment's running worst-case alignment.

After the linker assigns each section, including the first, the *running worst-case alignment* is the lesser of the alignment requirement for the section just assigned and the size alignment of that section. The *size alignment* is the maximum power of two that will evenly divide into the size of the section.

Example 4.28: Computation of running worst-case alignment

The linker has just assigned section **`.data`** from file **`factor.o`** to segment **`.data`**. The section's alignment requirement is 8 bytes, and its size is 6 bytes.

When linker assigns the section, the linker assumes padding was necessary to meet the alignment requirement of the section. This requirement is 8 bytes. The size alignment of the section is 2 bytes. (This is the largest power of 2 that will evenly divide the section's size of 6 bytes.) The minimum of these two values is 2 bytes, so the new running worst-case alignment is 2 bytes.

The worst-case number of bytes of padding needed is:

- the section's alignment requirement minus the segment's running worst-case alignment, or
- zero, if the running worst-case alignment is greater than the alignment requirement.

The actual number of bytes of padding needed may be fewer than the worst-case assumed by the linker during the assignment phase.

Example 4.29:

The previous section assignment left the segment with a running worst-case alignment of 2 bytes. The linker now needs to assign a section that has an alignment requirement of 4 bytes and a size of 16 bytes. The linker assumes that it needs 2 bytes of padding for this section (4 bytes - 2 bytes), so the segment must have at least 18 bytes left. The segment's new running worst-case alignment is 4 bytes (the lesser of 4 bytes and 16 bytes).

Example 4.30:

The segment **`.data`** has 10 bytes left. The last section assigned to it has a size of 4 and an alignment requirement of 4 bytes. The section **`consts`** matches this segment and does not match any other. It has an alignment requirement of 8 bytes and a size of 8 bytes. An error message from the linker tells us that the linker cannot assign this section to this segment because there is not enough room.

When the linker generates the list file, the last section in segment **`.data`** begins at address 0x64. Since its size is 4 bytes, the next available address is 0x68. This address is a multiple of 8, so the linker should not require padding to assign section **`consts`**. So why won't the section fit, since it is only 8 bytes, and there are 10 left?

The linker does not know the actual start address of segment **.data** during the assignment phase. The next available address in segment **.data** could have been an address other 0x68 that is not a multiple of 8. The last section assigned has an alignment requirement of 4 bytes. The alignment of the segment after padding and before the linker places this last section must be at least 4 bytes. The size of the last section is also 4 bytes, so the running worst-case alignment after the linker assigned this last section is 4 bytes.

When the linker attempts to assign section **consts**, the running worst-case alignment is 4 bytes. Up to 4 bytes of padding might be required in front of this section. Therefore, there must be 8 bytes for the section plus 4 bytes padding, or 12 bytes left in the segment, before the linker can assign this section.

The linker orders all sections with the same name together in a single list before assignment. After placement, it combines all of the like-named sections that are contiguous in the same segment into one larger section. It is likely that all of the sections that have a given name will have the same alignment requirements and a size that is modulo the alignment requirements.

If the above is true, as the linker assigns the like-named sections in turn to the same segment, it does not need to add padding after it assigns the first one. The running worst-case alignment will be the same as the required alignment for each of the like-named sections assigned after the first.

CHAPTER 5

LIST FILE

5.1 INTRODUCTION

The list file is a record of what goes into the output file generated by the linker. Specifically, the list file shows information regarding segments generated, sections, and symbols. The list file does not show the actual data in a given section or segment. To view that, you must use an object/executable file reader of some sort, e.g., a dumper or a debugger.

The list file is divided into three types of listings: segment, section, and symbol. You can direct MELD to generate a list file and determine which listings it contains by using the listing switches: **`-LIST`**, **`-seg`**, **`-sym`**, **`-symn`**, **`-syma`**, and **`-xref`**.

The *segment listing* lists the segments defined in the output file. If the listing is requested, it appears first in the list file. It shows various segment attributes such as where each segment starts in memory and how large it is. The linker will suppress 0 size segments. For more information on the segment listing, see *Section 5.3.2, Detailed Look at the Segment Listing*.

The section listing, if requested, is next in the list file. It lists the sections defined in the output file, as well as which segment each section is within. For each output section, it shows information such as what input sections from what source files were combined to create it, what symbols are defined in it, and what symbols are referenced in it. Output sections of size 0 are suppressed. For more information on the section listing, see *Section 5.3.3, Detailed Look at the Section Listing*.

The symbol listing, if requested, is last. It lists all symbols in the output file, with their values, sizes, and sections. For more information on the symbol listing, see *Section 5.3.4, Detailed Look at the Symbol Listing*.

5.2 LIST FILE COMMAND LINE SWITCHES

The command line listing switches, which cause a list file to be generated and determine its contents are: **`-LIST`**, **`-seg`**, **`-sym`**, **`-symn`**, **`-syma`**, and **`-xref.`**

5.2.1 The -LIST Switch

The **-LIST** switch specifies the name of the list file and sets default list switches. The **-LIST** switch takes a single argument, the name of the list file to be generated.

If no other listing switch appears on the command line or in command files, then the **-LIST** switch will set default listing contents. The default list file contents for the **-LIST** switch are:

- segment listing and section listing, as if **-seg** is specified
- symbol listing, as if **-symn** is specified
- Symbol cross-references in the section listing, as if **-xref** is specified

If the **-LIST** switch appears on the command line or in command files with one or more of the other listing switches, then **-LIST** only names the list file; the other switches determine what goes into the file (**-LIST** will not set the default listing contents).

If **-LIST** does not appear on the command line or in command files, but one or more of the other list file switches does, then the list file will be named **linker.lst**, and will have the contents specified by the listing switches provided.

The **-LIST** switch can appear multiple times on the command line or in command files. However, if it appears more than once, the linker still only generates one list file. This file is named according to the last **-LIST** switch.

If the file specified by the **-LIST** switch exists, the linker will overwrite it. Care must be taken not to follow the **-LIST** switch with an object or library file name. The linker could overwrite this file with a list file.

Example 5.1:

```
meld -o main -def linker.ldf startup.o main.o -LIST main.lst
```

This generates a list file named **main.lst.** File **main.lst** will contain a segment listing, a section listing, a symbol listing, and symbol cross-references in the section listing.

Example 5.2:

```
meld -o main -def linker.ldf startup.o main.o -seg -xref
     -LIST main1.lst -LIST main2.lst
```

This produces a list file named **main2.lst** that contains a segment listing, a section listing, and symbol cross-references within the section listing, but no symbol listing. Since **-seg** and **-xref** were specified, the defaults for the **-LIST** switch were not turned on. The **-LIST** switch is used only to name the list file.

Example 5.3:

```
meld -o main -def linker.ldf startup.o main.o -symn -xref
```

This produces a list file named **linker.lst** that contains a section listing with symbol cross-references, and a symbol listing that is sorted by name.

5.2.2 The -seg Switch

The **-seg** switch directs the linker to include in the list file the segment and section listings. Multiple **-seg** switches can appear on the command line or in command files; they have no additional effect beyond the first **-seg**.

Example 5.4:

```
meld -o main -def linker.ldf startup.o main.o -seg
```

This produces a list file named **linker.lst** with a segment listing and a section listing, without symbol cross-references. The list file for this example is in Figure 5-1.

```
*** Segment Listing ***
Segment Name    Start   Init_Size    Total_Size   Prot Type
.text           0x20000 0x90         0x90         RW   STD
stack           0x40000 0x0          0x40000      RW   RESERVED
        Total Init Size: 0x90
        Total Size: 0x90

*** Section Listing ***

Section: .text        Segment: .text
[start: 0x20000       size: 0x8C    type: init index: 1]

composite section        Location   Source File
.text                    0x20000    main.o
.text                    0x20008    startup.o

Section: .sdata2      Segment: .text
[start: 0x20090       size: 0x0     type: init index: 4]

composite section        Location   Source File
.sdata2                  0x20090    main.o
```

Figure 5-1. List File with -seg Switch

For more information on the listing generated by the **-seg** switch, see *Section 5.3.2, Detailed Look at the Segment Listing.*

5.2.3 The -xref Switch

The **-xref** switch directs MELD to list the symbol cross-references. This is a section by section listing of the symbols defined in each section and the symbols referenced in each section. Since the listings are by section, they appear with the section listings. If you do not request a section listing but ask for a cross-reference, the linker gives you a section listing anyway.

Multiple **-xref** switches can appear on the command line; they have no additional effect beyond the first **-xref**.

Example 5.5:

```
meld -o main -def linker.ldf startup.o main.o -xref
```

This produces a list file named **linker.lst** with section listings that have symbol cross-references. The list file for this example is in Figure 5-2.

```
*** Section Listing ***
Section: .text        Segment: .text
[start: 0x20000       size: 0x8C    type: init index: 1]

composite section        Location   Source File
.text                    0x20000    main.o
.text                    0x20008    startup.o

symbols defined:
.text                 0x20000 main       0x20000
__start               0x20008

symbols referenced:
_SDA_BASE_            _SDA2_BASE_
main

Section: .sdata2      Segment: .text
[start: 0x20090       size: 0x0     type: init index: 4]

composite section        Location   Source File
.sdata2                  0x20090    main.o

symbols defined:
.sdata2                  0x20090
```

Figure 5-2. List File with -xref Switch

For more information on the listing generated by the **-xref** switch, see *Section 5.3.3, Detailed Look at the Section Listing.*

5.2.4 The -sym, -symn, -syma Switches

The **-sym**, **-symn**, and **-syma** switches all direct the linker to generate a symbol listing in the list file. Note that **-sym** and **-symn** are aliases for each other; they have identical functionality.

The **-sym** and **-symn** switches direct the linker to generate a symbol listing that is alphabetically sorted by name, in ascending ASCII character order. More than one **-sym** or **-symn** can appear on the command line; however, any additional -**sym** or **-symn** switches will have no additional effect beyond the first one.

The**-syma** switch directs the linker to generate a symbol listing that is sorted by ascending symbol value. More than one**-syma** can appear on the command line; however, any additional **-syma** switches will have no additional effect beyond the first one.

If **-syma** appears on the command line along with **-sym** or **-symn**, then the linker generates two symbol listings in the list file; the first is sorted by name, and the second by symbol value.

Example 5.6:

```
meld -o main -def linker.ldf startup.o main.o -symn
```

This produces a list file named **linker.lst** with a symbol listing that is sorted by name. The list file for this example is in Figure 5-3.

```
*** Symbol Listing (sorted by name) ***
Symbol Name           Value   Size  Type  Vis   Section
.data                 0x40000 0x0   SECT  LOCL  5
.sdata                0x40000 0x0   SECT  LOCL  6
.sdata2               0x20090 0x0   SECT  LOCL  4
.text                 0x20000 0x90  SECT  LOCL  1
_SDA2_BASE_           0x0     0x0   NONE  GLOB  ABSOL
_SDA_BASE_            0x0     0x0   NONE  GLOB  ABSOL
__start               0x20008 0x0   DATA  GLOB  1
main                  0x20000 0x0   DATA  GLOB  1
main.c                0x0     0x0   NONE  LOCL  ABSOL
```

Figure 5.3. List File with -symn Switch

For more information on the listing generated by the **-sym** or **-symn** switch, see *Section 5.3.4, Detailed Look at the Symbol Listing*.

Example 5.7:

```
meld -o main -def linker.ldf startup.o main.o -syma
```

This produces a list file named **linker.lst** with a symbol listing that is sorted by value. The list file for this example is in Figure 5-4.

```
*** Symbol Listing (sorted by value) ***
Value    Symbol Name           Size    Type   Vis    Section
0x0      main.c                0x0     NONE   LOCL   ABSOL
0x0      _SDA_BASE_            0x0     NONE   GLOB   ABSOL
0x0      _SDA2_BASE_           0x0     NONE   GLOB   ABSOL
0x20000  .text                 0x90    SECT   LOCL   1
0x20000  main                  0x0     DATA   GLOB   1
0x20008  __start               0x0     DATA   GLOB   1
0x20090  .sdata2               0x0     SECT   LOCL   4
0x3FFFC  _startup.udata        0x0     NONE   GLOB   ABSOL
0x40000  .data                 0x0     SECT   LOCL   5
0x40000  .sdata                0x0     SECT   LOCL   6
```

Figure 5-4. List File with -syma Switch

For more information on the listing generated by the **-syma** switch, see *Section 5.3.4, Detailed Look at the Symbol Listing.*

Example 5.8

```
meld -o main -def linker.ldf startup.o main.o -syma -sym
```

This produces a list file named **linker.lst** with two symbol listings: one sorted by name and one sorted by value. The example listing is not shown, but is essentially the same as the listings in Figure 5.3 and Figure 5.4 concatenated.

5.3 DETAILED LIST FILE CONTENTS

5.3.1 List File Structure and Pagination

The list file has three types of listings: a segment listing, a section listing, and a symbol listing. A cross-reference listing is really a section listing with additional information. A section listing is always output with a segment listing.

The list file is paginated. At the top of each page is a header naming the version of MELD that generated it, the name of the target file, and the page number. Each different type of listing begins on a new page.

5.3.2 Detailed Look at the Segment Listing

The segment listing lists the segments contained in the output file and information about those segments. The segment listing, if one is present, is at the beginning of the list file. At most one segment listing will appear in a list file.

The main body of the segment listing is a list of the segments the linker has generated, with one line of information per segment. At the start of the list is a header line describing the type of information listed.

As an example:

```
Segment Name  Start    Init_Size  Total_Size  Prot   Type
.text         0x20000  0x90       0x90        RW     STD
```

There are six fields:

- **`Segment Name`** — This is the name of the segment.
- **`Start`** — This is the start address of the segment, in hexadecimal.
- **`Init_Size`** — This is the number of bytes in the segment's initialized data space, in hexadecimal. This initialized data exists in the output object file and is extracted by the loader or debugger. Initialized data is always at the start of the segment.
- **`Total_Size`** — This is the number of bytes in the segment, including both initialized and uninitialized data, in hexadecimal. The difference between **`Total_Size`** and **`Init_Size`** is the number of bytes of uninitialized data. Uninitialized data is always at the end of the segment, after any initialized data.
- **`Prot`**—This is the access protection flags set for the segment. Three flags are possible: **`R`** means the segment is readable (has read access), **`W`** means the segment as writable (has write access), **`X`** means the segment as executable (has execute access).

  The default for a standard segment or a reserved segment is **`RW`**, meaning the segment is readable and writable. A romcopy segment never has access protection flags set.

  This field is typically used by loaders. An embedded application will likely ignore these flags; that is, there will be no memory-access protection provided. If memory-access protection is used, it will likely be setup by the startup code and not affected by any tools which extract the segments from the executable.

- **`Type`** — This is the type of segment. A standard segment is shown as **`STD`**, a reserved segment as **`RESERVED`**, and a romcopy segment as **`ROMCOPY`**. For information on segment types, see *Section 4.5.2, Segment Type.*

  In Figure 5-1, the **`stack`** segment is a reserved segment with no sections assigned to it. It could have been declared a standard segment, but, if it had, a loader or debugger might attempt to initialize it to zero. By declaring the space reserved, this will not occur.

  The main body of the segment listing contains two lines for romcopy segments. The second tells what segment is copied by the romcopy segment. For example, if **`.idata`** is a romcopy of **`.data`**, its segment listing lines might look like this:

```
Segment Name   Start    Init_Size   Total_Size  Prot Type
.idata         0x20130  0x30        0x30        R    ROMCOPY
        (segment .data is rom copy of segment .data)
```

After the main segment by segment information in the segment listing, there are two lines that look like this:

```
Total Init Size: 0x160
Total Size: 0x160
```

The first line shows the sum of the **`Init_Size`** values for all of the segments. The second line shows the sum of the **`Total_Size`** values for all of the segments. When summing the sizes, the linker does not include reserved segments, since they are not output. The linker does include romcopy segments, since they occupy space in ROM, just like the segments in RAM that they copy.

5.3.3 Detailed Look at the Section Listing

The section listing provides information about each of the sections that is output. The sections are listed in the order that they are defined in the segments. If a section list is present, it follows the segment listing. If there is no segment list, then the section listing comes first.

The cross-reference listing contains information about symbols and is listed section by section. A cross-reference will be present in the section listing only if a cross-reference is requested. Asking for a cross-reference will automatically cause a section listing to occur.

Figure 5-6 is an excerpt from a section listing, with cross-reference information shown.

```
Section: .text    Segment: .text
[start: 0x20000   size: 0x130    type: init index: 1]

composite section     Location   Source File
.text                 0x20000    startup.o
.text                 0x20088    factorial.o
.text                 0x20110    fact.o

symbols defined:
.text             0x20000        fact_loop          0x20124
fact_end          0x20134        __start            0x20000
main              0x20088        fact               0x20110

symbols referenced:
_SDA_BASE_            _SDA2_BASE_
main                  factorials
fact
```

Figure 5-5. Section Listing Excerpt

NOTE

Section listings do not contain information about special sections that only a loader or debugger would care about. For instance, symbol table sections, string table sections, debug information and relocation sections are not shown.

For example, these sections, if declared, would be in shown in a section listing: **`.text`**, **`.data`**, **`my__data`**, and **`.bss`**. Even if they are declared, these sections would not be: **`.symtab`**, **`.strtab`**, and **`.debug`**.

Each section listed contains three kinds of information: section information, composite section information, and a symbol cross-reference.

5.3.3.1 Section Information

The section listing starts with information that describes various attributes of the section, such as its name, start address, size and type. Here is an example of section information:

```
Section: .text      Segment: .text
[start: 0x20000  size: 0x130    type: init    index: 1]
```

The fields of the section information are:

- **`Section`** — This is the section's name.
- **`Segment`** — This gives the name of the segment that contains the section. Segment and section names may use the same names without conflict.
- **`start`** — This is the start address of the section, in hexadecimal.
- **`size`** — This is number of bytes contained in the section, in hexadecimal.
- **`type`** — This is the section's type. Sections that contain code or data to initialize the sections are initialized sections. These sections have type **`init`**. Other sections contain no data in the executable and are used to reserve space for variables that need no initial values. These sections have type **`uninit`**.
- **`index`** — This is the section's index in the output file's section header table. The symbol listing tells you which section the symbol is defined in by specifying the section index.

5.3.3.2 Composite Section Information

The linker combines input object sections with the same name into a larger section, where possible. The composite section information tells which sections, from which input object files or libraries members, were combined to form the output section. This is the composite section information from Figure 5-5:

```
composite section     Location      Source File
.text                 0x20000       startup.o
.text                 0x20088       factorial.o
.text                 0x20110       fact.o
```

There are three fields per line in the composite section information:

- **`composite section`** — This is the name of the input sections which were combined into the output section.
- **`Location`** — This is the memory address, in hexadecimal, where the input object section is assigned in the output file.
- **`Source File`** — This is name of the input object file or library member that the input section came from. For a library member, this field will be the name of the library, followed by the name of the object member in the library.

5.3.3.3 Symbol Cross-Reference

The symbol cross-reference information lists the symbols that are defined or referenced in a given section. The cross-reference information appears in the section listing only if the **`-xref`** flag is present, or by default if **`-LIST`** and no other listing flags are present. (See *Section 5.2.1, The -LIST Switch*.) Figure 5-6 shows the cross-reference information from Figure 5-5 for the **`.text`** section.

```
symbols defined:
.text           0x20000          fact_loop       0x20124
fact_end        0x20134          start           0x20000
main            0x20088          fact            0x20110

symbols referenced:
_SDA_BASE_          _SDA2_BASE_
main                factorials
fact
```

Figure 5-6. Symbol Cross Reference

There are two parts to the symbol cross-reference information: **`symbols defined`** and **`symbols referenced`**. If there are no symbols defined in the section, the **`symbols defined`** header is suppressed; if there are no symbols referenced, the **`symbols referenced`** header is suppressed. The contents of the two parts are:

- **`symbols defined`** — This lists the symbols defined in the section, along with their address values, in hexadecimal.
- **`symbols referenced`** — This lists the symbols that are referenced (used) in the section. Note that a symbol defined in a section may also be referenced in the section. When the value (address) of a referenced symbol is determined, code or data that references the symbol must be adjusted to reflect the new value.

5.3.4 Detailed Look at the Symbol Listing

The symbol listing shows the names of the symbols in the executable file, along with other relevant data. The symbol listing, if present, is at the end of the list file. Symbol listings may be listed by symbol name in ascending alphabetical order or by symbol value listed in ascending numerical order. There may be two symbol listings, sorted each way.

The symbol listing will show section names, since these are symbols in the executable. It will show the debugging sections. The listing will filter out the symbols in debugging sections, since these are often extremely numerous.

Figure 5-7 is an example symbol listing, showing both of the sort orderings.

```
*** Symbol Listing (sorted by name) ***
Symbol Name           Value   Size    Type   Vis    Section
.data                 0x40000 0x2C    SECT   LOCL   5
.debug                0x0     0x128   SECT   LOCL   7
.sdata                0x40030 0x0     SECT   LOCL   6
.sdata2               0x20130 0x0     SECT   LOCL   4
.text                 0x20000 0x130   SECT   LOCL   1
_SDA2_BASE_           0x0     0x0     NONE   GLOB   ABSOL
_SDA_BASE_            0x0     0x0     NONE   GLOB   ABSOL
__start               0x20000 0x0     DATA   GLOB   1
_startup.udata        0x4002C 0x0     NONE   GLOB   ABSOL
fact                  0x20108 0x0     DATA   GLOB   1
fact_loop             0x2011C 0x0     DATA   LOCL   1
factorials            0x40000 0x0     DATA   GLOB   5
main                  0x20088 0x0     DATA   GLOB   1

*** Symbol Listing (sorted by value) ***
Value   Symbol Name   Size    Type    Vis    Section
0x0     .debug        0x128   SECT    LOCL   7
0x0     .line         0x4E    SECT    LOCL   8
0x0     _SDA_BASE_    0x0     NONE    GLOB   ABSOL
0x0     _SDA2_BASE_   0x0     NONE    GLOB   ABSOL
0x20000 .text         0x130   SECT    LOCL   1
0x20000 __start       0x0     DATA    GLOB   1
0x20088 main          0x0     DATA    GLOB   1
0x20108 fact          0x0     DATA    GLOB   1
0x2011C fact_loop     0x0     DATA    LOCL   1
0x20130 .sdata2       0x0     SECT    LOCL   4
0x40000 .data         0x2C    SECT    LOCL   5
0x40000 factorials    0x0     DATA    GLOB   5
0x4002C _startup.udata        0x0     NONE   GLOB   ABSOL
0x40030 .sdata        0x0     SECT    LOCL   6
```

Figure 5-7. Symbol Listing

In either sort ordering, for each symbol the listing shows these fields: **Symbol Name**, **Value**, **Size**, **Type**, **Vis** and **Section**. The order of the fields changes slightly depending on the sort order of the listing. The fields are:

Symbol Name This is the symbol's name.

Value This is the symbol's address value or absolute value, in hexadecimal.

In most cases, the value is the address where the symbol is defined. For instance, in Figure 5-7 the value of **fact** is x20108 — this is its address.

In some cases, the symbol is not defined inside a section. It has a value which may or may not represent an address. This is called an absolute value. For instance, the value of **_startup.udata** is 0x0. This symbol was defined in the linker definition file.

In other cases, the value is not used, and is generally 0. For instance, the value of section symbol **.debug** (which is never mapped into memory) is 0x0.

Size This is the size of the thing the symbol describes. The size is 0 if the symbol object does not have a size, or the language tools either do not know or do not state the size of the object. For instance, code labels do not have sizes. Some symbols, such as section symbols, always have sizes.

Type This is the type of the symbol. Most symbols, for example labels such as **fact**, are symbols of type **DATA**. This is true whether the label is a code or data label. Section symbols are given a type of **SECT**. Absolute symbols are type **NONE**. Debug symbols are also type **NONE**, but these are filtered out of the listing.

Vis This is the visibility of a symbol. There are three possible values ELF provides: local (shown as **LOCL**), weak (shown as **WEAK**), and global (shown as **GLOBL**).

A symbol's visibility determines whether the symbol can be accessed by its name from outside the object module (object file or library member) that contains its definition. In order for an object module to reference a symbol by name that is defined in another object module, that symbol must be global or weak.

For executable files, visibility is usually irrelevant, since all of the objects are already combined into one, and no one should be referencing symbols outside. (Dynamically loaded executable files are an exception.)

For input object modules, and partial re-link output object files (using the **-r** command-line switch), this value is relevant.

Local symbols, such as **fact_loop** in Figure 5-7, are only accessible within the object module where in they are defined. There can be multiple local symbols of the same name in different object modules. Their names will not conflict.

Global symbols, such as **fact**, can be referenced by object modules other than the ones in which they are defined. Only one global symbol of any particular name can be defined in all of the object modules you link together.

Like global symbols, weak symbols can be referenced from object modules other than the ones in which they are defined. Unlike global symbols, there can be multiple weak symbols with same name, or weak symbols with the same names as global symbols. The linker encounters another weak or global symbol by the same name as one that has already been defined, the linker will convert that weak symbol to a local symbol.

If a global symbol is defined after a weak one with the same name has been defined, then the linker will use the global symbol as the symbol with global visibility and convert the previously-defined weak symbol to local.

When a weak symbol is converted to local, all references to that symbol name inside the object module where the weak symbol is defined will refer to the weak symbol's definition. References outside the object module will continue to use the pre-existing weak or global definition.

A listing of a weak symbol would look like this:

```
0x20000 main 0x0   DATA WEAK    1
```

Section This is the index of the section where the symbol is defined. Most symbols of type **DATA** or **SECT** are defined and contained in some section defined in the output file. For instance, from Figures 5-5 and 5-7, **fact** is defined in a section whose index is 1, which is section **.text**. Some symbols have a section number of **ABSOL**, which really is not a section index. This means that the symbol as an absolute value. For example, the symbol **_startup.udata** is defined in the linker definition file.

Other symbols may have section index of **COMM**. This means the symbol is a common symbol, defining an uninitialized common area to be created by the linker in the **.bss** section. When a symbol is common, the value shown is the alignment requirement for the common area to be created.

The linker converts a common symbol to a global symbol at the start of an area reserved in a **.bss** section that is big enough for the common symbol's size. Therefore, you will not see common symbols in the symbol table unless you have done a partial link (using the **-r** command-line switch). A common symbol listing looks like this:

```
0x8 comm_area 0x40 NONE GLOB COMM
```

APPENDIX A

LINKER ERROR MESSAGES

This appendix lists linker error messages. If a message has an associated file, that file's name precedes the error message. If an error occurs in the linker definition file (LDF), the appropriate line number precedes the error message. For library (archive) file members, the linker shows the library member name and the library name.

Note the three types of error messages:

- *Warnings:* The linker issues a warning message if it finds a condition that might indicate something wrong. The linker continues the link process, creating an output file.
- *Errors:* The linker issues an error message if it finds a problem severe enough to prevent creation of an output file. Most such problems are errors in command-line specifications or linker definition file (LDF) segment/symbol definitions. After issuing such an error message, the linker tries to continue processing input files, checking for any additional problems.
- ***Fatal Errors:*** The linker issues a fatal error message if it finds a problem severe enough to stop the link process immediately. An example is an incorrect or unrecognizable object file.

NOTE:

If a fatal error message indicates a corrupted or unusable library file or object file, check whether the file was created with a tool that produces System V file formats. For example, the archiver tools that come packaged with many operating systems do not produce archive files that MELD accepts. Use the MAR archiver instead.

`attempt to divide by zero in definition file symbol` *`symbol`*

Error: A divisor in an expression given as the value for symbol *symbol* resolved to zero. Edit the LDF to fix the expression.

`attempt to divide by zero in segment` *`segment`*

Error: A divisor in an expression given as a specifier value for segment *segment* resolved to zero. Edit the LDF to fix the expression.

attempt to extract string from non-string section

Fatal Error: Internal file information is incorrect: a reference to an area that should have contained strings (such as section or symbol names) failed. Probable cause: a corrupted object file or a non-ELF file.

bad string table offset address : *member-header-name-field* in archive

Fatal Error: The value in a library member header's name field is incorrect. The value is neither the member name nor the offset to the member name in the library's archive string table. Probable cause: a corrupted archive file or an invalid archive file type.

cannot use -T or -D switch when defining segments via def file

Error: You specified a **-T** or **-D** switch to set the start address of the default **.text** or **.data** segment. You also supplied an LDF that contains segment definitions, so the linker did not create the default segments. To specify the start addresses of LDF segments, use the **start** specifier, not the **-T** or **-D** switch.

command file *filename* is not a valid file name

Error: The command-line **-f** switch specified an invalid file name. Provide a valid file name.

corrupt or illegal format for object file

Fatal Error: Unsuccessful attempt to read an ELF section header, symbol, relocation entry, or other such item from an object file. Probable cause: a corrupted object file or a non-ELF file.

corrupt or illegal format for object file header

Fatal Error: Unsuccessful attempt to read the ELF header of an object file. Probable cause: a corrupted object file or a non-ELF file.

could not find library *filename*

Error: The command-line **-l** switch specified an inappropriate string for a library file name. (To construct the library file name, the linker bracketed the string between the prefix **lib** and the extension **.a**. But the linker could not find a file of this name in the current directory or in any directory in the library path list.) Specify a different string for the **-l** switch or use the **-L** switch to provide the name of the directory (path) where the linker can find the file.

deferred relocation symbol *symbol* not output

Error: An error occurred inside the linker while it was writing the executable output file. Refer this problem to Motorola software support.

duplicate global definition of symbol *symbol*

Error: Two or more of the object files or library members linked together, or the LDF and one or more files or members, define the symbol as a globally visible symbol. The second definition generates this error message. The linker does not tell you the first place that defines the symbol. Remove all but one global definition of the symbol.

duplicate symbol definition for *symbol*

Error: The LDF contains two symbol declarations for the name *symbol*. Edit the LDF: eliminate one of the declarations or rename the symbol in one of the declarations.

duplicate symbol *symbol*

Warning: Code defines the specified symbol multiple times, as permitted by the **-dup** switch. (Without the **-dup** switch, the linker would consider multiple symbol definitions an error.)

end address is less than start address for segment *segment*

Error: Segment *segment* has an end address that is less than its start address. Fix this in the LDF.

end specifier not allowed with size, maxsize or minsize

Error: You cannot use the **end** specifier with the **size**, **maxsize** or **minsize** specifier. As the **end** specifier limits the segment size, using another size-limiting specifier is redundant or contradictory. Edit the LDF to remove one of the specifiers.

entry symbol *symbol* not defined

Error: The linker cannot find *symbol*, the symbol specified as the entry point for the output file. Such a symbol must be defined and globally visible. If the symbol is defined in a library member, the linker will only extract that member if some object file or member in the link refers to a symbol defined in that library member. The default entry symbol is **__start**.

If you do not want an entry symbol in your output object, use the **-noent** switch. Otherwise, make sure that the entry label is defined, has global visibility, and is included in the linked object files or library members.

expected a positive or 0 number following the -error switch

Error: The **-error** switch lacks its 0 or positive-number parameter value. Provide this value, in hexadecimal, decimal, or octal format.

expected a positive or 0 number following the -warn switch

Error: The **-warn** switch lacks its 0 or positive-number parameter value. Provide this value, in hexadecimal, decimal, or octal format.

expected at least one section name template between '()'

Error: No section name template is in the **section** specifier. Edit the LDF to add at least one section name template.

expected a ';' to terminate symbol declaration

Error: A symbol declaration lacks its semicolon (**;**) terminator. Edit the LDF to add a semicolon.

expected closing > to follow <

Error: A module name template string (in a **section** specifier) begins with an open angle (**<**), but lacks a close angle (**>**). Edit the LDF: end the string with a close angle.

expected command file name after -f switch

Error: The **-f** switch lacks its command filename parameter value. Provide a valid command filename after the switch.

expected directive to terminate with ';'

Error: A **checkoverlap**, **nocheckoverlap** or **singlesection** directive lacks its semicolon (**;**) terminator. Edit the LDF to add a semicolon.

expected either ';' or symbol visibility followed by ';'

Error: The symbol expression in a symbol declaration lacks either its semicolon (**;**) terminator or its optional symbol-visibility value. Edit the LDF to add a semicolon or insert a symbol visibility value.

expected entry label to follow -ent switch

Error: The **-ent** switch lacks its symbol-name parameter value. Provide a valid entry label (symbol name) after the switch.

expected expression after unary sign

Error: A minus sign (**-**) lacks a valid expression. Edit the LDF: provide a valid expression after the minus sign.

expected integer for alignment value

Error: The value given for an **alignment** specifier is not a hexadecimal, octal, or decimal number, 0 to 65535. Edit the LDF: specify a value in this range.

expected integer for priority value

Error: The value given for a **priority** specifier is not a hexadecimal, octal, or decimal number, 1 to 65535. Edit the LDF: specify a value in this range.

expected list file name after the -LIST switch

Error: The **-LIST** switch lacks its list filename parameter value. Provide a valid filename after the switch. (If you name a file that already exists, the linker will overwrite the file.)

expected matching ')'

Error: While parsing an expression, the linker found a sub-expression that begins with an open parenthesis [**(**] but lacks a close parenthesis [**)**]. Edit the LDF to add the close parenthesis.

expected module name template between <>

Error: In a **section** specifier, angles (**<>**) do not enclose a module name template. Edit the LDF: insert a module name template between the angles.

expected name of linker definition file following -def switch

Error: The parameter value for the **-def** switch is not the name of a linker definition file. Provide a valid LDF name as the parameter value.

expected number after pad switch

Error: The **-pad** switch lacks its pad-byte parameter value. Provide this numeric value (not an expression) after the switch, in hexadecimal, decimal, or octal format.

expected number, expression or '(' expression ')'

Error: During expression processing, the linker expected one of these values: a simple number, a valid sub-expression, or a sub-expression in parentheses. Edit the LDF: add one of these values.

expected output file name to follow -o switch

Error: The **-o** switch lacks its parameter value: the name of the output object file. Provide a valid filename after the switch. Alternatively, remove the switch: the linker will use the default file **a.out** in the current directory.

expected path specifier following -L switch

Error: The **-L** switch lacks its parameter value: the name of a directory (path) to add to the end of the library path list. Provide a valid path name after the switch.

expected romcopy(<segment name>)

Error: A segment is declared as type romcopy, but the **romcopy** keyword lacks its parameter value: a segment name, in parentheses. Edit the LDF to correct the syntax.

expected section or module name template in section specifier

Error: The **section** specifier of a segment declaration lacks its parameter value: either a section name template or a module name template. Edit the LDF: provide an appropriate template.

expected segment name

Error: The **segment** keyword lacks its parameter value: the name of the segment. Edit the LDF: provide an appropriate segment name.

expected symbol name after symdefine or symboldefine keyword

Error: The **symdefine** or **symboldefine** keyword lacks its argument: a valid symbol name. Edit the LDF: provide the symbol-name argument.

expected type of segment

Error: A **type** specifier in a segment declaration has an incorrect type parameter value. Edit the LDF: provide a valid parameter value.

expected valid expression inside ()

Error: During expression processing, the linker found parentheses [**()**] that do not enclose a valid sub-expression. Edit the LDF: insert a valid sub-expression or remove the parentheses.

expected valid segment type

Error: The value of a segment's **type** specifier is not valid. Edit the LDF to provide one of the valid values: **standard**, **reserved** or **romcopy**.

expected value for start address of .data segment

Error: The **-D** switch lacks its parameter value. Provide this numeric value (not an expression) after the switch, in hexadecimal, decimal, or octal format.

expected value for start address of .text segment

Error: The **-T** switch lacks its parameter value. Provide this numeric value (not an expression) after the switch, in hexadecimal, decimal, or octal format.

expected weak or global for visibility in symbol declaration

Error: The symbol visibility value (at the end of the symbol declaration) is not valid. Edit the LDF to provide one of the valid values: **weak** or **global**.

expected write, read, or execute for protection

Error: The **protect** specifier value is not valid. Edit the LDF to provide one of the a valid values: **read**, **write** or **execute**. Alternatively, provide a list of these values, separating values with commas.

expected '=' or ':' after *keyword* keyword

Error: No appropriate operator (**=**, **:**, or **:=**) follows the specified keyword. Edit the LDF to insert one of these operators.

expected ';' to terminate segment declaration

Error: A segment declaration lacks its semicolon (**;**) terminator. Edit the LDF to add a semicolon.

expected '(<section name>)' following *keyword* keyword

Error: The *keyword* keyword lacks its argument: a section name, enclosed in parentheses. Edit the LDF: provide the section-name argument.

expected '(<segment name>)' following *keyword* keyword

Error: The *keyword* keyword lacks its argument: a segment name, enclosed in parentheses. Edit the LDF: provide the segment-name argument.

expected '(<symbol name>)' following addrof keyword

Error: The **addrof** keyword lacks its argument: a symbol name, enclosed in parentheses. Edit the LDF: provide the symbol-name argument.

extraneous comma in section specifier

Warning: A **section** specifier in an LDF segment declaration contains an inappropriate comma before the final close parenthesis [**)**].

file name *filename* is not valid

Error: The *filename* value specified with the **-o** switch is not valid. Provide a valid filename.

first symbol not undef in ELF object file

Fatal Error: The first symbol in an ELF object file was not the *undefined symbol* required by the ELF format. Probable cause: a corrupted object file or a non-ELF file.

inconsistent object types, must be all hosted or embedded

Error: Object files do not have the same embedded flag setting, so the linker does not know whether the application is hosted or embedded. Make sure that all object files are consistent: all must have their embedded flags set, or all must have their embedded flags cleared.

incorrect processor type for object file

Error: The linker processes only object files for the PowerPC processor family. (ELF files have flags that indicate the processor family they support.) Provide an object file generated by EABI-conforming tools.

invalid bit field replacement : symbol *symbol* addend: *addend* symbol [offset]: *symbolValue* reloc type: *relocation*

Error: The linker could not perform the bit-field-replacement relocation operation on symbol *symbol* as the *addend* value is outside its parameters. The high 16 bits of *addend* must specify a bit position between 0 and 31, the low 16 bits of *addend* must be a length value between 1 and 32, and the sum of the high 16 bits and the low 16 bits must not exceed 32.

Other values of this message may help you find the problem:

- **offset** is or is not in this message, according to the relocation type:
  - if **offset** is not in this message, ***symbolValue*** is the value of symbol *symbol*.
  - if **offset** is in this message, ***symbolValue*** is the relative offset of the relocation.
- ***relocation*** is the type of relocation operation.

The *PowerPC Embedded Application Binary Interface* and *System V Application Binary Interface PowerPC Processor Supplement* have information about relocation types.

invalid command line syntax

Error: The linker found something on the command line (or in a command file) that is not a file name or a switch. Edit the command line (or the command file) to correct this.

invalid command switch '-*switch*'

Error: The linker found something on the command line (or in a command file) that starts with a hyphen, but is not a valid switch. Change this item to a valid switch or remove it.

invalid ELF object file format

Error: The linker found a file that should be an ELF object file but is not. Provide an ELF object file.

invalid expression in *specifier* specifier

Error: The *specifier* specifier does not have a valid segment expression. Edit the LDF to correct the expression.

invalid expression in symbol declaration

Error: In the symbol declaration, the value expression of the symbol is not valid. Edit the LDF to correct the expression.

invalid expression inside ()

Error: During expression processing, the linker found parentheses [**()**] that do not enclose a valid sub-expression. Edit the LDF: insert a valid sub-expression or remove the parentheses.

invalid linker definition file command

Error: An LDF line begins with an invalid keyword. Edit the LDF to provide one of the valid declaration or directive keywords: **segment**, **symboldefine**, **symdefine**, **singlesection**, **checkoverlap**, or **nocheckoverlap**.

invalid module name template *module*

Error: The module name template (enclosed in angles) of a **section** specifier is not valid. Edit the LDF: provide a valid template. The template may include the ***** or **?** wildcard characters.

invalid path specifier following -L switch

Error: The **-L** switch parameter value is not a valid directory (path) name. Provide a valid path name after the switch.

list file *filename* is not a valid file name

Error: The *filename* value is not a valid file name. Provide a valid file name.

low 2 bits must be zero : symbol *symbol* addend: *addend* symbol [offset]: *symbolValue* reloc type: *relocation*

Error: The linker could not perform the relocation operation on symbol *symbol* as the operation produced an address that is not word-aligned (that is, the least significant two bits are not both zero). Such a relocation operation usually specifies the target addresses of a PowerPC processor branch instruction. Look for a section (created from your source code) that does not have the correct alignment requirements. Check the definition of symbol *symbol* to confirm that it is at least word aligned.

Other values of this message may help you find the alignment problem:

- **offset** is or is not in this message, according to the relocation type:
  - if **offset** is not in this message, ***symbolValue*** is the value of symbol *symbol*.
  - if **offset** is in this message, ***symbolValue*** is the relative offset of the relocation.
- ***addend*** is a value added to ***symbolValue***.
- ***relocation*** is the type of relocation operation.

The *PowerPC Embedded Application Binary Interface* and *System V Application Binary Interface PowerPC Processor Supplement* have information about relocation types.

matching segments have no room for section *section*, file *object-file*

Error: An input section matches the **section** specifiers of one or more segments, but none of the segments has room for the section. Edit the segment declarations in the LDF to make room for the section.

maxsize specifier not allowed with size or end specifier

Error: You cannot use the **maxsize** specifier with the **size** or **end** specifier. As the **maxsize** specifier limits the segment size, using another size-limiting specifier is redundant or contradictory. Edit the LDF to remove one of the specifiers.

min size of segment *segment* has cyclic dependency on other segments

Error: The **minsize** specifier of segment *segment* includes an expression that depends on some attribute of another segment. But that attribute eventually depends on the size of this segment. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`minsize specifier not allowed with size or end specifier`

Error: You cannot use the **`minsize`** specifier with the **`size`** or **`end`** specifier. As the **`minsize`** specifier limits the segment size, using another size-limiting specifier is redundant or contradictory. Edit the LDF to remove one of the specifiers.

`module name template exceeds max length of `*`length`*

Error: A module name template (in a **`section`** specifier) exceeds the maximum length allowed. Edit the LDF to shorten the template.

`multiple sections named `*`section`* **`exist`**

Warning: In response to a **`singlesection`** directive, the linker checked for identically named sections in output file segments: multiple sections have the name *section*. Check the LDF to determine why the **`section`** specifiers do not create a single section. Also check whether some input sections named *section* are initialized sections and others are uninitialized.

`number string too large to be represented internally : `*`number-string`*

Fatal Error: The linker interprets string *number-string* to be a number, but the value is too large for the internal limitations of the linker.

`number string too long : `*`number-string`*

Fatal Error: String *number-string* is too long.

`object file contains unsupported relocation section type`

Error: The linker does not accept relocation information from sections of type **`SHT_REL`** in input object files or library members. If you are using only Motorola tools (which support and generate EABI-conforming object files), call Motorola software support. For information on this section type, see *System V Application Binary Interface*.

`object file does not have a symbol table`

Fatal Error: The linker cannot resolve symbol addresses because the input object file does not include a symbol table.

`object file not correct version or format`

Error: An input object file does not have the correct version or format. Make sure that the file is an ELF file, generated by compatible tools.

object file not in relocatable form

Error: An input object file is in executable form (which the linker generates), but the file should be in relocatable form (which a compiler or assembler generates). Make sure that you are linking only input files generated by EABI-compliant tools.

offset *string-table-offset* is outside string table, size *string-table-size* library

Fatal Error: The field value for a library member header name is an invalid offset to the member's name in the library archive string table. Probable cause: a corrupted archive file or an invalid archive file type.

only integer numbers allowed

Error: The linker found a poorly-formed numeric value, such as **1E-05**. Edit the LDF to correct the value.

only one *specifier* specifier per segment allowed

Error: A segment declaration contains more than one *specifier* specifier. Edit the LDF: remove all but one of these specifiers.

only one linker definition file allowed

Error: Multiple **-def** switches specified multiple linker definition files. Merge these files into one, or remove all but one **-def** switch.

pad byte must be between 0 and 255

Error: The parameter value for the **-pad** switch is not a number, 0 to 255. Specify a value in this range.

parameter exceeded max length permitted

Fatal Error: A command-line parameter is too long.

priority specifier not allowed for romcopy segments

Error: A romcopy segment (which cannot have assigned sections) improperly has a **priority** specifier. Edit the LDF to remove the **priority** specifier, or change the segment type.

protect specifier not allowed for reserved segments

Error: A reserved segment (which is not loaded into memory) improperly has a **protect** specifier. Edit the LDF to remove the **protect** specifier, or change the segment type.

quoted string in options file does not have ending quote

Error: The string begins with a double quote, but lacks an ending quote (or the ending quote is on a different line). Use beginning and ending double quotes on the same line.

relocation overflow : symbol *symbol* addend: *addend* symbol [offset]: *symbolValue* reloc type: *relocation*

Error: The linker could not perform the relocation operation on symbol *symbol* as the result does not fit in the intended code or data field. Look in the source for the specified object file or library member, and correct any references to symbol *symbol*. To determine the field size, you may need to check your PowerPC processor reference manual for information on the PowerPC processor instruction set.

Other values of this message may help you find the alignment problem:

- **offset** is or is not in this message, according to the relocation type:
  - if **offset** is not in this message, ***symbolValue*** is the value of symbol *symbol*.
  - if **offset** is in this message, ***symbolValue*** is the relative offset of the relocation.
- ***addend*** is a value added to ***symbolValue***.
- ***relocation*** is the type of relocation operation.

The *PowerPC Embedded Application Binary Interface* and *System V Application Binary Interface PowerPC Processor Supplement* have information about relocation types.

relocation *relocation* virtual address *address* does not meet alignment requirements

Warning: An instruction or data field to be relocated at address *address* does not meet the alignment requirements of the *relocation* relocation type. Check *PowerPC Embedded Application Binary Interface* or *System V Application Binary Interface PowerPC Processor Supplement* to find the relocation type's alignment requirement. Generate a list file to find which section, from which source file, is at address *address*. Confirm the word alignment of all code sections in that source file. Similarly, confirm that all data fields are aligned by the amount of their length (the maximum amount is 4 bytes).

romcopy segment *segment* has section specifier(s)

Error: The romcopy segment segment (which cannot have assigned sections), improperly has a **section** specifier. Edit the LDF to remove the **section** specifier, or change the segment type.

romcopy segment *segment1* is copy of unprocessed segment *segment2*

Error: The *segment1* romcopy segment is a copy of the *segment2* segment, which contains a problem. (A previous error message reported the problem in the *segment2* segment.) The linker cannot process the *segment1* segment until you correct the problem in the *segment2* segment.

sdata base needs to address too large a range of *range*

Error: Sections **.sdata** and **.sbss** (which form a small data area) are too large or too far apart for the linker to set the base pointer, **_SDA_BASE_**. (The linker must establish **_SDA_BASE_** so that its value is within a signed 16-bit offset of every byte in the small data area.) If the data in these sections requires more than 64K bytes, reduce the data. Otherwise, edit the LDF to bring the sections closer together. The *range* value is the total span (in bytes) of the sections.

sdata2 base needs to address too large a range of *range*

Error: Sections **.sdata2** and **.sbss2** (which form a small data area) are too large or too far apart for the linker to set the base pointer, **_SDA2_BASE_**. (The linker must establish **_SDA2_BASE_** so that its value is within a signed 16-bit offset of every byte in the small data area.) If the data in these sections requires more than 64K bytes, reduce the data. Otherwise, edit the LDF to bring the sections closer together. The *range* value is the total span (in bytes) of the sections.

sectafter(*section*) references non-existent section

Error: The **sectafter** keyword's*section* argument does not specify an existing section. Link in an object file that contains this section. Alternatively, edit the LDF to correct this argument value.

sectend(*section*) references non-existent section

Error: The **sectend** keyword's argument does not specify an existing section. Link in an object file that contains this section. Alternatively, edit the LDF to correct this argument value.

section name template exceeds max length of *length*

Error: A section name template (in a **section** specifier) exceeds the maximum length allowed. Edit the LDF to shorten the template.

section specifiers must be inside '()'

Error: Parentheses do not enclose the section and module templates of the **section** specifier. Edit the LDF to add the parentheses.

sectsize(*section*) references non-existent section

Error: The **sectsize** keyword's section argument does not specify an existing section. Link in an object file that contains this section. Alternatively, edit the LDF to correct this argument value.

sectstart(*section*) references non-existent section

Error: The **sectstart** keyword's argument does not specify an existing section. Link in an object file that contains this section. Alternatively, edit the LDF to correct this argument value.

segafter(*segment*) references non-existent segment

Error: The **segafter** keyword's argument does not specify an existing segment. Edit the LDF to declare segment *segment* or correct the argument value.

segend(*segment*) references non-existent segment

Error: The **segend** keyword's argument does not specify an existing segment. Edit the LDF to declare segment *segment* or correct the argument value.

segisize(*segment*) references non-existent segment

Error: The **segisize** keyword's argument does not specify an existing segment. Edit the LDF to declare segment *segment* or correct the argument value.

segment name *segment* not unique

Error: Two segment declarations specify the name *segment*. Edit the LDF to remove or rename one of the segments. (Note that a segment name *can* duplicate a section or symbol name.)

segment *segment* end address less than the start address

Error: Segment *segment* has an end address that is less than its start address. Fix this in the LDF.

segment *segment* has min size *minsize* > max size of *maxsize*

Error: The **minsize** specifier value of segment *segment* exceeds the **maxsize** specifier value. Edit the LDF: change either value to correct this problem, or remove one of the specifiers.

segment *segment* has more than one romcopy reference

Error: Multiple romcopy segments improperly copy segment *segment*. Edit the LDF so that at most one romcopy segment copies segment *segment*.

segment *segment* is a romcopy of itself

Error: Romcopy segment *segment* improperly copies itself. Edit the LDF so that segment *segment* instead copies a standard segment.

segment *segment* is romcopy of nonexistent segment *copied-segment*

Error: Romcopy segment *segment* copies a non-existing segment. Edit the LDF so that segment *segment* copies an existing standard segment.

segment *segment* is romcopy of segment *copied-segment* which is not a standard segment

Error: Romcopy segment *segment* copies a segment of the wrong type. Edit the LDF so that segment *segment* copies a standard segment.

segment *segment* must have at least one section specifier

Error: Segment *segment* is a standard segment but lacks any **section** specifier. Edit the LDF: either add a **section** specifier or change the segment type.

segments *segment1* and *segment2* overlap

Error: Doing overlap checking, the linker found overlapping segments *segment1* and *segment2*. Edit the LDF so that the segments do not overlap.

segsize(*segment*) references non-existent segment

Error: The **segsize** keyword's argument does not specify an existing segment. Edit the LDF to declare segment *segment* or correct the argument value.

segstart(*segment*) references non-existent segment

Error: The **segstart** keyword's argument does not specify an existing segment. In the LDF, either define the segment or correct the argument value.

size, end, minsize or maxsize not allowed for romcopy segment *segment*

Error: Romcopy segment *segment* improperly has a **size**, **end**, **minsize**, or **maxsize** specifier. Edit the LDF to remove the specifier.

size, end, or maxsize expression for segment *segment* cannot be reduced

Error: The linker could not reduce to an integer value the **size**, **end** or **maxsize** specifier expression for segment *segment*, probably because of a cycle. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`size expression has cycle in definition file symbol symbol`

Error: The value expression for symbol *symbol* has a cycle that involves the **`size`** function. This **`size`** function refers to the size of one segment, which depends on an attribute of another segment. But that attribute eventually depends on the size of *this* segment. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`size specifier not allowed with end specifier`

Error: The LDF improperly uses the **`size`** specifier with the **`end`** specifier. This is redundant or contradictory. Edit the LDF to remove one of the specifiers.

`size specifier not allowed with maxsize or minsize specifier`

Error: The LDF improperly uses the **`size`** specifier with a **`minsize`** or **`maxsize`** specifier. This is redundant or contradictory. Edit the LDF to remove one of the specifiers.

`start address of segment segment cannot be reduced`

Error: The linker could not reduce to an integer value the start address of segment *segment*. This probably is because of a cycle involving a second segment for which the linker cannot resolve the start address, the second segment being the argument of some function. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`start expression has cycle in definition file symbol symbol`

Error: The value expression for symbol *symbol* has a cycle that involves the **`segstart`** function. This **`segstart`** function refers to the start of one segment, which depends on some attribute of another segment. But that attribute eventually depends on the start of *this* segment. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`start expression has cycle in segment segment`

Error: The **`start`** specifier value expression for segment *segment* has a cycle that involves the **`segstart`** function. This **`segstart`** function refers to the start of one segment, which depends on some attribute of another segment. But that attribute eventually depends on the start of *this* segment. Edit the LDF to break this unresolvable cyclic dependency. (Paragraph 4.10.1 explains such cyclic constraints.)

`string table referenced but not found in library`

Fatal Error: The field value for a library member header name is an offset to the member's name in the library archive string table, but the linker cannot find a string table. Probable cause: a corrupted archive file or an invalid archive file type.

switch *switch* expected space before parameter

Error: Linker switch *switch* lacks the required space before its parameter value. Insert a space between the switch and the value.

symbol not contained in a section : symbol *symbol* addend: *addend* symbol [offset]: *symbolValue* reloc type: *relocation*

Error: The linker could not perform a relocation operation on the start of the section that contains symbol *symbol*, because the symbol is absolute (that is, no section contains symbol *symbol*). Look in the source for this object file or library member and correct any references to the symbol.

Other values of this message may help you find the problem:

- **offset** is or is not in this message, according to the relocation type:
  - if **offset** is not in this message, ***symbolValue*** is the value of symbol *symbol*.
  - if **offset** is in this message, ***symbolValue*** is the relative offset of the relocation.
- ***addend*** is a value added to ***symbolValue***.
- ***relocation*** is the type of relocation operation.

The *PowerPC Embedded Application Binary Interface* and *System V Application Binary Interface PowerPC Processor Supplement* have information about relocation types.

symbol not in small data area : symbol *symbol* addend: *addend* symbol [offset]: *symbolValue* reloc type: *relocation*

Error: The linker could not perform the relocation operation on symbol *symbol*, as no small data area contains the symbol. Look in the source for this object file or library member and correct any references to the symbol.

Other values of this message may help you find the problem:

- **offset** is or is not in this message, according to the relocation type:
  - if **offset** is not in this message, ***symbolValue*** is the value of symbol *symbol*.
  - if **offset** is in this message, ***symbolValue*** is the relative offset of the relocation.
- ***addend*** is a value added to ***symbolValue***.
- ***relocation*** is the type of relocation operation.

The *PowerPC Embedded Application Binary Interface* and *System V Application Binary Interface PowerPC Processor Supplement* have information about relocation types.

symbol string too long : *string*

Fatal Error: String *string* is too long to be a symbol name.

symbol *symbol* truncated in library

Warning: An object-file or library-member symbol has a very long name. Internal limitations force the linker to truncate this name. Accordingly, comparisons or searches that use this symbol name may yield incorrect results.

there were no object files presented for linking

Error: You provided only library files to the linker. Provide at least one object file.

truncating library member name *member* in library

Warning: A library (archive) file member has a very long name. Internal limitations force the linker to truncate this name. Accordingly, comparisons or searches that use this member name may yield incorrect results.

unable to access section *index*

Fatal Error: The *index* entry in the object file symbol table specifies a section that contains a symbol, but no section header table entry corresponds to *index*. Probable cause: a corrupted symbol table or a non-ELF file.

unable to access symbol *index*

Fatal Error: No symbol seems to have the symbol-table-index value *index*. Probable cause: a corrupted relocation operation or a non-ELF file.

unable to allocate additional memory

Fatal Error: The linker cannot obtain more system memory space. Probable cause: full file system or hard disk.

unable to allocate memory for section references

Fatal Error: The linker cannot obtain more system memory space for constructing section content tables. Probable cause: full file system or hard disk.

unable to append lib to search path, too long

Fatal Error: A **-l** switch specified a library; a **-L** switch specified a library path list. But the linker could not construct a full file name from these items, as their combined length exceeds internal limitations of the linker.

unable to convert library member size from library

Fatal Error: The size field of a library member header does not specify a decimal integer. Probable cause: a corrupted archive file or an invalid archive file type.

unable to find symbol table

Fatal Error: The output file's section header table lacks an entry for a symbol table. Probable cause: a corrupted object file.

unable to identify file as object or library file

Error: An item on the command line (or in a command file) does not seem to be a switch or switch parameter. Accordingly, the item should be the name of an ELF object file or a library (archive) file, but this does not seem to be correct. Check the spelling and syntax of all items in the command line; if the item is a filename, make sure that the file is an object or library file.

unable to match section *section*, file *object* to any segment, check segment defs

Error: The linker could not assign section *section* to a segment, as no segment had a matching **section** specifier. Edit the LDF so that a section specifier matches section *section* (and its file or member).

unable to open command file *filename*

Error: The linker could not open the specified command file. Check the *filename* spelling; make sure that the file exists and that it has at least read access.

`unable to open file, check name, paths, and permissions`

Error: The linker cannot open an input object or library file. The file may not exist in the specified directory, the file may not have the correct permissions, or the file may be too short to be an object or library file. Make sure the file exists and has the correct permissions.

`unable to open linker definition file`

Error: The **`-def`** switch specified an LDF that the linker cannot open. Make sure the file exists and has the correct permissions.

`unable to open listing file`

Error: The linker cannot open the listing file for writing. Make sure that the file exists, that the linker can overwrite the file, that the file name is valid, that the linker can create the file, and that the file system is not full.

`unable to open output object file`

Fatal Error: The linker could not open the output file. Probable cause: file or directory permission violation.

`unable to read member header from library`

Fatal Error: The linker could not read header information from a library (archive) file. Probable cause: a corrupted archive file or an invalid archive file type.

`unable to read object file`

Fatal Error: The linker cannot read from an output file opened previously.

`unable to read output object file`

Fatal Error: The linker cannot read ELF format information from an output file opened previously.

`unable to read string table in library`

Fatal Error: The linker could not read the library file string table information. Probable cause: a corrupted archive file or an invalid archive file type.

`unable to read symbol table count from library`

Fatal Error: The linker could not read the symbol table entry count in the library file symbol table. Probable cause: a corrupted archive file or an invalid archive file type.

`unable to read symbol table member offset from library`

Fatal Error: The linker could not read a member's offset in the library file symbol table. Probable cause: a corrupted archive file or an invalid archive file type.

unable to read symbol table string

Fatal Error: The linker could not read a symbol table name in the library file symbol table. Probable cause: a corrupted archive file or an invalid archive file type.

unable to reduce absolute symbol *symbol*

Error: The linker could not reduce the value expression for symbol *symbol*. The expression probably has a cycle involving a function whose argument is a segment, a section or another symbol. Edit the LDF to correct the expression or remove the cycle. (Paragraph 4.10.1 explains such cyclic constraints.)

unable to reduce end, maxsize or size expression for segment *segment*

Error: Segment *segment* contains an **end**, **maxsize**, or **size** specifier, but the linker could not reduce the specifier's value expression to an integer value. The expression probably has a cycle involving a function whose argument is another segment whose start address or size cannot be reduced. Edit the LDF to remove the cycle. (Paragraph 4.10.1 explains such cyclic constraints.)

unable to reduce start expression to determine size of segment *segment*

Error: Segment *segment* has **start** and **end** specifiers, but the linker could not reduce both specifiers' value expressions to integer values. Either expression probably has a cycle involving a function whose argument is another segment whose start address or size cannot be reduced. Edit the LDF to remove the cycle. (Paragraph 4.10.1 explains such cyclic constraints.)

unable to reopen file

Fatal Error: The linker cannot re-open a command file (to resume processing command information) after processing command information from a different command file.

unable to write to object file

Fatal Error: The linker cannot write relocation symbols to the output file. Probable cause: full file system.

unable to write to output file

Fatal Error: The linker cannot write the ELF header, symbol-table entries, or other such entries of the output file. Probable cause: full file system.

unable to write to output object file

Fatal Error: The linker cannot write ELF format information to the output file. Probable cause: full file system.

unrecognized segment option

Error: In a segment declaration an expression is out of place, a keyword is not correct, or there is some other incorrect syntax. Edit the LDF to correct the segment declaration.

unresolved symbol : *symbol*

Error: At least one object file or library member referenced symbol *symbol*, but the linker could not find the symbol's definition. Include in the link the object file or library member that defines the symbol.

unsupported relocation type: *type*

Fatal Error: The linker did not recognize the *type* value of an object-file or library-member relocation operation. For more information about relocation types, see the *PowerPC Embedded Application Binary Interface* or *System V Application Binary Interface PowerPC Processor Supplement.*

unsupported implicit relocations

Fatal Error: Object-file or library-member sections of type **SHT_REL** contain implicit relocation instructions, which the linker does not accept. (For more information about section type **SHT_REL**, see the *System V Application Binary Interface*.) If you are using only Motorola tools (which support and generate EABI-conforming object files), call Motorola software support.

APPENDIX B

MOTOROLA ARCHIVER

The Motorola Archiver (MAR) creates and maintains archive (library) files containing groups of files. These archive files conform to the System V Release 4 archive file format.

Although the archiver can combine files of any format into archives, a typical use is creating libraries of relocatable ELF object files. The Motorola Link Editor (MELD) or another linker can efficiently search through such a library to resolve external symbol differences. For example, you might use the archiver to create a library of input/output routines. Each time you used the linker to build an executable file you would be able to link the same archive file, instead of linking specific input/output routines.

An archive file has a global symbol table, which the archiver updates each time you change the content or order of member files. The linker uses this table to search for symbol definitions efficiently, even if it makes multiple passes of the archive file.

B.1 COMMAND SYNTAX

The MAR command syntax is either:

mar [-]*action* [*modifiers*] [*position*] [*archive*] [*member ...*]

where:

-	An optional leading hyphen, which the archiver permits for users already familiar with typical UNIX-style commands.
action	One action value from Table B-1: delete, replace, move, and so forth.
modifiers	Optional values from Table B-2; these values modify the archiver action. You need not use spaces to separate modifier values.
position	Name of a member file, used with **a**, **b**, or **i** modifier values to specify where a member file should be inserted or moved.
archive	Name of the archive file.
member	Name of a file to be made a member file of the archive, or the name of an existing member file to be modified.

or

mar -f *option_file*

where

option_file Name of a file whose contents are of the syntax:

[-]*action* [*modifiers*] [*position*] [*archive*] [*member ...*]

Options in the file may be separated by spaces, tabs, or the ends of lines. Names of files (***position***, ***archive***, and ***member***) may be enclosed in double quotes (**"**).

(For an on-screen summary of archiver command syntax, enter the archiver command without any parameter values.)

The archiver returns the value 0 when it successfully carries out an archiver action. A non-zero return value indicates an error.

Table B-1. Archiver Action Values

Action	Effect	Comments
d	Deletes specified ***member*** files from ***archive***.	
r	Replaces specified ***member*** files in ***archive***. If any ***member*** files are new, adds them to ***archive*** in command-line order. Creates ***archive*** if it does not already exist.	If the command line lacks any **a**, **b**, or **i** modifier, adds any new ***member*** files to the end of ***archive***. If the command creates ***archive***, prints an archive-created message to **stdout** unless the command line includes the **c** option. If ***member*** files of the command line have the same base name, the archiver adds multiple ***member*** files of identical names to ***archive***.
q	Quickly appends specified ***member*** files to ***archive*** in command-line order; does not check whether ***member*** files already exist in ***archive***. Creates ***archive*** if it does not already exist.	If the command creates ***archive***, prints archive-created message to **stdout** unless the command line includes the **c** option. If ***member*** files of the command line have the same base name, the archiver adds multiple ***member*** files of identical names to ***archive***. Similarly, if ***member*** files of the command line duplicate names of ***member*** files already in ***archive***, ***archive*** can end up with multiple ***member*** files of identical names.
t	Prints table of contents for each specified ***member*** file, to **stdout**. ***Member*** names appear in their order in ***archive***.	If the command line lacks any ***member*** names, prints table of contents for all ***member*** files of ***archive***.

Table B-1. Archiver Action Values (Continued)

Action	Effect	Comments
S	Prints ***archive*** file's global symbol table to **stdout**. Symbol names appear in their order in ***archive***.	This action value must be upper case.
p	Prints contents for each specified ***member*** file, to **stdout**. Contents appear in their order in ***archive***.	If the command line lacks any ***member*** names, prints contents for all ***member*** files of ***archive***.
m	Moves specified ***member*** files to a new location in ***archive***; moved ***member*** files keep their same relative order.	If the command line lacks **a**, **b**, or **i** modifier, moves specified ***member*** files to the end of ***archive***. Specified ***member*** files must already exist in ***archive***.
x	Extracts specified ***member*** files from ***archive***. Writes such files to the current directory, leaving them unchanged in ***archive***.	If the command line lacks any ***member*** names, extracts all ***members*** of ***archive***. Creates extracted files as new, with user as owner. But if directory already had a previous file of a ***member*** file's name and permissions allow, the archiver retains the previous file's owner and group attributes, overwriting its contents with ***member****'s*.
V	Prints MAR version information to **stdout**.	This action value must be upper case.

Table B-2. Archiver Modifier Values

Modifier	Effect	Comments
u	With the **r** action, replaces a ***member*** file of ***archive*** only if the corresponding disk file is newer.	Has no effect with actions other than **r**.
a *position*	With the **m** action, moves specified existing ***member*** files after the ***position*** file. With the **r** action, (1) replaces specified existing ***member*** files without moving them, and (2) adds specified new ***member*** files after the ***position*** file.	Requires ***position*** argument, which must be an existing ***member*** name of ***archive***. Has no effect with actions other than **m** or **r**.

Table B-2. Archiver Modifier Values (Continued)

Modifier	Effect	Comments
b ***position***	With the **m** action, moves specified existing ***member*** files before the ***position*** file. With the **r** action, (1) replaces specified existing ***member*** files without moving them, and (2) adds specified new ***member*** files before the ***position*** file.	Requires ***position*** argument, which must be an existing ***member*** name of ***archive***. Has no effect with actions other than **m** or **r**. The **i** option is identical.
i ***position***	With the **m** action, moves specified existing ***member*** files before the ***position*** file. With the **r** action, (1) replaces specified existing ***member*** files without moving them, and (2) adds specified new ***member*** files before the ***position*** file.	Requires ***position*** argument, which must be an existing ***member*** name of ***archive***. Has no effect with actions other than **m** or **r**. The **b** option is identical.
o	With the **x** action, sets modification times of extracted ***member*** files to times when ***member*** files were added to ***archive***.	User must be owner or super user of extracted members. Has no effect with actions other than **x**.
v	With the **d**, **r**, **q**, **m**, or **x** action, gives file-by-file description of the operation progress. With the **t** action, prints the table of contents as follows: each member name on a separate line, preceded by permissions, user id and group id numbers, size in bytes, and modification time. With the **S** action, prints symbols on separate lines, with offset number of bytes from the start of the ***member*** defining the symbol and the defining ***member's*** name. With the **p** action, adds a header to the contents printing of each ***member***: a newline, the ***member*** name in angles, and two more newlines.	Prints to **stdout** ***d-member*** for each ***member*** deleted, ***r-member*** for each ***member*** replaced, ***a-member*** for each ***member*** added, ***q-member*** for each ***member*** quickly replaced, ***m-member*** for each ***member*** moved, and ***x-member*** for each ***member*** extracted. Has no effect with the **V** action.
c	With the **r** or **q** action, suppresses any archive-created message	Has no effect with actions other than **r** or **q**.

Table B-2. Archiver Modifier Values (Continued)

Modifier	Effect	Comments
s	With the **t**, **S**, **p**, or **x** action, regenerates hidden symbol-table ***member*** file of ***archive***.	Has no effect with other actions.
l	With the **d**, **r**, **q**, or **m** action, uses the current working directory (instead of default) to create a temporary working file. Does the same thing with the **t**, **S**, **p**, or **x** action, if the command also includes the **s** option.	Has no effect with the **V** action. Has no effect with the **t**, **S**, **p**, or **x** action, if the command lacks the **s** option.

B.2 MAR TEMPORARY FILES

During operation, the archiver creates temporary files for interim calculations and value storage. If you have defined the environment variable **TMPDIR**, the archiver tries to create such temporary files in the directory that **TMPDIR** specifies. If you have not defined **TMPDIR**, the archiver tries to create its temporary files in directory **/tmp**.

If you wish the archiver to create its temporary files in the current working directory, you may use the **l** modifier, which overrides the **/tmp** directory and any path that **TMPDIR** specifies.

B.3 ARCHIVE EXAMPLES

Example 1: Create a new archive (**libthing.a**) by quickly appending four files, giving a file-by-file description of the progress:

```
$ mar -qv libthing.a file3.o file2.o file1.o file4.o
mar: created libthing.a
q - file3.o
q - file2.o
q - file1.o
q - file4.o
```

Example 2: Show the expanded table of contents for archive file **libthing.a**:

```
$ mar -tv libthing.a
rw-rw-r-- 227 /1   1024 Aug 01 17:00 1995 file3.o
rw-rw-r-- 227 /1    512 Aug 01 17:00 1995 file2.o
rw-rw-r-- 227 /1   2048 Aug 01 17:00 1995 file1.o
rw-rw-r-- 227 /1   2048 Aug 01 17:00 1995 file4.o
```

Example 3: Move **`file2.o`** and **`file3.o`** between **`file1.o`** and **`file4.o`** (note that **`file4.o`** is the ***`position`*** value). Then show the table of contents for the modified archive file, to verify that **`file2.o`** and **`file3.o`** maintained their order in Example 2.

```
$ mar -mvb file4.o libthing.a file2.o file3.o
m - file3.o
m - file2.o
$ mar -t libthing.a
file1.o
file3.o
file2.o
file4.o
```

Example 4: Move **`file3.o`** between **`file2.o`** and **`file4.o`**, then show the table of contents for the modified archive file, to verify that files are in numerical order:

```
$ mar -mva file2.o libthing.a file3.o
m - file3.o
$ mar -t libthing.a
file1.o
file2.o
file3.o
file4.o
```

Example 5: Replace **`file2.o`** and **`file4.o`**, add a new file **`file0.o`** before **`file1.o`**, then show the updated table of contents:

```
$ mar -rvb file1.o libthing.a file0.o file4.o file2.o
r - file2.o
r - file4.o
a - file0.o
$ mar -t libthing.a
file0.o
file1.o
file2.o
file3.o
file4.o
```

APPENDIX C

MOTOROLA S-RECORD GENERATOR

The Motorola S-Record Generator (MSREC) converts ELF-format relocatable object files to S-records: text strings in a special ASCII format, suitable for RS-232 transmission. You can download S-records to PROM programmers or download them directly to a target device for execution. Section C.5 describes the structure of S-record files.

The S-record generator also can generate S-records from non-ELF files, for reliable transfer of programs between different types of computer systems.

C.1 COMMAND SYNTAX

The simple representation of MSREC command syntax is:

msrec [[*option parameter*]...] *input_file*

where:

option	An option value, from Table C-1.
parameter	A parameter value from Table C-1.
input_file	The name of the object file to be converted.

But actual **msrec** commands are more detailed than this simple syntax may suggest, because:

- Most *options* require a *parameter* value of a specific type.
- Certain *options* must not have any *parameter* value.
- The **-mem** option requires two *parameter* values.
- The **-n** option may contain multiple **n** and **r** parameter values, as in **-n nrrrn**.
- You may use just the first letter of most options.

Table C-1 explains such information with regard to all option and parameter values. (For a screen summary of these options, enter the **msrec** command with no option or parameter values.)

Accordingly, a complete representation of the command syntax is:

```
msrec  [-e[ntry] address|symbol]
       [-seg|+seg segment]
       [-sec|+sec section]
       [-i[gnore]]
       [-C control_file]
       [-o output_file_base]
       [-S1|-S2|-S3]
       [-l[ine] number]
       [-b[ase] address]
       [-r[eloc] offset]
       [-s[um]]
       [-p[ad] pattern]
       [-m[em] address:address]
       [-c[om] "string"]
       [-w[ait] number]
       [-n (n|r)...]
       [-h[elp]]
       [-q[uiet]]
       [-v[er]]
       [-dw (8|16|32)]
       [-dm (8|16|32|64)]
       [-dd number]
       [-ds number]
       [-f] input_file
```

Should an **msrec** command include the same option multiple times, the S-record generator ignores all but the last. For example, in the command **msrec -S3 -S1 a.out**, the S-record generator ignores the **-S3** option, but follows the **-S1** option (limiting output for data records to type S1 S-records). Another such example is the **-o** option, which specifies the base name for output files: the **msrec** command could have many **-o** options, but only the last would take effect.

But these syntax errors will cause the S-record generator to abort, without generating any output files:

- Two or more **-c** options in an **msrec** command.
- An **msrec** command that includes **-seg** or **+seg** options, and also **-sec** or **+sec** options.
- Two or more input files.

Table C-1. S-Record Generator Options

Option	Effect	Parameter	Comments
-entry	Specifies the entry point (that is, the address of the instruction to which control will be passed). The S-record generator assigns this value to the address field of the termination S-record.	***address*** — the address value for the termination record's address field. May be in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the address value a multiple of decimal 1024. ***symbol*** — a member of the ELF symbol table of the input ELF file. Must be of global or weak binding; must be of type object, function, or section. The address field of the termination S-record receives the address that corresponds to the symbol.	A loader or other such program that later processes S-records uses the entry-point value. If the parameter value is a symbol, but the symbol is not found (or if the binding or type information is bad) the termination record's address field receives the value 0. If the command does not include this option, the address field of the termination record receives the e_entry value of the ELF file header.
-seg	Excludes from S-record generation the specified segment.	***segment*** — name of an ELF-file segment.	Cannot be used with **-sec** or **+sec** options. Use a separate **-seg** option for each segment you exclude.
+seg	Includes for S-record generation the specified segment.	***segment*** — name of an ELF-file segment that has non-zero memory size, particularly one not of type PT_LOAD.	Cannot be used with **-sec** or **+sec** options. Use a separate **+seg** option for each segment you include. If the command does not include this option, the S-record generator considers all type PT_LOAD segments of non-zero memory size for S-record generation.
-sec	Excludes from S-record generation the specified section.	***section*** — name of an ELF-file section, such as **.text** or **.data**.	Cannot be used with **-seg** or **+seg** options. Use a separate **-sec** option for each section you exclude.

Table C-1. S-Record Generator Options (continued)

Option	Effect	Parameter	Comments
+sec	Include for S-record generation the specified section.	***section*** — name of an ELF-file section, such as **.text** or **.data**, that has non-zero memory size and the SHF_ALLOC attribute flag, particularly if its type is not SHT_PROGBITS.	Cannot be used with **-seg** or **+seg** options. Use a separate **+sec** option for each section you include. If the command includes at least one **+sec** option, the S-record generator considers for S-record generation all type SHT_PROGBITS sections that have the SHF_ALLOC attribute flag and non-zero memory size.
-ignore	Limits consideration for S-record generation to segments or sections specifically included via **+seg** or **+sec** options.	None.	If the command does not include this option, the S-record generator considers for S-record generation all type PT_LOAD segments of non-zero memory size, or (if the command includes any **-sec** or **+sec** options), all type SHT_PROGBITS sections of non-zero memory size that have the SHF_ALLOC attribute flag set.
-C	Specifies the control file. (Section C.4 gives more information about control files.)	***control_file*** — name of a control file.	If the command does not include this option, the S-record generator considers only options specifically part of the **msrec** command line. This option must be upper case.
-o	Specifies the base name for all output files, so that output files do not go to the standard output. (Section C.6 gives more information about output files.)	***output_file_base*** — base filename for S-record output files.	If the command does not include this option, all generated S-records go to the standard output.

Table C-1. S-Record Generator Options (continued)

Option	Effect	Parameter	Comments
-S1, **-S2**, or **-S3**	Limits the output for data records to the specified S-record type.	None.	If the command does not include this option, the S-record generator outputs the smallest S-record type that can accommodate the data's addresses.
-line	Overrides the default maximum number of printable characters allowed in an S-record.	***number*** — integer value, in decimal range 14—254. May be in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the value a multiple of decimal 1024.	If the command does not include this option, MSREC sets 80 as the default maximum.
-base	Specifies an alternate base memory address. Uses this address value in the S-record that has the lowest starting address, adjusting all other S-record address fields appropriately.	***address*** — the address value for the alternate base memory address. May be in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the address value a multiple of decimal 1024.	Can cause address values to wrap around the end of the 32-bit address space. If the command does not include this option, S-records will contain address values derived from ELF input file segment start addresses (or section start addresses if the command includes **+sec** or **-sec** options).
-reloc	Specifies an offset from the base memory address. The S-record generator adds this offset (as a 2's complement 32-bit number) to the base memory address, adjusting other S-record address fields accordingly.	***offset*** — **-** or **+** character, with an integer value (in octal, decimal, or hexadecimal format). An optional **k** or **K** suffix makes the address value a multiple of decimal 1024.	Can cause address values to wrap around the end of the 32-bit address space. The base memory address is the address-field value of the S-record that corresponds to the beginning of the segment (or section) that has the lowest starting address. If the command does not include this option, S-record address fields will reflect segment/section start addresses of the ELF input file, or values specified by the **-base** option.

Table C-1. S-Record Generator Options (continued)

Option	Effect	Parameter	Comments
-sum	Sends to standard output each output file's data checksum. (Checksum is a one-byte hexadecimal value: the least significant byte of the 1's complement of the byte-value sum of all data fields of all S1, S2, and S3 S-records.)	None.	Overrides the -quiet option. If the command does not include this option, the S-record generator silently calculates checksums and writes them into the S-records.
-pad	Activates a padding character for S-record output. For all undefined input-file addresses in the low-high address range, the S-record generator produces S-records with the *pattern* value in the data field.	***pattern*** — Value for one byte, specified as an octal, decimal, or hexadecimal integer, or specified as one printable ASCII character.	The **-base**, **-reloc**, and **-mem** options can modify the low-high address range. If the command does not include this option, the S-record generator does no padding.
-mem	Specifies alternate lowest and highest addresses to be considered for S-record generation.	***address:address*** — the alternate address values. May be in octal, decimal, or hexadecimal format. Optional **k** or **K** suffixes make the address values multiples of decimal 1024.	A colon must separate the two address values. These addresses override input-file default lowest and highest addresses (which the **-base** and **-reloc** options can modify). The S-record generator considers for S-record generation only data within the specified address range.
-com	Puts the specified string value in the data field of all type S0 S-records.	***string*** — printable ASCII text.	Double quotes must enclose *string* value. If the command does not include this option, the data field of all type S0 S-records has the value *input_file*.

Table C-1. S-Record Generator Options (continued)

Option	Effect	Parameter	Comments
-wait	Directs the S-record generator to wait the specified number of seconds between the output of one S-record and the next S-record (to the standard output).	***number*** — integer value, in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the value a multiple of decimal 1024.	This option is meaningless if the **-o** option directs output to a file other than the standard output. If the command does not include this option, S-records go to the standard output as fast as they are generated.
-n	Specifies alternative end sequences for successive output S-records.	***n*** — line-feed character. ***r*** — carriage-return character.	This option can have multiple parameter values, for such an option as **-n nnrrn**. If the command does not have this option, each S-record (line of output) ends with a line-feed (0xA) character.
-help	Prints a summary of options allowed.	None.	Prints summary only to the standard output.
-quiet	Suppresses all warning messages.	None.	
-ver	Prints version and copyright information.	None.	Prints information only to the standard output.
-dw	Specifies device width in number of bits.	8, 16, or 32.	Parameter value must not exceed the **-dm** option value; the default value is 32. This option is valid only if the command also includes the **-o** option.
-dm	Specifies memory bus width in number of bits.	8, 16, 32, or 64	Parameter value must equal or exceed the **-dw** option value; the default value is 32. This option is valid only if the command also includes the **-o** option.

Table C-1. S-Record Generator Options (continued)

Option	Effect	Parameter	Comments
-dd	Specifies device depth upper limit (number of virtual sets or sets of output files).	***number*** — integer value, in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the value a multiple of decimal 1024.	This option is valid only if the command also includes the **-o** and **-ds** options. If the command includes the **-ds** option, the default depth value is 26 (for levels a—z); otherwise, the default depth value is 1.
-ds	Specifies size of individual device or output file, in bytes.	***number*** — integer value, in octal, decimal, or hexadecimal format. An optional **k** or **K** suffix makes the value a multiple of decimal 1024.	This option is valid only if the command also includes the **-o** option. The default size is unlimited. Specifying a small device size for a large input file can force the creation of multiple sets of output files.
-f	Optional input file designator	None.	Precedes the mandatory *input-file* value of the command.

C.2 SEGMENTS AND SECTIONS

There are two ways to consider the contents of an ELF-format file:

- A collection of *segments* (each of which consists of *sections*). This perception fits with the executable (or loader) view of the ELF file.
- A collection of *sections*. This perception fits with the linker view of the ELF file.

The S-record generator's default behavior is to generate S-records for all loadable *segments* of an ELF executable input file. The S-record generator can exercise finer control over embedded PowerPC ELF files produced by the Motorola Embedded C Compiler, as these files can associate a name or label with each segment. The **-seg** option lets you exclude, by name, specific loadable segments from S-record generation. (Use a separate **-seg** option for each segment you exclude.) The **+seg** option lets you include for S-record generation segments of non-zero memory size not marked loadable. (Use a separate **+seg** option for each such segment you include.)

For absolute control, you can use the **-ignore** option, which tells the S-record generator to generate S-records only for segments you specify via **+seg** options.

Alternatively, the S-record generator can generate S-records from the *sections* of the ELF executable input file. For this behavior, you must use the **-sec** or **+sec** options in the **msrec** command line. Either option triggers a section default arrangement: the S-record generator generates S-records for all loadable sections of type SHT_PROGBITS. To exclude specific loadable sections, use the **-sec** option. (Use a separate **-sec** option for each section you exclude.) To specifically include sections of either type SHT_PROGBITS or SHT_NOBITS, use the **+sec** option. (Use a separate **+sec** option for each section you include.) In compiled programs, the **.text** and **.data** sections will be of type SHT_PROGBITS, and the **.bss** section will be of type SHT_NOBITS.

Using the **-ignore** option excludes all sections but those you include via **+sec** options.

C.3 ROMCOPY SEGMENTS

The S-record generator supports the *romcopy* feature of embedded PowerPC ELF executable files. This feature lets you maintain an image of RAM data in device ROM. A romcopy segment contains duplicate data of a RAM segment. But the romcopy segment also contains directions that its duplicate data be loaded at a different, non-RAM, address in device memory.

RAM data must reside in writeable memory, as it typically includes variable values that the program can change during execution. An example of such RAM data is the initialized data (**.data**) of a compiled C program. Correct execution is possible only if the variables in the program's **.data** area have been initialized to known values before execution begins. In a working embedded system, part of system initialization would be copying such known values from their romcopy addresses to the appropriate RAM addresses.

Note the two common stages for developing embedded system code that will reside in ROM:

1. **During early development**, you repeatedly download system data and text to RAM, instead of programming them into ROM. This facilitates and speeds iterations of the code-test-debug cycle, and makes it easy to patch code in memory.
2. **When code is finished or almost finished**, you program the code into EPROMs and install it into the embedded target system.

Also note that romcopy segments are marked as loadable, but RAM segments are not. The S-record generator's default behavior, when generating S-records from segments, is to include segments marked loadable. This means that the S-record generator generates S-records for the romcopy segments, but not for the RAM segments that contain the same data.

Accordingly, the two ways to download data of RAM segments correspond to the code-development stages.

1. **During early development**, use the **+seg** option to specify the RAM segments. (The segments must not have memory size zero.) The S-record generator generates S-records for the RAM segments just as it generates S-records for the ROM segments. Downloading writes both the RAM S-records and the ROM S-records to system RAM.

   This download initializes **.data** variables to the proper startup values; these values are ready for the downloaded program when its execution begins. The program can change these values as appropriate. To restore the original startup RAM values, download the RAM S-records again.

2. **When code is finished or almost finished**, duplicate the RAM-segment data in romcopy segments. The S-record generator generates S-records for the romcopy segments, as they are marked loadable. You can download the S-records for both the **.text** (instructions) and romcopy segments to a device programmer, for burning into an EPROM. Startup code must copy the romcopy-segment data to the appropriate RAM locations before you execute the program. Each time you execute the program, the startup initialization routines copy the startup values from device ROM to system RAM.

   (The Embedded Tools Getting Started Guide includes an example of startup code that copies romcopy-segment data to the **.data** area.)

   This second method also is appropriate when downloading into target-system RAM for debug. In this situation, the startup code copies the romcopy-segment data from one location in system RAM to a second location in system RAM. (The second location is the one appropriate for the startup RAM values.) This allows you to test and debug the startup code in RAM before you program it into EPROM. Additionally, this method reinitializes variables whenever you restart the downloaded program, without requiring another download.

C.4 CONTROL FILE FORMAT

A control file is an ASCII text file that consists of **msrec**-command options and parameter values. Specifying the control file in your **msrec** command, via the **-C** option, has the same effect as including the individual options and parameter values in the **msrec** command. A control file helps you avoid typing mistakes for **msrec** commands that have many options and parameter values. A control file saves time if you will enter a sequence of **msrec** commands that share many identical options and parameter values.

Use your standard text editor to write a control file. The text can be any of the **msrec**-command options and parameter values, except the **-C** option.

- Spaces must separate options from each other and from their parameter values, just as in the **msrec** command. The new-line and carriage-return characters can serve as spaces, in this regard.
- The input filename and the output base filename may be enclosed in double quotes (**"**).
- You cannot nest command files. (This is why you cannot include the **-C** option in command files.)
- The **#** character starts a comment string, which continues until a new-line, carriage-return, or end-of-file character. The S-record generator does not include the comment or the **#** character in its command stream of options and parameter values.
- The control file should not contain shell variables, functions, or aliases. As no shell program evaluates control-file text, the S-record generator would try to use such variables, functions, or aliases as command options or parameter values.

For example, suppose that control file **test1** contains this sequence:

```
-e 0x1000          #entry point set to 0x1000
+sec theta         #include section theta
-sec iota          #exclude section iota
-line 100          #set S-record length
-S3                #use only S3 S-records
a.out              #the ELF input file
```

If so, entering the command

```
msrec -C test1
```

has the same effect as entering the command

```
msrec -e 0x1000 +sec theta -sec iota -line 100 -S3 a.out
```

C.5 S-RECORD FORMAT

An S-record file consists of a sequence of formatted ASCII-character strings: each such string is an S-record. There is no significance to the order of S-records within an S-record file, except that the last record must be of type S7, S8, or S9.

S-records are strings of five fields: type, count, address, data, and checksum. Each byte of binary data is encoded as a two-character hexadecimal number: the first character represents the high-order four bits of the byte, and the second character represents the low-order four bits.

The diagram below shows the general format of an S-record. Table C-2 shows the composition of each field; Table C-3 lists the types of S-records.

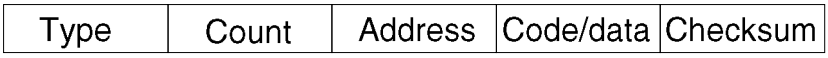

Table C-2. S-Record Field Composition

Field	Printable Characters	Contents
Type	2	S-record type: S0, S1, S2, S3, S7, S8, or S9.
Count	2	Number of remaining character pairs in the S-record.
Address	4, 6, or 8	The 2-, 3-, or 4-byte address at which the data field is to be loaded in memory.
Code/data	0 — n	From 0 to n bytes of executable code, memory-loadable data, or descriptive information.
Checksum	2	The least significant byte of the one's complement of the sum of values represented by the character pairs making up the count, address, and code/data fields.

A line feed usually ends each S-record. Should you need additional or different S-record terminators, use the **-nn** or **-nr** command option. Alternatively, your transmitting program may be able to specify terminators.

Table C-3. S-Record Types

Type	Hexadecimal Type	Description
S0	0x5330 (first byte: 0x53, second byte:0x30)	Header record. The address field normally is filled with zeros. The code/data field contains a description of the following block of S-records.
S1	0x5331	Data record. The address field contains a 2-byte address. The code/data field contains memory-loadable data.
S2	0x5332	Data record. The address field contains a 3-byte address. The code/data field contains memory-loadable data.
S3	0x5333	Data record. The address field contains a 4-byte address. The code/data field contains memory-loadable data.
S7	0x5337	Termination record. The address field contains a 4-byte address, which can signal an entry point (that is, the address of the instruction to which control is to be passed). There is no code/data field.
S8	0x5338	Termination record. The address field contains a 3-byte address, which can signal an entry point (that is, the address of the instruction to which control is to be passed). There is no code/data field.
S9	0x5339	Termination record. The address field contains a 2-byte address, which can signal an entry point (that is, the address of the instruction to which control is to be passed). There is no code/data field.

There is only one termination record for each block of S-records. Normally, there is only one header record, although multiple header records are possible.

This example shows a typical S-record file:

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
S11300100002000800082629001853812341001813
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

This file consists of one S0 record, four S1 records, and one S9 record.

The explanation of the S0 record is:

S0	Type indicator S0, indicating a header record.
06	Hexadecimal 06, indicating that six character pairs (or ASCII bytes) follow.
00 00	Four-character, 2-byte address field; zeros.
48 44 52	ASCII H, D, and R are "HDR."
1B	Checksum of the S0 record.

The explanation of the first S1 record is:

S1	Type indicator S1, indicating a code/data record to be loaded at a 2-byte address.
13	Hexadecimal 13, indicating that 19 character pairs follow. These pairs represent a 2-byte address, 16 bytes of binary data, and a 1-byte checksum.
00 00	Four-character, 2-byte address field containing hexadecimal address 0x0000: the loading address for the data that follows.
28 5F 24 5F 22 12 22 6A 00 04 24 29 00 08 23 7C	The character pairs representing the actual binary data.
2A	Checksum of the S1 record.

Similarly, the second and third S1 records contain 0x13 (19) character pairs; these records end with checksums 13 and 52, respectively. The fourth S1 record contains 0x07 character pairs and has a checksum of 92.

The explanation of the S9 record is:

S9	Type indicator S9, indicating a termination record.
03	Hexadecimal 03, indicating that three character pairs follow.
00 00	Four-character, 2-byte address field; zeros.
FC	Checksum of the S9 record.

C.6 MSREC OUTPUT FILES

The S-record generator's default behavior is to send all the S-records it generates to the standard output. If your **msrec** command specifies an output base filename, via the **-o** option, the S-record generator uses the base filename, after removing any file extension, to construct the output file name. If there is only one output file, it writes the S-records to the base filename with the extension **.mx** appended. For example, if your **msrec** command includes the expression **-o ../alpha.ext**, the S-record generator sends all valid output to file **../alpha.mx**.

You can download this output file directly to target memory. Alternatively, you can download this output file to a device programmer for programming an EPROM device, a flash-memory device, or other such device.

Some embedded target hardware holds program code in an array of EPROM devices or in other multiple programmable memory devices, so you may need to divide the output S-records among the devices. Accordingly, you can have the S-record generator generate multiple output files, each of which contains part of the total program code. Each output file corresponds to one programmable memory device; you can download each output file, in turn, to the device programmer.

Use the command options **-dm** (memory bus width), **-dw** (device width), **-ds** (device size), and **-dd** (device depth) to control the number and content of S-records written to each output file. Use these same options to generate multiple output files, to be mapped to multiple programmable devices.

When hardware design dictates that each access to the EPROM address range must get information simultaneously from multiple devices, use the **-dm** and **-dw** options. An access to the first address within the EPROM address range actually accesses the first location of multiple devices.

For example, many systems use two byte-wide EPROM devices for 16-bit memory accesses. For each fetch of a 16-bit instruction, the processor reads the one byte of each EPROM device. One device supplies the upper byte of the instruction, and the other device supplies the lower byte of the instruction. For such a situation, your **msrec** command should specify an 8-bit device width (**-dw 8**) and a 16-bit memory bus width (**-dm 16**): the S-record generator automatically programs the upper bytes of instructions into one output file and the lower bytes of instructions into a second output file.

Another common design practice, when one EPROM device is too small to contain all the code, is having two consecutively addressed EPROM devices for the code. In this situation, your **msrec** command should include the **-ds** option to specify the device size. This way, the S-record generator knows to write any overflow from the first device to a second output file, for programming to the second device.

Multiple output files share the same base filename, but have digit-letter extensions that make the files unique. For example, if your **msrec** command includes the expressions **-o beta**, **-dw 8**, **-dm 32** , and **-ds 16K**, the S-record generator sends valid output to files **beta.1a**, **beta.2a**, **beta.3a**, **beta.4a**, **beta.1b**, **beta.2b**, **beta.3b**, **beta.4b**, and so forth. Figure C-1 represents such files for an input file of 300 kilobytes. (Section C.7.1 explains this calculation in more detail.)

←——————Bus width (32 bits)——————→			
Device width ←——(8 bits)——→			
beta.1a	beta.2a	beta.3a	beta.4a
beta.1b	beta.2b	beta.3b	beta.4b
beta.1c	beta.2c	beta.3c	beta.4c
beta.1d	beta.2d	beta.3d	beta.4d
beta.1e	beta.2e	beta.3e	beta.4e

Figure C-1. Beta Example Output Files

Beyond determining the number of output files, the **-dw** and **-dm** options affect the arrangement of contents. For example, suppose that your **msrec** command includes the expressions **-o gamma**, **-dw 16**, and **-dm 64**. The S-record generator assigns output bytes 0, 1, 8, 9, and so forth to file **gamma.1a**; bytes 2, 3, 10, 11, and so forth to file **gamma.2a**; bytes 4, 5, 12, 13, and so forth to file **gamma.3a**; and bytes 6, 7, 14, 15, and so forth to file **gamma.4a**. Figure C-2 shows this byte assignment. (Section C.7.2 explains this calculation in more detail.)

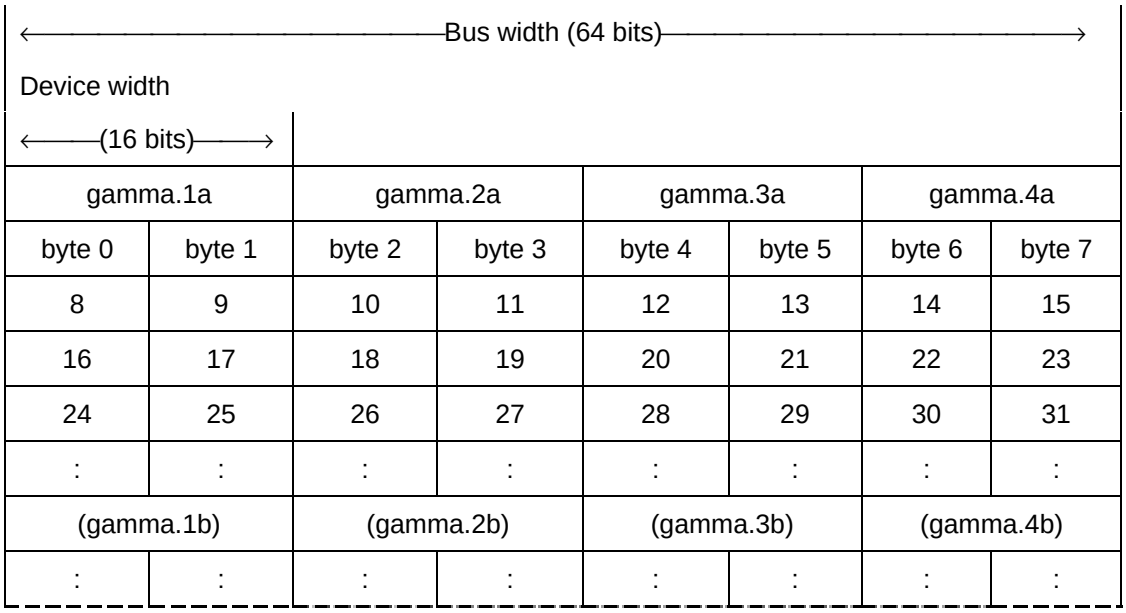

Figure C-2. Gamma Example Byte Assignment

For another example, suppose that your **`msrec`** command includes the expressions **`-o delta`**, **`-dw 32`**, and **`-dm 64`**. The S-record generator assigns output bytes 0, 1, 2, 3, 8, 9, 10, 11, and so forth to file **`delta.1a`**; and bytes 4, 5, 6, 7, 12, 13, 14, 15, and so forth to file **`delta.2a`**. Figure C-3 shows this byte assignment. (Section C.7.3 explains this calculation in more detail.)

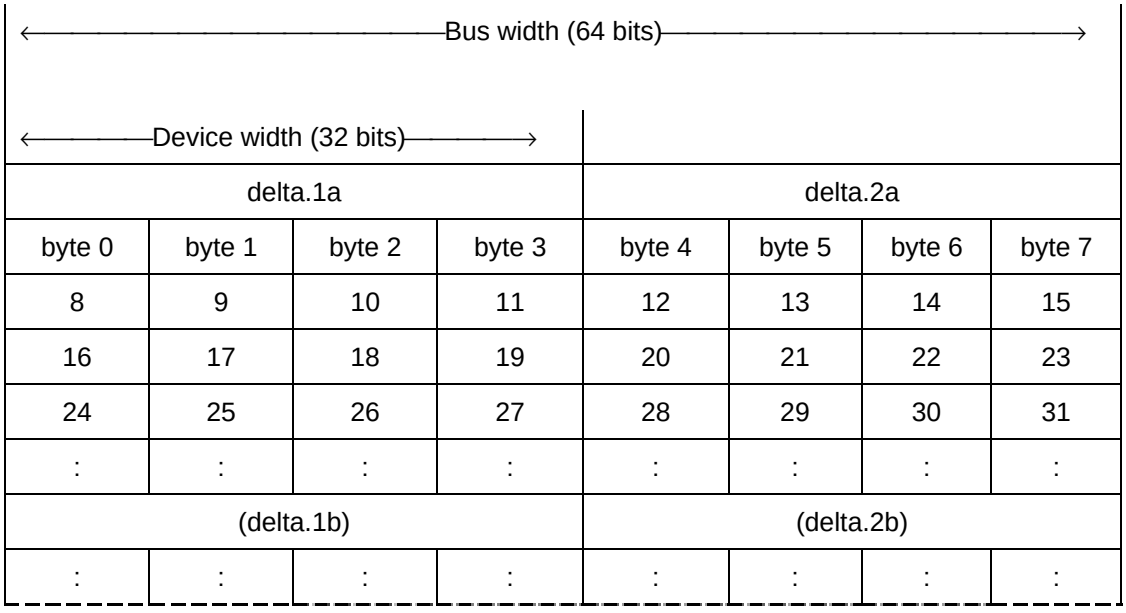

Figure C-3. Delta Example Byte Assignment

A third example shows the effect of the **-dd** (device depth) option. Suppose that your **msrec** command includes the expressions **-o epsilon**, **-dw 16**, **-dd 3**, and **-ds 512K**. (The **-dm** option defaults to the value 32.) The S-record generator generates the six output files that Figure C-4 shows. The **-dd** option value 3 limits the depth of output files, so that there cannot be files **epsilon.1d**, **epsilon.2d**, **epsilon.1e**, **epsilon.2e**, or so forth.

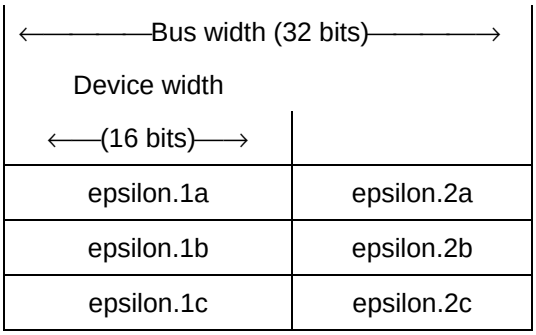

Figure C-4. Epsilon Example Output Files

C.7 OUTPUT FILE CALCULATIONS

Section C.6 showed how the S-record generator outputs a matrix of files. Each row consists of 1, 2, 4, or 8 files: the quotient of the bus width (**dm**) divided by the device width (**dw**). The size of the input file and the device size (**ds**) determine the number of rows (the depth), although the device depth (**-dd**) option can limit that depth.

The formula for the maximum number of rows is:

```
input bytes / ((dm / dw) ds)
```

in which the symbol **/** indicates rounded-up integer division.

C.7.1 Beta Example

The beta example had option values **-dm 32**, **-dw 8**, **-ds 16K**, and input file size 300 kilobytes. The maximum number of rows is:

```
300K / ((32 / 8) 16K)
= 300K / (4 × 16K)
= 300K / 64K
= 4.68, rounded up to 5
```

Accordingly, the maximum number of rows for this example is 5, which Figure C-1 shows.

But suppose that the target system had only four rows of four 16K × 8-bit memory devices: you would have included the **-dd 4** command option. An input file of 300 kilobytes would not have fit into four rows of output files. The S-record generation would have failed; the S-record generator would have issued the message, *Output file depth cannot exceed the range A-D.*

C.7.2 Gamma Example

The gamma example showed byte assignments for output files. To understand byte assignments, assume that one row of memory devices holds *words*, which consist of *subwords*. Each *word* is as wide as the memory bus (**dm**); each *subword* is as wide as the device width (**dw**). Each *subword* can consist of 1, 2, or 4 bytes. The S-record generator assigns byte *n* to an output-file column, according to this formula:

```
1 + ((n div bytes per subword) % (dm / dw))
```

in which the symbol **div** indicates truncating integer division, the symbol **%** indicates modulus (remainder) division, and the symbol **/** indicates rounded-up integer division.

The gamma example had option values **-dm 64**, and **-dw 16**. The 16-bit **dw** value means that each subword consists of 2 bytes. To find the column position of byte 14, we use the formula above:

```
1 + ((14 div 2) % (64 / 16)
= 1 + (7 % 4)
= 1 + 3
= 4
```

And Figure C-2 shows that the S-record generator assigned byte 14 to file **gamma.4a**.

C.7.3 Delta Example

The delta example also showed byte assignments for output files. This example had option values **-dm 64**, and **-dw 32**. The 32-bit **dw** value means that each subword consists of 4 bytes. To find the column position of byte 14, we use the formula:

```
1 + ((n div bytes per subword) % (dm / dw))
```

in which the symbol **div** indicates truncating integer division, the symbol **%** indicates modulus (remainder) division, and the symbol **/** indicates rounded-up integer division.

```
1 + ((14 div 4) % (64 / 32)
= 1 + (3 % 2)
= 1 + 1
= 2
```

And Figure C-3 shows that the S-record generator assigned byte 14 to file **delta.2a**.

C.8 ZETA EXAMPLE

For this final example, assume that your **msrec** command includes the expressions **+seg .data +seg .text -i -o zeta -dw 8 -ds 0x50**. Also assume that your input file consists of a 320-byte (0x140-byte) **.data** segment linked at 0x0 and a 376-byte (0x178-byte) **.text** segment linked at 0x200. These input bytes cover the address range 0x0 — 0x377. Figure C-5 represents this input file.

Address	Contents
0	.data segment
1	
2	
:	
13E	
13F	
140	(undefined)
141	
:	
1FE	
1FF	
200	.text segment
210	
:	
376	
377	

Figure C-5. Zeta Example Input File

Figure C-6 represents the S-record generator output files. Each row of output files consists of **dm / dw** files: as the default **-dm** value is 32, this number is 32 / 8 = 4 files. Each row covers 4 * 0x50 = 320 (0x140) bytes.

zeta.1a		zeta.2a		zeta.3a		zeta.4a		
0	.data 0	0	.data 1	0	.data 2	0	.data 3	
1	4	1	5	1	6	1	7	
2	8	2	9	2	10	2	11	.data
:	:	:	:	:	:	:	:	segment
								0x140
7D	134	7D	135	7D	136	7D	137	bytes
7E	138	7E	139	7E	13A	7E	13B	
7F	.data 13C	7F	.data 13D	7F	.data 13E	7F	.data 13F	

zeta.1b		zeta.2b		zeta.3b		zeta.4b		
0		0		0		0		
1		1		1		1		
:		:		:		:		(undefined)
2F		2F		2F		2F		
30	.text 200	30	.text 201	30	.text 202	30	.text 203	
31	204	31	205	31	206	31	207	.text
:	:	:	:	:	:	:	:	segment
								0x80
7E	278	7E	279	7E	27A	7E	27B	bytes
7F	.text 27C	7F	.text 27D	7F	.text 27E	7F	.text 27F	

zeta.1c		zeta.2c		zeta.3c		zeta.4c		
0	.text 280	0	.text 281	0	.text 282	0	.text 283	
1	284	1	285	1	286	1	287	.text
:	:	:	:	:	:	:	:	segment
3C	370	3C	371	3C	372	3C	373	0xF8
3D	.text 374	3D	.text 375	3D	.text 376	3D	.text 377	bytes
3E		3E		3E		3E		
:		:		:		:		(undefined)
7F		7F		7F		7F		

Figure C-6. Zeta Example Output File

The first row (files **zeta.1a**, **zeta.2a**, **zeta.3a**, and **zeta.4a**) covers the address range 0x0 — 0x13F. Each file contains 320 / 4 = 80 (0x50) bytes. The **.data** segment fits perfectly into this row of files.

The second row (files **zeta.1b**, **zeta.2b**, **zeta.3b**, and **zeta.4b**) covers the address range 0x140 — 0x27F. The starting address for the **.text** segment, 0x200, is in the middle of this range. Accordingly, only part of the second row's range holds **.text** bytes: 0x200 — 0x27F, which is 0x27F - 0x200 + 1 = 128 (0x80) bytes long. Each file in this row holds 128 / 4 = 32 (0x20) bytes. (The S-record generator does not generate S-records for the range 0x140 — 0x1FF, as no input file contents were defined for the range.)

The rest of the **.text** segment fits into the third row (files **zeta.1c**, **zeta.2c**, **zeta.3c**, and **zeta.4c**). Each file has 0x3E **.text** bytes.

INDEX