



ELF Linker/Locator and Archiver User's Guide

92G6922

Third Edition (August 1996)

This edition of the *IBM ELF Linker/Locator and Archiver User's Guide* applies to the IBM ELF Linker version 4.1h and to subsequent versions until otherwise indicated in new versions or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

IBM does not warrant that the products in this publication, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying product descriptions are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department YM3A
P.O. Box 12195
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Copyright © 1995-96, MetaWare® Incorporated, Santa Cruz, CA

©Copyright International Business Machines Corporation 1995-96. All rights reserved.

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Patents and Trademarks

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

IBM and the IBM logo are registered trademarks of the International Business Machines Corporation.

All MetaWare product names are trademarks or registered trademarks of MetaWare Incorporated.

Other terms which are trademarks are the property of their respective owners.

Contents

About This Book	vii
Notational and Typographic Conventions.....	vii
File-Naming Conventions	viii
Where to Go for More Information	viii
 1 Introduction	 1
1.1 MetaWare ELF Linker/Locator.....	1
1.2 MetaWare Archiver	2
1.3 Utilities	2
1.3.1 ELF-to-Hex Conversion Utility.....	2
1.3.2 The elfdump Binary Dump Utility.....	2
 2 Using the ELF Linker/Locator.....	 5
2.1 Invoking the Linker	5
2.1.1 Invoking the Linker from the High C/C++ Driver	5
2.1.2 Invoking the Linker from the System Command Line.....	7
2.2 Linker Options Summary	8
2.2.1 Options Reserved for Developing Embedded Applications.....	11
2.2.2 Option Name Conflicts.....	12
2.3 Linker Commands Summary.....	13
2.4 Linker Invocation Examples.....	15
 3 Linker Command-Line Options	 17
3.1 Specifying Command-Line Arguments in a File	17
3.2 Linker Options Reference.....	18
 4 Linker Command Files	 37
4.1 Overview of Linker Command Files.....	37
4.1.1 Command-File Conventions.....	37
4.1.2 Classification of Sections in Executable Files.....	38
4.1.3 Using Wildcards in File-Name References	39
4.2 Linker Commands Reference	40
4.2.1 More About the SECTIONS Command.....	52

5	AT&T-Style Map-File Directives	59
5.1	Overview of Input Map Files.....	59
5.2	Input Map File Directives	60
5.3	Sample Input Map File.....	65
6	Additional Linker Topics	67
6.1	Special Linker-Defined Symbols.....	67
6.2	Linking Incrementally.....	68
6.3	Generating Relocation Fix-Ups Versus Local Copies	69
6.4	Rewiring Shared Function Calls Through the PLT	70
6.5	Initializing RAM from a Program in ROM	71
6.6	How the Linker Processes Archive Libraries	72
6.6.1	Undefined Reference Errors in Libraries	73
6.6.2	Multiple Definition Errors in Libraries	73
6.7	Dynamic Versus Static Linking	74
7	Linker Error Messages.....	75
7.1	Linker Error Message Severity Levels.....	75
7.1.1	Warnings	75
7.1.2	Errors.....	75
7.1.3	Terminal Errors	76
7.2	Linker Error Messages in the Map Listing File	76
8	Utilities	77
8.1	Using the MetaWare Archiver	77
8.1.1	Invoking the Archiver from the Command Line.....	78
8.1.2	Archiver Options.....	79
8.1.3	Specifying Archiver Command-Line Arguments in a File ..	80
8.1.4	Archiver Invocation Examples.....	80
8.2	Using the elf2hex Conversion Utility	80
8.2.1	Invoking the ELF-to-Hex Converter	81
8.2.2	Conversion Option Reference	82
8.2.3	elf2hex Invocation Examples	85
8.3	Using the elfdump Binary Dump Utility	86
8.3.1	Invoking elfdump	87
8.3.2	elfdump Command-Line Options	88

A	Working with PROMs and Hex Files	89
A.1	Hex File Overview	89
A.2	PROM Device Model	90
A.2.1	PROM Device Characteristics	90
A.2.2	A Single PROM Bank	90
A.2.3	Multiple PROM Banks	91
A.3	Hex Output-File Naming Conventions	92
A.3.1	Single Hex File (No Size or Number of Banks Specified)	92
A.3.2	Multiple Hex Files	93
A.4	Hex File Formats	94
A.4.1	Motorola Hex Record Format	94
A.4.2	Extended Tektronix Hex Record Format	96
A.4.3	Mentor QUICKSIM Modelfile Format	98
B	OS/2 Module-Definition Files	101
B.1	Module-Definition Files	101
B.2	Module Statements	101
B.2.1	Supported OS/2 Keywords	102
B.2.2	Legacy Support: OS/2 2.x Keywords	103
B.3	Module Statement Reference	103
B.4	Sample Module-Definition Files	118
B.4.1	Module-Definition File for an Application	118
B.4.2	Module-Definition File for a DLL	119
	Index	121

Tables

Table 2.1	Linker Options Summary.....	8
Table 2.2	Linker Options for Developing Embedded Applications	11
Table 2.3	Conflicting (Duplicate) Linker/Compiler Options	13
Table 2.4	Linker AT&T-Style Map-File Directives Summary.....	14
Table 2.5	Linker Commands Summary	14
Table 3.1	Default Hex Conversion Characteristics.....	32
Table 8.1	Archiver Command-Line Options	79
Table 8.2	ELF-to-Hex Options Summary	82
Table 8.3	elfdump Command-Line Options.....	88
Table A.1	Start and EOF Characters in Motorola S3 Record Format.....	95
Table A.2	Header Field Components in Extended Tektronix Hex Format Data Record	98
Table B.1	Module-Definition Keywords.....	102
Table B.2	OS/2 2.x Legacy Support Keywords.....	103
Table B.3	Optional Attributes for the CODE Statement	104
Table B.4	Optional Attributes for the DATA Statement	106
Table B.5	Optional Attributes for the SEGMENTS Statement.....	116

Figures

Figure 8.1	Example of File Partitioning for ROM	85
Figure 8.2	Sample Output of Extended Tektronix Hex Format	86
Figure 8.3	Structure Dumps of ELF Object and Executable Files	87
Figure A.1	A Single Bank of PROM Devices	91
Figure A.2	Two Banks of PROM Devices	92
Figure A.3	Data Record in Motorola S3 Record Format	95
Figure A.4	Data Record in Extended Tektronix Hex Format	97

About This Book

This **ELF Linker/Locator and Archiver User's Guide** describes how to use the MetaWare linker and archiver for object modules and libraries conforming to the Executable and Linking Format (ELF). This user's guide also describes how to use the ELF-to-hex conversion utility and other utilities.

Notational and Typographic Conventions

This manual uses several notational and typographic conventions to visually differentiate text.

Convention	Meaning
Courier	Indicates program text, input, output, file names
Bold Courier	Indicates commands, keywords, command-line options, literal options
<i>Italic Courier</i>	Indicates formal parameters to be replaced by names or values you specify; also indicates input on the command line
<i>Italics</i>	Indicates special terms and definitions
{ x y z }	Indicates that one (and only one) of the options separated by vertical bars and enclosed in curly braces can be selected
[x y z]	Indicates that none, one, some, or all of the options separated by vertical bars and enclosed in brackets can be selected
...	Indicates multiple entries of the same type
	Separates choices within brackets

File-Naming Conventions

This manual uses the following file-naming conventions:

File Type	File-Name Extension
C source file	.c
C++ source file	.cpp
Object file	.o
Archive library	.a .lib (OS/2 target only)
Shared object library (or DLL)	.so .dll (OS/2 target only)

*Embedded
PowerPC
targets only*

Note:

For the High C/C++ Embedded PowerPC Toolset, all MetaWare run-time libraries are static libraries whose names are of the form `libname.a`.

Where to Go for More Information

See the readme file! The file `readme`, in your main High C/C++ directory, identifies last-minute changes and describes any special files on the distribution. Read the `readme` file before using the ELF linker/locator, archiver, or other utilities.

The following MetaWare manuals provide information about the High C/C++ compiler and MetaWare's implementation of the C and C++ languages and libraries:

- The **High C/C++ Programmer's Guide** describes how to use the High C/C++ compiler and provides system-specific information.
- The **High C/C++ Language Reference** describes the syntax and semantics of the C and C++ language and the MetaWare High C/C++ extensions to the C and C++ languages.
- The **High C Library Reference** provides descriptions of C functions.

- The **High C++ I/O Streams Library Reference** gives information about C++ input and output streams.

For a complete listing of all documents associated with this product, as well supported standards and specifications, see the *About This Book* chapter of the **High C/C++ Programmer's Guide**.

1

Introduction

This chapter presents an overview of the tools documented in this manual, including the ELF linker/locator and archiver, the ELF-to-hex conversion utility, and other utilities.

1.1 MetaWare ELF Linker/Locator

The ELF linker/locator reads and writes files in relocatable or absolute Executable and Linking Format (ELF). Specifically, it supports ELF as documented in the **System V Application Binary Interface**, Third Edition, from UNIX System Laboratories (ISBN 0-13-100439-5).

The linker is language-independent. You can use it to link any ELF-conformant object file, including files produced by different compilers or assemblers, or files from libraries built by a UNIX SVR4-compatible archiver. You can link files dynamically or statically. You can also link files incrementally.

For information about using the ELF linker/locator, see the following chapters and appendixes:

- Chapter 2: *Using the ELF Linker/Locator*
- Chapter 3: *Linker Command-Line Options*
- Chapter 4: *Linker Command Files*
- Chapter 5: *AT&T-Style Map-File Directives*
- Chapter 6: *Additional Linker Topics*
- Chapter 7: *Linker Error Messages*
- Chapter 8: *Utilities*
- Appendix A: *Working with PROMs and Hex Files*
- Appendix B: *OS/2 Module-Definition Files*

1.2 MetaWare Archiver

The MetaWare archiver groups independently developed object files into an *archive library* (a collection of object files, or *members*, residing in a single file) to be accessed by the linker. When the linker reads an archive, it extracts only the object files necessary to resolve external references.

For information about using the archiver, see §8.1: *Using the MetaWare Archiver*.

1.3 Utilities

The ELF linker/locator and archiver come with additional utilities, which are documented in Chapter 8: *Utilities*.

1.3.1 ELF-to-Hex Conversion Utility

The ELF-to-hex conversion utility converts binary files to an ASCII hexadecimal format. (This feature is also available through linker option **-x**.) The converter produces one or more ASCII hex files from an executable ELF file. The input object file is an executable file produced by the linker.

For information about using the ELF-to-hex converter, see the following:

- §8.2: *Using the elf2hex Conversion Utility*
- Appendix A: *Working with PROMs and Hex Files*

1.3.2 The **elfdump** Binary Dump Utility

The **elfdump** utility produces a structure dump or hexadecimal dump of some or all of the ELF object files and shared libraries (DLLs) compiled with High C/C++.

You can also use **elfdump** to dump executable files.

For information about using **elfdump**, see the following:

- §8.3: *Using the elfdump Binary Dump Utility*

Using the ELF Linker/Locator

This chapter, which describes how to invoke the ELF linker/locator, contains the following sections:

- §2.1: *Invoking the Linker*
- §2.2: *Linker Options Summary*
- §2.3: *Linker Commands Summary*
- §2.4: *Linker Invocation Examples*

2.1 Invoking the Linker

You can invoke the MetaWare ELF linker/locator automatically, from the High C/C++ driver command, or manually, from the system command line. (Invoking the linker from the driver is the recommended method.)

The ELF linker/locator supports the use of argument files, AT&T-style command files (input map files), and a second, more robust form of command files. These different types of files are documented in the following:

- §3.1: *Specifying Command-Line Arguments in a File*
- Chapter 5: *AT&T-Style Map-File Directives*
- Chapter 4: *Linker Command Files*

The following sections explain the two linker invocation methods. See §2.4: *Linker Invocation Examples* for more information.

2.1.1 Invoking the Linker from the High C/C++ Driver

Note: We recommend you use the driver command (**hc** or **hcsuffix**) to compile and link High C/C++ programs. See the **High C/C++ Programmer's Guide** for the exact name of

the driver for your target processor. In this manual, we use “**hc**” to generically represent the driver command.

The **hc** driver command invokes the compiler — and then automatically invokes the linker — to compile and link High C/C++ programs. For example, this command:

```
hc file1.c file2.c
```

creates object files `file1.o` and `file2.o` (corresponding to the source files), then links them together to create the executable output file.

Note: By default, if you compile a single source file, the driver removes the object file after the link. To keep the object file, specify option **-Hkeep** on the driver command line.

Compiling without invoking the linker With compiler option **-c**, you can direct the **hc** driver to compile the source files and *not* invoke the linker. For example, this command:

```
hc -c file1.c file2.c
```

creates object files `file1.o` and `file2.o` (corresponding to the source files), then stops.

Passing files directly to the linker The driver assumes that any file specification on the **hc** command line that does not have a recognizable source-file extension is a linker input file (either an object file or a library). The driver passes such files directly to the linker.

See *File-Naming Conventions* in the *About This Book* chapter of this manual for a list of default source-file extensions; see the **High C/C++ Programmer's Guide** for a more comprehensive list.

Linking in run-time library files The linker also links the High C/C++ run-time libraries with the object files to resolve any external references. To list the libraries being linked in, add option **-v** to the **hc** command line.

Naming the output file By default, the output file is named `a.out`. You can use **hc** option **-o** to change the name of the output file. For example, this command:

```
hc -o test.out file1.c file2.c
```

creates object files `file1.o` and `file2.o`, then links them together to create the executable output file `test.out`.

Conflicting linker and compiler option names Most linker options you specify on the **hc** command line are passed to the linker. However, in cases where a linker option duplicates a compiler option, the driver assumes the option is a compiler option. To force such options to be passed to the linker, use the compiler option **-Hldopt=opt**. For example:

```
hc file1.c file2.c -Hldopt=-v
```

This invocation passes option **-v** (also a compiler option) to the linker. See Table 2.3, *Conflicting (Duplicate) Linker/Compiler Options*, for information about duplicate options. See the **High C/C++ Programmer's Guide** for information about invoking and controlling the compile-and-link process.

2.1.2 Invoking the Linker from the System Command Line

Note: The ELF linker/locator is named **ld** for native applications and **ldsuffix** for cross-platform applications, where *suffix* represents the target processor.

See the **Installation Guide** for the exact name of the linker for your target. In this manual, we use “**ld**” to generically represent all forms of the ELF linker/locator command.

Assuming the ELF linker/locator is in your execution path, you can invoke it manually with the following command:

```
ld [options] file [file ...] [@arg_file ...]
```

Note: If the linker is not in your execution path, you must use the full pathname of **ld** to invoke the linker.

The linker command-line arguments have the following meanings:

<i>options</i>	Linker options, separated by spaces. See §2.2: <i>Linker Options Summary</i> for a complete list of linker options.
<i>file</i>	Name of object file (relocatable input file) or archive file, separated by whitespace.
<i>arg_file</i>	Name of an argument file. See §3.1: <i>Specifying Command-Line Arguments in a File</i> for more information.

You can specify options in any order. You can intersperse options with object files and argument files on the command line. You can also place options before or after files. However, some options affect the behavior of subsequent options; for example, option **-B static** affects how the linker interprets option **-l name**.

Command files See Chapter 5: *AT&T-Style Map-File Directives* and Chapter 4: *Linker Command Files* for information about the two supported types of command files.

Order of archive files is significant The order of the archive libraries on the command line is important in resolving external symbol references, because the linker reads the files in the order you list them. The linker searches an archive library only to resolve those unresolved symbols that are pending when it encounters that archive. The linker searches a library repeatedly, as required, to resolve intra-library references.

You can name libraries on the command line more than once, if necessary, to resolve mutual references. See §6.6: *How the Linker Processes Archive Libraries* for more information on this topic.

2.2 Linker Options Summary

Table 2.1 summarizes the general ELF linker/locator options. See Table 2.1, *Linker Options Summary* for information about a specialized subset of linker options. All linker options are described in more detail in §3.2: *Linker Options Reference*. See §2.1: *Invoking the Linker* for information about how to use linker options.

Table 2.1 *Linker Options Summary*

Option	Meaning
-A <i>cmd_file</i>	Specifies a linker command file
-b	Equivalent to -B noplt -B nocopies
-B all_archive	Extracts all members from archive libraries
-B dynamic	Searches for shared libraries before static

Table 2.1 Linker Options Summary (Continued)

<i>Embedded application development</i>	Option	Meaning
	-B help	Displays information about -B options reserved for embedded development (see Table 2.1, <i>Linker Options Summary</i>)
	-B lstrip	Strips local symbols from symbol table
	-B nocopies	Does not make local copies of shared variables
	-B noplt	Does not implicitly map symbols into the PLT
	-B static	Searches only for static libraries; does not accept shared libraries (DLLs)
	-B symbolic	Binds intra-module global-symbol references to symbol definitions within the link
	-C crossref	Displays a cross-reference listing of global symbols
	-C globals	Displays a sorted list of global symbols with their mappings
	-C page=<i>n</i>	Specifies the number of lines per page in the displayed listing
	-C sections	Displays a listing of section mappings
	-C symbols	Displays a listing of all symbols in the generated output symbol table
	-C unmangle	Displays all C++ names in unmangled form in generated listing
	-d n	Specifies static linking
	-d y	Specifies dynamic linking
	-e <i>entry_name</i>	Specifies program entry point
	-G	Generates a dynamic or shared library
	-h <i>name</i>	Specifies name used to refer to the generated shared library (DLL)
	-H	Displays linker command-line syntax help screen
	-i <i>deffile</i>	Reads OS/2-style <i>deffile</i> for linker definitions

Table 2.1 Linker Options Summary (Continued)

Option	Meaning
-I <i>name</i>	Writes <i>name</i> into the program header as the interpreter's pathname
-J <i>file</i>	Limits the dynamic symbol table to the listed imported names
-l <i>name</i>	Includes a named library from a path search specified with option -L
-L <i>paths</i>	Specifies one or more search paths for named libraries
-m	Prints a memory map listing file to standard output; same as -C sections -C globals
-M <i>imap_file</i>	Reads an input map file (AT&T-style command file)
-o <i>out_file</i>	Specifies the name of the output file
-q	Does not display copyright message
-Q n	Does not place linker version number information in the generated output file
-Q y	Places linker version number information in the generated output file
-r	Generates relocatable object file for incremental linking
-s	Strips the symbol table and debugging information from the output object file
-t	Suppresses warnings for multiply defined common symbols of unequal sizes
-u <i>ext_name</i>	Creates an undefined symbol
-v	Displays linker version number prior to linking
-w	Suppresses all warning messages
-x help	Displays information about -x options for hex record generation (see Table 2.1, <i>Linker Options Summary</i>)
-YP , <i>paths</i>	Changes the default library search directories

*Embedded
application
development*

Table 2.1 Linker Options Summary (Continued)

Option	Meaning
-z defs	Does not allow undefined symbols
-z nodefs	Allows undefined symbols
-z sections	Diagnoses unreferenced files and input sections
-z text	Does not allow output relocations against read-only sections

2.2.1 Options Reserved for Developing Embedded Applications

Table 2.1 summarizes certain linker options that are reserved for developing embedded applications. These special options are documented in detail in §3.2: *Linker Options Reference*.

Caution: In general, these special options should not be used to develop non-embedded applications — that is, applications that run on an existing operating system environment (such as an application that runs on Solaris).

Table 2.2 Linker Options for Developing Embedded Applications

Option	Meaning
-B base= <i>0xaddress</i>	Specifies origin address (in hexadecimal); same as -B start_addr
-B hardalign	Forces each output segment to align on a page boundary
-B help	Displays information about special -B options used only for developing embedded applications
-B kernel	Generates a kernel-space-loadable output file
-B movable	Makes the executable file dynamically movable
-B noallocdyn	Does not map dynamic tables in virtual memory
-B nodemandload	Ignores boundary issues when mapping sections

Table 2.2 Linker Options for Developing Embedded Applications

Option	Meaning
-B noheader	Does not include ELF header in loadable segments
-B pagesize=size	Specifies page size of target processor
-B rogot	Places the <code>.got</code> section in a read-only segment
-B rogotplt	Places the <code>.got</code> and <code>.plt</code> sections in a read-only segment
-B roplt	Places the <code>.plt</code> section in a read-only segment
-B start_addr=0xaddress	Specifies origin address (in hexadecimal); same as -B base
-B wired	Generates non-swappable output file
-x	Generates a hex file; uses default values for all conversion characteristics
-xc [=] [t] [l] [d]	Generates a hex file; specifies section types to be converted (text, lit, and/or data)
-x help	Displays information about -x options for hex record generation
-xi=banks	Generates a hex file; specifies number of interleaved banks
-x{m q t}	Generates a hex file in one of these formats: <ul style="list-style-type: none"> • Motorola S3 Record (-xm) • Mentor QUICKSIM Modelfile (-xq) • Extended Tektronix Hex Record (-xt)
-xn=word_size	Generates a hex file; specifies width of memory device, in bits
-xo=name	Generates a hex file; specifies file's name
-xs=size	Generates a hex file; specifies memory device size and width using <i>EKxW</i> or <i>EMxW</i> format

2.2.2 Option Name Conflicts

A few linker option names conflict with the names of compiler options. Table 2.3 summarizes these conflicting linker/compiler options. When you invoke the `hc` driver to compile, assemble, and link a program, use compiler option `-Hldopt` to pass a duplicate option to the linker (see §2.1.1: *Invoking the Linker from the High C/C++ Driver* for more information).

Table 2.3 *Conflicting (Duplicate) Linker/Compiler Options*

Option	Meaning As a Linker Option	Meaning As a Compiler Option
<code>-h name</code>	Uses <i>name</i> to refer to the generated shared library (DLL)	Searches Help files for information about <i>name</i>
<code>-H</code>	Displays linker command-line syntax Help screen	Displays names of files included during compilation
<code>-M name</code>	Reads <i>name</i> , an input map file (AT&T-style command file)	Specifies <i>name</i> as the memory model
<code>-v</code>	Displays linker version number prior to linking	Displays options and toggles being passed to the linker

2.3 Linker Commands Summary

The ELF linker/locator supports two kinds of command files:

- AT&T-style input map files, implemented with linker option `-M input_map_file` and documented in Chapter 5: *AT&T-Style Map-File Directives*
- a more robust command-file format inherited from other linker implementations, implemented with option `-A command_file` and documented in Chapter 4: *Linker Command Files*

Table 2.4 summarizes the AT&T-style input map-file directives implemented with option `-M`.

Table 2.5 summarizes the linker commands implemented with option `-A`.

Table 2.4 Linker AT&T-Style Map-File Directives Summary

Type of Directive	Purpose
Output section declaration	Creates a new output section in the output file, or changes the attribute values of an existing output section
Mapping directive	Tells the linker how to map input sections to output sections
Size-symbol declaration	Defines a new global/absolute symbol that represents the size, in bytes, of the specified output section
Output-section order specification	Specifies an ordering of output sections in the generated executable

Table 2.5 Linker Commands Summary

Command	Meaning
ARGS	Specifies linker command-line arguments in a command file
DEMANDLOAD	Specifies an executable that can be demand-loaded from disk
INITDATA	Initializes control sections at run time
LOAD	Reads input files
NODEMANDLOAD	Specifies an executable that cannot be demand-loaded from disk
PUBLIC	Reassigns the value of an external symbol
RESADD	Reserves memory location by address
RESNUM	Reserves memory location by size
SECTIONS	Specifies order of output sections
START	Specifies program entry point

2.4 Linker Invocation Examples

This section provides several examples of invoking the linker.

Example 1 In this example, the linker reads command file `cmd_file` and generates an executable file called `out_file`:

```
ld -A cmd_file -o out_file -m > mmap_file
```

Option `-m` directs the linker to generate a memory map listing file, which is redirected to the file `mmap_file`.

Example 2 In this example, the linker first reads from file `cmd_file`. Then it reads object files `mod1.o` and `mod2.o`. Finally, it writes the output file to `a.out` (no output file was specified with option `-o`) and writes the memory map listing file to `mmap_file`.

```
ld -A cmd_file -m mod1.o mod2.o > mmap_file
```

Assuming file `cmd_file` contains this code:

```
LOAD test1.o
LOAD test2.o
LOAD lib1.a
```

the linker reads the following files:

```
test1.o
test2.o
lib1.a
mod1.o
mod2.o
```

Example 3 This is an example of *incremental linking*:

```
ld -o rel_file.o -r mod1.o mod2.o
```

The linker links object files `mod1.o` and `mod2.o` and produces relocatable output file `rel_file.o`. Option `-r` specifies that the output file should be relocatable rather than an executable, so it can be linked again.

Note: The relocatable output file `rel_file.o` is essentially a new, bigger object file. It becomes executable only after you link it *without* the `-r` option.

Linker Command-Line Options

This chapter, which describes command-line options available for the ELF linker/locator, contains the following sections:

§3.1: *Specifying Command-Line Arguments in a File*

§3.2: *Linker Options Reference*

3.1 Specifying Command-Line Arguments in a File

You can place frequently used linker command-line arguments in an *argument file*. An argument file is an ASCII text file in which you enter command-line options and arguments the same way you would enter them on the linker command line. You can use as many lines as necessary. (A newline is treated as whitespace.)

To specify to the linker that a file is an argument file, enter the name of the file on the command line preceded by an “at” symbol (@). For example:

```
ld @argument_file
```

When the linker encounters an argument on the command line that starts with @, it assumes it has encountered an argument file. The linker immediately opens the file and processes the arguments contained in it.

Argument files
versus
command files
versus
input map files

Do not confuse argument files with linker command files, which you specify with option **-A**, or linker input map files, which you specify with option **-M**.

- *Argument files* enable you to place command-line arguments in a file for convenience and as a workaround to the line-length limitation on DOS commands.
- *Linker command files* contain linker commands that enable you to manipulate the placement of sections within an output file and perform

other fine-tuning operations. See Chapter 4: *Linker Command Files* for more information.

- *Linker input map files* contain special mapping directives that enable you to declare segments and specify segment attributes, control input-section mapping, and perform other section-mapping operations. See Chapter 5: *AT&T-Style Map-File Directives* for more information.

3.2 Linker Options Reference

This section provides detailed information about each linker option. See §2.1: *Invoking the Linker* and §2.4: *Linker Invocation Examples* for information about when and how to use linker options.

-A *cmdfile* — Specify linker command file

Specifies the named file as a linker command file. The command file must contain one or more linker options or commands. When you use this option, you must supply a command-file name; the linker does not assume a default name.

See Chapter 4: *Linker Command Files* for a description of linker command files.

-b — Do not do special processing of shared symbols

Note: Option **-b** is equivalent to the combination **-B nocopies** **-B noplt**.

Dynamic linking only Option **-b** applies only when you are producing dynamically linked executables. It directs the linker to do no special processing for relocations that reference symbols in shared objects.

This option causes the linker to generate output code that is more efficient, but less shareable. The code is less shareable because the system's dynamic loader is forced to modify code that would otherwise be read-only, and therefore shareable with other processes running the same executable.

See §6.3: *Generating Relocation Fix-Ups Versus Local Copies* and §6.4: *Rewiring Shared Function Calls Through the PLT* for more information.

Note: Use the **-b** option only when you are creating an executable in dynamic-link mode.

When you do not use the **-b** option, the linker creates special position-independent relocations for references to functions defined in shared objects, and arranges for data objects defined in shared objects to be copied into the memory image of the executable at run time.

-B all_archive — Extract all members of an archive library

Causes the linker to extract all members of an archive library. This option is typically used to construct a shared library from an archive library. For example, this command:

```
ld liby.a -G -B all_archive -o liby.so
```

directs the linker to extract all members from archive library `liby.a` and create shared library `liby.so`.

-B base=0xaddress — Specify the origin address in hexadecimal

Embedded Specifies the starting address of the first output section. By default, the
development only starting address is based on a convention determined by the operating system. Same as **-B start_addr**.

-B dynamic — Search for shared library libname when processing -l name

Dynamic linking only Specifies that subsequent occurrences of option **-l name** direct the linker to search for shared libraries ahead of static libraries.

More specifically, the **-B dynamic** option directs the linker to interpret subsequent occurrences of the **-l name** option to include the shared library `libname.so`, `libname.dll`, or `name.dll`, in that order.

If it does not find the shared library, the linker resorts to including the static library `libname.a` or `name.lib`, in that order.

Options **-B dynamic** and **-B static** work as a binding mode toggle; you can alternate them any number of times on the command line.

Note: Option **-B dynamic** is useful only when you specify dynamic linking, because the linker does not accept shared libraries when it is linking statically.

For more information about specifying libraries, see the **-l name** option.

-B hardalign — Force each output segment to align on a page boundary

Embedded development only Directs the linker to align the start of each output segment on a page boundary.

Note: A *segment* is a grouping of control sections that are loaded as a unit.

-B help — Display information about a subset of -B options reserved for embedded development

Displays a summary screen of the following **-B** options, which are designed specifically for developing embedded applications:

-B base=0xaddress	-B pagesize=size
-B hardalign	-B rogot
-B kernel	-B rogotplt
-B movable	-B roplt
-B noallocdyn	-B start_addr=0xaddress
-B nodemandload	-B wired
-B noheader	

See Table 2.1, *Linker Options Summary* for a summary of these options.

Caution: In general, these special **-B** options should not be used to develop non-embedded applications — that is, applications that run on an existing operating system environment (such as an application that runs on Solaris).

-B kernel — Designate output file as “kernel-space-loadable”

Embedded development (OS/2 targets) only Directs the linker to designate that the output file is to be loaded into the kernel space. To designate the output file as such, the linker sets the `PF_K` (0x02000000) bit for each segment being loaded.

-B lstrip — Strip local symbols from symbol table

Directs the linker to strip all local symbols from the output file symbol table. The only symbols that remain are those that are global.

If you are linking incrementally (option **-r**), the linker retains those local symbols that are referenced from a relocation table.

-B movable — Make dynamic executable file movable

Embedded Directs the linker to render the dynamic executable file so that its origin
development only address can be altered at load time (in a manner similar to a shared library).

-B noallocdyn — Do not map dynamic tables in virtual memory

Embedded Directs the linker to not map dynamic tables (.dynamic, .rel, and so on)
development only in virtual memory. That is, the linker designates the associated sections as “not allocable”.

Option **-B noallocdyn** accomodates dynamic loaders that read relocation information directly from the ELF file instead of reading the information from virtual memory after the file is loaded.

-B nocopies — Do not make local copies of shared variables; insert relocation fix-ups

When linking an executable that references a variable within a shared library (DLL), the linker can take one of two courses of action:

- | | |
|--------------------------------------|---|
| <i>Relocation fix-up</i> | 1. It can insert a relocation fix-up for each reference to the symbol, in which case the dynamic loader modifies each instruction that references the symbol’s address. |
| <i>Local copy of shared variable</i> | 2. It can make a local copy of the variable and arrange for the dynamic linker to “rewire” the shared library so as to reference the local copy. |

By default, the linker makes a local copy of the variable and arranges for the shared library to reference that local copy. Option **-B nocopies** forces the linker to insert relocation fixups, instead.

Note: When you use option **-G** (generate a shared library), option **-B nocopies** is automatically turned On.

See §6.3: *Generating Relocation Fix-Ups Versus Local Copies* for a more detailed discussion of this topic.

-B nodemandload — Ignore boundary issues when mapping sections

Embedded Informs the linker that the output file does not need to be demand-page
development only loadable. Directs the linker to ignore page-boundary issues when mapping sections into the output file.

-B noheader — Do not include ELF header in loadable segments

Embedded Suppresses the inclusion of the ELF header in the loadable segments of a
development only dynamically linked executable or a shared library (DLL).

-B noplt — Do not implicitly map symbols into the PLT of the executable file

By default, the linker maps function entry-point symbols imported from shared libraries (DLLs) into the Procedure Linkage Table (PLT). All references to such functions within the executable are “rewired” to reference the PLT entry instead. Therefore, only the PLT entry needs modifying at load time.

If you specify **-B noplt**, the linker does not implicitly create PLT entries. Instead, the linker arranges for each individual call to be subject to a fixup at load time.

Option **-B noplt** slows down load time, but can increase execution speed slightly. (Because there is no overhead of jumping through the PLT, calls to functions within shared libraries will be slightly faster at run time.)

However, option **-B noplt** can render the executable less shareable with other processes that are running the same program. Use option **-B noplt** if it is not necessary for your code to be shareable across processes. See §6.4: *Rewiring Shared Function Calls Through the PLT* for a more detailed discussion of this topic.

-B pagesize=size — Specify page size of target processor in bytes

Embedded Specifies the memory page size to be used when mapping sections into the
development only ELF file. Sections are mapped so that:

```
file_offset % page_size == address % page_size.
```

The default page size is operating-system dependent.

-B rogot — Place the .got section in a read-only section

Embedded By default, the linker places the Global Offset Table (.got) section into
development only writable memory. This is usually required because the dynamic loader must perform relocations on the GOT. However, some implementations of the dynamic loader require that the .got section be mapped as a read-only section.

Option **-B rogot** directs the linker to place the .got section in a read-only section. (Presumably, the loader temporarily write-enables the page as it is relocated.)

-B rogotplt — Place the .got and .plt sections in read-only sections

Embedded Equivalent to the combination:
development only **-B rogot -B roplt**

-B roplt — Place the .plt section in a read-only section

Embedded By default, the linker places the Procedure Linkage Table (.plt) section
development only into writable memory. This is usually required because the operating system's dynamic loader must perform relocations on the PLT.

However, some implementations of the dynamic loader require that the .plt section be mapped as a read-only section.

Option **-B roplt** directs the linker to place the .plt section in a read-only section. (Presumably, the loader temporarily write-enables the page as it is relocated.)

x86 targets only **Note:** For x86 targets, the linker makes the .plt section read-only. Each PLT entry has a corresponding GOT table entry in which the linker inserts the jump address. (The GOT is writable.) Because these GOT table entries exist, there is no need for the dynamic loader to modify the PLT entries for the x86.

-B start_addr=0xaddress — Specify origin address

Embedded Specifies the origin address in hexadecimal. Equivalent to the **-B base**
development only option.

-B static — Search for static library libname when processing -l name

Dynamic linking only Specifies that subsequent occurrences of option **-l name** direct the linker to search only for static libraries.

More specifically, the **-B static** option directs the linker to interpret subsequent occurrences of the **-l name** option to include the static library `libname.a` or `name.lib`, in that order.

Options **-B static** and **-B dynamic** work as a toggle; you can alternate them any number of times on the command line.

See **-B dynamic** for more information.

Note: For information about specifying libraries, see the **-l name** option.

-B *symbolic* — Bind intra-module global-symbol references to symbol definitions within the link

Dynamic linking only Binds intra-library references to their symbol definitions within the shared library, if definitions are available.

When you do not specify **-B *symbolic***, references to global symbols within shared libraries (DLLs) can be overridden at load time by like symbols being exported in the executable or in preceding shared libraries.

-B *wired* — Generate non-swappable output file

Embedded development (OS/2 targets) only Directs the linker to render the output file non-swappable. OS/2 uses the PF_M bit (0x04000000) to render a segment non-swappable; that is, once that segment is loaded into memory, the virtual memory manager does not swap it out.

To designate the output file as non-swappable, the linker sets the PF_M bit for each output segment.

-C *listing_type* — Display listing of specified type

Displays a listing with attribute specified by *listing_type*. The argument *listing_type* can be any of the following predefined values:

crossref	Displays a cross-reference listing of global symbols.
globals	Displays a sorted list of global symbols with their mappings.
page=<i>n</i>	Specifies the number of lines per page in the displayed listing. Default <i>n</i> = 60. To suppress page ejects, set <i>n</i> = 0 (zero).
symbols	Displays a listing of all symbols in the generated output symbol table.
unmangle	Used with the other -C options to display C++ names in an unmangled form.

To specify multiple listing attributes, enter multiple **-C** options. For example,

```
ld -C crossref -C page=50 -C sections file.o
```

The **-C** listing is sent to standard output. To save the listing, redirect it to a file.

Note: Option **-m** is equivalent to the combination
-C sections -C globals.

-d{y|n} — Generate a dynamically or statically linked executable

Specifies dynamic or static linking. The default is **-d y**, dynamic linking.

- **-d y** specifies dynamic linking.
- **-d n** specifies static linking.

Note: If no shared library (DLL) references are required, the linker generates a statically linked executable, even if you do not specify **-d n**.

-e entry_name — Specify program entry point

Specifies the name of the point where program execution should start. This option is useful for loading stand-alone programs. The default entry-point name for the linker is `_start`. The name *entry_name* overrides the default entry-point name.

-G — Generate a shared library (DLL)

Dynamic linking only Produces a shared library (also known as a DLL).

-h name — Use name to refer to the generated shared library

Dynamic linking only Records *name* in the dynamic section of the object file. *name* replaces the file name for the object. The dynamic loader uses *name* at run time and records it in executable elements linked with the object.

This option applies only when you build a shared library (DLL).

-H — Display linker command-line syntax help screen

Displays on standard output a summary page of linker command-line syntax, options, and flags.

See also options **-B help** and **-x help**.

-i deffile — Read OS/2-style deffile for linker definitions

Embedded development (OS/2 targets) only Reads the module-definition file *deffile*, which contains definitions for the linker.

See Appendix B: *OS/2 Module-Definition Files* for more information.

-I *name* — Write *name* into the program header as the interpreter's pathname

Dynamic linking only When you specify dynamic linking, the name of the dynamic loader is used as the interpreter by default. Specify **-I *name*** to override the default and write *name* into the program header as the path name of the interpreter. When the program is executed, control is passed to the interpreter.

This option is not applicable if you specify static linking.

-J *file* — Export only the symbols listed in *file* when generating a shared library

Dynamic linking only Limits the dynamic symbol table to the names listed in *file* and any imported names. *file* is a file containing a subset of the list of names exported from the shared library being built. Each symbol name must appear on a line by itself without any preceding or embedded whitespace.

If you omit the **-J** option, the names of all global symbols appear in the table.

-l *name* — Search for library whose name contains *name*

This option provides an abbreviated way of referencing a library that the linker is to search for external references. The linker expands *name* to the full name of the library by adding a suffix and an optional prefix, as follows:

```
[lib]name.{a|dll|lib|so}
```

You can specify **-l *name*** on the command line any number of times. The placement of the **-l *name*** option is significant, because the linker searches for a library as soon as it encounters the library's name.

How the Linker Determines Which Library to Search For

When you dynamically link an application, the options **-B *static*** and **-B *dynamic*** serve as a *binding mode* toggle that dictates how the linker processes library-name option **-l *name***.

Searching for shared libraries (DLLs) first If **-B dynamic** is in effect and you specify **-l name**, the linker searches the library search paths until it finds one of the following libraries, in the order given here:

1. `libname.so`
2. `libname.dll`
3. `name.dll`
4. `libname.a`
5. `name.lib`

Searching for static-link libraries, only If **-B static** is in effect and you specify **-l name**, or if you are statically linking your application, the linker searches the library search paths until it finds one of the following static-link libraries, in this order:

1. `libname.a`
2. `name.lib`

When **-B static** is in effect, the linker does not search for or accept shared libraries (DLLs).

How the Linker Determines the Library Search Paths

Environment variable `LD_LIBRARY_PATH` consists of one or more lists of directories in which the linker will search for the libraries you specify with **-l name**.

One LD_LIBRARY_PATH directory list If you define `LD_LIBRARY_PATH` as follows:

```
LD_LIBRARY_PATH=dir_list
```

where `dir_list` is a list of directories separated by colons (:) on UNIX-based host systems or by semicolons (;) on DOS-based host systems, the linker searches for libraries in the following directory locations, in the order given:

1. All directories you specified with option **-L paths**.
2. The directories in `dir_list`.
3. The directories specified in environment variable `LIBPATH` or, if `LIBPATH` is not defined, the host system's built-in list of standard library directories.

Two LD_ If you define LD_LIBRARY_PATH as follows:
LIBRARY_PATH
directory lists LD_LIBRARY_PATH=*dir_list_1*|*dir_list_2*

where each *dir_list* item is a list of directories separated by colons (:) or semicolons (;) as previously described, the linker searches for libraries in the following directory locations, in the order given:

1. the directories in *dir_list_1*
2. all directories you specified with option **-L** *paths*
3. the directories in *dir_list_2*
4. the directories specified in environment variable LIBPATH or, if LIBPATH is not defined, the host system's built-in list of standard library directories

Note: LD_LIBRARY_PATH can have a maximum of two *dir_list* items. On UNIX-based host systems *only*, you can use a semicolon (;) instead of the vertical bar (|) to separate the two *dir_list* items in the LD_LIBRARY_PATH definition.

Standard library If LIBPATH is not defined, the linker searches the following default
directories host-system library directories:

UNIX VR4 and Solaris hosts: /usr/ccs/lib:/usr/lib:
 /usr/local/lib

SunOS host: /lib:/usr/lib:/usr/local/lib

DOS and OS/2 hosts: \lib

-L *paths* — Specify search path to resolve -l *name* specifications

Directs the linker to search in the specified directory for a library named with a subsequent option **-l**. The linker searches *paths* first, then the default directories.

The *paths* argument represents a list of directories separated by colons (:) on UNIX-based host systems or by semicolons (;) on DOS-based host systems.

See also the entries for options **-l *name*** and **-YP, *dirlist***.

-m — Write a memory map listing file to standard output

Generates a memory map listing file and sends the result to standard output. This map listing file explains how sections are allocated in virtual memory. It also contains symbol tables that show the absolute locations of each local and global symbol.

Option **-m** is equivalent to this combination:

-C globals -C sections

To save the contents of a memory map listing file, redirect the linker output to a file. For example:

```
ld -A command_file -m > map_listing_file
```

-M *imap_file* — Read an input map file (AT&T-style command file)

The linker automatically maps input sections from object files to output sections in executable files.

To change the default linker mapping when you specify a static link, use the **-M** input map-file option.

Note: The **-M** option is useful only when you specify static linking.

See Chapter 5: *AT&T-Style Map-File Directives* for more information about option **-M** and input map files.

For a complete discussion of input map files, see Appendix B: *Mapfile Option* in the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**.

-o *out_file* — Specify name of the output file

Specifies the name of the output file. The default output file name is `a.out`. The format of the output file is ELF (Executable and Linking Format).

-q — Do not display copyright message

Suppresses display of a MetaWare ELF linker/locator copyright message.

-Q {*y* | *n*} — Specify whether version information appears in output file

Option **-Q *y*** places ELF linker/locator version-number information in the generated output file; option **-Q *n*** suppresses placement of this version information.

Option **-Q y** is the default.

-r — Generate relocatable object file for incremental linking

Static linking option Causes the linker to generate relocatable output that can be used as input to subsequent links. The linker resolves all possible external references and adds relocation information for the next link.

Undefined external references to other files can still exist in the output object file. These undefined references are reported in the memory map listing file, if you specified one. For more information, see §6.2: *Linking Incrementally*.

Note: When you specify both option **-r** and option **-s** (to strip symbol tables), the linker strips only non-global symbols.

-s — Strip symbols and debugging information from the output file's symbol table

Causes the linker to strip the output object file's symbol table of all symbols and debugging information — except those symbols required for further relocating (for example, symbols required by option **-r**).

If you specify option **-r**, the linker strips only non-global symbols. By default, the linker writes a symbol table to the object file.

-t — Suppress warnings about multiply defined common symbols of unequal sizes

Directs the linker to not generate a warning for multiply defined symbols of unequal sizes. Such cases can arise when the linker is resolving common blocks to each other, or to exported definitions.

-u *ext_name* — Create undefined symbol *ext_name*

Directs the linker to enter *ext_name* as an unresolved symbol in the symbol table prior to reading input files. A typical use of this option is to force the extraction of an archive member that defines a symbol.

-v — Display linker version number prior to linking

Causes the linker to write its version number and a copyright banner to standard output, and then proceed with the linking process. The version number is always displayed in a memory map listing file, if you generate one.

Note: Displaying the version number is enabled by default, so there is no need to use option **-v**. Option **-v** is provided for backward compatibility only.

To suppress the display of the version number, use option **-q**.

-w — Suppress all warning messages

Suppresses all linker warning messages. See Chapter 7: *Linker Error Messages* for more information about linker warning and error messages.

-x — Generate a hex file; use default values for all conversion characteristics

Embedded development only Instructs the linker to generate one or more ASCII hex files suitable for programming PROMs (in addition to the ELF executable file).

Note: See Appendix A: *Working with PROMs and Hex Files* for detailed information about hex-file formats.

Appending flags or attributes to -x To specify characteristics of the hex file, you can append a flag or attribute to option **-x**, as listed here:

- xc** Specify section types to be converted
- xi** Specify number of interleaved banks
- x{m | q | t}** Generate a hex file in one of these formats:
 - Motorola S3 Record (m)
 - Mentor QUICKSIM Modelfile (q)
 - Extended Tektronix Hex Record (t)
- xn** Specify width of memory device, in bits
- xo** Specify hex file's name
- xs** Specify memory device size and width

These variations on option **-x** are documented in greater detail on the following pages.

Default conversion characteristics Used by itself without attributes or flags, option **-x** specifies default values for all hex conversion characteristics.

Table 3.1 Default Hex Conversion Characteristics

Conversion Characteristic	Default Value
Section types	Convert data, lit, and text sections
Number of interleaved banks	One bank (option -xi=1)
Width of memory device	Widest memory device (option -xn=32)
Name of hex file	Derived from executable input file
Size (depth and width) of memory devices	One absolute hex file
Hex format of output file	Motorola S3 Record format (option -xm)

Setting multiple hex conversion characteristics To set multiple hex conversion characteristics, repeat option **-x** for each additional attribute or flag. You can specify attributes and flags in any order. In cases where you can specify values for an attribute, the values do not have to be separated by commas or whitespace.

-xc [=] [t] [l] [d] — Generate a hex file; specify section types to be converted

Embedded development only Specifies the section types to be converted (dumped). The arguments to **-xc**, which indicate the section types to be converted, can be one or more of the following:

- d Specifies data sections
- l Specifies lit sections (read-only data sections)
- t Specifies text sections

Default = tld.

If you do not specify option **-xc** (but you do specify any other **-x** option), the linker converts all three section types.

-x help — Display information about -x options for hex record generation

Embedded development only Displays a summary screen of **-x** options that are designed specifically for generating hexadecimal records.

-xi=banks — Generate a hex file; specify number of interleaved banks

Embedded development only Specifies the number of interleaved banks. Value *banks* can be 1, 2, or 4.
Default *banks* = 1.

-x{m|q|t} — Generate a hex output file in specified hex record format

Embedded Directs the linker to generate a hex output file in one of the following
development only formats:

- **-xm**: Motorola S3 Hex Record format. See §A.4.1: *Motorola Hex Record Format* for more information.
- **-xq**: Mentor QUICKSIM Modelfile Hex Record format. See §A.4.3: *Mentor QUICKSIM Modelfile Format* for more information.
- **-xt**: Extended Tektronix Hex Record format. See §A.4.2: *Extended Tektronix Hex Record Format* for more information.

Default = **-xm**.

Note: Options **-xm**, **-xq**, and **-xt** are mutually exclusive.

-xn=word_size — Generate a hex file; specify width of memory device, in bits

Embedded Specifies width of a memory device in bits. Value *word_size* must be
development only equal to or greater than the width (W) specified by the value *size* of option **-xs**. Valid values for *word_size* are 8, 16, or 32.

Default *word_size* = 32.

-xo=name — Generate a hex file; specify file's name

Embedded Specifies the name of the generated hex file.
development only

If the link requires multiple hex files, each file is named *name.suf*, where *suf* is a three-character suffix that denotes the row, bank, and bit or nybble position of the device.

If you do not specify option **-xo** (but you do specify any other **-x** option), the name of the hex file(s) is derived from the name of the executable input file.

For a discussion of naming conventions for hex output files, see §A.3: *Hex Output-File Naming Conventions*.

-xs=size — Generate a hex file; specify memory device size and width

Embedded Identifies the size of the memory devices (in kilobytes or megabytes) and the
development only width (in bits), using the format *EKxW* or *EMxW*, where *E*, *K*, *M*, and *W* are defined as follows:

- E* Specifies the *depth* of the device for which the output is being prepared. These are valid values for *E*:
- 1, 2, 4, 8, 16, 32, 64, 128, 256, or 512 kilobytes (*EK*) or 1 megabyte (*EM*)
- W* Specifies the *width* of the device in bits. These are valid values for *W*:
- 4, 6, 8, 16, or 32 bits.
- If you do not specify *W*, the linker uses a default width of 8 bits.
- K* Specifies that the depth is represented in kilobytes.
- M* Specifies that the depth is represented in megabytes.

For example, a 32K device that is eight bits wide could be specified as 32Kx8 or as 32K, while a 256K device that is 16 bits wide would be specified as 256Kx16.

If you do not specify option **-xs** (but you do specify any other **-x** option), the linker output is one absolute hex file containing all data and code.

When you specify QUICKSIM hex format (option **-xq**), the linker ignores option **-xs**.

-YP,paths — Use *paths* as default search paths to resolve subsequent -l *name* specifications

Changes the default library search directories to *paths* by overriding environment variable `LIBPATH`. Directs the linker to search directories in *paths* to resolve subsequent **-l *name*** specifications.

- On UNIX hosts, use colons (:) to separate the items in *paths*.
- On DOS hosts, use semicolons (;).

You can also use environment variable `LD_LIBRARY_PATH` to specify library search directories.

See options **-L *path*** and **-l *name*** for more information.

-z defs — Do not allow undefined symbols

Forces a fatal error if any undefined symbols remain at the end of the link. This is the default.

-z nodefs — Allow undefined symbols

Allows undefined symbols.

You can use **-z nodefs** to build an executable in dynamic mode and link with a shared library (DLL) that has unresolved references in routines not used by that executable.

Note: For this link-with-unresolved-references to work, dynamic loading must be in “lazy binding” mode.

Caution: Use the **-z nodefs** option with caution or address faults might occur at run time.

-z sections — Diagnose unreferenced files and input sections

Directs the linker to diagnose files and input sections that are not referenced. Displays the diagnostic information on `stdout`.

You can use this diagnostic information to discover unreferenced object files inadvertently included in the heap. The diagnostic information can also provide hints about how a module might need to be divided if entire control sections are not referenced.

-z text — Do not allow output relocations against read-only sections

Dynamic linking only Forces a fatal error if any relocations against non-writable, allocatable sections remain at the end of the link.

Use the **-z text** option in dynamic mode only.

Linker Command Files

This chapter, which describes the `-A command_file` commands and how to use them with the ELF linker/locator, contains the following sections:

§4.1: *Overview of Linker Command Files*

§4.2: *Linker Commands Reference*

See §2.3: *Linker Commands Summary* for a definition of the two kinds of command files supported by the ELF linker/locator.

4.1 Overview of Linker Command Files

The ELF linker/locator can read a series of commands provided in a *command file*, which you specify with linker option `-A` followed by the name of the command file. For example:

```
ld obj_file.o -A cmd_file.txt
```

You can specify multiple command files. The effect is as if they were concatenated.

Note: Command files are normally used in contexts that require static linking. When you create a dynamically linked application, the effects of a command file might be constrained by the required conventions of the dynamic loader.

4.1.1 Command-File Conventions

The following conventions apply to linker command files:

- Command names are case sensitive. They must be uppercase.
- Commands and their arguments can start in any column.
- Arguments must be separated from their command by whitespace.

- Names must be separated from each other by commas or spaces, depending on the format of the command.
- Comments start with a semicolon (;) or pound sign (#) and end with a newline.
- Only one command per line is permitted. To continue a line, put a backslash (\) at the end of the line. The **SECTIONS** command is an exception: it can cross line boundaries without requiring a backslash (\).
- Numbers must be specified as C constants (for example, 0x89ABC).
- When using commands that name a specific section, you must specify an input section with reference to the input object file in which it exists. For example, this command:

```
.text!text!mod.o
```

refers to an input section named `.text` of type `text` that resides in the input file `mod.o`.

You specify output sections by their names and/or types. You can omit extensions if there is no ambiguity in determining the section name. For example, each of the following commands:

```
.text!text  
.text
```

specifies an output section named `.text` of type `text`, if there is only one output section of type `text`.

- You can use wildcard characters to reference file names in command files. See §4.1.3: *Using Wildcards in File-Name References* for more information.

4.1.2 Classification of Sections in Executable Files

Sections in executable files are classified according to type:

- A `text` section is read-only and contains executable code.
- A `lit` section is read-only but contains data; for example, string constants and **const** variables.

- A data section contains writable data.
- The bss section is a writable data section that is initialized to zeroes when the program is loaded. The bss section does not occupy space in the object file.

4.1.3 Using Wildcards in File-Name References

When you refer to file names within linker command files, you can use the wildcard characters described in this section. These wildcards behave like the corresponding UNIX regular-expression characters.

*** — Match zero or more characters**

Examples:

- `abc*` matches any name beginning with `abc`.
- `*abc` matches any name that ends with `abc`.
- `*abc*` matches any name that has `abc` as a substring.

In the context of the linker, `*(*)` matches any archive member name; `/lib/libc.a(*)` matches any member of `/lib/libc.a`.

? — Match exactly one character

Example: `abc?` matches any name that begins with `abc` followed by any valid character.

[abc] — Match any one of the characters a, b, or c

Example: `file.[ao]` matches `file.a` or `file.o`

[^abc] — Match any character except a, b, or c

Example: `file.[^abc]` matches `file.x`, where `x` is any valid character *except* `a`, `b`, or `c`.

[a-z] — Match any character in the range a through z

Example: `file.[a-z]` matches `file.a`, `file.b`, and so on, up to `file.z`.

[^a-z] — Match any character except those in the range a through z

Example: `file.[^a-z]` matches `file.x`, where `x` is any character *except* those in the range `a` through `z`.

\ — Escapes other wildcard characters

Examples:

- `file.*` matches `file.*`
- `file.\?` matches `file.?`

4.2 Linker Commands Reference

This section describes each supported linker command. Commands are arranged alphabetically.

Note: In the following **Syntax** descriptions, bolded punctuation characters such as “[” and “{” are *meta-characters*, which indicate choices of arguments, repeating arguments, and so on.

These meta-characters are described in the section *Notational and Typographic Conventions* in the *About This Book* chapter at the beginning of this manual.

Do *not* enter meta-characters in your command files.

ARGS

Specify linker command-line arguments in a command file

Syntax **ARGS** [*arguments*]

Argument	Description
<i>arguments</i>	Any valid linker command-line argument. See Chapter 2: <i>Using the ELF Linker/Locator</i> for information on valid linker command-line arguments.

Description Use the **ARGS** command to specify command-line arguments (options and input files) in a linker command file. If the first command-line argument begins with a dash (-), you can omit the **ARGS** keyword.

The linker processes the arguments in the linker command file exactly as if they had appeared on the command line at the position of option **-A** (which specifies the command file).

Example 1 This **ARGS** command names the output file as `file` and generates a linker memory map file:

```
ARGS -o file -m
```

Example 2 This example names the output file as `file` and generates a linker memory map file:

```
-o file -m
```

In this example, the **ARGS** command is not needed, because the line begins with a dash.

DEMANDLOAD / NODEMANDLOAD

Specify an executable that can or cannot be demand-loaded from disk

Syntax **DEMANDLOAD** [*page_size*]
NODEMANDLOAD

Argument	Description
<i>page_size</i>	The page size (in bytes) of the target memory configuration. The value must be an integral power of 2. The default value is operating-system dependent. Typical values are 4096 and 65536.

Description The **DEMANDLOAD** command specifies that the generated executable be of a form that can be *demand-loaded* from disk.

Demand-loading, also called *demand-paging*, is a process available in virtual storage systems whereby a program page is copied from external storage to main memory when needed for program execution.

Specifically, given any section *s* with virtual address *address(s)* and file-offset *offset(s)*, the following condition is true:

$$\text{offset}(s) \bmod \text{page_size} == \text{address}(s) \bmod \text{page_size}$$

The **NODEMANDLOAD** command specifies that the generated executable is not to be demand-loaded from disk.

Example This **DEMANDLOAD** command generates an executable that is demand-loadable from a system that uses 65,536-byte pages:

```
DEMANDLOAD 65536
```

INITDATA

Specify control sections to initialize at run time

Syntax **INITDATA** *section* [,*section*] . . .

Argument	Description
<i>section</i>	Section designation

The *section* argument takes *one* of the following forms:

1. *section_name!section_type!object_file*
2. *section_name!section_type*
3. *section_name!!object_file*

where the subcomponents of the *section* argument have the following meanings:

<i>section_name</i>	Name of the output section.
<i>section_type</i>	Section type. The only permissible types are <code>data</code> and <code>lit</code> .
<i>object_file</i>	Name of the input object file containing the section.

Description The **INITDATA** command specifies control sections to be initialized at run time. This capability is required for programs that must run in ROM.

Note: The **INITDATA** command cannot be applied to `text` sections. Also, to function reliably, the executable file must be statically linked. If you are performing an incremental link (option **-r**), the linker ignores the **INITDATA** command.

The **INITDATA** command causes the linker to do the following:

- Create a new `lit` section named `.initdat`.
- Fill `.initdat` with a table representing the contents of the applicable control sections. The linker reclassifies the specified sections as `bss` sections.
- Generate an external symbol named `_initdat`, whose address is the absolute starting address of the `.initdat` section. (This facilitates the copy mechanism at run time.)

Call `_initcopy()` to initialize the sections

Your program must call the library function `_initcopy()` at the start of program execution to initialize the control sections from the table in `.initdat`.

Note: `_initcopy()` is available in object format (in the High C/C++ standard library) and in source format (in the High C/C++ `lib` directory).

For a discussion of how to use the `_initcopy()` function, see §6.5: *Initializing RAM from a Program in ROM*.

You can give multiple section names to the **INITDATA** command. You can use the **SECTIONS** command to provide absolute addresses for both the `.initdat` section and the destination data sections. If you do not supply an absolute address for `.initdat`, the linker allocates it as an ordinary `lit` section. You can also specify a `bss` section — the `_initcopy()` function will initialize it to 0 (zero).

If you specify an input section name (that is, if you associate the section with an input object file), the section must be the only resident of its associated output section.

Example 1 This **INITDATA** command causes the linker to create an `.initdat` section containing a copy of the contents of sections `.lit` and `.data`:

```
INITDATA .lit,.data
```

When the application invokes the function `_initcopy()`, sections `.lit` and `.data` are automatically reinitialized from the `.initdat` section.

Example 2 This **INITDATA** command causes `_initcopy()` to initialize all data sections and `bss` sections at run time, regardless of how those sections are named:

```
INITDATA !data,!bss
```

LOAD

Read input files

Syntax `LOAD input_file[,input_file]...`

Argument	Description
<i>input_file</i>	Name of input object file or library

Description The **LOAD** command specifies input object files to be linked as if you specified them on the command line. Input files can be any of the following:

- relocatable files from the assembly process
- relocatable files resulting from incremental linking
- libraries

The linker uses an input file's internal format to determine the nature of the file.

When you specify input files both on the command line and in a linker command file, the linker reads the input files in the order it encounters them. For example, given the following command line:

```
ld file1.o file2.o -A command_file file3.o
```

the linker does the following, in this order:

1. Reads the files `file1.o` and `file2.o`.
2. Opens the file `command_file` and processes any commands in it (including **LOAD** commands).
3. Reads `file3.o`.

Processing archive libraries The linker searches libraries in the order it encounters them. When the linker encounters an archive library, it searches the library only to resolve those unresolved symbols that are pending when it encounters that archive. Thus, backward references from one library to another are not resolved. The

linker searches a library repeatedly, as required, to resolve intra-library references.

To resolve mutual external references, specify the name of the library again. See §6.6: *How the Linker Processes Archive Libraries* for additional information.

Example This **LOAD** command reads all input sections from the specified object files in the order they are encountered:

```
LOAD file1.o, file2.o, file3.o, libc.lib
```

NODEMANDLOAD Specify an executable that cannot be demand-loaded from disk

Syntax **NODEMANDLOAD** [*page_size*]

Description See the entry for **DEMANDLOAD**.

PUBLIC Reassign the value of an external symbol

Syntax **PUBLIC** *symbol=value*

Argument	Description
<i>symbol</i>	External symbol
<i>value</i>	Value of the external symbol; must be a C constant

Description The **PUBLIC** command changes or defines the value of an external (or public) symbol at link time, after the linker has read in all input files.

- If the symbol is already defined externally, the linker changes its value to the specified value.
- If the symbol is not already defined, the linker enters it into the symbol table so it will be available to resolve external references.

The linker regards any value defined by the **PUBLIC** command as absolute. The linker does not locate the value in any relocatable section, and it does not relocate the value relative to any section base.

The value you specify in the **PUBLIC** command can be in decimal, octal, or hexadecimal format.

Example This **PUBLIC** command defines the value of public symbol `PUB` to be 100:

```
PUBLIC PUB=100
```

RESADD

Reserve memory location by address

Syntax **RESADD** *low,high*

Argument	Description
<i>low</i>	Low address of the memory to be reserved. Must be a numeric C constant.
<i>high</i>	High address of the memory to be reserved. Must be a numeric C constant.

Description The **RESADD** command reserves addresses *low* to *high*, inclusive. Nothing is loaded into this area. It is essentially a “hole” in memory.

Example This **RESADD** command reserves 3,000 hex bytes of memory, starting at address 0x7000:

```
RESADD 0x7000,0x9FFF
```

RESNUM

Reserve memory location by size

Syntax **RESNUM** *low,size*

Argument	Description
<i>low</i>	Low address of the memory to be reserved. Must be a numeric C constant.
<i>size</i>	An offset, in bytes. <i>size</i> indicates the amount of memory to be reserved.

Description The **RESNUM** command reserves addresses *low* to *low + (size - 1)*, inclusive. Nothing is loaded into this area — it is essentially a “hole” in memory.

Example This **RESNUM** command reserves 3,000 hex bytes of memory, starting at address 0x7000:

```
RESNUM 0x7000,0x3000
```

SECTIONS

Specify order of output sections

Note: Before you use the **SECTIONS** command, we recommend that you read through this entire section, including the examples, and read §4.2.1: *More About the SECTIONS Command* to get a general idea of the possible forms the **SECTIONS** command can take. Doing so will facilitate your understanding of this command's complex syntax.

Syntax At the highest level of abstraction, this is the syntax of the **SECTIONS** command:

```
SECTIONS {output_group_spec ...}
```

output_group_spec The *output_group_spec* specifier in a **SECTIONS** command determines how the linker groups output sections in memory. You must replace the *output_group_spec* specifier with one of the following forms:

1. **GROUP** *address_spec*: { *output_sect_spec* }
2. *output_sect_spec*

These two forms of *output_group_spec*, in turn, can contain additional keywords, specifiers, and variables.

See §4.2.1.1: *The output_group_spec Specifier* for a complete discussion of this specifier.

output_sect_spec The *output_sect_spec* specifier in an *output_group_spec* names an output section, and (optionally) assigns to it an absolute address or causes it to be appropriately aligned when allocated in memory. *output_sect_spec* uses this syntax:

```
{sect_name | *} [TYPE sect_type] [address_spec ...]:  
  [{input_sect_spec ...}]
```

See §4.2.1.2: *The output_sect_spec Specifier* for a complete discussion of this specifier.

input_sect_spec The *input_sect_spec* specifier in an *output_sect_spec* allows you to select input sections within the input file(s) by name or type. *input_sect_spec* uses this syntax:

```
input_file[ (member) ] {sect_spec . . . }
```

See §4.2.1.4: *The input_sect_spec Specifier* for a complete discussion of this specifier.

Description The **SECTIONS** command overrides the default conventions the linker uses to place output sections in an executable output file. You use the **SECTIONS** command to control how the linker combines input sections into output sections, and how it allocates output sections in memory.

Syntax conventions You must observe the following syntax conventions when using the **SECTIONS** command:

- You can specify multiple **SECTIONS** commands. The effect is as if they were concatenated.
- Lines within the **SECTIONS** command can freely cross line boundaries.
- You must enter non-meta-character punctuation as shown. However, if you do not use the *input_sect_spec* specifier, you can omit its enclosing braces. (See *Notational and Typographic Conventions* in the *About This Book* chapter for an explanation of meta-characters.)

Keywords specific to the SECTIONS command The **SECTIONS** command's syntax uses a variety of keywords (also called *directives*), as well as specifiers and variables that you define. The following keywords are part of the syntax of the **SECTIONS** command. These keywords are reserved by the linker in the context of the **SECTIONS** command. They must be all uppercase.

Keyword	Description
ADDRESS	Specifies an absolute starting address for an output section or group
ALIGN	Specifies the alignment boundary for an output section
ALIGN_INPUT	Specifies an alignment boundary for each input section allocated to an output section, relative to the beginning of the output section
GROUP	Specifies a list of output sections that are to be treated as one entity by the linker algorithm

Keyword	Description
TYPE	Specifies a section type for input or output sections

For a more in-depth discussion of the syntax and components of a **SECTIONS** command, see §4.2.1: *More About the SECTIONS Command*.

Example 1 This example shows how to separate read-only memory (ROM) from read/write memory (RAM). The command file and object files are named on the command line as follows:

```
ld -A file.cmd mod1.o mod2.o
```

The command file, `file.cmd`, specifies where to locate the output sections. Suppose `file.cmd` contains the following:

```
SECTIONS {
    GROUP ADDRESS 0x200000 : { # ROM start address
        .text :
        .lit :
    }
    GROUP ADDRESS 0x300000 : { # RAM start address
        .data :
        .bss :
    }
}
```

The first **GROUP** directive tells the linker to locate the executable code (`.text`) and read-only data (`.lit`) together at one address (`0x200000`).

The second **GROUP** directive tells the linker to locate the initialized data (`.data`) and uninitialized data (`.bss`) together at a second address (`0x300000`).

Note: The addresses in Example 1 are arbitrary.

Example 2 This example is more complex than Example 1. Suppose you want to link three files (`mod1.o`, `mod2.o`, and `mod3.o`) to produce an output file with `.data` sections combined first, `.bss` sections combined next, then followed by two `.text` sections.

Name the command file and object files on the command line as follows:

```
ld -A file.cmd mod1.o mod2.o mod3.o
```

The command file, `file.cmd`, contains the following:

```
SECTIONS {
  GROUP ADDRESS 0x400 : { # Group starts at 0x400
    outdata : {          # 1st output section
      * {.data}          # Combine data sections
    }
    .bss ALIGN 0x80 : {  # First 0x80 byte
                        #   boundary after
                        #   outdata section.
      * {TYPE bss}      # Combine bss sections
    }
    usertext : {         # 3rd output section
      mod1.o {mycode}    # Section mycode of
                        #   mod1.o linked into
                        #   section usertext
    }
  }
  .text ADDRESS 0x4000 : { # 4th output section
    * {.text}             # Section mycode of
                        #   mod2.o linked after
                        #   all .text sections
    mod2.o {mycode}
  }
}
```

In this example, the linker arranges the output sections in the following order at the indicated locations:

- `outdata` (located at 0x400)
- `.bss` (located at the next 0x80th boundary after `outdata`)
- `usertext` (immediately following `.bss`)
- `.text` (located at 0x4000)

The **GROUP** directive ensures that the linker allocates the `outdata`, `.bss`, and `usertext` sections consecutively as a group.

The `.text` section contains the `.text` sections of all three files, followed by section `mycode` of `mod2.o`. If previous sections have already overwritten this address, the linker issues an error message and does not generate an executable.

Example 3 This **SECTIONS** command:

```
SECTIONS {
  .text ADDRESS 0xc0004000:
    * TYPE text:
```



```

* TYPE lit:
* TYPE data:
* TYPE bss:
}

```

arranges the output sections in order as follows:

- `.text` (located at address `0xc0004000`)
- any other output sections of type `text`
- any output sections of type `lit`
- any output sections of type `data`
- any output sections of type `bss`

The specification `* TYPE text:` instructs the linker to insert *all* output sections of type `text`, except those explicitly designated elsewhere.

Example 4 This **SECTIONS** command:

```

SECTIONS {
* ADDRESS 0xc0004000:
* TYPE lit:
* TYPE data:
* TYPE bss:
}

```

puts all output sections that do not match elsewhere at address `0xc0004000`, followed by all sections of type `lit`, `data`, and `bss`.

START

Specify program entry point

Syntax **START** [*symbol* | *value*]

Argument	Description
<i>symbol</i>	Name of an exported symbol
<i>value</i>	An address; must be a C integer constant

Description The **START** command specifies the program's entry point. You can use this command to specify a particular entry point or to override a previously defined (or default) entry point.

When you link an executable file (that is, relocate it), the entry point of the output matches the entry point of the input, relocated appropriately.

If you do not specify the entry point (with the **START** command or with linker option **-e**), the linker assumes that the entry point is `_start`.

If you have specified a symbol as the entry point (or if the entry point has defaulted to `_start`), the linker issues an error diagnostic if the symbol remains unresolved at the end of link time.

Example 1 This **START** command sets the entry point to the address of the symbol `begin`:

```
START begin
```

Example 2 This **START** command sets the entry point to address 800:

```
START 0x800
```

4.2.1 More About the **SECTIONS** Command

This section provides additional information about using these specifiers and keywords in a **SECTIONS** command:

- specifiers
 - *output_group_spec*
 - *output_sect_spec*
 - *address_spec*
 - *input_sect_spec*
- keywords
 - **GROUP**
 - **TYPE**
 - **ADDRESS**
 - **ALIGN**
 - **ALIGN_INPUT**

4.2.1.1 The *output_group_spec* Specifier

Syntax The *output_group_spec* specifier in a **SECTIONS** command uses this syntax:

```
[ GROUP [ address_spec ] : { ] output_sect_spec ... [ } ]
```

which means any of the following can be an *output_group_spec* specifier:

```
GROUP address_spec : { output_sect_spec ... }
GROUP : { output_sect_spec ... }
output_sect_spec
```

GROUP keyword You can use the **GROUP** keyword to force a set of output sections to be allocated contiguously in a specified order. For allocation purposes, the linker treats output sections enclosed by a **GROUP** keyword as a single entity. The entire group must fit into memory, and it will not be split around an absolute section.

Example

```
SECTIONS {
    GROUP ADDRESS 0x80000 : {
        .text:
        .lit:
    }
    .data:
    .bss:
}
```

In this example, the linker maps the `.text` and `.lit` sections consecutively, starting at address `0x80000`, and maps the `.data` and `.bss` sections at subsequent locations (subject to page constraints).

4.2.1.2 The *output_sect_spec* Specifier

Typically, the *output_sect_spec* specifier determines the order of output sections in memory. However, the combination of assigning output sections to absolute addresses and using the “next-fit” algorithm for locating relocatable sections can cause the linker to locate a relocatable output section *lower* in memory than an absolute section that preceded it in the **SECTIONS** command.

Syntax The *output_sect_spec* specifier in a **SECTIONS** command uses this syntax:

```
{sect_name | *} [ TYPE sect_type ] [ address_spec ... ] :
    [ {input_sect_spec ...} ]
```

The subcomponents of the *output_sect_spec* have the following meanings:

<i>sect_name</i>	Represents the name of an output section.
*	Wildcard character — matches any section name that is not otherwise explicitly specified.
<i>sect_type</i>	Represents the type of an output section. Must be preceded by the TYPE keyword.
<i>address_spec</i>	Designates the starting address of an output section or group, as well as constraints on the starting address.
<i>input_sect_spec</i>	Names a specific input file. Can be a regular expression as documented in §4.1.3: <i>Using Wildcards in File-Name References</i> .

Description Each *output_sect_spec* specifier names an output section and, optionally, assigns it to an absolute address, or causes it to be aligned appropriately when it is allocated to memory.

The executable output file produced by the linker contains these output sections in the order they appear in the **SECTIONS** command. Additional output sections created in the process of assigning input sections can follow the ones created explicitly with the **SECTIONS** command.

TYPE keyword By default, the type of each output section is determined by the type of the first input section assigned to it. However, you can explicitly assign the type with the **TYPE** keyword.

Using a wildcard You can also use the wildcard character (*) in place of an output section name. The wildcard designation defines an implicit group of sections. For example, the following designation instructs the linker to group all output sections of type *text* consecutively:

```
* TYPE text
```

See Example 1 and Example 4 (in the description of **SECTIONS**) for ways to use the wildcard character.

Note: The wildcard character does *not* apply to output sections explicitly named elsewhere.

Example

```
SECTIONS {
    .text : { * {TYPE text TYPE lit} }
    * TYPE data :
    * TYPE bss :
}
```

In this example, the linker maps all input sections of type `text` or `lit`, regardless of their names, into output section `.text`. All data sections follow, and then all `bss` sections.

4.2.1.3 The *address_spec* Specifier

The *address_spec* specifier in a **SECTIONS** command consists of one or more of the following keywords:

- | | |
|-----------------------------|--|
| ADDRESS <i>n</i> | Assigns an absolute starting address of an output section or group. Cannot be used with ALIGN . |
| ALIGN <i>n</i> | Causes the allocation address of the output section to be allocated to the next <i>n</i> -byte boundary where it will fit. Cannot be used with ADDRESS . |
| ALIGN_INPUT <i>n</i> | Causes each input section to be aligned to the specified boundary relative to the start of the output section. The linker adds padding between input sections to preserve alignment. |

ADDRESS, ALIGN, and ALIGN_INPUT keywords The keywords **ADDRESS**, **ALIGN**, and **ALIGN_INPUT** take an integer parameter, *n*, which can be given in the form of a C constant. Parameter *n* must be a power of 2 that is no less than the default alignment of the applicable sections or group.

Observe the following restrictions when using these keywords:

- **ALIGN** can be used with any output section or **GROUP** keyword.
- **ALIGN_INPUT** can be used with any output section, but not with the **GROUP** keyword.
- **ADDRESS** cannot be used for output sections within a **GROUP** statement. Because all the sections within a group are allocated together, the whole group must be made absolute.

4.2.1.4 The *input_sect_spec* Specifier

Within each *output_sect_spec* specifier, you can include an optional list of *input_sect_spec* specifiers enclosed in curly braces. The *input_sect_spec* adds a finer level of control to the **SECTIONS** command by enabling you to specify precisely which sections to store in each output section.

Syntax The *input_sect_spec* specifier in a **SECTIONS** command uses this syntax:

```
input_file [ (member) ] { sect_spec . . . }
```

The subcomponents of the *input_sect_spec* have the following meanings:

<i>input_file</i>	Regular expression; represents one or more input object files or archive libraries.
<i>member</i>	Denotes an archive member of an input archive library. Can also be a regular expression.
<i>sect_spec</i>	Specifies a section by name or type.

The *sect_spec* specifier can be one of the following:

<i>sect_name</i>	Represents the section name
TYPE <i>sect_type</i>	Represents the section type

Description Each *input_sect_spec* specifier can name a specific input file, or it can refer to all input files. Additionally, you can select individual input sections in the input file(s) by name or type.

Pattern matching A particular input section can match the pattern of two or more *input_sect_spec* specifiers. For example, each of the following specifications matches a section named `.text` of type `text` in file `file.o`:

1. `* {TYPE text}`
2. `file.o {.text}`
3. `file.o {TYPE text}`

In such a case, the linker attempts to resolve the ambiguity by associating the input section with the *input_sect_spec* that is the *most specific*, where *most specific* is determined as follows:

- If a subset of the matching *input_sect_spec* specifiers have an explicit file name, those that use the wildcard character (*) are eliminated.

Thus, in the preceding example, case (1.) would be eliminated. (Explicit file name `file.o` supercedes *.)

- If one of the matching *input_sect_spec* specifiers has an explicit section name, it is considered more specific than one that has a section type only.

Thus, in the preceding example, the linker would choose case (2.) to refer to section `.text` of file `file.o`. (Explicit section name `.text` supercedes section type name **TYPE** `text`.)

- If, after applying these criteria, more than one matching *input_sect_spec* remains, the linker chooses the one that appears first in the **SECTIONS** command.

Mapping an unassigned input section to an output section If any unassigned input sections remain after all **SECTIONS** commands have been read, the linker maps each one into an output section according to the following rules:

- If the input section is absolute, the linker maps it into an absolute output section of the same name.
- If an *output_sect_spec* with the same name as the input section exists, and it has no *input_sect_spec* specifiers, the linker maps the input section into the associated output section.
- Otherwise, the linker maps all unassigned input sections of a given name into a single output section of the same name.

Examples These are examples of valid *input_file* specifications:

```
*/libc.a(*)
qsort.o
*
libtools.a(printer.o)
```

This is an example of an *input_sect_spec* specification in a **SECTIONS** command:

```
*/libtest.a(file.o) {TYPE .text}
```

The following example puts all input sections of type text that are not from an archive library into output section `.text`, and puts all text sections from archive libraries into section `.lib.text`. All `lit`, `data`, and `bss` sections follow, in that order.

```
SECTIONS {
    # Put all .o text sections first:
    .text:{
        *.o { TYPE text }
    }
    # Put all text sections of archives into
    # .lib.text:
    .lib.text: {
        *(*) { TYPE text }
    }
    * TYPE lit:
    * TYPE data:
    * TYPE bss:
}
```


AT&T-Style Map-File Directives

This chapter, which describes the **-M** *input_map_file* directives (AT&T-style map-file commands) and how to use them with the ELF linker/locator, contains the following sections:

§5.1: *Overview of Input Map Files*

§5.2: *Input Map File Directives*

§5.3: *Sample Input Map File*

See §2.3: *Linker Commands Summary* for a definition of the two kinds of command files supported by the ELF linker/locator.

5.1 Overview of Input Map Files

The linker automatically maps input sections from object files to output sections in executable files. You can change the default linker mapping when you specify a static link.

The ELF linker/locator can read a series of commands provided in an *input map file*, which you specify with linker option **-M** followed by the name of the map file. For example:

```
ld obj_file.o -M imapfile.txt
```

You can specify multiple input map files. The effect is as if they were concatenated.

See the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**, Appendix B: *Mapfile Option*, for complete input map file details.

You can use AT&T-style input map files to do the following:

- declare output sections and specify values for section attributes such as section type, permissions, addresses, length, and alignment
- control mapping of input sections to output sections

- declare a global-absolute symbol that can be referenced from object files
- specify output section order

Note: The only sections you should list in input map files are those sections that will be allocated at load time.

Example In this linker command line:

```
ld -M map_file file.o -l name
```

input map file `map_file` specifies the mapping directives. The linker uses library `libname.a` or `name.lib` to resolve external symbols.

5.2 Input Map File Directives

You can include the following types of directives in an input map file:

Type of Directive	Purpose
Output section declaration	Creates a new output section in the output file, or changes the attribute values of an existing output section
Mapping directive	Tells the linker how to map input sections to output sections
Size-symbol declaration	Defines a new global/absolute symbol that represents the size, in bytes, of the specified output section
Output-section order specification	Specifies an ordering of output sections in the generated executable

These directive types are covered in more detail in the following sections.

Output section declaration

Create or modify output section in executables

Syntax `osect_name[= attr_value [...]];`

- `osect_name` is an identifier, the name of an output section.
- The `attr_value` arguments can be one or more of the following:

Argument	Description	Valid Values
<code>sect_type</code>	Loaded by the loader at load time Read by the loader at load time	LOAD NOTE
<code>permiss_flags</code>	Any permutation of read, write, and execute	? [R W X]
<code>virtual_address</code>	Hexadecimal address	V <i>value</i>
<code>physical_address</code>	Hexadecimal address	P <i>value</i>
<code>length</code>	Integer up to 2 ³²	L <i>value</i>
<code>alignment</code>	Any integer	A <i>value</i>

Note: The `attr_value` argument is referred to as `segment_attribute_value` in the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**.

Description An output section declaration creates a new output section in the executable file or changes the attribute values of an existing output section.

Example In this example:

```
segment_1 = LOAD V0x30005000 L0x1000;
```

`segment_1` is declared as follows:

- to be loaded at load time (`sect_type = LOAD`)
- starting at virtual address 30005000 (`virtual_address = V0x30005000`)
- for a length of hexadecimal 1000 bytes (`length = L0x1000`)

Because `segment_1` is declared as type **LOAD**, it defaults to read + write + execute (**RWX**) — unless you specify otherwise with a *permiss_flags* value.

Mapping directive

Specify how to map input sections to output sections

Syntax `osect_name : [sect_attr_value [...]]
[: file_name [...]] ;`

- *osect_name* is an identifier, the name of an output section.
- *file_name* is the name of an object file, which can be a regular expression (see §4.1.3: *Using Wildcards in File-Name References*).
- The *sect_attr_value* arguments can be one or more of the following:

Argument	Description	Valid Values
<i>s_name</i>	Any valid section name	
<i>s_type</i>	No bits section	\$NOBITS
	Note section	\$NOTE
	Processor-specific sections	\$PROGBITS
<i>s_flags</i>	Any permutation of allocatable, writable, and executable	? [!] [A W X]

Note: The *sect_attr_value* parameter is referred to as *section_attribute_value* in the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**.

Description Mapping directives tell the linker how to map input sections to output sections.

The *entrance criteria* for a specific output section are the *sect_attr_value* parameters an input section must have to map into that output section.

Example In this example:

```
segment_1 : $PROGBITS ?AX : file_1.o file_2.o ;
segment_1 : .bss ;
```

segment_1 is mapped with the following:

- all sections from the object files file_1.o and file_2.o that are processor specific (\$PROGBITS) and allocatable + executable (?AX)
- all .bss sections

Size-symbol declaration

Define new symbol representing size of an output section

Syntax *sect_name @ symbol_size_name*

Description Size-symbol declarations define a new global or absolute symbol that represents the size, in bytes, of the specified output section. This symbol can be referenced in your object files.

symbol_size_name can be any valid High C/C++ identifier.

Example This example assigns the size name Protected_Data_Section_Size to sect_1.

```
sect_1 @ Protected_Data_Section_Size
```

Section-ordering directive

Specify ordering of output sections in executable

Note: This feature is an extension of the implementation described in **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**, Appendix B: *Mapfile Option*.

Syntax *) sect_name [. . .] ;*

Description A output-section ordering directive specifies the ordering of output sections in the generated executable, subject to any explicit virtual-address specification. (Explicit virtual-address assignment overrides an output-section ordering specification.)

The components of this directive have the following meanings:

-) Designates ordering of sections
- segment_name* Names an output section to be ordered

When you use the output-section ordering directive, the linker places only those sections listed in the directive at the named virtual address. For example, suppose you have these sections from an ELF file:

```
.text
.rodata
.plt
.data
.got
.bss
```

and these directives in the input map file:

```
.text = V0x1000
) .text .rodata .data ;
```

Suppose the default starting address for the linker to use is 0x8048000. If the linker creates an executable file with the given sections, the linker will map the sections as follows:

- The sections named in the input map file (.text, .rodata, and .data) will appear at virtual start address 0x1000.
- The sections not named in the input map file (.plt, .got, and .bss) will begin at the default start address (0x8048000) and will be mapped accordingly.

In this example, if you want all the input sections to be packed into output sections starting at 0x1000, use an input map file with these directives:

```
.text = V0x1000
) .text .rodata .plt .data .got .bss ;
```

Note: The sections that must be mapped into different output sections will be mapped onto consecutive pages, as long as you do not specify otherwise. This is the same as with default mapping at the default starting address.

Example 1 This command maps sections `.text`, `.got`, `.plt`, `.data`, and `.bss` consecutively, starting at address `0x1000`:

```
.text = V0x1000 ;
) .text .got .plt .data .bss ;
```

Example 2 This command maps sections `.data`, `.data1`, and `.bss` consecutively, starting at address `0x1000`, and maps sections `.text` and `.lit` starting at address `0x8000000`:

```
.data = V0x1000 ;
.text = V0x8000000 ;
) .data .data1 .bss ;
) .text .lit ;
```

Example 3 This command accomplishes the same as Example 2. Although the two output-section ordering directives have been combined into one, the order and starting addresses are unchanged — section `.text` still starts at address `0x8000000` and `.lit` follows it:

```
.data = V0x1000 ;
.text = V0x8000000 ;
) .data .data1 .bss .text .lit ;
```

5.3 Sample Input Map File

The following is an example of a default input map file:

```
text=LOAD ?RX;
text:$PROGBITS ?A!W;

data=LOAD ?RWX;
data:$PROGBITS ?AW;
data:$NOBITS ?AW;

my_note=NOTE;
my_note:$NOTE
```

This map file declares three output sections (`text`, `data`, and `my_note`) and sets their permissions and attributes.

- The `text` section is declared to be a load section, with read and execute permissions set. This `text` section is mapped with allocatable and

non-writable, processor-specific sections from within the link. It is loaded by the loader.

- The `data` section is also declared to be a load section, with the read, write, and execute permissions set. This `data` section is mapped with allocatable and writable, processor-specific, no-bit sections from object files specified on the command-line. It is also loaded at load time.
- The `my_note` section is declared to be a note section. This `my_note` section is mapped with note sections from object files specified on the command line. It is read by the loader at load time.

Additional Linker Topics

This chapter, which provides additional information about using the ELF linker/locator, contains the following sections:

§6.1: *Special Linker-Defined Symbols*

§6.2: *Linking Incrementally*

§6.3: *Generating Relocation Fix-Ups Versus Local Copies*

§6.4: *Rewiring Shared Function Calls Through the PLT*

§6.5: *Initializing RAM from a Program in ROM*

§6.6: *How the Linker Processes Archive Libraries*

§6.7: *Dynamic Versus Static Linking*

6.1 Special Linker-Defined Symbols

A program might need to determine the starting address and size of an output control section. The linker supports this capability by defining special symbols that are mapped to the starting and ending address of each control section of an executable. The linker defines these symbols only if an unresolved reference to them appears at the end of the link.

Note: Linker-defined symbols are defined only when you generate an executable file. They remain undefined when you perform incremental linking.

Naming conventions The symbols are named according to the following convention (as they would be referenced from C):

`_fsection_name` Set to the address of the start of `section_name`

`_esection_name` Set to the first byte following `section_name`

The linker removes a preceding dot character (.) of the section name. Thus, the symbols `_ftext` and `_etext` would be used to access the starting and ending address of the section named `.text`.

Finding section size You can determine the size of a section by subtracting the two addresses. In C, you do this by declaring the symbols as imported **char** arrays. For example, the following C code fragment illustrates how to access the address of the `.data` section and determine its size:

```
// Linker sets to address of .data
extern char _fdata[];
// Linker sets to address beyond .data
extern char _edata[];
main() {
    printf(".data address = 0x%lx;
           size = 0x%lx\n",
           _fdata,
           _edata - _fdata);
}
```

Duplicate section names If more than one output section exists with the same name, the linker only defines the special symbols for the first section (as it appears in the section table).

6.2 Linking Incrementally

Incremental linking is the process of combining two or more relocatable object files into a single relocatable object file. You can use the resulting file as input to additional linker invocations. Option **-r** instructs the linker to perform incremental linking.

For example, given the object files `t1.o`, `t2.o`, and `t3.o`, the following command combines them into a single relocatable object file `t123.o`:

```
ld -r -o t123.o t1.o t2.o t3.o
```

Common blocks When it performs incremental linking, the linker does not resolve common blocks (also known as *common symbols*). Common blocks are global symbols with an associated length but no explicit definition. Ordinarily, the linker is responsible for assigning appropriate addresses for such symbols. Common blocks are left to be resolved in the final link.

The object file resulting from an incremental link is *not* executable, even if all symbols are resolved. This is because instructions with associated relocation information are not necessarily in a form suitable for execution.

If you specify option **-s** (strip), the linker strips only local symbols and debugging information from the output file. All global symbols remain, in order to be available for future links.

6.3 Generating Relocation Fix-Ups Versus Local Copies

When linking an executable that references a variable within a shared library (DLL), the linker can do either of the following:

- insert a relocation fix-up for each reference to the symbol
- make a local copy of the variable and arrange for each reference to the symbol to reference the local copy instead

By default, the linker makes local copies of shared variables. To insert relocation fix-ups, specify option **-B nocopies**. See Chapter 3: *Linker Command-Line Options* for more information.

*Relocation fix-ups
can make code be
non-shareable*

Inserting relocation fix-ups can render a significant portion of an executable non-shareable with other processes if those other processes frequently reference variables exported from the shared library.

The dynamic loader must fix up these references at load time; this load-time fix-up results in `copy-on-write` faults. Pages containing such references cannot be shared with other processes that happen to be running the same executable.

However, if you compile the program position-independently (that is, you specify compiler option **-Kpic**), all global variables are referenced through the global offset table (GOT). In this case, only a single reference to each symbol exists and specifying option **-B nocopies** has no negative effect on resource utilization.

*Making local
copies keeps
code shareable*

Making a local copy for each shared variable has the advantage that none of the instructions referencing such symbols within the executable need to be fixed up at load time.

The linker allocates the local copies within a `bss` section of the executable. The dynamic linker initializes the local copy at load time and arranges for all references to the variable from the shared library to reference the local copy.

Rewiring the references in the shared library this way is seldom a problem, because shared libraries are typically compiled position-independently, so only the Global Offset Table references need changing.

6.4 Rewiring Shared Function Calls Through the PLT

In multi-tasking operating systems, a single program (for example, a text editor) can be executing multiple times concurrently in separate processes. In most virtual-memory operating systems, particularly those based on UNIX and Solaris, the code of such a “multi-client” program is loaded only once in physical memory. Each process then maps that program’s memory into its own address space. In this way, multiple processes execute a single copy of the program in memory.

If a multi-client program contains references to a shared library, the sharing becomes more complex. The operating system’s dynamic loader must resolve every reference to a shared library symbol. Because a shared library can be mapped at arbitrary virtual addresses at load time, each process running the same program might map the associated shared libraries at a different virtual address.

Any page modified by the dynamic loader to resolve shared-library references cannot be shared with other processes. As soon as the dynamic loader modifies a page, a `copy-on-write` fault occurs. Once such a fault occurs, the operating system makes a private copy of that page for the process.

If a program contains shared library references throughout, a significant portion of the program will not be shareable. This means that, when two or more processes are running such a program, the processes need extra memory that would not otherwise be required.

To help alleviate this problem of non-shareable code, the linker automatically “rewires” all function calls into shared libraries so that they go through the Procedure Linkage Table (PLT). Because the PLT is in the address space of the executable, the dynamic loader needs to fix up only the pages of the PLT at load time. The rest of the code can be shareable across processes.

If you specify linker option **-B noplt**, the linker will not implicitly create PLT entries. See Chapter 3: *Linker Command-Line Options* for more information.

6.5 Initializing RAM from a Program in ROM

Programs that you place in ROM must have their writable data sections initialized in RAM at run time. You can accomplish this with linker command **INITDATA**. **INITDATA** designates sections to be initialized at run time. (For details about the syntax and usage of **INITDATA**, see the entry for this command in §4.2: *Linker Commands Reference*.)

For example, the following command informs the linker that all data sections are to be initialized at run time:

```
INITDATA !data
```

Some configurations also require that **lit** sections (read-only data) be placed in RAM. To do this, you can extend the previous **INITDATA** command as follows:

```
INITDATA !data,!lit
```

*Initializing
sections designated
by **INITDATA***

To initialize sections designated by **INITDATA** at run time, you must invoke the library function `_initcopy()`, which is available in both source and object formats on the High C/C++ distribution. This function initializes the appropriate sections from a table constructed by the linker in section `.initdat`.

Note:

Function `_initcopy()` works even if there is no `.initdat` section, in which case it does nothing. This means you can unconditionally invoke the code regardless of whether you used the **INITDATA** command.

Calling Typically, you call `_initcopy()` at the start of `main()`, as follows:

```
_initcopy()
    #include <stdlib.h> /* Includes declaration */
                        /* of _initcopy()      */

    ...
    main(int argc, char **argv){
        _initcopy();
        ...
    }
```

If the loader does not automatically zero bss sections, you can use `_initcopy()` to do the work:

```
INITDATA !bss
```

See the entry for **INITDATA** in Chapter 4: *Linker Command Files* for more information.

6.6 How the Linker Processes Archive Libraries

This section describes how the ELF linker/locator processes archive libraries at link time. The linker follows the UNIX SVR4 convention for importing object code from archive libraries.

The order of archive libraries on the linker command line is significant. The linker processes the libraries in the order they appear, from left to right. Each library is “current” only once. The linker resolves undefined symbols in the program’s symbol table on a “first see, first use” basis.

If the *program symbol table* (the *PST*) contains any undefined symbols, the linker searches the current library’s global symbol table to see if any symbols in the library can resolve undefined symbols in the PST. If the library contains a global symbol that can resolve an undefined symbol, the linker imports the object file defining that symbol into the link.

When the linker imports an object file from a library, it adds the entire symbol table for that file to the PST. If this import adds any new undefined symbols to the PST, the linker searches the current library’s global symbol table again, attempting to resolve the new undefined symbols. If other object files in the current library contain global symbols that can resolve undefined symbols in the PST, the linker imports those object files to the link.

This search-and-import process continues until the linker can find no more symbols in the current library's global symbol table to resolve undefined symbols in the PST. The linker then moves on to process the next input file on the linker command line.

To search a library more than once, specify it multiple times on the command line. This might be necessary if two libraries have mutual references.

6.6.1 Undefined Reference Errors in Libraries

A common linking problem is an undefined reference that occurs even though the symbol is defined in a library. Typically, this problem occurs when two libraries have mutual dependencies.

For example, suppose a member in library `lib1.a` references a symbol defined in library `lib2.a`, and vice versa. To force the linker to search `lib1.a` again after searching `lib2.a`, you must specify `lib1.a` a second time after `lib2.a`, as follows:

```
ld f1.o f2.o ... lib1.a lib2.a lib1.a ...
```

This second specification of `lib1.a` is necessary because the linker does not recursively rescan libraries to resolve references. The linker reads libraries only as it encounters them.

Note: A better solution to this problem is to organize your libraries so that they have no circular dependencies.

6.6.2 Multiple Definition Errors in Libraries

Another common linking problem is multiple definitions that occur because a symbol is defined in more than one library. This problem is often caused by having a symbol defined in multiple archive members, and having the members define a different set of symbols.

For example, suppose that member `lib1.a(member1.o)` defines the symbols `_alpha` and `_beta`, and that member `lib2.a(member2.o)` defines the symbols `_alpha` and `_gamma`. Suppose that either `_alpha` or

`_beta` is unresolved when the linker encounters `lib1.a`. This condition forces the extraction of `member1.o`.

Further, suppose that members extracted from `lib1.a` reference `_gamma`. This forces the extraction of `member2.o` from `lib2.a`. As a result, `_alpha` gets defined more than once.

The proper solution to this problem is to design program files so that such conditions do not exist.

6.7 Dynamic Versus Static Linking

The ELF linker/locator can link files dynamically or statically. Dynamic and static linking differ in the way they address external references in memory (that is, in the way they connect a symbol referenced in one module of a program with its definition in another):

- *Static linking* assigns addresses in memory for all included object modules and archive members at link time.
- *Dynamic linking* leaves symbols defined in shared libraries (DLLs) unresolved until run time. The dynamic loader maps contents of DLLs into the virtual address space of your process at run time.

Dynamic linking allows many object modules to share the definition of an external reference, while static linking creates a copy of the definition in each executable.

Specifically, when you execute multiple copies of a statically linked program, each instance has its own copy of any library function used in the program. When you execute multiple copies of a dynamically linked program, just one copy of a library function is loaded into physical memory at run time, to be shared by several processes.

Linker Error Messages

This chapter contains information designed to assist you in understanding error messages generated by the linker. It contains the following sections:

§7.1: *Linker Error Message Severity Levels*

§7.2: *Linker Error Messages in the Map Listing File*

7.1 Linker Error Message Severity Levels

The linker generates diagnostic error messages at three severity levels: warnings, errors, and terminal errors.

7.1.1 Warnings

Warnings typically draw your attention to minor inconsistencies. For example:

```
Input section xxx of file xxx is of type xxx, but
it is being assigned to output section yyy with
different type yyy.
```

The linker still generates a valid output file when a warning occurs.

7.1.2 Errors

Errors indicate severe conditions, such as an unresolved symbol or a symbol that has been defined multiple times. The linker does not generate an output file when such an error occurs.

7.1.3 Terminal Errors

Terminal errors occur when the linker encounters an invalid or corrupt input file (or archive), or when an inconsistency occurs within one of the linker's internal data structures. The linker immediately aborts when it encounters a terminal error. The linker might or might not generate an output file. If the linker generates an output file, the file might be partial or corrupted in some manner.

7.2 Linker Error Messages in the Map Listing File

When a non-fatal error occurs during the link process, the linker writes a message to standard output and to the memory map listing file, if you specified one. Error messages contained in the map listing file provide a record that you can review later to diagnose linker errors.

When a terminal error occurs, the linker does not generate the map listing file, so the only record of what occurred is the set of diagnostic messages the linker sent to standard output.

8

Utilities

This chapter, which documents the archiver and other utilities in the High C/C++ toolset, contains the following sections:

§8.1: *Using the MetaWare Archiver*

§8.2: *Using the elf2hex Conversion Utility*

§8.3: *Using the elfdump Binary Dump Utility*

8.1 Using the MetaWare Archiver

The MetaWare archiver, **ar**, is a management utility that groups independently developed object files into an *archive library* (a collection of object files, also called *members*, residing in a single file) to be accessed by the linker. When the linker reads an archive file, it extracts only those object files necessary to resolve external references.

The MetaWare archiver does the following:

- creates an archive library
- deletes, adds, or replaces one or more members in an archive library
- extracts one or more members from an archive library
- lists the members included in an archive library
- maintains a list of externally defined names and the associated archive member that defines the name

Note: The MetaWare archiver does not support all UNIX **ar** options. Specifically, the UNIX **ar** options **-p** (print) and **-q** (quick append) are not supported.

8.1.1 Invoking the Archiver from the Command Line

The command-line syntax used to run the archiver consists of the name of the librarian program, followed by a list of options, one or more file names, and (possibly) one or more archive members. When errors occur during the processing of a library, the archiver prints an error message to standard output.

Assuming the archiver directory is on your execution path, you invoke the archiver with one of the following commands. For detailed descriptions of archiver options, see Table 8.1 in §8.1.2: *Archiver Options*.

Method 1 To replace (or add) members of an archive, use this syntax:

```
ar -r[v] archive file...
```

Method 2 To delete members of an archive, use this syntax:

```
ar -d[v] archive member...
```

Method 3 To display symbol table entries of an archive, display names of members in an archive, or extract members from an archive, use this syntax:

```
ar{-s|-t|-x}[v] archive[member...]
```

Method 4 To reconstruct an archive's symbol table, use this syntax:

```
ar -s[v] archive
```

Method 5 To merge members of one archive into another, use this syntax:

```
ar{-m|-M}[v] archive input_archive[member...]
```

These are the archiver command-line arguments and their definitions:

<i>archive</i>	The name of an archive library.
<i>input_archive</i>	The name of an archive library.
<i>file</i>	A relocatable object file.
<i>member</i>	A member contained in the archive library. If you omit this argument, the command applies to all entries in the archive library.

8.1.2 Archiver Options

Table 8.1 summarizes the archiver options. A dash (-) before the option is not required. No spaces are allowed between multiple options.

Table 8.1 Archiver Command-Line Options

Option	Meaning
-d	Delete member(s) from an archive library.
-h	Display archiver command synopsis.
-m	Extract members from one archive library and insert them into another. If the members already exist in the second archive library, the archiver does not replace them.
-M	Extract members from one archive library and insert them into another. If the members already exist in the second archive library, the archiver overwrites them.
-r	Replace (or add) member(s) in an archive library.
-s	Reconstruct the symbol table of an archive library.
-S	Display the symbol table entries of an archive library. If you do not specify any archive members, the archiver assumes all members of the archive.
-t	Display names of members in an archive library. If you do not specify any archive members, the archiver assumes all members of the archive.
-x	Extract specified member(s) from an archive library. If you do not specify any archive members, the archiver assumes all members of the archive. Option -x does not alter the archive library.
-v	Display verbose output of archiver actions. <ul style="list-style-type: none"> • When used with options -d, -r, or -x, option -v provides a file-by-file description of the actions performed. • When used with option -t, option -v provides a long listing of information for each file. • When used with options -s or -S, option -v is ignored.

8.1.3 Specifying Archiver Command-Line Arguments in a File

You can place frequently used archiver command-line arguments in an *argument file*. Simply enter command-line arguments in a text file in the same manner as you would enter them on the command line. You can use as many lines as necessary. (A newline is treated as whitespace.)

To specify to the archiver that a file is an argument file, enter the name of the file on the command line, preceded by an “at” symbol (@). For example:

```
ar @argument_file
```

When the archiver encounters an argument on the command line that starts with @, it assumes it has encountered an argument file. The archiver immediately opens the file and processes the arguments contained in it.

8.1.4 Archiver Invocation Examples

This section presents several archiver invocation examples.

Example 1 This example deletes object file `filename.o` from library `libx.a`. It provides verbose output:

```
ar -dv libx.a filename.o
```

Example 2 This example extracts object files `survey.o` and `table.o` from library `liborgnl.a`. It does not provide verbose output:

```
ar -x liborgnl.a survey.o table.o
```

Example 3 This example replaces the object files in library `libnew.a` with new object files named in the archiver argument file `newobjs.txt`. It provides verbose output:

```
ar -rv libnew.a @newobjs.txt
```

8.2 Using the `elf2hex` Conversion Utility

Note: For background information on hex files and PROM devices, see Appendix A: *Working with PROMs and Hex Files*.

This section describes the ELF-to-hex converter, **elf2hex**, a stand-alone utility for converting binary files to an ASCII hexadecimal format. This utility produces one or more ASCII hex files from an executable ELF file.

This conversion utility can write the hex file in any of the supported formats, including the following:

- Motorola S3 Record format (the default)
- Extended Tektronix Hex format
- Mentor QUICKSIM Modelfile format

(See §A.4: *Hex File Formats* for a discussion of these formats.)

The converter can, if required, partition the hex file into a set of files that are suitable for programming PROMs in a PowerPC-based system.

See §A.2: *PROM Device Model* for a discussion of the device model assumed by the converter.

8.2.1 Invoking the ELF-to-Hex Converter

You invoke the ELF-to-hex converter with the **elf2hex** command:

```
elf2hex [ option . . . ] input_file
```

These are the command-line arguments:

<i>option</i>	Conversion options, separated by spaces. See §8.2.2: <i>Conversion Option Reference</i> for a complete list of options.
<i>input_file</i>	Executable or relocatable ELF input file. This is the only required argument.

If you execute **elf2hex** with no options, it assumes default values for all options. If, by chance, you specify an incorrect option, **elf2hex** writes a self-explanatory error message to the standard error device.

Table 8.2 summarizes the ELF-to-hex conversion options.

The dash before the option is not required.

Table 8.2 *ELF-to-Hex Options Summary*

Option	Description
<code>-c list</code>	Specifies the section types to be converted
<code>-i bank</code>	Selects the number of interleaved banks
<code>-m</code>	Specifies the hex format of the output file (Motorola)
<code>-n word_size</code>	Specifies the width of the memory device
<code>-o name</code>	Specifies the name of the generated hex file
<code>-q</code>	Specifies the hex format of the output file (QUICKSIM)
<code>-Q</code>	Suppresses the copyright message
<code>-s size</code>	Identifies the size (or length) of the memory device(s)
<code>-t</code>	Specifies the hex format of the output file (Tektronix)

8.2.2 Conversion Option Reference

`-c list` — Specify section types to be converted

Lists section types to be converted. Argument *list* indicates the section types to be converted; it can be one or more of the following:

- d data sections
- l lit sections (literal: read-only data sections)
- t text sections

Default = `-c tld`.

If you do not specify option `-c`, `elf2hex` converts all three section types.

`-i bank` — Select the number of interleaved banks

Selects the number of interleaved banks. The value of *bank* can be 1, 2, or 4.

Default *bank* = 1.

`-m` — Specify a hex file in Motorola S3 Record format

Directs the linker to generate a hex output file in Motorola S3 Record format. See §A.4.1: *Motorola Hex Record Format* for more information.

See also corresponding hex output-file format options `-q` (Mentor QUICKSIM Modelfile format) and `-t` (Extended Tektronix Hex Record).

Default = `-m`.

Note: Options `-m`, `-q`, and `-t` are mutually exclusive.

`-n word_size` — Specify width of the memory device

Specifies width of the memory device. The value specified in argument `word_size` must be equal to or greater than the `size` specified by `elf2hex` option `-s`. Valid values for `word_size` are 8, 16, or 32.

Default `word_size` = 32.

`-o name` — Specify the name of the generated hex file

Specifies the name of the generated hex file. If multiple hex files are required, each file is named `name`, followed by a suffix that denotes the row, bank, and bit or nybble position of the device.

If you do not specify option `-o`, the name of the hex file(s) is derived from the name of the executable input file. See §A.3: *Hex Output-File Naming Conventions* for a description of hex file-naming conventions.

`-Q` — Suppress copyright message

Suppresses the copyright message. Option `-Q` is `off` by default.

`-q` — Specify a hex file in Mentor QUICKSIM Modelfile format

Directs the linker to generate a hex output file in Mentor QUICKSIM Modelfile format. See §A.4.3: *Mentor QUICKSIM Modelfile Format* for more information.

See also corresponding hex output-file format options `-m` (Motorola S3 Record format) and `-t` (Extended Tektronix Hex Record format).

Default = `-m`.

Note: Options `-m`, `-q`, and `-t` are mutually exclusive.

`-s size` — Identify size of memory device

Identifies the size (length) of the memory devices, using the format `EKxW` or `EMxW`, where *E*, *K*, *M*, and *W* are defined as follows:

- E* Specifies the depth of the device for which the output is being prepared. These are valid values for *E*:
1, 2, 4, 8, 16, 32, 64, 128, 256, or 512 kilobytes (*EK*) or 1 megabyte (*EM*)
- W* Specifies the width of the device in bits. These are valid values for *W*:
4, 6, 8, 16, or 32 bits.

If you do not specify *W*, the linker uses a default width of 8 bits.
- K* Specifies that the depth is represented in kilobytes.
- M* Specifies that the depth is represented in megabytes.

For example, an eight-bit-wide 32K device could be specified as 32Kx8 or simply 32K. If you do not specify option **-s**, the output is one absolute hex file with all data and code at virtual addresses.

When you specify QUICKSIM hex format (with option **-q**), option **-s** is ignored.

-t — Specify a hex file in Extended Tektronix Hex Record format

Directs the linker to generate a hex output file in Extended Tektronix Hex Record format. See §A.4.2: *Extended Tektronix Hex Record Format* for more information.

See also corresponding hex output-file format options **-m** (Motorola S3 Record format) and **-q** (Mentor QUICKSIM Modelfile format).

Default = **-m**.

Note: Options **-m**, **-q**, and **-t** are mutually exclusive.

8.2.3 elf2hex Invocation Examples

This section presents two `elf2hex` invocation examples.

Example 1 The following command converts all sections of input file `a.out`, using Motorola S3 32-bit format for the output record. It also sets the size to eight-bit-wide, 64K memory devices.

```
elf2hex -m -s 64kx8 a.out
```

`elf2hex` generates four hex files, with default file names `a.a00`, `a.a08`, `a.a16`, and `a.a24`. See Figure 8.1.

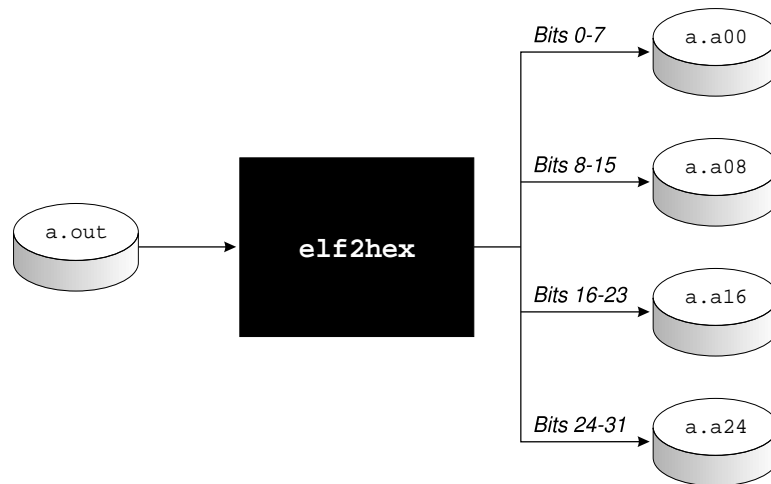


Figure 8.1 *Example of File Partitioning for ROM*

If the data exceeds the specified device size, `elf2hex` generates additional files of the same size, with extensions `.b00`, `.b08`, `.b16` and `.b24`. If the device size is again exceeded, additional sets of files are generated; their file-name extensions begin with the letter `c`, then `d`, and so forth. See §A.3: *Hex Output-File Naming Conventions* for a discussion of hex file naming conventions.

Example 2 The following command generates a hex file in Extended Tektronix Hex format. The output file is named `test.hex`.

```
elf2hex -t -o test a.out
```

If the input file is relocatable, `elf2hex` issues a warning message but continues with the translation, ignoring the relocation information.

In this example, the output file appears as follows:

```
%4A6CE44000250101105E40017E158101180340837002008300030082400301792172450101
%4A6C6440200340839002008300030082410301792172450101030179047245010115836000
%4A6414404001FF82FF160061601560600461616100A4FF61FD15828201A800801970400101
%4A65F440600300790172450101A0FF00FE7040010124797E01247F7F798379790225797901
%4A68344080CE0087793E00807FC000007A157E0100030279009279797FCE0081792479817F
%4A691440A0147E7E798379790225797901CE0087793600007FC000007A157F8100257D7D68
%4A63E440C003007600817976021477797D0300780003007900147979771E00787915767601
%32605440E04179760AACFF79F870400101C0000080157D7D68
%0A81A44000
```

Figure 8.2 Sample Output of Extended Tektronix Hex Format

You can partition the translated object file into a set of files, each suitable for programming one of the read-only memories in a multi-PROM system.

For details on hex file formats, see §A.4: *Hex File Formats*.

8.3 Using the `elfdump` Binary Dump Utility

This section describes the ELF binary dump utility, `elfdump`, a stand-alone tool for dumping information from ELF object and executable files.

Object files compiled with High C/C++ conform to the Executable and Linking Format (ELF). After compiling, you can use the `elfdump` utility to produce a structure dump of some or all of the resulting ELF object files and shared libraries (DLLs). You can also use `elfdump` to dump executable files.

Note: You cannot apply `elfdump` directly to archive libraries. Extract the objects with `ar`, then apply `elfdump` to them.

Figure 8.3, *Structure Dumps of ELF Object and Executable Files* shows some typical `elfdump` outputs.

8.3.1 Invoking `elfdump`

You invoke `elfdump` as follows:

```
elfdump [options] file[ file ...]
```

These are the command-line arguments:

- option* Options that specify the category of information to be dumped. See §8.3.2: *elfdump* Command-Line Options for a complete list of options.
- file* ELF object file, shared object file, or executable file. This is the only required argument.

The file names are separated from one another by whitespace.

Object File Dump	Shared Object or Executable Dump
ELF Header	ELF Header
Sections...	Program Headers
Relocation Tables (optional)	Sections...
Symbol Table	Relocation Tables (optional)
	Symbol Table and *Dynamic Symbol Table
	*Dynamic Table
	*Dump of Import Section (OS/2 only)
	*Hash Buckets

*Present only in a dynamically linked executable

Figure 8.3 Structure Dumps of ELF Object and Executable Files

8.3.2 `elfdump` Command-Line Options

You can list `elfdump` options separately with individual hyphens (separated by whitespace), or all together (not separated by whitespace) following a single hyphen. For example, these `elfdump` commands are equivalent:

```
elfdump -s -r -p obj_file.o
elfdump -srp obj_file.o
```

Table 8.3 lists the command-line options available for `elfdump`.

Table 8.3 `elfdump` Command-Line Options

Option	Information Dumped
-C	<code>Cppfilt</code> style output
-D	Dynamic table
-h	ELF header
-H	Hash table contents
-i	OS/2-style import sections
-o	<code>Cppfilt</code> style output with ordinals
-p	Program header
-r	Relocation header sections
-s	Section headers
-t	Symbol table(s)
-x	OS/2-style exports
-X	Section contents of debug/line sections
-z	Section contents

Defaults The default option setting is **-DhHprst**. You cancel the default setting by specifying any other command-line option or combination of command-line options.

By default, `elfdump` sends the dump to standard output. If you want to save the dump, redirect it to a file.

Working with PROMs and Hex Files

This appendix, which provides background information about hex files and PROMs, contains the following sections:

§A.1: *Hex File Overview*

§A.2: *PROM Device Model*

§A.3: *Hex Output-File Naming Conventions*

§A.4: *Hex File Formats*

For information about using linker option `-x` to generate hex files, see §3.2: *Linker Options Reference*. For information about the ELF-to-hex conversion utility, `elf2hex`, see §8.2: *Using the elf2hex Conversion Utility*.

A.1 Hex File Overview

The term *hex file* is short for *hexadecimal record file*. A hex file is a representation in ASCII format of a binary image file. You can use linker option `-x` to generate a hex file. You can also generate a hex file from an existing executable by invoking the `elf2hex` conversion utility.

A hex file consists of pairs of ASCII characters, with each pair representing an eight-bit byte. For example, a byte with a value of 7E in a binary file is represented in a hex file as ASCII 7 followed by ASCII E — which requires two bytes. With devices that are four bits wide, each nybble in the hex file is denoted as a two-digit hex pair, with the first digit being zero (0). The contents of the hex file are divided into records, with one record per line.

A hex file is generally more than twice the size of the corresponding binary file. Though hex files are larger than binary files, they are generally easier to work with. For example, most computers transfer ASCII data more reliably over a serial line, such as when downloading an executable to a debugger or PROM burner.

Hex files can be configured to be directly loaded into a single PROM device. This might require generating multiple hex files from a single byte stream.

A.2 PROM Device Model

When the linker generates hex files (because you specified linker option `-x` or invoked the `elf2hex` utility), it assumes a specific PROM model. This section describes the assumed device model.

A.2.1 PROM Device Characteristics

These are key characteristics of PROM devices:

- Width* • A PROM device has a width, which is the number of bits that each unit of data occupies in the device. Supported widths are 4, 8, 16, or 32 bits.
- Size or length* • A PROM device has a finite size or length. This is the maximum number of bytes the device can hold. The length must be an integral power of two (for example, 32K).
- Word width and banks* • PROM devices can be combined into words. The word width can be 8, 16, or 32 bits. This value represents the number of bits accessible in a single read or write to the group (or bank) of devices. The default word size is 32 bits.
- Multiple banks* • PROM devices can be further configured into multiple banks. If you specify two banks, every other word unit is placed in each bank. If you specify four banks, every fourth word is placed in a bank, and so on. You can define one, two, or four banks. The default is one bank.

A.2.2 A Single PROM Bank

Device width and word width together determine how a bank of devices is configured. For example, assume a device width of 8 bits and a word width of 32 bits. Arrays of words are stored in four parallel devices, with each device containing every fourth data byte, as shown in Figure A.1.

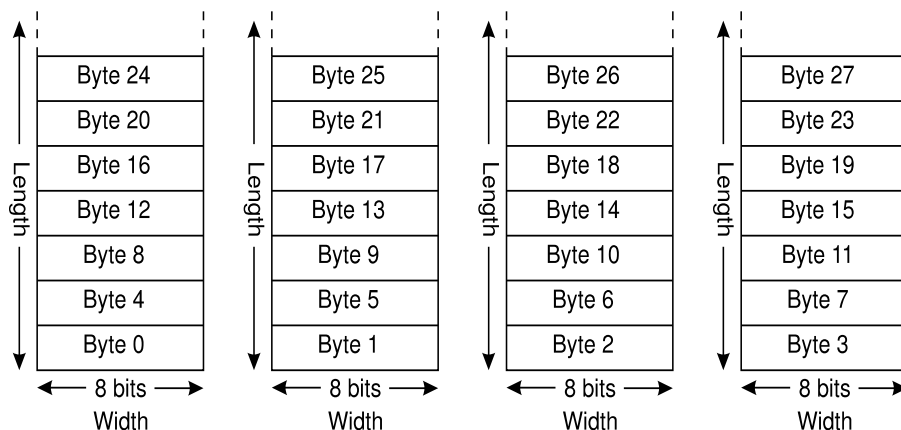


Figure A.1 A Single Bank of PROM Devices

In the case of a single PROM bank (of 8-bit devices and 32-bit words), four hex files are generated — one file per device. The files are named according to the conventions described in §A.3: *Hex Output-File Naming Conventions*.

A.2.3 Multiple PROM Banks

In a multi-bank PROM configuration, data ordinarily placed into a single device is interleaved across two or four devices. For example, in a two-bank configuration (still assuming devices are eight bits wide), one bank contains even-numbered words, and the other contains odd-numbered words, as shown in Figure A.2.

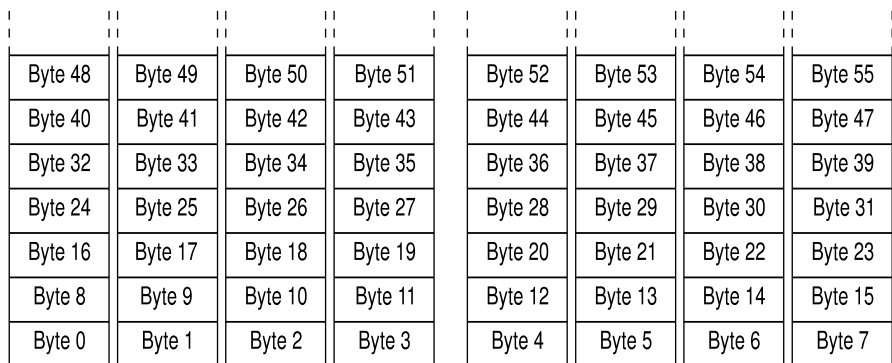


Figure A.2 Two Banks of PROM Devices

In a four-bank configuration, each device has every fourth byte; in an eight-bank configuration, each device has every eighth byte, and so on.

A.3 Hex Output-File Naming Conventions

When you generate hex output files (using linker option **-x** or the **elf2hex** utility), one or more files are required, depending on the number of PROM banks you specified and the size (length) of the targeted memory device. This section describes naming conventions for the hex output file(s).

For a discussion of PROMs and related terminology, see §A.2: *PROM Device Model*.

A.3.1 Single Hex File (No Size or Number of Banks Specified)

If you do not specify a size and a bank number when you generate the hex file, the file has the same name as the executable file, but with the suffix **.hex**. For example, given this command:

```
ld -x file
```

the linker produces the hex output file **file.hex**.

Note: If you use linker option **-xo** or **elf2hex** option **-o** to specify the name of the generated hex file, the file will have the specified name. The suffix **.hex** is not appended in this case.

A.3.2 Multiple Hex Files

When multiple hex files are required, the file names are distinguished by suffixes that denote the row, bank (if more than one is specified), and bit or nybble position of the device. The naming convention depends on the number of banks you specify.

A.3.2.1 One Bank Specified

If more than one hex file is required, and you specify only one bank, each hex file has a suffix composed of two elements, as shown here:

`file.{row}{bit}`

where the suffix elements have the following meanings:

- | | |
|------------|--|
| <i>row</i> | A single letter that denotes the device row.
a denotes the first row, b denotes the second row, and so on.

Multiple rows occur when the data size exceeds the device length specified (linker option -xs or elf2hex option -s). |
| <i>bit</i> | A two-digit decimal integer that denotes a bit position, starting from the right-most byte of a word: <ul style="list-style-type: none"> 00 Denotes the right-most byte 08 Denotes the second byte from the right 16 Denotes the third byte from the right 24 Denotes the fourth byte from the right (which happens to be the first byte for a configuration with 32-bit words) This number is based on the specified device width (linker option -xn or elf2hex option -n). |

Based on these considerations, the file `file.a08` would contain the bytes corresponding to the first instance of the second PROM (in an eight-bit-wide configuration).

A.3.2.2 Multiple Banks Specified

If more than one hex file is required, and you specify multiple banks, each hex file has a suffix composed of three elements, as follows:

`file.{row}{bank}{nybble}`

where the suffix elements have the following meanings:

<i>row</i>	<p>A single letter that denotes the device row. a denotes the first row, b denotes the second row, and so on.</p> <p>Multiple rows occur when the data size exceeds the device length specified (linker option -xs or elf2hex option -s).</p>
<i>bank</i>	<p>A single letter that denotes the bank. a denotes the first bank, b denotes the second, and so on.</p>
<i>nybble</i>	<p>The bit position divided by 4 (for example, 24 becomes 6, 16 becomes 4, and so on).</p> <p>This number is based on the specified device width (linker option -xn or elf2hex option -n).</p> <p>The nybble position (rather than the bit position) is used to ensure that the suffix contains no more than three characters, to conform to DOS file-naming conventions.</p>

A.4 Hex File Formats

Sections §A.4.1 through §A.4.3 summarize supported hex formats.

A.4.1 Motorola Hex Record Format

Motorola hex records, known as *S-records*, are identified by one of the following pairs of start characters:

S0	Optional sign-on record, usually found at the beginning of the data file
S1, S2, or S3	Data records
S7, S8, or S9	End-of-file (EOF) records

Note: Hex records for processors running ELF applications use the S3 and S7 start/EOF characters. This is known as *S3 format*.

A.4.1.1 Data Records (Start Characters S1 — S3)

Motorola data records contain the following components:

- header characters
- byte-count field
- address field
- data field
- checksum field

Figure A.3 illustrates a Motorola S3 data record.

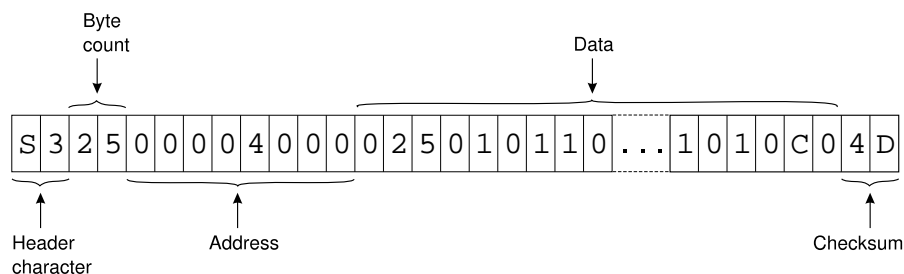


Figure A.3 Data Record in Motorola S3 Record Format

Header characters Each data record begins with a header that contains a pair of start characters, S1, S2, or S3. The start characters specify the length of the data record's

address field. Table A.1 lists the address field length specified by each pair of start characters and the associated end-of-file character(s).

Table A.1 Start and EOF Characters in Motorola S3 Record Format

Start Characters	End-of-File Characters	Address Field Length
S1	S9	Four characters (two bytes)
S2	S8 or S9	Six characters (three bytes)
S3	S7	Eight characters (four bytes)

Byte-count field The two-character byte-count field represents the number of eight-bit data bytes, starting from the byte count itself and ending with the last byte of the data field.

Note: This is a *byte* count, not a *character* count. The character count is twice the value of the byte-count field.

Address field The address field is four, six, or eight characters long. It contains the physical base address where the data bytes are to be loaded.

Data field The data field contains the actual data to be loaded into memory. Each byte of data is represented by two hexadecimal characters.

Checksum field The checksum field is two characters long. It contains the one's complement of the sum of the bytes in the record, from the byte count to the end of the data field.

A.4.1.2 End-of-File Records (S7 — S9)

End-of-file (EOF) records begin with a header containing a pair of start characters (S7, S8 or S9). The pair used depends on the number of bytes in the address field (see Table A.1, *Start and EOF Characters in Motorola S3 Record Format*).

Following the header is a byte count (03), an address (0000), and a two-character checksum. There is no data field in EOF records. The EOF record contains the entry-point address.

A.4.2 Extended Tektronix Hex Record Format

The Extended Tektronix hex record format has three types of records:

Symbol	Holds program section information
Termination	Indicates the end of a module
Data	Contains a header field, a length-of-address field, a load address, and the actual object code

A.4.2.1 Data Record

Extended Tektronix Hex Format data records contain the following components:

- header field
- length of address field
- load address
- object code

Figure A.4 illustrates the format of an Extended Tektronix Hex Format data record.

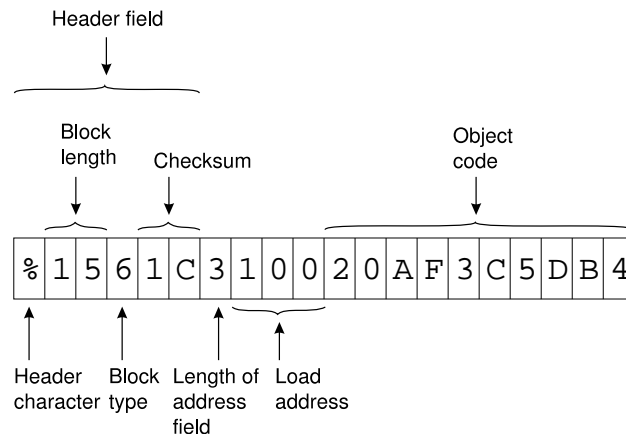


Figure A.4 Data Record in Extended Tektronix Hex Format

Header field A header field starts with a special *header character* and contains *block length*, *block type*, and *checksum* values.

Table A.2 lists and describes the components of an Extended Tektronix Hex Format data record's header field. The *width* of a component is the number of ASCII characters it contains.

Table A.2 Header Field Components in Extended Tektronix Hex Format Data Record

Header Field Component	Width	Description
Header character	1	% character; indicates records in Extended Tektronix Hex format
Block length	2	Number of characters in record, minus the %
Block type	1	Type of record: 6 = data 3 = symbol 8 = termination
Checksum	2	A two-digit hexadecimal sum of all the values in the record, except the % and the checksum itself

Length of address field This field is a one-digit hexadecimal integer representing the length of the address field in bits. A 0 (zero) signifies an address-field length of 16 bits.

Load address The load address specifies where the object code will be located. This is a variable-length number that can contain up to 16 characters.

Object code The remaining characters of the data record contain the object code. Each byte of data is represented by two hexadecimal characters.

A.4.3 Mentor QUICKSIM Modelfile Format

A Mentor QUICKSIM modelfile defines the contents of ROM or RAM devices implemented in a logic design. The format of the modelfile follows these rules:

- All data and addresses must be in hexadecimal format.
- All comments are preceded by a pound sign (#).
- The contents of a single memory location are specified as *address / data*. See the following example.
- The contents of a range of memory locations are specified as *(low_address - high_address) / data*. See the following example.
- Letters can be uppercase or lowercase.

Example This example is a portion of a QUICKSIM file.

```
...
00004410 / 30000000 ;
00004411 / 000107FC ;
00004412-0000441B / 00010838 ;
0000441C / 000107F4 ;
0000441D / 00010838 ;
0000441E / CA3107EC ;
0000441F / 89E10FAC ;
00004420 / 89010A20 ;
00004421 / F0F10F60 ;
00004422-00004431 / 00010A20 ;
00004432 / 00010DDC ;
...
```

Note: The device-length specification (linker option **-xs** or **elf2hex** option **-s**) is ignored for QUICKSIM records.

B

OS/2 Module-Definition Files

This appendix, which discusses OS/2 module-definition files and module statements, contains the following sections:

§B.1: *Module-Definition Files*

§B.2: *Module Statements*

§B.3: *Module Statement Reference*

§B.4: *Sample Module-Definition Files*

Note: The information in this appendix applies *only* if you are developing OS/2 applications for the PowerPC.

B.1 Module-Definition Files

When targeting the PowerPC IBM Microkernel and OS/2, the ELF linker/locator can accept a *module-definition* file. Module-definition files, which usually have the extension `.DEF`, provide the linker with information about the application or dynamic link library (DLL) being linked. To specify a module-definition file, use linker option `-i`. (See Chapter 3: *Linker Command-Line Options* for details.)

A module-definition file describes various characteristics of an executable or DLL: what it is (an application or a DLL), its name, attributes, exported functions, imported functions, and so on.

B.2 Module Statements

Module-definition files contain one or more *module statements*. A module statement consists of a module-definition keyword, its required parameters (if any), and optional parameters as needed. Table B.1, *Module-Definition Keywords* summarizes the module statement keywords supported by the

ELF linker/locator. Individual descriptions of the module statement keywords begin in §B.3: *Module Statement Reference*.

Note: Table B.2, *OS/2 2.x Legacy Support Keywords* lists obsolete keywords that the ELF linker/locator will accept *but ignore*; these obsolete keywords are accepted for legacy support only.

Including comments To include a source-level comment in a module-definition file, begin the comment with a semicolon (;).

B.2.1 Supported OS/2 Keywords

The ELF linker/locator supports the following keywords in PowerPC IBM Microkernel and OS/2 module-definition files:

Table B.1 *Module-Definition Keywords*

Keyword	What It Does
BASE	Specifies default starting address for writable data
CODE	Gives default attributes for code segments
DATA	Gives default attributes for data segments
EXETYPE	Identifies operating system
EXPORTS	Defines exported functions
HEAPSIZE	Specifies local heap size
IMPORTS	Defines imported functions
LIBRARY	Names a dynamic-link library
NAME	Names an application
ROBASE	Specifies default starting address for code
SEGMENTS	Gives attributes for specific segments

B.2.2 Legacy Support: OS/2 2.x Keywords

The ELF linker/locator can *read* module-definition files written for OS/2 2.x, but it *responds* only to the keywords specific to PowerPC IBM

Microkernel and OS/2. In providing this “legacy support” for OS/2 2.x, the linker accepts *but ignores* the following OS/2 2.x keywords in module-definition files:

Table B.2 OS/2 2.x Legacy Support Keywords

OS/2 2.x Keyword	Purpose
OLD	Preserves import information
PHYSICAL_DEVICE	Identifies the module as a physical device driver
PROTMODE	Specifies that module runs only in protected mode
SESSION	Defines type of execution environment
STACKSIZE	Specifies local stack size
STUB	Adds DOS executable file to beginning of module
VERSION	Specifies a version number for the module
VIRTUAL_DEVICE	Identifies the module as a virtual device driver

B.3 Module Statement Reference

BASE

Syntax **BASE** = *address*

Description The **BASE** statement specifies the default starting address for writable data. Required parameter *address* is any valid C constant.

Note: To specify the starting address for read-only sections (code sections), see the **ROBASE** command.

Example Specify preferred load address for writable data as 128K:

```
BASE = 131072    ; decimal for 128K
```

CODE

Syntax **CODE** [*attribute* ...]

Description The **CODE** statement defines the default attributes for code segments within a library or application.

The optional *attribute* parameters specify the following:

- whether the code segment can be both executed and read
- whether the code segment is readable and writable
- the read / write / execute status of the code segment
- whether the READWRITE code segment is shared among processes

One or more of the following *attribute* options can appear after the **CODE** statement, in any order. (Note: In the following table, a ✓ in the **Def** column indicates the *default option*.)

Table B.3 *Optional Attributes for the CODE Statement*

Attribute	Options	Def	Description
Read/ write/ execute status (choose one)	EXECUTEONLY		The code segment can only be executed.
	EXECUTEREAD or EXECUTE	✓	The code segment is readable and executable.
	READONLY or READ		The code segment is read-only.
	READWRITE or WRITE		The code segment is readable and writable.
Share status (choose one)	SHARED		One copy of the READWRITE code segment is loaded and shared among all processes that access the module. Signified by setting the PF_S bit (0x01000000) of the associated program header.
	NONSHARED	✓	The READWRITE code segment cannot be shared; it must be loaded separately for each process.

Attribute Rules Use only one option from each attribute set. If you do not specify an option, the linker supplies the default for that set.

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the following attribute options for the **CODE** statement:

Discard status	DISCARDABLE or NONDISCARDABLE
I/O privilege	IOPL or NOIOPL
Load status	LOADONCALL or PRELOAD
Move status	FIXED or MOVABLE
Swap status	RESIDENT or SWAPPABLE

Example This application **CODE** statement sets defaults for module code segments so they are read-only and cannot be shared (NONSHARED is the default supplied by the linker for applications):

```
CODE READ
```

DATA

Syntax **DATA**[*attribute* | *attribute* | ...]

Description The **DATA** statement defines the default attributes for data segments within a library or application. The optional *attribute* parameters specify the following:

- whether the data segment can be both executed and read
- whether the data segment is readable and writable
- the read/write/execute status of the data segment
- whether the READWRITE data segment is shared among processes

One or more of the following attribute options can appear after the **DATA** statement, in any order. (Note: In the following table, a ✓ in the **Def** column indicates the *default option*.)

Table B.4 *Optional Attributes for the DATA Statement*

Attribute	Options	Default	Description
Read/write/execute status (choose one)	EXECUTEONLY		The data segment can only be executed.
	EXECUTEREAD or EXECUTE	✓	The data segment is readable and executable.
	READONLY or READ		The data segment is read-only.
	READWRITE or WRITE		The data segment is readable and writable.
Share status (choose one)	SHARED		One copy of the READWRITE data segment is loaded and shared among all processes that access the module. Signified by setting the PF_S bit (0x01000000) of the associated program header.
	NONSHARED	✓	The READWRITE data segment cannot be shared; it must be loaded separately for each process.

Attribute Rules Use only one option from each attribute set. If you do not specify an option, the linker supplies the default for that set.

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the following attribute options for the **DATA** statement:

Automatic data segment	NONE, SINGLE, or MULTIPLE
Discard status	DISCARDABLE or NONDISCARDABLE
I/O privilege	IOPL or NOIOPL
Load status	LOADONCALL or PRELOAD
Move status	FIXED or MOVABLE
Swap status	RESIDENT or SWAPPABLE

Example This **DATA** statement sets defaults for module data segments so they are read-only, and can be shared.

```
DATA READ SHARED
```

DESCRIPTION

Syntax **DESCRIPTION** *'string'*

Description The **DESCRIPTION** statement inserts its argument *string* into the library or application as a **.comment** section. The *string* is a one-line string enclosed in single quotation marks.

The **DESCRIPTION** statement is useful for placing copyright, source-control, or acknowledgement information into a library or application.

Example This **DESCRIPTION** statement inserts a **.comment** section containing the text Copyright 1996 MetaWare Incorporated:

```
DESCRIPTION 'Copyright 1996 MetaWare Incorporated'
```

EXEFORMAT

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **EXEFORMAT** statement.

EXETYPE

Syntax **EXETYPE** *attribute*

Description The optional **EXETYPE** statement specifies under which operating system the shared library (DLL) or application runs.

One of the following executable-type attributes appears after the **EXETYPE** statement:

Attribute (Use One)	Description
AIX	AIX applications and shared libraries
OS2	OS/2 applications and shared libraries (DLLs)
PN	Personality-neutral applications and shared libraries (operating-system independent)
SVR4	UNIX SVR4 applications and shared libraries

Attribute (Use One)	Description
UNKNOWN	Other applications and shared libraries

The **EXETYPE** statement sets bits in the header that identify the operating-system type; operating-system loaders check these bits. Using the **EXETYPE** statement provides an extra means of ensuring that the user does not run the DLL or application on an incorrect operating system.

Example This **EXETYPE** statement specifies that the application or library being defined runs under the OS/2 operating system:

```
EXETYPE OS2
```

EXPORTS

Syntax **EXPORTS**

```
entry_name[ = intrnl_name ][ @ordnl_number ][ NONAME ]
```

Description OS/2 supports an ordinal-number based facility for exporting or importing names across module boundaries.

The **EXPORTS** statement defines the names and attributes of the functions (in the module being defined) that are exported to other modules. In most cases, the **EXPORTS** statement is meaningful only for functions within DLLs.

The **EXPORTS** keyword indicates the beginning of the *export definitions*, with one definition per line. You must provide an export definition for each dynamic-link routine that you want available to other modules at run time.

The *entry_name* parameter is required, but the parameters *intrnl_name* and *ordnl_number* and the keyword **NO_NAME** are optional.

Parameter	Description
<i>entry_name</i>	Name of the export function as known to other modules
<i>intrnl_name</i>	Name of the export function as it appears within the module; by default, this internal name is the same as <i>entry_name</i>
<i>ordnl_number</i>	Ordinal number to be associated with the function

Parameter	Description
NONAME	Specifies that the linker must export the function by ordinal value only

Note: If you use the optional *ordnl_number* parameter, you can invoke the function's entry point by name or number.

Legacy Support For legacy support purposes only, the ELF linker/locator accepts the following **EXPORTS** statement format:

```
EXPORTS
ent [=int] [@ord[ RESIDENTNAME ] ] [no_name] [p_words]
```

However, the ELF linker/locator *ignores* the RESIDENTNAME option and the *p_words* parameter.

Example This **EXPORTS** statement defines four export functions:

```
EXPORTS
WriteToPackage = writelin @8
TextIn        = string2
TestForValid  @3
open          @9 NONAME
```

These export functions can be accessed as follows:

- WriteToPackage can be accessed either by its exported name or by an ordinal number. In the module's own source code, WriteToPackage is actually defined as writelin.
- TextIn, defined in the module's source code as string2, can be accessed only by its exported name.
- TestForValid, defined with the same name in the module's source code, can be accessed by its exported name or by an ordinal number.
- open can be referenced *only* by its ordinal number, 9.

HEAPSIZE

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **HEAPSIZE** statement.

IMPORTS

Syntax **IMPORTS**
 [*intrnl_name* =] *module_name.entry_point*

Description The **IMPORTS** statement complements the **EXPORTS** statement.

The **IMPORTS** statement defines the names and attributes of the application's or library's imported functions (those exported from other modules with an **EXPORTS** statement).

Note: The linker typically uses an import library to resolve external references to dynamic-link symbols. The **IMPORTS** statement provides an alternative technique for resolving such references within a module.

The **IMPORTS** keyword indicates the beginning of the *import definitions*, with one definition per line. You must provide an import definition for each dynamic-link routine you want to import from another module at run time; each import definition corresponds to one particular function.

The *module_name* and *entry_point* parameters are required, but the *intrnl_name* parameter is optional.

	Parameter	Description
<i>Optional</i>	<i>intrnl_name</i>	Name that the importing module uses when calling the function
<i>Required</i>	<i>module_name</i>	Name of the application or library that contains the function
<i>Required</i>	<i>entry_point</i>	Function to be imported; this can be a name or an ordinal value

An imported function has (potentially) three distinct names:

- the name that the exporting module uses to *define* the function (**EXPORTS** parameter *intrnl_name*)
- the name used as an entry point *between* modules (**EXPORTS** parameter *entry_name*, **IMPORTS** parameter *entry_point*)
- the name that the importing module uses to *call* the function (**IMPORTS** parameter *intrnl_name*)

If neither the importing module nor the exporting module uses the optional *intrnl_name* parameter, the function has only one name.

If either the importing module or the exporting module uses the optional *intrnl_name* parameter, the function can have two or three different names.

Example This **IMPORTS** statement defines three import functions:

```
IMPORTS
    samples1.TestRead
    ReadData = samples3.TestInputData
    WriteData = samples3.5
```

These import functions reside in the following modules:

- TestRead resides in module `samples1`.
- The second import function, known to the importing executable as `ReadData`, resides in module `samples3`, where it is defined as `TestInputData`.
- The third import function, known to the importing executable as `WriteData`, resides in module `samples3`, where it was assigned ordinal number 5.

LIBRARY

Syntax **LIBRARY** [*library_name*] [*initialization*] [*termination*]

Description The **LIBRARY** statement identifies the module as a dynamic-link library. The optional parameters specify the following:

- *library_name* the name of the library
- *initialization* the library initialization required
- *termination* the library termination required

One or more of the following options can appear after the **LIBRARY** statement, in any order. (Note: In this table, a ✓ in the **Def** column indicates the *default option*.)

Parameter	Option	Def	Description
<i>library_name</i>	N/A		Specifies the name of the library, as known by the operating system
<i>initialization</i>	INITNONE		No initialization routine
	INITGLOBAL	✓	Calls the initialization routine when the library module is initially loaded into memory
	INITINSTANCE		Calls the initialization routine each time a new process gains access to the library
	INITTHREAD		Calls the initialization routine each time a thread is created
<i>termination</i>	TERMNONE		No termination routine
	TERMGLOBAL	✓	Calls the termination routine when the library module is unloaded from memory
	TERMINSTANCE		Calls the termination routine each time a new process gives up access to the library

Parameter	Option	Def	Description
	TERMTHREAD		Calls the termination routine each time a thread gives up access to the library

The *library_name* parameter can be any valid file name. This name appears in the SONAME record of the dynamic linkage table (.dynamic). If you do not use *library_name*, the name of the library is the name of the executable, with the extension removed.

Restrictions The **LIBRARY** and **NAME** statements are mutually exclusive: if the module-definition file contains a **LIBRARY** statement, the **NAME** statement cannot appear.

Example This **LIBRARY** statement assigns the name `mylib` to the dynamic-link library and specifies that library initialization be performed each time a new process gains access:

```
LIBRARY mylib INITINSTANCE
```

By default, because a *termination* parameter is not specified, the library termination routine is called only when the library module is unloaded.

NAME

Syntax **NAME** [*application_name*][*application_type*]

Description The **NAME** statement identifies the executable file as an application.

The optional *application_name* parameter, which specifies the name of the application, can be any valid file name. If you do not use *application_name*, the name of the application is the name of the executable, with the extension removed.

Parameter	Option	Description
<i>application_name</i>	N/A	Specifies the name of the application, as known by the operating system

Parameter	Option	Description
<i>application_type</i>	WINDOWAPI	Presentation Manager application
	WINDOWCOMPAT	Application compatible with Presentation Manager
	NOTWINDOWCOMPAT	Application not compatible with Presentation Manager

The optional *application_type* parameter defines the type of the application.

- If the *application_type* is WINDOWAPI, the application uses the Presentation Manager (PM) API and must be executed in the PM environment.
- If the *application_type* is WINDOWCOMPAT, the application can run inside the PM, or it can run in a separate screen group. An application can be WINDOWCOMPAT if it uses the proper subset of keyboard, video, and mouse functions supported in PM applications.
- If the *application_type* is NOTWINDOWCOMPAT, the application must operate in a separate screen group from the Presentation Manager.

Restrictions If the **NAME** statement appears in the module-definition file, the **LIBRARY** statement cannot appear there.

If the module-definition file contains no **NAME** or **LIBRARY** statement, the system assumes that the file describes an application.

Example This **NAME** statement assigns the name `myprog` to the application and specifies that the application is PM-compatible:

```
NAME myprog WINDOWCOMPAT
```

OLD

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **OLD** statement.

PHYSICAL DEVICE

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **PHYSICAL DEVICE** statement.

PROTMODE

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **PROTMODE** statement.

ROBASE

Syntax **ROBASE** = *address*

Description The **ROBASE** statement specifies the default starting address for code. Required parameter *address* is any valid C constant.

Example Specify preferred load address for code as 128K:

```
ROBASE = 131072    ; decimal for 128K
```

SEGMENTS

Syntax **SEGMENTS**
['] *seg_name* ['] [*attribute ...*]

Description The **SEGMENTS** statement defines the attributes of segments in the library or application, on a segment-by-segment basis.

The **SEGMENTS** keyword indicates the beginning of the *segment definitions*, with one definition per line. The *seg_name* parameter is required in every segment-definition, but the *attribute* parameters are optional.

seg_name If *seg_name* conflicts with a module-definition keyword (such as **CODE** or **DATA**), you must use single quote characters (') to enclose *seg_name*.

attribute The optional *attribute* parameters, defined in Table B.5, *Optional Attributes for the SEGMENTS Statement*, specify the following segment attributes:

- whether the segment can be both executed and read
- whether the segment is readable and writable
- the read / write / execute status of the segment
- whether the READWRITE segment is shared among processes

Note: The *attribute* settings specified in a **SEGMENTS** statement override any applicable default attributes set in a **CODE** or **DATA** statement.

One or more of the following attribute options can appear after the **SEGMENTS** statement, in any order. (Note: In this table, a ✓ in the **Def** column indicates the *default option*.)

Table B.5 *Optional Attributes for the SEGMENTS Statement*

Attribute	Options	Def	Description
Read/write/execute status (choose one)	EXECUTEONLY		The segment can only be executed.
	EXECUTEREAD or EXECUTE	✓	The segment is readable and executable.
	READONLY or READ		The segment is read-only.
	READWRITE or WRITE		The segment is readable and writable.
Share status (choose one)	SHARED		One copy of the READWRITE data segment is loaded and shared among all processes that access the module. Signified by setting the PF_S bit (0x01000000) of the associated program header.
	NONSHARED	✓	The READWRITE data segment cannot be shared; it must be loaded separately for each process.

Attribute Rules Use only one option from each attribute set. If you do not specify an option, the linker supplies the default for that set.

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the following attribute options for the **SEGMENTS** statement:

Discard status	DISCARDABLE or NONDISCARDABLE
I/O privilege	IOPL or NOIOPL
Load status	LOADONCALL or PRELOAD
Move status	FIXED or MOVABLE
Swap status	RESIDENT or SWAPPABLE
Class name	CLASS 'class'

Example This **SEGMENTS** statement provides three segment definitions:

```

SEGMENTS
    segment1 READ WRITE
    segment2 EXECUTE SHARED
    segment3 READ SHARED

```

These segments have the following attributes:

- segment1 is readable and writable.
- segment2 is executable and shared.
- segment3 is readable and shared.

SESSION

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **SESSION** statement.

STACKSIZE

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **STACKSIZE** statement.

STUB

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **STUB** statement.

VERSION

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **VERSION** statement.

VIRTUAL DEVICE

Legacy Support For legacy support purposes only, the ELF linker/locator accepts *but ignores* the **VIRTUAL DEVICE** statement.

B.4 Sample Module-Definition Files

The following sections give some basic examples of module-definition files. These examples contain only a few of the possible statements you might want to use. For more details see the preceding sections in this appendix.

B.4.1 Module-Definition File for an Application

Module-definition files (usually ending with a `.def`) for applications are typically very simple. This is an example of a module-definition file for an application:

```
NAME hello.exe WINDOWCOMPAT
```

In this example, the **NAME** statement identifies the executable file as an application named `hello.exe`. The **WINDOWCOMPAT** option tells the linker that the application is compatible with the Presentation Manager.

Note: The `-o` command line option overrides the declaration of a file name for the application.

B.4.2 Module-Definition File for a DLL

Module-definition files for DLLs contain information describing the characteristics of the library. This is an example of a module-definition file for a DLL:

```

LIBRARY My_Lib INITINSTANCE TERMINSTANCE
IMPORTS
;*****
;* Functions imported from GOO *
;*****
GOO2          = GOO.992
GOO3          = GOO.993
EXPORTS
;*****
;* Functions exported to other modules *
;*****
moo=hoo1      @2000
hoo2          @2001
hoo3          @2010
hoo4          @2011

```

In this example, the **LIBRARY** statement identifies the module as a dynamic-link library called `My_Lib`. The **INITINSTANCE** option specifies that the initialization routine is called each time a process gains access to the library. The **TERMINSTANCE** option specifies that the termination routine is called each time a process gives up access.

The **IMPORTS** symbol declares the functions that are imported from another module, in this case module `GOO`, while the **EXPORTS** symbol declares functions that are available to other modules.

Index

A

linker option	-A — specify linker command file	18
	a.out, default name of linker output file	29
assigning output sections to	absolute addresses	53
SECTIONS keyword ADDRESS — assign	absolute starting address for an output section or	
	group	55
archiver option -r — replace (or	add) archive library members	79
linker command RESADD — reserve memory		
location by	address	46
SECTIONS keyword	ADDRESS — assign absolute starting address for	
	an output section or group	55
explicit virtual	address assignment	63
alter origin	address at load time	21
ROBASE statement — specify the default		
starting	address for code sections (read-only sections)	115
BASE statement — specify default starting	address for writable data sections	103
linker option -B base — specify origin	address in hexadecimal	19
linker option -B start_addr — specify origin	address in hexadecimal	23
	address_spec specifier in a SECTIONS command ...	55
assigning output sections to absolute	addresses	53
"next-fit"	algorithm for locating relocatable sections	53
SECTIONS keyword	ALIGN — specify alignment boundary for output	
	section	55
linker option -B hardalign — force each		
output segment to	align on a page boundary	20
SECTIONS keyword	ALIGN_INPUT — specify alignment boundary	
	for input section relative to output section	55
SECTIONS keyword ALIGN_INPUT — specify	alignment boundary for input section relative to	
	output section	55
	alignment boundary for output section	55
SECTIONS keyword ALIGN — specify	all_archive — extract all members of an archive	
linker option -B	library	19
	allocable"	21
designate sections as "not		
SECTIONS keyword GROUP — force a set of	allocated contiguously in a specified order	53
output sections to be	allocated in virtual memory	29
memory map listing file — explains how	ambiguous section names	57
sections are		
resolving		

	appending flags or attributes to linker option -x	31
EXETYPE statement — specify operating system	application runs	107
where the shared library (DLL) or	applications	11
linker options for developing embedded	ar — management utility that groups object files	
archiver	into an archive library	77
processing	archive libraries	44, 72
order of	archive libraries on the linker command line	8
linker option -B all_archive — extract all	archive library	19
members of an	archive library	78
member file contained in	archive library members and archive symbol tables .	79
manipulating	archive library, defined	2
invoking the	archiver	78
specifying	archiver ar — management utility that groups	
	object files into an archive library	77
	archiver command-line arguments in an "at" file	80
	archiver command-line options summary	79
	archiver error messages	78
linker command	ARGS — specify linker command-line arguments	
	in command file	40
linker commands and	arguments in a command file	37
specifying archiver command-line	arguments in an "at" file	80
specifying linker command-line	arguments in an "at" file	17
linker command ARGS — specify linker	arguments in command file	40
command-line	arguments, defined	7
linker command-line	ASCII format	80
elf2hex — utility to convert binary object	assign a type to an output section	54
file to hexadecimal	assign absolute starting address for an output	
SECTIONS keyword TYPE — explicitly	section or group	55
SECTIONS keyword ADDRESS —	assigning output sections to absolute addresses	53
specifying linker command-line arguments in	"at" file	17
an	"at" file	80
specifying archiver command-line arguments in	(AT&T-style command file) for linker directives	29
an	AT&T-style input map file directives	60
linker option -M — read input map file	AT&T-style input map-file directives summary	14
	attributes for code segments	104
CODE statement — define default	attributes for data segments	105
DATA statement — defines default	attributes of functions exported to other modules ..	108
EXPORTS statement — defines names and		

IMPORTS statement — define names and	attributes of functions imported	110
SEGMENTS statement — define	attributes of individual segments	115
appending flags or	attributes to linker option -x	31

B

linker option	-b — do not do special processing of shared symbols	18
linker option	-B all_archive — extract all members of an archive library	19
linker option	-B base — specify origin address in hexadecimal	19
linker option	-B dynamic — search for shared library when processing -l name	19
linker option	-B hardalign — force each output segment to align on a page boundary	20
linker option	-B help — display information about -B options used for embedded development only	20
linker option	-B kernel — designate output file as "kernel-space-loadable"	20
linker option	-B lstrip — strip local symbols from symbol table ..	20
linker option	-B movable — make dynamic executable file movable	21
linker option	-B noallocdyn — do not map dynamic tables in virtual memory	21
linker option	-B nocopies — do not make local copies of shared variables; insert relocation fix-ups	21
linker option	-B nodemandload — ignore boundary issues when mapping sections	21
linker option	-B noheader — do not include ELF header in loadable segments	21
linker option	-B noplt — do not implicitly map symbols into the PLT of the executable file	22
linker option	-B pagesize — specify page size of target processor in bytes	22
linker option	-B rogot — place the .got section in a read-only section	22
linker option	-B rogotplt — place the .got and .plt sections in a read-only section	23
linker option	-B roplt — place the .plt section in a read-only section	23
linker option	-B start_addr — specify origin address in hexadecimal	23
linker option	-B static — search for static library when processing -l name	23
linker option	-B symbolic — bind intra-module global symbol references to symbol definitions within the link ...	24

linker option -B wired — generate non-swappable output file	24
PROM bank, defined	90
elf2hex option -i — specify number of interleaved banks	82
linker hex conversion option -xi — specify number of interleaved banks	33
linker option -B base — specify origin address in hexadecimal	19
BASE statement — specify default starting address for writable data sections	103
elf2hex — utility to convert binary object file to hexadecimal ASCII format	80
elfdump — utility that produces a binary structure dump of ELF object files and shared libraries (DLLs)	86
binding mode toggle — static versus dynamic binding	26
common blocks, defined	68
linker option -B hardalign — force each output segment to align on a page boundary	20
SECTIONS keyword ALIGN_INPUT — specify alignment boundary for input section relative to output section	55
SECTIONS keyword ALIGN — specify alignment boundary for output section	55
linker option -B nodemandload — ignore boundary issues when mapping sections	21
using library function <code>_initcopy()</code> to zero bss sections	72

C

linker option -C — display listing of specified type	24
elf2hex option -c — specify section types to be converted	82
linker option -C crossref — display a cross-reference listing of global symbols	24
linker option -C globals — display a sorted list of global symbols with their mappings	24
linker option -C page — specify the number of lines per page in the displayed listing	24
linker option -C sections — display a listing of section mappings	24
linker option -C symbols — display a listing of all symbols in the generated output symbol table	24
linker option -C unmangle — used with other -C options to display C++ names in unmangled form	24
linker option -C unmangle — used with other -C options to display C++ names in unmangled form	24
invoking the linker from the High C/C++ driver	5
rewiring shared function calls through the PLT (procedure linkage table)	70
ROBASE statement — specify the default starting address for code sections (read-only sections)	115

	CODE statement — define default attributes for code segments	104
linker	command ARGS — specify linker command-line arguments in command file	40
linker	command DEMANDLOAD — demand load executable from disk	41
linker commands and arguments in a linker option -A — specify linker	command file	37
linker	command file	18
linker option -M — read input map file (AT&T-style	command file conventions	37
using wildcards in file-name references in linker	command file) for linker directives	29
linker	command files	39
linker	command INITDATA — specify control sections to initialize at run time	42, 71
invoking the linker from the system	command line	7
order of archive libraries on the linker	command line	8
linker	command LOAD — read input object files	44
linker	command NODEMANDLOAD — do not demand load executable from disk	41
linker	command PUBLIC — reassign the value of an external symbol	45
linker	command RESADD — reserve memory location by address	46
linker	command RESNUM — reserve memory location by size	46
linker	command SECTIONS — specify order of output sections	47
linker	command START — specify program entry point ..	51
specifying archiver	command-line arguments in an "at" file	80
specifying linker	command-line arguments in an "at" file	17
linker command ARGS — specify linker	command-line arguments in command file	40
linker	command-line arguments, defined	7
archiver	command-line options summary	79
elf2hex	command-line options summary	82
elfdump	command-line options summary	88
linker	command-line options summary	8
archiver option -h — display archiver	command-line syntax help screen	79
linker option -H — display linker	command-line syntax help screen	25
linker	commands and arguments in a command file	37
linker	commands reference	40
linker	commands summary	14
DESCRIPTION statement — insert string as a	.comment section	107
inserting	comments in a module-definition file	102

linker option -t — suppress warnings about multiple definitions of	common blocks, defined	68
conflicting (or duplicate) linker and	common symbols of unequal sizes	30
library function <code>_initcopy()</code> — initialize segment — a grouping of	common symbols, defined	68
linker command <code>INITDATA</code> — specify linker command file	compiler option -Hldopt — pass linker options to linker	7
file naming	compiler option names	7, 12
hex output file naming	compiling without invoking the linker	6
notational and typographic	control sections	43, 71
<code>SECTIONS</code> command syntax	control sections that are loaded as a unit	20
default hex	control sections to initialize at run time	42, 71
setting multiple hex	conventions	37
<code>elf2hex</code> — utility to	conventions	viii
generating relocation fix-ups versus local	conventions	92
linker option -B nocopies — do not make local	conventions	vii
linker option -V — display linker version number and	conversion characteristics	32
linker option -q — do not display	conversion characteristics	32
<code>elf2hex</code> option -Q — suppress entrance	convert binary object file to hexadecimal ASCII format	80
linker option -C	copies of shared variables	69
	copies of shared variables; insert relocation fix-ups ..	21
	copy-on-write fault	69, 70
	copyright banner prior to linking	30
	copyright message	29
	copyright message	83
	criteria for an output section	62
	crossref — display a cross-reference listing of global symbols	24
D		
archiver option	-d — delete archive library members	79
linker option	-d n — specify static linking	25
linker option	-d y — specify dynamic linking	25
<code>BASE</code> statement — specify default starting address for writable	data sections	103
linker option -s — strip symbols and	<code>DATA</code> statement — defines default attributes for data segments	105
	debugging information from the output file's symbol table	30, 69

output section	declaration — map file directive to create or modify output section in executable	61
size-symbol	declarations — define symbol for output section size	63
CODE statement — define	default attributes for code segments	104
DATA statement — defines	default attributes for data segments	105
	default hex conversion characteristics	32
	default host-system library directories	28
a.out,	default name of linker output file	29
_start—	default program entry point	25, 51
linker option -YP — use paths as	default search paths to resolve subsequent -l name specifications	34
ROBASE statement — specify the	default starting address for code sections (read-only sections)	115
BASE statement — specify	default starting address for writable data sections ..	103
linker hex conversion option -x — generate hex file; use multiple	default values	31, 92
linker option -t — suppress warnings about multiple	definition errors in libraries	73
linker option -B symbolic — bind intra-module global symbol references to symbol	definitions of common symbols of unequal sizes	30
linker option -z	definitions within the link	24
archiver option -d —	defs — do not allow undefined symbols; force fatal error	35
linker command NODEMANDLOAD — do not	delete archive library members	79
linker command	demand load executable from disk	41
	demand loading (or demand paging), defined	41
	DEMANDLOAD — demand load executable from disk	41
	DESCRIPTION statement — insert string as a .comment section	107
elf2hex option -n — specify width of memory	device	83
elf2hex option -s — specify size of memory	device	84
linker hex conversion option -xn — specify width of memory	device	33
linker hex conversion option -xs — specify size and width of memory	device	34
PROM	device characteristics	90
PROM	device model	90
PHYSICAL	DEVICE statement — obsolete	115
VIRTUAL	DEVICE statement — obsolete	118
mapping	directive — specify how to map input sections to output sections	62

section ordering	directive — specify ordering of output sections in executable	63
output section declaration — map file	directive to create or modify output section in executable	61
linker option -M — read input map file (AT&T-style command file) for linker	directives	29
AT&T-style input map file	directives	60
AT&T-style input map-file	directives summary	14
keywords (or	directives) specific to the SECTIONS command	48
default host-system library	directories	28
archiver option -t —	display archive library member names	79
archiver option -S —	display archive library symbol table entries	79
archiver option -h —	display archiver command-line syntax help screen ..	79
linker option -C page — specify the number of lines per page in the	displayed listing	24
LIBRARY statement — identify module as a	DLL	112
linker option -G — generate a shared library	(DLL)	25
linker option -h — use name to refer to the generated shared library	(DLL)	25
EXETYPE statement — specify operating system where the shared library	(DLL) or application runs	107
linker option -J — export only the symbols listed in file when generating a shared library	(DLL); limit dynamic symbol table	26
elfdump — utility that produces a binary structure dump of ELF object files and shared libraries	(DLLs)	86
searching for shared libraries	(DLLs)	27
invoking the linker from the High C/C++	driver	5
passing linker options via the High C/C++	driver	7
passing files directly from the	driver to the linker	6
elfdump — utility that produces a binary structure	dump of ELF object files and shared libraries (DLLs)	86
conflicting (or	duplicate section names	68
linker option -B	duplicate) linker and compiler option names	7, 12
dynamic — search for shared library when processing -l name	dynamic — search for shared library when processing -l name	19
dynamic binding	dynamic binding	26
dynamic executable file movable	dynamic executable file movable	21
dynamic linking	dynamic linking	25
dynamic linking, defined	dynamic linking, defined	74
binding mode toggle — static versus		
linker option -B movable — make		
linker option -d y — specify		

linker option -J — export only the symbols listed in file when generating a shared library (DLL); limit	dynamic symbol table	26
linker option -B noallocdyn — do not map	dynamic tables in virtual memory	21
E		
linker option	-e — specify program entry point	25
linker option -B noheader — do not include elfdump — utility that produces a binary structure dump of	ELF header in loadable segments	21
	ELF object files and shared libraries (DLLs)	86
	elf2hex — utility to convert binary object file to hexadecimal ASCII format	80
	elf2hex command-line options summary	82
	elfdump — utility that produces a binary structure dump of ELF object files and shared libraries (DLLs)	86
	elfdump command-line options summary	88
	embedded applications	11
linker options for developing	embedded development only	20
linker option -B help — display information about -B options used for	entrance criteria for an output section	62
linker command START — specify program	entry point	51
linker option -e — specify program	entry point	25
_start — default program	entry point	25, 51
	environment variable LD_LIBRARY_PATH ..	27, 34
	environment variable LIBPATH	27, 34
archiver	error messages	78
linker	error messages	75
linker	error messages in the memory map listing file	76
multiple definition	errors in libraries	73
undefined reference	errors in libraries	73
linker	errors, defined	75
linker terminal	errors, defined	76
NAME statement — identifies	executable file as an application	113
	EXEFORMAT statement — obsolete	107
	EXETYPE statement — specify operating system where the shared library (DLL) or application runs	107
	explicit virtual address assignment	63
SECTIONS keyword TYPE —	explicitly assign a type to an output section	54
linker option -J —	export only the symbols listed in file when generating a shared library (DLL); limit dynamic symbol table	26

	EXPORTS statement — defines names and attributes of functions exported to other modules	108
elf2hex option -t — specify	Extended Tektronix Hex Record format	84
linker hex conversion option -xt — specify	Extended Tektronix Hex Record format	33
	Extended Tektronix Hex Record format, defined	96
	external references listed in memory map listing file	30
	undefined	
linker command PUBLIC — reassign the value of an	external symbol	45
linker option -B all_archive —	extract all members of an archive library	19
archiver option -x —	extract specified archive library members	79

F

linker option -z defs — do not allow undefined symbols; force	fatal error	35
linker option -z text — do not allow output relocations against read-only sections; force copy-on-write	fatal error	35
linker option -s — strip symbols and debugging information from the output using wildcards in	fault	69, 70
linker hex conversion option -x — generate hex	file's symbol table	30, 69
archiver ar — management utility that groups object	file-name references in linker command files	39
linker option -B nocopies — do not make local copies of shared variables; insert relocation	file; use default values	31, 92
generating relocation	files into an archive library	77
appending	fix-ups	21
library	fix-ups versus local copies of shared variables	69
using library	flags or attributes to linker option -x	31
rewiring shared	function _initcopy() — initialize control sections	43, 71
	function _initcopy() to zero bss sections	72
	function calls through the PLT (procedure linkage table)	70
position independent relocations for references to	functions defined in shared objects	19
EXPORTS statement — defines names and attributes of	functions exported to other modules	108
IMPORTS statement — define names and attributes of	functions imported	110

G

linker option	-G — generate a shared library (DLL)	25
---------------	--------------------------------------	----

linker option -B symbolic — bind intra-module	global symbol references to symbol definitions within the link	24
linker option -C crossref — display a cross-reference listing of	global symbols	24
linker option -C globals — display a sorted list of	global symbols in libraries	72
linker option -C linker option -C	global symbols with their mappings	24
linker option -B rogotplt — place the	globals — display a sorted list of global symbols with their mappings	24
linker option -B rogot — place the	.got and .plt sections in a read-only section	23
SECTIONS keyword	.got section in a read-only section	22
	GROUP — force a set of output sections to be allocated contiguously in a specified order	53

H

archiver option	-h — display archiver command-line syntax help screen	79
linker option	-H — display linker command-line syntax help screen	25
linker option	-h — use name to refer to the generated shared library (DLL)	25
linker option -B	hardalign — force each output segment to align on a page boundary	20
linker option -I — write interpreter pathname into program	header	26
linker option -B noheader — do not include ELF	header in loadable segments	21
linker option -B	HEAPSIZE statement — obsolete	109
linker option -x	help — display information about -B options used for embedded development only	20
archiver option -h — display archiver command-line syntax	help — display information about -x options for hex record generation	32
linker option -H — display linker command-line syntax	help screen	79
default	help screen	25
setting multiple	hex conversion characteristics	32
linker	hex conversion characteristics	32
elf2hex option -o — specify name of generated	hex conversion option -x — generate hex file; use default values	31, 92
	hex file	83
	hex file formats	94

elf2hex option -m — specify Motorola S3	hex output file naming conventions	92
elf2hex option -q — specify Mentor QUICKSIM Modelfile	Hex Record format	83
elf2hex option -t — specify Extended Tektronix	Hex Record format	83
linker hex conversion option -xm — specify Motorola S3	Hex Record format	84
linker hex conversion option -xq — specify Mentor QUICKSIM Modelfile	Hex Record format	33
linker hex conversion option -xt — specify Extended Tektronix	Hex Record format	33
Extended Tektronix	Hex Record format, defined	96
Mentor QUICKSIM Modelfile	Hex Record format, defined	98
Motorola S3	Hex Record format, defined	94
linker option -x help — display information about -x options for	hex record generation	32
elf2hex — utility to convert binary object file to	hexadecimal ASCII format	80
passing linker options via the	hexadecimal record files, defined	89
invoking the linker from the	High C/C++ driver	7
compiler option	High C/C++ driver	5
default	-Hldopt — pass linker options to linker	7
	host-system library directories	28

I

linker option	-i — read OS/2-style module-definition file for linker definitions	25
elf2hex option	-i — specify number of interleaved banks	82
linker option	-I — write interpreter pathname into program header	26
linker option -B nodemandload —	ignore boundary issues when mapping sections	21
linker option -B noplt — do not	implicitly map symbols into the PLT of the executable file	22
	IMPORTS statement — define names and attributes of functions imported	110
linker option -r — generate relocatable object file for	incremental linking	30
	incremental linking — combine two or more relocatable object files into a single relocatable object file	68
position	independent relocations for references to functions defined in shared objects	19
library function	_initcopy() — initialize control sections	43, 71

using library function	<code>_initcopy()</code> to zero bss sections	72
linker command	INITDATA — specify control sections to initialize at run time	42, 71
library function <code>_initcopy()</code> —	initialize control sections	43, 71
	initializing RAM from a program in ROM	71
linker option -M — read	input map file (AT&T-style command file) for linker directives	29
AT&T-style	input map file directives	60
AT&T-style	input map-file directives summary	14
linker command LOAD — read	input object files	44
SECTIONS keyword ALIGN_INPUT — specify alignment boundary for	input section relative to output section	55
linker option -z sections — diagnose unreferenced files and mapping directive — specify how to map	input sections	35
	input sections to output sections	62
	input_sect_spec specifier in a SECTIONS command — specify which input sections to store in each output section	56
linker option -B nocopies — do not make local copies of shared variables; DESCRIPTION statement —	insert relocation fix-ups	21
	insert string as a .comment section	107
elf2hex option -i — specify number of	inserting comments in a module-definition file	102
linker hex conversion option -xi — specify number of	interleaved banks	82
linker option -I — write	interleaved banks	33
linker option -B symbolic — bind	interpreter pathname into program header	26
	intra-module global symbol references to symbol definitions within the link	24
	invoking the archiver	78
compiling without	invoking the linker	6
	invoking the linker from the High C/C++ driver	5
	invoking the linker from the system command line ...	7

J

linker option	-J — export only the symbols listed in file when generating a shared library (DLL); limit dynamic symbol table	26
---------------	--	----

K

linker option -B	kernel — designate output file as "kernel-space- loadable"	20
SECTIONS	keyword ADDRESS — assign absolute starting address for an output section or group	55

SECTIONS	keyword ALIGN — specify alignment boundary for output section	55
SECTIONS	keyword ALIGN_INPUT — specify alignment boundary for input section relative to output section	55
SECTIONS	keyword GROUP — force a set of output sections to be allocated contiguously in a specified order ..	53
SECTIONS	keyword TYPE — explicitly assign a type to an output section	54
OS/2 2.x legacy support	keywords	103
OS/2 module-definition files and	keywords	101
	keywords (or directives) specific to the SECTIONS command	48

L

linker option	-l — search for library whose name contains "name" ..	26
linker option	-L — specify search path to resolve -l name specifications	28
linker option -B dynamic — search for shared library when processing	-l name	19
linker option -B static — search for static library when processing	-l name	23
linker option -L — specify search path to resolve	-l name specifications	28
linker option -YP — use paths as default search paths to resolve subsequent environment variable	-l name specifications	34
OS/2 2.x	LD_LIBRARY_PATH	27, 34
PROM size (or length), defined	legacy support keywords	103
environment variable	length), defined	90
global symbols in	LIBPATH	27, 34
multiple definition errors in	libraries	72
processing archive	libraries	73
searching for static-link	libraries	44, 72
undefined reference errors in	libraries	27
elfdump — utility that produces a binary structure dump of ELF object files and shared	libraries	73
searching for shared	libraries (DLLs)	86
order of archive	libraries (DLLs)	27
archiver ar — management utility that groups object files into an archive	libraries on the linker command line	8
linker option -G — generate a shared	library	77
linker option -h — use name to refer to the generated shared	library (DLL)	25
	library (DLL)	25

EXETYPE statement — specify operating system where the shared	library (DLL) or application runs	107
linker option -J — export only the symbols listed in file when generating a shared default host-system linking in run-time	library (DLL); limit dynamic symbol table	26
	library directories	28
	library files	6
	library function <code>_initcopy()</code> — initialize control sections	43, 71
using	library function <code>_initcopy()</code> to zero bss sections	72
manipulating archive	library members and archive symbol tables	79
determining	library search paths	27
linker option -B dynamic — search for shared	LIBRARY statement — identify module as a DLL	112
linker option -B static — search for static	library when processing -l name	19
linker option -l — search for	library when processing -l name	23
linker option -J — export only the symbols listed in file when generating a shared	library whose name contains "name"	26
library (DLL);	limit dynamic symbol table	26
linker option -C page — specify the number of	lines per page in the displayed listing	24
rewiring shared function calls through the PLT (procedure	linkage table)	70
compiling without invoking the	linker	6
passing files directly from the driver to the	linker	6
conflicting (or duplicate)	linker and compiler option names	7, 12
order of archive libraries on the	linker command line	8
specifying	linker command-line arguments in an "at" file	17
	linker command-line arguments, defined	7
	linker command-line options summary	8
	linker commands summary	14
	linker error messages	75
	linker error messages in the memory map listing file	76
	linker errors, defined	75
invoking the	linker from the High C/C++ driver	5
invoking the	linker from the system command line	7
	linker options for developing embedded applications	11
compiler option -Hldopt — pass	linker options to linker	7
passing	linker options via the High C/C++ driver	7
	linker terminal errors, defined	76
	linker warnings, defined	75
determining section size with	linker-defined symbols	68
special	linker-defined symbols	67
linker option -d n — specify static	linking	25

linker option -d y — specify dynamic	linking	25
linker option -r — generate relocatable object file for incremental	linking	30
incremental	linking — combine two or more relocatable object files into a single relocatable object file	68
dynamic	linking in run-time library files	6
static	linking, defined	74
linker option -C globals — display a sorted	linking, defined	74
linker option -C page — specify the number of lines per page in the displayed	list of global symbols with their mappings	24
linker error messages in the memory map	listing	24
saving the contents of a memory map	listing file	76
undefined external references listed in	listing file	29
memory map	listing file	30
memory map	listing file — explains how sections are allocated in virtual memory	29
linker option -m — write memory map	listing file to standard output	29
linker option -C symbols — display a	listing of all symbols in the generated output symbol table	24
linker option -C crossref — display a cross-reference	listing of global symbols	24
linker option -C sections — display a	listing of section mappings	24
linker option -C — display	listing of specified type	24
linker command	LOAD — read input object files	44
linker command NODEMANDLOAD — do not demand load executable from disk	load time	21
alter origin address at	loadable segments	21
linker option -B noheader — do not include ELF header in	loading (or demand paging), defined	41
demand	local copies of shared variables	69
generating relocation fix-ups versus	local copies of shared variables; insert relocation fix-ups	21
linker option -B nocopies — do not make	local symbols from symbol table	20
linker option -B lstrip — strip	locating relocatable sections	53
"next-fit" algorithm for	location by address	46
linker command RESADD — reserve memory	location by size	46
linker command RESNUM — reserve memory	lstrip — strip local symbols from symbol table	20
linker option -B		

M

archiver option	-m — merge archive members; do not replace existing members	79
-----------------	---	----

archiver option	-M — merge archive members; replace existing members	79
linker option	-M — read input map file (AT&T-style command file) for linker directives	29
elf2hex option	-m — specify Motorola S3 Hex Record format	83
linker option	-m — write memory map listing file to standard output	29
linker option -C unmangle — used with other -C options to display C++ names in unmangled form		24
linker option -B noalldyn — do not map dynamic tables in virtual memory		21
linker option -M — read input map file (AT&T-style command file) for linker directives		29
output section declaration — map file directive to create or modify output section in executable		61
AT&T-style input map file directives		60
mapping directive — specify how to map input sections to output sections		62
linker error messages in the memory map listing file		76
saving the contents of a memory map listing file		29
undefined external references listed in memory map listing file		30
memory map listing file — explains how sections are allocated in virtual memory		29
linker option -m — write memory map listing file to standard output		29
linker option -B noplt — do not implicitly map symbols into the PLT of the executable file		22
AT&T-style input map-file directives summary		14
mapping directive — specify how to map input sections to output sections		62
linker option -B nodemandload — ignore mapping sections		21
boundary issues when mapping unassigned input sections		57
linker option -C globals — display a sorted list of global symbols with their mappings		24
linker option -C sections — display a listing of section mappings		24
manipulating archive library member file contained in archive library		78
output_sect_spec specifier in a SECTIONS command — determines order of output sections members and archive symbol tables		79
in memory		53
specify the page size of target memory configuration		41
elf2hex option -n — specify width of memory device		83
elf2hex option -s — specify size of memory device		84
linker hex conversion option -xn — specify memory device		33
width of		

linker hex conversion option -xs — specify size and width of	memory device	34
linker command RESADD — reserve	memory location by address	46
linker command RESNUM — reserve	memory location by size	46
linker error messages in the	memory map listing file	76
saving the contents of a	memory map listing file	29
undefined external references listed in	memory map listing file	30
	memory map listing file — explains how sections are allocated in virtual memory	29
linker option -m — write	memory map listing file to standard output	29
elf2hex option -q — specify	Mentor QUICKSIM Modelfile Hex Record format .	83
linker hex conversion option -xq — specify	Mentor QUICKSIM Modelfile Hex Record format .	33
	Mentor QUICKSIM Modelfile Hex Record format, defined	98
	merge archive members; do not replace existing members	79
archiver option -m —	merge archive members; replace existing members .	79
archiver option -M —	Modelfile Hex Record format	83
elf2hex option -q — specify Mentor QUICKSIM		
linker hex conversion option -xq — specify Mentor QUICKSIM	Modelfile Hex Record format	33
output section declaration — map file directive to create or inserting comments in a	Modelfile Hex Record format, defined	98
linker option -i — read OS/2-style sample OS/2	modify output section in executable	61
elf2hex option -m — specify	module-definition file	102
linker hex conversion option -xm — specify	module-definition file for linker definitions	25
	module-definition files	118
	module-definition files and keywords	101
	Motorola S3 Hex Record format	83
	Motorola S3 Hex Record format	33
	Motorola S3 Hex Record format, defined	94
linker option -B	movable — make dynamic executable file movable	21
	multiple definition errors in libraries	73
linker option -t — suppress warnings about	multiple definitions of common symbols of unequal sizes	30
setting	multiple hex conversion characteristics	32
N		
elf2hex option	-n — specify width of memory device	83
elf2hex option -o — specify	name of generated hex file	83
linker hex conversion option -xo — specify	name of generated hex file	33
a.out, default	name of linker output file	29
linker option -o — specify	name of output file	29

	NAME statement — identifies executable file as an application	113
conflicting (or duplicate) linker and compiler option	names	7, 12
duplicate section	names	68
EXPORTS statement — defines	names and attributes of functions exported to other modules	108
IMPORTS statement — define	names and attributes of functions imported	110
linker option -C unmangle — used with other -C options to display C++ file	names in unmangled form	24
hex output file	naming conventions	viii
	naming conventions	92
	naming the output file	6
	"next-fit" algorithm for locating relocatable sections	53
linker option -B	noallocdyn — do not map dynamic tables in virtual memory	21
linker option -B	nocopies — do not make local copies of shared variables; insert relocation fix-ups	21
linker option -z	nodefs — allow undefined symbols	35
linker command	NODEMANDLOAD — do not demand load executable from disk	41
linker option -B	nodemandload — ignore boundary issues when mapping sections	21
linker option -B	noheader — do not include ELF header in loadable segments	21
linker option -B wired — generate	non-swappable output file	24
linker option -B	nopltd — do not implicitly map symbols into the PLT of the executable file	22
designate sections as	"not allocable"	21
elf2hex option -i — specify	notational and typographic conventions	vii
linker hex conversion option -xi — specify	number of interleaved banks	82
linker option -C page — specify the	number of interleaved banks	33
	number of lines per page in the displayed listing	24

O

elf2hex option	-o — specify name of generated hex file	83
linker option	-o — specify name of output file	29
linker option -r — generate relocatable	object file for incremental linking	30
elf2hex — utility to convert binary	object file to hexadecimal ASCII format	80
linker command LOAD — read input	object files	44
elfdump — utility that produces a binary structure dump of ELF	object files and shared libraries (DLLs)	86

incremental linking — combine two or more relocatable	object files into a single relocatable object file	68
archiver ar — management utility that groups position independent relocations for references to functions defined in shared	object files into an archive library	77
EXEFORMAT statement —	objects	19
HEAPSIZE statement —	obsolete	107
OLD statement —	obsolete	109
PHYSICAL DEVICE statement —	obsolete	114
PROTMODE statement —	obsolete	115
SESSION statement —	obsolete	115
STACKSIZE statement —	obsolete	117
STUB statement —	obsolete	117
VERSION statement —	obsolete	118
VIRTUAL DEVICE statement —	obsolete	118
EXETYPE statement — specify	OLD statement — obsolete	114
	operating system where the shared library (DLL) or application runs	107
linker option -A — specify linker command file		18
linker option -b — do not do special processing of shared symbols		18
linker option -B all_archive — extract all members of an archive library		19
linker option -B base — specify origin address in hexadecimal		19
linker option -B dynamic — search for shared library when processing -l name		19
linker option -B hardalign — force each output segment to align on a page boundary		20
linker option -B help — display information about -B options used for embedded development only		20
linker option -B kernel — designate output file as "kernel- space-loadable"		20
linker option -B lstrip — strip local symbols from symbol table		20
linker option -B movable — make dynamic executable file movable		21
linker option -B noallocdyn — do not map dynamic tables in virtual memory		21
linker option -B nocopies — do not make local copies of shared variables; insert relocation fix-ups		21
linker option -B nodemandload — ignore boundary issues when mapping sections		21

linker	option -B noheader — do not include ELF header in loadable segments	21
linker	option -B noplt — do not implicitly map symbols into the PLT of the executable file	22
linker	option -B pagesize — specify page size of target processor in bytes	22
linker	option -B rogot — place the .got section in a read-only section	22
linker	option -B rogotplt — place the .got and .plt sections in a read-only section	23
linker	option -B ropplt — place the .plt section in a read-only section	23
linker	option -B start_addr — specify origin address in hexadecimal	23
linker	option -B static — search for static library when processing -l name	23
linker	option -B symbolic — bind intra-module global symbol references to symbol definitions within the link	24
linker	option -B wired — generate non-swappable output file	24
linker	option -C — display listing of specified type	24
elf2hex	option -c — specify section types to be converted ...	82
linker	option -C crossref — display a cross-reference listing of global symbols	24
linker	option -C globals — display a sorted list of global symbols with their mappings	24
linker	option -C page — specify the number of lines per page in the displayed listing	24
linker	option -C sections — display a listing of section mappings	24
linker	option -C symbols — display a listing of all symbols in the generated output symbol table	24
linker	option -C unmangle — used with other -C options to display C++ names in unmangled form	24
archiver	option -d — delete archive library members	79
linker	option -d n — specify static linking	25
linker	option -d y — specify dynamic linking	25
linker	option -e — specify program entry point	25
linker	option -G — generate a shared library (DLL)	25
archiver	option -h — display archiver command-line syntax help screen	79
linker	option -H — display linker command-line syntax help screen	25

linker	option -h — use name to refer to the generated shared library (DLL)	25
compiler	option -Hldopt — pass linker options to linker	7
linker	option -i — read OS/2-style module-definition file for linker definitions	25
elf2hex	option -i — specify number of interleaved banks	82
linker	option -I — write interpreter pathname into program header	26
linker	option -J — export only the symbols listed in file when generating a shared library (DLL); limit dynamic symbol table	26
linker	option -l — search for library whose name contains "name"	26
linker	option -L — specify search path to resolve -l name specifications	28
archiver	option -m — merge archive members; do not replace existing members	79
archiver	option -M — merge archive members; replace existing members	79
linker	option -M — read input map file (AT&T-style command file) for linker directives	29
elf2hex	option -m — specify Motorola S3 Hex Record format	83
linker	option -m — write memory map listing file to standard output	29
elf2hex	option -n — specify width of memory device	83
elf2hex	option -o — specify name of generated hex file	83
linker	option -o — specify name of output file	29
linker	option -q — do not display copyright message	29
linker	option -Q — specify whether version information appears in output file	29
elf2hex	option -Q — suppress copyright message	83
elf2hex	option -q — specify Mentor QUICKSIM Modelfile Hex Record format	83
linker	option -r — generate relocatable object file for incremental linking	30
archiver	option -r — replace (or add) archive library members	79
archiver	option -S — display archive library symbol table entries	79
archiver	option -s — reconstruct archive library symbol table	79
elf2hex	option -s — specify size of memory device	84
linker	option -s — strip symbols and debugging information from the output file's symbol table	30, 69
archiver	option -t — display archive library member names	79

linker	option -t — suppress warnings about multiple definitions of common symbols of unequal sizes .	30
elf2hex	option -t — specify Extended Tektronix Hex Record format	84
linker	option -u — create unresolved (undefined) symbol .	30
linker	option -V — display linker version number and copyright banner prior to linking	30
archiver	option -v — display verbose output of archiver actions	79
linker	option -w — suppress all warnings	31
appending flags or attributes to linker	option -x	31
archiver	option -x — extract specified archive library members	79
linker hex conversion	option -x — generate hex file; use default values	31, 92
linker	option -x help — display information about -x options for hex record generation	32
linker hex conversion	option -xc — specify section types to be converted .	32
linker hex conversion	option -xi — specify number of interleaved banks ..	33
linker hex conversion	option -xm — specify Motorola S3 Hex Record format	33
linker hex conversion	option -xn — specify width of memory device	33
linker hex conversion	option -xo — specify name of generated hex file	33
linker hex conversion	option -xq — specify Mentor QUICKSIM Modelfile Hex Record format	33
linker hex conversion	option -xs — specify size and width of memory device	34
linker hex conversion	option -xt — specify Extended Tektronix Hex Record format	33
linker	option -YP — use paths as default search paths to resolve subsequent -l name specifications	34
linker	option -z defs — do not allow undefined symbols; force fatal error	35
linker	option -z nodefs — allow undefined symbols	35
linker	option -z sections — diagnose unreferenced files and input sections	35
linker	option -z text — do not allow output relocations against read-only sections; force fatal error	35
conflicting (or duplicate) linker and compiler	option names	7, 12
linker	options for developing embedded applications	11
linker option -x help — display information about -x	options for hex record generation	32
archiver command-line	options summary	79
elf2hex command-line	options summary	82

elfdump command-line	options summary	88
linker command-line	options summary	8
linker option -C unmangle — used with other	options to display C++ names in unmangled form ...	24
-C	options to linker	7
compiler option -Hldopt — pass linker	options used for embedded development only	20
linker option -B help — display information	options via the High C/C++ driver	7
about -B		
passing linker		
SECTIONS keyword GROUP — force a set of	order	53
output sections to be allocated contiguously	order of archive libraries on the linker command line	8
in a specified	order of output sections	47
linker command SECTIONS — specify	order of output sections in memory	53
output_sect_spec specifier in a SECTIONS	ordering directive — specify ordering of output	
command — determines	sections in executable	63
section	origin address at load time	21
alter	origin address in hexadecimal	19
linker option -B base — specify	origin address in hexadecimal	23
linker option -B start_addr — specify	OS/2 2.x legacy support keywords	103
	OS/2 module-definition files and keywords	101
linker option -i — read	OS/2-style module-definition file for linker	
linker option -o — specify name of	definitions	25
linker option -B wired — generate	output file	29
non-swappable	output file	24
linker option -Q — specify whether version	output file	29
information appears in	output file	29
a.out, default name of linker	output file	6
naming the	output file as "kernel-space-loadable"	20
linker option -B kernel — designate	output file naming conventions	92
hex		
linker option -s — strip symbols and	output file's symbol table	30, 69
debugging information from the	output relocations against read-only sections; force	
linker option -z text — do not allow	fatal error	35
input_sect_spec specifier in a SECTIONS	output section	56
command — specify which input sections to	output section	62
store in each	output section declaration — map file directive to	
entrance criteria for an	create or modify output section in executable	61
size-symbol declarations — define symbol for	output section size	63

linker command SECTIONS — specify order of mapping directive — specify how to map input sections to	output sections	47
section ordering directive — specify ordering of	output sections	62
output_sect_spec specifier in a SECTIONS command — determines order of assigning	output sections in executable	63
linker option -B hardalign — force each	output sections in memory	53
linker option -C symbols — display a listing of all symbols in the generated	output sections to absolute addresses	53
	output segment to align on a page boundary	20
	output symbol table	24
	output_group_spec specifier in a SECTIONS command	52
	output_sect_spec specifier in a SECTIONS command — determines order of output sections in memory	53
P		
linker option -C	page — specify the number of lines per page in the displayed listing	24
linker option -B hardalign — force each output segment to align on a	page boundary	20
specify the	page size of target memory configuration	41
linker option -B	pagesize — specify page size of target processor in bytes	22
demand loading (or demand	paging), defined	41
compiler option -Hldopt —	pass linker options to linker	7
	passing files directly from the driver to the linker	6
linker option -L — specify search	passing linker options via the High C/C++ driver	7
linker option -I — write interpreter	path to resolve -l name specifications	28
determining library search	pathname into program header	26
linker option -YP — use	paths	27
	paths as default search paths to resolve subsequent -l name specifications	34
	pattern matching rules for section names	56
	PHYSICAL DEVICE statement — obsolete	115
	PLT (procedure linkage table)	70
rewiring shared function calls through the	PLT of the executable file	22
linker option -B noplt — do not implicitly map symbols into the	.plt section in a read-only section	23
linker option -B roplt — place the	.plt sections in a read-only section	23
linker option -B rogotplt — place the .got and	position independent relocations for references to functions defined in shared objects	19

rewiring shared function calls through the PLT	(procedure linkage table)	70
linker option -B pagesize — specify page size of target	processor in bytes	22
elfdump — utility that produces a binary structure dump of ELF object files and shared libraries (DLLs)		86
linker command START — specify program entry point		51
linker option -e — specify program entry point		25
_start — default program entry point		25, 51
linker option -I — write interpreter pathname into program header		26
initializing RAM from a program in ROM		71
PST — program symbol table		72
	PROM device characteristics	90
	PROM device model	90
	PROTMODE statement — obsolete	115
	PST — program symbol table	72
linker command PUBLIC — reassign the value of an external symbol		45

Q

linker option -q — do not display copyright message		29
linker option -Q — specify whether version information appears in output file		29
elf2hex option -Q — suppress copyright message		83
elf2hex option -q — specify Mentor QUICKSIM Modelfile Hex Record format		83
elf2hex option -q — specify Mentor QUICKSIM Modelfile Hex Record format		83
linker hex conversion option -xq — specify Mentor QUICKSIM Modelfile Hex Record format		33
Mentor QUICKSIM Modelfile Hex Record format, defined		98

R

linker option -r — generate relocatable object file for incremental linking		30
archiver option -r — replace (or add) archive library members		79
initializing RAM from a program in ROM		71
linker option -M — read input map file (AT&T-style command file) for linker directives		29
linker command LOAD — read input object files		44
linker option -i — read OS/2-style module-definition file for linker definitions		25

linker option -B rogot — place the .got section in a	read-only section	22
linker option -B rogotplt — place the .got and .plt sections in a	read-only section	23
linker option -B ropplt — place the .plt section in a	read-only section	23
ROBASE statement — specify the default starting address for code sections	(read-only sections)	115
linker option -z text — do not allow output relocations against	read-only sections; force fatal error	35
linker command PUBLIC —	reassign the value of an external symbol	45
archiver option -s —	reconstruct archive library symbol table	79
hexadecimal	record files, defined	89
elf2hex option -m — specify Motorola S3 Hex	Record format	83
elf2hex option -q — specify Mentor QUICKSIM Modelfile Hex	Record format	83
elf2hex option -t — specify Extended Tektronix Hex	Record format	84
linker hex conversion option -xm — specify Motorola S3 Hex	Record format	33
linker hex conversion option -xq — specify Mentor QUICKSIM Modelfile Hex	Record format	33
linker hex conversion option -xt — specify Extended Tektronix Hex	Record format	33
Mentor QUICKSIM Modelfile Hex	Record format, defined	98
Extended Tektronix Hex	Record format, defined	96
Motorola S3 Hex	Record format, defined	94
linker option -x help — display information about -x options for hex	record generation	32
undefined	reference errors in libraries	73
using wildcards in file-name	references in linker command files	39
undefined external	references listed in memory map listing file	30
position independent relocations for	references to functions defined in shared objects	19
linker option -B symbolic — bind intra-module global symbol	references to symbol definitions within the link	24
linker option -r — generate	relocatable object file for incremental linking	30
incremental linking — combine two or more	relocatable object files into a single relocatable object file	68
"next-fit" algorithm for locating	relocatable sections	53
linker option -B nocopies — do not make local copies of shared variables; insert	relocation fix-ups	21
generating	relocation fix-ups versus local copies of shared variables	69

linker option -z text — do not allow output	relocations against read-only sections; force fatal error	35
position independent	relocations for references to functions defined in shared objects	19
archiver option -r — do not	replace (or add) archive library members	79
archiver option -m — merge archive members;	replace existing members	79
do not	replace existing members	79
archiver option -M — merge archive members;	RESADD — reserve memory location by address ...	46
linker command	reserve memory location by address	46
linker command RESADD —	reserve memory location by size	46
linker command RESNUM —	RESNUM — reserve memory location by size	46
linker command	resolve -l name specifications	28
linker option -L — specify search path to	resolve subsequent -l name specifications	34
linker option -YP — use paths as default	resolving ambiguous section names	57
search paths to	rewiring shared function calls through the PLT (procedure linkage table)	70
	ROBASE statement — specify the default starting address for code sections (read-only sections) ...	115
linker option -B	rogot — place the .got section in a read-only section	22
linker option -B	rogotplt — place the .got and .plt sections in a read-only section	23
initializing RAM from a program in	ROM	71
linker option -B	roplt — place the .plt section in a read-only section	23
linking in	run-time library files	6

S

archiver option	-S — display archive library symbol table entries	79
archiver option	-s — reconstruct archive library symbol table	79
elf2hex option	-s — specify size of memory device	84
linker option	-s — strip symbols and debugging information from the output file's symbol table	30, 69
elf2hex option -m — specify Motorola	S3 Hex Record format	83
linker hex conversion option -xm — specify Motorola	S3 Hex Record format	33
Motorola	S3 Hex Record format, defined	94
	sample module-definition files	118
linker option -l —	saving the contents of a memory map listing file	29
linker option -B dynamic —	search for library whose name contains "name"	26
linker option -B static —	search for shared library when processing -l name ...	19
linker option -L — specify	search for static library when processing -l name	23
	search path to resolve -l name specifications	28

determining library	search paths	27
linker option -YP — use paths as default	search paths to resolve subsequent -l name specifications	34
	searching for shared libraries (DLLs)	27
	searching for static-link libraries	27
linker option -B rogotplt — place the .got and .plt sections in a read-only	section	23
DESCRIPTION statement — insert string as a .comment	section	107
entrance criteria for an output	section	62
output	section declaration — map file directive to create or modify output section in executable	61
linker option -B rogot — place the .got	section in a read-only section	22
linker option -B roplt — place the .plt	section in a read-only section	23
linker option -C sections — display a listing of	section mappings	24
pattern matching rules for	section names	56
resolving ambiguous	section names	57
duplicate	section names	68
SECTIONS keyword ALIGN_INPUT — specify alignment boundary for input	section ordering directive — specify ordering of output sections in executable	63
SECTIONS keyword ALIGN_INPUT — specify alignment boundary for input	section relative to output section	55
size-symbol declarations — define symbol for output	section relative to output section	55
determining	section size	63
elf2hex option -c — specify	section size with linker-defined symbols	68
linker hex conversion option -xc — specify	section types to be converted	82
linker command SECTIONS — specify order of output	section types to be converted	32
linker option -B nodemandload — ignore boundary issues when mapping	sections	47
"next-fit" algorithm for locating relocatable	sections	21
BASE statement — specify default starting address for writable data	sections	53
library function _initcopy() — initialize control	sections	103
mapping unassigned input	sections	43, 71
using library function _initcopy() to zero bss	sections	57
ROBASE statement — specify the default starting address for code	sections	72
	sections (read-only sections)	115

linker option -z	sections — diagnose unreferenced files and input sections	35
linker option -C	sections — display a listing of section mappings	24
linker command	SECTIONS — specify order of output sections	47
memory map listing file — explains how	sections are allocated in virtual memory	29
designate	sections as "not allocable"	21
address_spec specifier in a	SECTIONS command	55
keywords (or directives) specific to the	SECTIONS command	48
output_group_spec specifier in a	SECTIONS command	52
using wildcards in	SECTIONS command	54
output_sect_spec specifier in a	SECTIONS command — determines order of output sections in memory	53
input_sect_spec specifier in a	SECTIONS command — specify which input sections to store in each output section	56
	SECTIONS command syntax conventions	48
linker option -B rogotplt — place the .got and .plt	sections in a read-only section	23
section ordering directive — specify ordering of output	sections in executable	63
	SECTIONS keyword ADDRESS — assign absolute starting address for an output section or group	55
	SECTIONS keyword ALIGN — specify alignment boundary for output section	55
	SECTIONS keyword ALIGN_INPUT — specify alignment boundary for input section relative to output section	55
	SECTIONS keyword GROUP — force a set of output sections to be allocated contiguously in a specified order	53
	SECTIONS keyword TYPE — explicitly assign a type to an output section	54
segment — a grouping of control	sections that are loaded as a unit	20
assigning output	sections to absolute addresses	53
linker command INITDATA — specify control	sections to initialize at run time	42, 71
mapping directive — specify how to map input	sections to output sections	62
linker option -z text — do not allow output	sections; force fatal error	35
relocations against read-only	segment — a grouping of control sections that are loaded as a unit	20
linker option -B hardalign — force each	segment to align on a page boundary	20
output		
linker option -B noheader — do not include	segments	21
ELF header in loadable		
CODE statement — define default attributes	segments	104
for code		

DATA statement — defines default attributes for data	segments	105
	SEGMENTS statement — define attributes of individual segments	115
	SESSION statement — obsolete	117
	setting multiple hex conversion characteristics	32
rewiring	shared function calls through the PLT (procedure linkage table)	70
elfdump — utility that produces a binary structure dump of ELF object files and searching for	shared libraries (DLLs)	86
linker option -G — generate a	shared libraries (DLLs)	27
linker option -h — use name to refer to the generated	shared library (DLL)	25
EXETYPE statement — specify operating system where the	shared library (DLL)	25
linker option -J — export only the symbols listed in file when generating a	shared library (DLL) or application runs	107
linker option -B dynamic — search for position independent relocations for references to functions defined in	shared library (DLL); limit dynamic symbol table ...	26
linker option -b — do not do special processing of	shared library when processing -l name	19
generating relocation fix-ups versus local copies of	shared objects	19
linker option -B nocopies — do not make local copies of	shared symbols	18
linker command RESNUM — reserve memory location by	shared variables	69
size-symbol declarations — define symbol for output section	shared variables; insert relocation fix-ups	21
PROM	size	46
linker hex conversion option -xs — specify	size	63
elf2hex option -s — specify	size (or length), defined	90
linker option -B pagesize — specify page	size and width of memory device	34
	size of memory device	84
	size of target processor in bytes	22
	size-symbol declarations — define symbol for output section size	63
linker option -t — suppress warnings about multiple definitions of common symbols of unequal	sizes	30
linker option -C globals — display a	sorted list of global symbols with their mappings	24
linker option -L — specify search path to resolve -l name	specifications	28
linker option -YP — use paths as default search paths to resolve subsequent -l name	specifications	34

address_spec	specifier in a SECTIONS command	55
output_group_spec	specifier in a SECTIONS command	52
output_sect_spec	specifier in a SECTIONS command — determines order of output sections in memory	53
input_sect_spec	specifier in a SECTIONS command — specify which input sections to store in each output section	56
	STACKSIZE statement — obsolete	117
linker option -m — write memory map listing file to	standard output	29
	_start — default program entry point	25, 51
linker command	START — specify program entry point	51
linker option -B	start_addr — specify origin address in hexadecimal	23
SECTIONS keyword ADDRESS — assign absolute	starting address for an output section or group	55
ROBASE statement — specify the default	starting address for code sections (read-only sections)	115
BASE statement — specify default	starting address for writable data sections	103
linker option -B	static — search for static library when processing -l name	23
linker option -d n — specify	static linking	25
	static linking, defined	74
binding mode toggle —	static versus dynamic binding	26
searching for	static-link libraries	27
input_sect_spec specifier in a SECTIONS command — specify which input sections to	store in each output section	56
DESCRIPTION statement — insert	string as a .comment section	107
linker option -B lstrip —	strip local symbols from symbol table	20
linker option -s —	strip symbols and debugging information from the output file's symbol table	30, 69
elfdump — utility that produces a binary	structure dump of ELF object files and shared libraries (DLLs)	86
	STUB statement — obsolete	118
archiver command-line options	summary	79
AT&T-style input map-file directives	summary	14
elf2hex command-line options	summary	82
elfdump command-line options	summary	88
linker command-line options	summary	8
linker commands	summary	14
linker option -w —	suppress all warnings	31
elf2hex option -Q —	suppress copyright message	83
linker option -t —	suppress warnings about multiple definitions of common symbols of unequal sizes	30
linker command PUBLIC — reassign the value of an external	symbol	45

linker option -u — create unresolved (undefined)	symbol	30
size-symbol declarations — define linker option -B symbolic — bind intra-module global	symbol for output section size	63
linker option -B lstrip — strip local symbols from	symbol references to symbol definitions within the link	24
linker option -C symbols — display a listing of all symbols in the generated output	symbol table	20
linker option -J — export only the symbols listed in file when generating a shared library (DLL); limit dynamic	symbol table	24
linker option -s — strip symbols and debugging information from the output file's PST — program	symbol table	26
manipulating archive library members and archive	symbol table	30, 69
linker option -B	symbol table	72
linker option -b — do not do special processing of shared	symbol tables	79
linker option -C crossref — display a cross-reference listing of global	symbolic — bind intra-module global symbol references to symbol definitions within the link ...	24
linker option -z nodefs — allow undefined determining section size with linker-defined special linker-defined	symbols	18
linker option -C	symbols	24
linker option -s — strip	symbols	35
linker option -B lstrip — strip local global	symbols	68
linker option -B noplt — do not implicitly map	symbols	67
linker option -t — suppress warnings about multiple definitions of common	symbols — display a listing of all symbols in the generated output symbol table	24
linker option -C globals — display a sorted list of global common	symbols and debugging information from the output file's symbol table	30, 69
linker option -z defs — do not allow undefined	symbols from symbol table	20
SECTIONS command	symbols in libraries	72
archiver option -h — display archiver command-line	symbols into the PLT of the executable file	22
	symbols of unequal sizes	30
	symbols with their mappings	24
	symbols, defined	68
	symbols; force fatal error	35
	syntax conventions	48
	syntax help screen	79

linker option -H — display linker command-line	syntax help screen	25
invoking the linker from the EXETYPE statement — specify operating	system command line	7
	system where the shared library (DLL) or application runs	107
T		
archiver option	-t — display archive library member names	79
linker option	-t — suppress warnings about multiple definitions of common symbols of unequal sizes	30
elf2hex option	-t — specify Extended Tektronix Hex Record format	84
linker option -B lstrip — strip local symbols from symbol	table	20
linker option -C symbols — display a listing of all symbols in the generated output symbol	table	24
linker option -J — export only the symbols listed in file when generating a shared library (DLL); limit dynamic symbol	table	26
linker option -s — strip symbols and debugging information from the output file's symbol	table	30, 69
PST — program symbol	table	72
rewiring shared function calls through the PLT (procedure linkage	table)	70
manipulating archive library members and archive symbol	tables	79
linker option -B noalldyn — do not map dynamic	tables in virtual memory	21
specify the page size of	target memory configuration	41
linker option -B pagesize — specify page size of	target processor in bytes	22
elf2hex option -t — specify Extended	Tektronix Hex Record format	84
linker hex conversion option -xt — specify Extended	Tektronix Hex Record format	33
Extended	Tektronix Hex Record format, defined	96
linker	terminal errors, defined	76
linker option -z	text — do not allow output relocations against read-only sections; force fatal error	35
linker option -C — display listing of specified	type	24
SECTIONS keyword	TYPE — explicitly assign a type to an output section	54
elf2hex option -c — specify section	types to be converted	82

linker hex conversion option -xc — specify	
section	types to be converted 32
notational and	typographic conventions vii

U

linker option	-u — create unresolved (undefined) symbol 30
mapping	unassigned input sections 57
	undefined external references listed in memory
	map listing file 30
	undefined reference errors in libraries 73
linker option -z nodefs — allow	undefined symbols 35
linker option -z defs — do not allow	undefined symbols; force fatal error 35
linker option -u — create unresolved	(undefined) symbol 30
linker option -t — suppress warnings about	
multiple definitions of common symbols of	
linker option -C	unequal sizes 30
	unmangle — used with other -C options to display
	C++ names in unmangled form 24
linker option -z sections — diagnose	unreferenced files and input sections 35
linker option -u — create	unresolved (undefined) symbol 30
archiver ar — management	utility that groups object files into an archive library 77
elfdump —	utility that produces a binary structure dump of
	ELF object files and shared libraries (DLLs) 86
elf2hex —	utility to convert binary object file to hexadecimal
	ASCII format 80

V

linker option	-V — display linker version number and copyright
	banner prior to linking 30
archiver option	-v — display verbose output of archiver actions 79
environment	variable LD_LIBRARY_PATH 27, 34
environment	variable LIBPATH 27, 34
generating relocation fix-ups versus local	
copies of shared	variables 69
linker option -B nocopies — do not make	
local copies of shared	variables; insert relocation fix-ups 21
archiver option -v — display	verbose output of archiver actions 79
linker option -Q — specify whether	version information appears in output file 29
linker option -V — display linker	version number and copyright banner prior to
	linking 30
	VERSION statement — obsolete 118
explicit	virtual address assignment 63
	VIRTUAL DEVICE statement — obsolete 118

linker option -B noallocdyn — do not map dynamic tables in	virtual memory	21
memory map listing file — explains how sections are allocated in	virtual memory	29

W

linker option	-w — suppress all warnings	31
linker option -t — suppress	warnings about multiple definitions of common symbols of unequal sizes	30
linker	warnings, defined	75
elf2hex option -n — specify	width of memory device	83
linker hex conversion option -xn — specify	width of memory device	33
linker hex conversion option -xs — specify size and	width of memory device	34
PROM	width, defined	90
using	wildcards in file-name references in linker command files	39
using	wildcards in SECTIONS command	54
linker option -B	wired — generate non-swappable output file	24
BASE statement — specify default starting address for	writable data sections	103
linker option -I —	write interpreter pathname into program header	26
linker option -m —	write memory map listing file to standard output	29

X

appending flags or attributes to linker option	-x	31
archiver option	-x — extract specified archive library members	79
linker hex conversion option	-x — generate hex file; use default values	31, 92
linker option	-x help — display information about -x options for hex record generation	32
linker hex conversion option	-xc — specify section types to be converted	32
linker hex conversion option	-xi — specify number of interleaved banks	33
linker hex conversion option	-xm — specify Motorola S3 Hex Record format	33
linker hex conversion option	-xn — specify width of memory device	33
linker hex conversion option	-xo — specify name of generated hex file	33
linker hex conversion option	-xq — specify Mentor QUICKSIM Modelfile Hex Record format	33
linker hex conversion option	-xs — specify size and width of memory device	34
linker hex conversion option	-xt — specify Extended Tektronix Hex Record format	33

Y

linker option -YP — use paths as default search paths to
resolve subsequent -l name specifications 34

Z

linker option -z defs — do not allow undefined symbols; force
fatal error 35

linker option -z nodefs — allow undefined symbols 35

linker option -z sections — diagnose unreferenced files and input
sections 35

linker option -z text — do not allow output relocations against
read-only sections; force fatal error 35

using library function `_initcopy()` to zero bss sections 72

