ELF Linker and Utilities

User's Guide

Version 4.2

SA14-2563-00

Fifth edition (August 2000)

This edition of the IBM ELF Linker and UtilitiesUser's Guide applies to the IBM ELF Linker version 4.2 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 DOCUMENT “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 use of the information herein shall be free from third party intellectual property claims.

IBM does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and or changes in the product(s) and/or program(s) described in this document at any time. This document does not imply a commitment by IBM to supply or make generally available the product(s) described herein.

All performance data contained in this document was obtained in a specific environment, and is presented as an illustration. The results obtained in other operating environments may vary.

Address comments about this document to:

IBM Corporation
Department 5OB/Building 667
Box 12195
RTP, NC 27709

Printed in the United States of America.

4 3 2 1

IBM may have patents or pending patent applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, North Castle Drive, Armonk, NY 10504, United States of America.

Contents

Tables

Figures

About This Book

This **ELF Linker and Utilities 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
`Italic Courier`	Indicates formal parameters to be replaced by names or values you specify; also indicates input on the command line
Emphasized text	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 or braces

Terminology Conventions

A dynamically linked library (DLL) is also known as a shared object library in UNIX terminology. In this manual, the term *DLL* is used to refer to these libraries.

Where to Go for More Information

The main `readme` file describes any special files and provides informaion that was not available at the time the manuals were published.

The *Where to Go for More Information* section in the **High C/C++ Programmer's Guide** describes the following:

- Documents in the High C/C++ Toolset
- C and C++ Programming Documents
- Processor-Specific Documents
- Specifications and ABI Documents

For information about the Executable and Linking Format (ELF), see the **TIS Portable Formats Specification Version 1.1**. TIS Committee, 1993.

```
http://developer.intel.com/vtune/tis.htm
```

Using the ELF Linker 1

This chapter describes how to invoke the MetaWare linker; it contains the following sections:

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

See the **Programmer's Guide** for detailed information about using the driver.

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

Note: See the **Installation Guide** for the exact name of the driver command for your target processor. In this manual, we use **hc** to generically represent the driver command.

The High C/C++ driver automatically invokes the linker on files it recognizes as linker input files. For example, the following driver command invokes the linker to link together object files `file1.o` and `file2.o` and produce the executable output file:

```
hc file1.o file2.o
```

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

See the **Programmer's Guide** for information about file extensions recognized by the driver.

1.1.1 Resolving Conflicts in Linker and Compiler Option Names

The driver passes most linker options you specify on the driver command line to the linker. However, in cases where a linker option and a compiler option have the same name, the driver assumes the option is a compiler option.

To force the driver to pass linker options to the linker, use driver option **`-Hldopt`**. For example, the following command passes option **`-V`** (also a compiler option) to the linker:

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

See the **Programmer's Guide** for more information about using option **`-Hldopt`**.

For information about individual linker options, including whether an equivalent compiler option exists, see §2.1: *Linker Options Reference*.

1.2 Invoking the Linker from the Command Line

Note: See the **Installation Guide** for the exact name of the linker command for your target. In this manual, we use **`ld`** to generically represent the linker command.

This is the command-line syntax for invoking the linker:

```
ld [options] input_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.

- *`options`* is a series of optional command-line options. (See Chapter 2: *Linker Command-Line Options* for information about linker options.)
- *`input_file`* is the name of an object file (relocatable input file), archive file, or command file.

  The order of archive libraries on the command line is important in resolving external symbol references, because of the way the linker reads the files. (See §6.6: *How the Linker Processes Archive Libraries* and option **`-Bgrouplib`** for more information on this topic.)

- *`arg_file`* is the name of an optional argument file. (See §1.3: *Specifying Command-Line Arguments in a File* for information about argument files.)

1.3 Specifying Command-Line Arguments in a File

You can place frequently used linker command-line options 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 or driver command line. You can use as many lines as necessary. A newline is treated as whitespace.

Specifying an Argument File on the Driver Command Line

To specify a linker argument file on the **`hc`** command line, use driver option **`-Hldopt`**, and precede the argument-file name with the "at" symbol (`@`); for example:

```
hc  -Hldopt=@arg_file file.c
```

See the **Programmer's Guide** for information about driver option **`-Hldopt`**.

Specifying an Argument File on the Linker Command Line

To specify an argument file on the linker command line, enter the name of the file on the command line preceded by the "at" symbol (`@`); for example:

```
ld  @argument_file file.o
```

Note: Do not confuse argument files with linker command files.

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.

Command files contain linker commands that specify the placement of sections within an output file and perform other fine-tuning operations. For more information about command files, see Chapter 3: *Using Linker Command Files*.

1.4 Linking in Run-Time Library Files

The linker links the High C/C++ run-time libraries with the object files to resolve any external references. To list the libraries being linked in, specify driver option **`-V`**.

Embedded targets only

Note: For embedded targets, all MetaWare run-time libraries are static libraries whose names are of the form `lib`*`name`*`.a`.

Linker Command-Line Options 2

This chapter describes the linker command-line options; it contains the following sections:

§2.1: *Linker Options Reference*

Linker command-line options determine how the linker links object files, and what output is produced.

You can specify linker options in any order. You can intersperse options with file names on the command line. You can also place options before or after file names.

Note: Some options affect the behavior of subsequent options (for example, options **`-Bstatic`** and **`-Bdynamic`** affect how the linker interprets option **`-l`**), so be careful in what order you specify them.

2.1 Linker Options Reference

This section provides detailed information about each linker option. See §1.1: *Invoking the Linker from the High C/C++ Driver* for information about using linker options with the **`hc`** driver. See §1.2: *Invoking the Linker from the Command Line* for information about using linker options on the linker command line.

`-A` *`cmd_file`* — (Deprecated) Process an AMD-style linker command file

Note: Option **`-A`** has been deprecated, because the linker by default processes AMD-style command files as SVR3-style command files. See Chapter 3: *Using Linker Command Files* for more information.

`-b` — Do not do any special processing of shared symbols

Dynamic linking only

Option **`-b`** 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.

Note: Option **`-b`** is equivalent to the following combination:

```
-Bnocopies -Bnoplt
```

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.

Note: Use option **`-b`** 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.

For more information, see the following:

- §6.3: *Generating Relocation Fix-Ups Versus Local Copies*
- §6.4: *Rewiring Shared Function Calls Through the PLT*

`-Ball_archive` — Extract all members of an archive library

Option **`-Ball_archive`** causes the linker to extract all members of an archive library. This option is typically used to construct a DLL from an archive library. For example, this command directs the linker to extract all members from archive library `liby.a` and create DLL `liby.so`:

```
ld  liby.a  -G  -Ball_archive  -o liby.so
```

To display all undefined functions in an archive, invoke the linker with that archive alone:

```
ld -Ball_archive libc.a
```

This technique is useful for embedded development, because it shows what operating-system functions the archive depends on, which the embedded developer must provide.

`-Ballocatecommon` — Force allocation of common data

Incremental linking only

Option **`-Ballocatecommon`** forces the allocation of common data when you specify option **`-r`** for incremental linking. See §6.2: *Linking Incrementally* for more information.

Note:	Option **`-Ballocatecommon`** has no effect if you do not specify option **`-r`**.

`-Bbase=0x`*`address`*`[:0x`*`address`*`]` — Specify the origin address in hexadecimal

Embedded development only

Option **`-Bbase`** specifies the origin of the `.text` section. The linker uses `0x`*`address`* as the base address of the `.text` section. If the linker is generating a demand-loadable executable file, it might place the ELF header at this address. To keep the linker from placing the header at the specified address, use **`-Bnodemandload`** or **`-Bnoheader`**.

If you specify a second `0x`*`address`*, the linker uses that as the base address of the `.data` section.

By default, the starting address of the `.text` section is based on a convention determined by the operating system.

Same as **`-Bstart_addr`**.

`-Bcopydata` — Create an `INITDATA` entry for all writable data

Embedded development only

Option **`-Bcopydata`** directs the linker to create an **`INITDATA`** entry to initialize from “initdata” tables all writable data sections at start up. For more information, see the listing for **`INITDATA`** in §4.4: *SVR3-Style Command Reference*.

`-Bdefine:`*`sym`*`=`*`expr`* — Define a public symbol

Option **`-Bdefine`** defines *`sym`* as a public symbol with the value *`expr`*. This option has the same effect as defining a symbol with an Assignment command in a linker command file. (See §4.4: *SVR3-Style Command Reference* for information about the Assignment command.)

`-Bdynamic` — Search for DLL `lib`*`name`* when processing option `-l`

Dynamic linking only

Option **`-Bdynamic`** specifies that subsequent occurrences of option **`-l`** direct the linker to search for DLLs before it searches for static libraries.

More specifically, the **`-Bdynamic`** option directs the linker to interpret subsequent occurrences of option **`-l`** *`name`* to include the DLL `lib`*`name`*`.so`, `lib`*`name`*`.dll`, or *`name`*`.dll`, in that order.

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

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

Note: Option **`-Bdynamic`** is useful only when you specify dynamic linking, because the linker does not accept DLLs when it is linking statically.

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

`-Bgrouplib` — Scan archives as a group

Option **`-Bgrouplib`** directs the linker to scan archives as a group, so that mutual dependencies between archives are resolved without requiring that an archive be specified multiple times.

When the linker encounters an archive on the command line, it "merges" the contents of the archive with any preceding archives, then scans the result for unresolved references. This process is repeated for each subsequent archive the linker encounters.

Note: If a symbol is exported by more than one archive, the earliest one will always be extracted.

`-Bhardalign` — Force each output segment to align on a page boundary

Embedded development only

Option **`-Bhardalign`** 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.

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

Embedded development only

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

`-Bbase`
`-Bcopydata`
`-Bhardalign`
`-Bmovable`
`-Bnoallocdyn`
`-Bnodemandload`
`-Bnoheader`
`-Bnozerobss`
`-Bpagesize`
`-Bpictable`
`-Brogot`
`-Brogotplt`
`-Broplt`
`-Bstart_addr`
`-Bzerobss`

Caution: These special **`-B*`** options should not be used to develop non-embedded applications; that is, applications that run on an existing operating system (such as an application that runs on Solaris). An executable produced with these options generally will not load or run properly in a non-embedded environment.

`-Blstrip` — Strip local symbols from symbol table

Option **`-Blstrip`** 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.

`-Bmovable` — Make dynamic executable file movable

Embedded development only

Option **`-Bmovable`** directs the linker to render the dynamic executable file so that its origin address can be altered at load time (in a manner similar to a DLL).

`-Bnoallocdyn` — Do not map dynamic tables in virtual memory

Embedded development only

Option **`-Bnoallocdyn`** directs the linker to not map dynamic tables (`.dynamic`, `.rel`, and so on) in virtual memory. That is, the linker designates the associated sections as “not allocable”.

Option **`-Bnoallocdyn`** accommodates dynamic loaders that read relocation information directly from the ELF file instead of reading the information from virtual memory after the file is loaded.

`-Bnocopies` — Do not make local copies of shared variables; insert relocation fix-ups

Option **`-Bnocopies`** forces the linker to insert relocation fix-ups, instead of making local copies of shared variables.

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

1. The linker 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.
2. The linker can make a local copy of the variable and arrange for the dynamic linker to "rewire" the DLL so as to reference the local copy.

By default, the linker makes a local copy of the variable and arranges for the DLL to reference that local copy.

Note: When you use option **`-G`** (generate a DLL), option **`-Bnocopies`** is automatically turned `On`.

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

`-Bnodemandload` — Ignore boundary issues when mapping sections

Embedded development only

Option **`-Bnodemandload`** informs the linker that the output file does not need to be demand-page loadable, and directs the linker to ignore page-boundary issues when mapping sections into the output file.

`-Bnoheader` — Do not include ELF header in loadable segments

Embedded development only

Option **`-Bnoheader`** suppresses the inclusion of the ELF header in the loadable segments of a dynamically linked executable (DLL).

`-Bnoplt` — Do not implicitly map symbols into the PLT of the executable file

By default, the linker maps function entry-point symbols imported from 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 modification at load time.

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

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

However, option **`-Bnoplt`** can render the executable less shareable with other processes that are running the same program. Use option **`-Bnoplt`** if it is not necessary for your code to be shareable across processes.

For a more detailed discussion of this topic, see §6.4: *Rewiring Shared Function Calls Through the PLT*.

`-Bnozerobss` — Do not zero `bss` sections at run time

Embedded development only; PowerPC targets only

For PowerPC targets, the linker by default adds an **`INITDATA`** entry in start-up code to explicitly zero the following sections at run time: `.bss`, `.bss2`, `.sbss`, and `.sbss2` (that is, option **`-Bzerobss`** is the default for PowerPC targets).

Option **`-Bnozerobss`** directs the linker to not zero the `bss` sections at run time. When you specify this option, the program loader is responsible for zeroing the `bss` sections at load time.

See also option **`-Bzerobss`** and SVR3-style command **`INITDATA`**.

`-Bpagesize=`*`size`* — Specify page size of target processor in bytes

Embedded development only

Option **`-Bpagesize`** specifies the memory page size to be used when mapping sections into the ELF file. Sections are mapped so that:

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

The default page size is operating-system dependent.

`-Bpictable[=[text|data][,`*`name`*`]]` — Generate run time fix-up table

Embedded development only; PowerPC and ARM targets only

Option **`-Bpictable`** directs the linker to generate the necessary tables so that the text and data segments can each be moved to arbitrary addresses at start-up. These tables are processed by start-up code.

If you specify `text`, then the fix-up tables are placed in the read-only text segment. If you specify `data`, the fix-up tables are placed in the writable data segment.

`name` is the global symbol to be used by start-up code to refer to the base of the table. The default value for *`name`* is `__PICTABLE __`.

`-Bpurgedynsym` — Export only those symbols referenced by DLLs being linked against

Dynamic linking only

Option **`-Bpurgedynsym`** reduces the size of the dynamic symbol table of a dynamically linked executable. When you specify **`-Bpurgedynsym`**, the linker exports only those symbols necessary for dynamic linking, and those that are explicitly imported by DLLs that are being linked against. Only those symbols appear in the dynamic symbol table (`.dynsym`).

Caution: If the executable dynamically loads a DLL at run time and the DLL contains a reference to a symbol within the executable, you should not use option **`-Bpurgedynsym`**.

`-BpurgeNEEDED` — Include only explicitly referenced DLLs in the "needed" list of a generated module

Dynamic linking only

Option **`-BpurgeNEEDED`** specifies that the "needed" list of the generated module contains only those DLLs that are explicitly referenced. The "needed" list of a generated module contains the names of DLLs that must be loaded when the resulting module is loaded. By default, all DLLs specified on the linker command line appear in the list, regardless of whether there are explicit references to them.

`-Brel16` — Use certain non-ABI relocation types

PowerPC targets only

When you specify option **`-Brel16`**, the linker replaces certain ABI relocation types with non-ABI relocation types, as follows:

ABI	Non-ABI
`R_PPC_ADDR16_LO`	`R_PPC_REL16_LO (=200)`
`R_PPC_ADDR16_HI`	`R_PPC_REL16_HI (=201)`
`R_PPC_ADDR16_HA`	`R_PPC_REL16_HA (=202)`

Non-ABI relocation types behave like `R_PPC_RELATIVE` in that there is no target symbol referenced.

Option **`-Brel16`** affects the way the dynamic relocation table is constructed for PowerPC-based DLLs. For generated DLLs to be loadable at run time, the DLL loader must understand the non-ABI relocation types and process them appropriately. Such is not the case under current UNIX (Solaris) systems.

If all modules of the DLL are compiled with the **`-Kpic`** compiler option (that is, they are position-independent), the **`-Brel16`** option is moot because no `R_PPC_ADDR16*` type relocations will be generated.

`-Brogot` — Place the `.got` section in a read-only section

Embedded development only

By default, the linker places the Global Offset Table (`.got`) section into 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 **`-Brogot`** 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.)

`-Brogotplt` — Place the `.got` and `.plt` sections in read-only sections

Embedded development only

When you specify option **`-Brogotplt`**, the linker places the `.got` and `.plt` sections in a read-only section.

Option **`-Brogotplt`** is equivalent to the following combination:

```
-Brogot -Broplt
```

`-Broplt` — Place the `.plt` section in a read-only section

Embedded development only

By default, the linker places the Procedure Linkage Table (`.plt`) section 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 **`-Broplt`** 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.)

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.

`-Bstart_addr``=0x`*`address`*`[:0x`*`address`*`]` — Specify origin address in hexadecimal

Embedded development only

Option **`-Bstartaddr`** specifies the origin of the `.text` section. The linker uses `0x`*`address`* as the base address of the `.text` section. If the linker is generating a demand-loadable executable file, it might place the ELF header at this address. Use **`-Bnodemandload`** or **`-Bnoheader`** to keep the linker from placing the header at the specified address.

If you specify a second `0x`*`address`*, the linker uses that as the base address of the `.data` section.

By default, the starting address is based on a convention determined by the operating system.

Same as **`-Bbase`**.

`-Bstatic` — Search for static library `lib`*`name`* when processing `-l` *`name`*

Static linking only

Options **`-Bstatic`** specifies that subsequent occurrences of option **`-l`** direct the linker to search only for static libraries.

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

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

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

`-Bsymbolic` — Bind intra-module global symbol references to symbol definitions within the link

Dynamic linking only

Option **`-Bsymbolic`** binds intra-library references to their symbol definitions within the DLL, if definitions are available.

When you do not specify **`-Bsymbolic`**, references to global symbols within DLLs can be overridden at load time by like symbols being exported in the executable or in preceding DLLs.

`-Bzerobss` — Zero bss sections at run time instead of load time

Embedded development only

Option **`-Bzerobss`** directs the linker to add an **`INITDATA`** entry in start-up code to zero the following sections at run time: `.bss`, `.bss2`, `.sbss`, and `.sbss2`. Ordinarily, the program loader is responsible for zeroing these sections at load time.

Note: Start-up code must call the C run-time function `_initcopy()` to actually zero the `.bss` section.

PowerPC targets only

For PowerPC targets, option **`-Bzerobss`** is the default; to disable this initialization, you must specify option **`-Bnozerobss`**.

For more information, see SVR3-style command **`INITDATA`**.

`-C` *`listing_type`* — Display listing of specified type

Option **`-C`** displays a listing with the attribute specified by *`listing_type`*. The listing is sent to standard output. To save the listing, redirect it to a file.

`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=`*`n`*	Specifies the number of lines per page in the displayed listing. The default value for *`n`* is `60`. To suppress page ejects, set *`n`* to `0` (zero).
`sections`	Displays a listing of section mappings with global symbols interspersed.
`sectionsonly`	Displays a listing of section mappings.

`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 unmangled form.

To specify multiple listing attributes, specify option **`-C`** multiple times; for example:

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

Option **`-m`** is equivalent to **`-C`** `sections`.

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

Option **`-d`** 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 DLL references are required, the linker generates a statically linked executable, even if you do not specify **`-d`** `n`.

`-D` *`cmd_file`* — (Deprecated) Process a Diab-style a command file

Note: Option **`-D`** has been deprecated, because the linker by default processes Diab-style command files as SVR3-style command files. See Chapter 3: *Using Linker Command Files* for more information.

Caution: Option **`-D`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

`-e` *`entry_name`* — Specify program entry point

Option **`-e`** 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 DLL

Dynamic linking only

Option **`-G`** produces a dynamically linked library (DLL).

`-h` *`name`* — Use *name* to refer to the generated DLL

Dynamic linking only

Ordinarily, when a DLL is referenced as an input file to the linker, the name of the DLL is inserted in the “needed” list of the generated executable (or DLL). By default, this name is the full path name of the file that contains the DLL.

Option **`-h`** designates an alternate name to appear in the “needed” list of any executable that references this DLL. *`name`* is usually a relative path name.

For example, the following command instructs the linker to refer to `/lib/libX.so` as simply `libX.so` in the generated DLL:

```
ld -G -h libX.so a.o b.o c.o -o /lib/libX.so
```

Option **`-h`** has meaning only in conjunction with option **`-G`**.

Caution: Option **`-h`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

`-H` — Display linker command-line syntax help screen

Option **`-H`** displays on standard output a summary page of the linker command-line syntax, options, and flags.

See also options **`-Bhelp`** and **`-x`**`help`.

Caution: Option **`-H`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

`-I` *`name`* — Write *name* into the program header as the path name of the dynamic loader

Dynamic linking only

A dynamically linked executable contains an “interpreter” entry in the program header table, which identifies the path of the dynamic loader. (The default path is target dependent.) Option **`-I`** *`name`* overrides the default and writes *`name`* into the program header as the path name of the dynamic loader.

Caution: Option **`-I`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

`-J` *`file`* — Export only the symbols listed in *`file`* when generating a DLL

Dynamic linking only

Option **`-J`** 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 DLL being built. Each symbol name must appear on a line by itself without any preceding or embedded whitespace.

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

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

Option **`-l`** directs the linker to search for a library whose name contains *`name`*, in order to resolve 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 option **`-l`** on the command line any number of times. The placement of option **`-l`** is significant, because the linker searches for a library as soon as it encounters the library's name. See §6.6: *How the Linker Processes Archive Libraries* for more information about the order of archive libraries on the command line.

How the Linker Determines Which Library to Search For

When you dynamically link an application, the options **`-Bstatic`** and **`-Bdynamic`** serve as a binding mode toggle that dictates how the linker processes option **`-l`**.

If **`-Bdynamic`** is in effect and you specify **`-l`** *`name`*, the linker searches the library search paths until it finds one of the following DLLs, in the order given here:

1. `lib`*`name`*`.so`
2. `lib`*`name`*`.dll`
3. *`name`*`.dll`

4. `lib`*`name`*`.a`
5. *`name`*`.lib`

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

1. `lib`*`name`*`.a`
2. *`name`*`.lib`

For information about how the linker determines the library search paths, see the listing for option **`-L`**.

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

Option **`-L`** directs the linker to search in the specified directories for a library named with a subsequent option **`-l`** *`name`*.

`paths` is a list of directories separated by colons (`:`) on UNIX hosts, or by semicolons (`;`) on Windows/DOS hosts.

How the Linker Determines the Library Search Paths

Cross linking In cross linking, the linker searches for libraries in the following directory locations, in the order given:

1. Directories specified with option **`-L`**
2. Default search directories specified with option **`-YP`**

Host linking Environment variable `LD_LIBRARY_PATH` consists of one or more lists of directories that the linker will search for libraries specified with option **`-l`**.

You can define `LD_LIBRARY_PATH` as a single directory list:

```
LD_LIBRARY_PATH=dir_list
```

`dir_list` is a list of directories separated by colons (`:`) on UNIX hosts, or by semicolons (`;`) on Windows/DOS hosts.

The linker searches for libraries in the following directory locations, in the order given:

1. Directories specified with option **`-L`** *`paths`*
2. The directories in *`dir_list`*

3. Default search directories specified with option **`-YP`**
4. If option **`-YP`** is not specified, directories specified in environment variable `LIBPATH` or, if `LIBPATH` is not defined, the host system's built-in list of standard library directories

You can also define `LD_LIBRARY_PATH` as two directory lists:

```
LD_LIBRARY_PATH=dir_list_1|dir_list_2
```

In this case, the linker searches for libraries in the following directory locations, in the order given:

1. The directories in *`dir_list_1`*
2. Directories specified with option **`-L`** *`paths`*
3. The directories in *`dir_list_2`*
4. Default search directories specified with option **`-YP`**
5. If option **`-YP`** is not specified, 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 hosts only, you can use a semicolon (`;`) instead of the vertical bar (`|`) to separate the two *`dir_list`* items in the `LD_LIBRARY_PATH` definition.

Host-System Library Directories

If `LIBPATH` is not defined, the linker behaves like a native host linker, and searches the following default host-system library directories:

UNIX SVR4 hosts:	`/usr/ccs/lib:/usr/lib:/usr/local/lib`
SunOS hosts:	`/lib:/usr/lib:/usr/local/lib`
Windows/DOS hosts:	`\lib`

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

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

Option **`-m`** is equivalent to **`-C`**`sections`.

To save the listing file, redirect it to a file.

`-M` *`cmd_file`* — Process an SVR4-style command file (input map file)

Static linking only

Option **`-M`** specifies an SVR4-style command file for customizing the mapping of sections and symbols from object files to output files.

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

See Chapter 5: *SVR4-Style Command Files* for more information.

Caution: Option **`-M`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

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

Option **`-o`** 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

Option **`-q`** suppresses display of the linker copyright message.

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

Option **`-Q`** `y` places linker 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 only

Option **`-r`** 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.

`-R` *`pathlist`* — Define the search path used to resolve references to DLLs at run time

Dynamic linking only

Option **`-R`** designates the search path that the dynamic linker uses to resolve references to DLLs at run time. (The term *dynamic linker* refers to the linker/loader provided by the operating system.)

`pathlist` is a list of path names separated by colons (`:`) for UNIX targets, or by semicolons (`;`) for Windows/DOS targets.

- If you do not specify *`pathlist`*, the linker uses a default path, which varies depending on your system.
- If you specify this option more than once on the linker command line, only the last instance takes effect.

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

Option **`-s`** 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`**). By default, the linker writes a symbol table to the object file.

If you specify option **`-r`**, the linker strips only non-global symbols.

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

Option **`-t`** 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`*

Option **`-u`** 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 and copyright banner prior to linking

Option **`-V`** 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: The version number and copyright banner are displayed by default, so there is no need to use option **`-V`**; it is provided only for backward compatibility.

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

Caution: Option **`-V`** is also implemented as a compiler option. See §1.1.1: *Resolving Conflicts in Linker and Compiler Option Names* for more information.

`-w` — Suppress all warning messages

Option **`-w`** suppresses all linker warning messages. See Chapter 7: *Linker Error Messages* for more information about linker warning and error messages.

`-x[`*`attribute`*`]` — Generate a hex file

Embedded development only

Option **`-x`** 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.

Note: Option **`-x`** generates a new object file and its corresponding hex files as a single-step process. To convert an existing object file into hex files, use the **`elf2hex`** utility.

`attribute` specifies the characteristics of the hex file. 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 specify a value for an attribute, the value does not have to be separated from the attribute by commas or whitespace.

Table 2.1 lists valid values for *`attribute`*.

Table 2.1 Hex Conversion Characteristics

Attribute	Conversion Characteristic	**`elf2hex`** Option
`c` *`list`*	Specify section types to be converted.	**`-c`**
`i` *`bank`*	Select the number of interleaved banks.	**`-i`**
`help`	Display help about the **`-x`** option and its attributes	
`m`	Specify a hex file in Motorola S3 Record format	**`-m`**
`q`	Specify a hex file in Mentor QUICKSIM Modelfile format.	**`-q`**
`t`	Specify a hex file in Extended Tektronix Hex Record format.	**`-t`**
`n`	Specify width of the memory device	**`-n`**
`o`	Specify the name of the generated hex file	**`-o`**
`s`	Identify size of memory device.	**`-s`**

The attributes of option **`-x`**, except for `help`, are equivalent to options for the **`elf2hex`** conversion utility, and linker option **`-x`***`attribute`* takes the same arguments and has the same default as its **`elf2hex`** counterpart. If you do not specify *`attribute`*, option **`-x`** uses default values for all hex conversion characteristics. See §8.2.3: *elf2hex Option Reference* for more information.

`-Xcheck` — Check for inconsistent function calls

picoJava targets only

Option **`-Xcheck`** directs the linker to insert code to check for functions that are passed more or fewer arguments than the function definition specifies. Such an inconsistency can cause severe runtime problems on the picoJava architecture.

-Xsuppress_dot — Suppress leading period in symbol names

PowerPC targets only

Option **-Xsuppress_dot** instructs the linker to skip leading periods in symbol names when it resolves undefined symbols. This option is necessary when you link against the GreenHills-compiled libraries provided in the pSOS distribution.

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

Option **-YP** directs the linker to search the directories in *path* to resolve subsequent **-l** specifications. (The linker first searches the directories specified with option **-L**, and then the directories specified with option **-YP**.)

paths is a list of directories separated by colons (:) on UNIX hosts, or by semicolons (;) on Windows/DOS hosts.

See the listing for option **-L** for more information about how the linker determines the library search path.

-zdefs — Do not allow undefined symbols; force fatal error

Option **-zdefs** forces a fatal error if any undefined symbols remain at the end of the link. This is the default.

-zdup — Permit duplicate symbols

Option **-zdup** instructs the linker to issue a warning for duplicate global symbol definitions, instead of an error diagnostic.

-zlistunref — Diagnose unreferenced files and input sections

Option **-zlistunref** directs the linker to diagnose files and input sections that are not referenced, and to display 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.

When you use option **-zpurge** with option **-zlistunref**, the linker displays a table of the omitted sections.

`-znodefs` — Allow undefined symbols

Option **`-znodefs`** allows undefined symbols. You can use **`-znodefs`** to build an executable in dynamic mode and link with a 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 option **`-znodefs`** with caution, or address faults might occur at run time.

`-zpurge` — Omit unreferenced input sections

Option **`-zpurge`** transitively omits any unreferenced allocatable input section. When you use option **`-zpurge`** with option **`-zlistunref`**, the linker displays a table of the omitted sections.

- If an executable is being generated, the linker assumes the following sections are the roots from which control is entered:
  - the section containing the entry point
  - the section `.init`
  - the section `.fini`

  All sections are omitted, except those that are transitively accessible from these sections.

- If a DLL or relocatable object file is being generated (that is, if you specify linker option **`-G`** or linker option **`-r`**), all sections that export global symbols are considered to be the roots. In other words, only those sections containing global symbols, or sections that are transitively referenced from these sections, will be retained.

Option **`-zpurge`** can be used with compiler toggle `Each_function_in_own_section` to transitively eliminate unreferenced functions. See the **Programmer's Guide** for information about this toggle.

`-zpurgetext` — Omit unreferenced executable input sections

Option **`-zpurgetext`** behaves like option **`-zpurge`**, except that option **`-zpurgetext`** omits only unreferenced *executable* input sections. Option **`-zpurgetext`** prevents the linker from throwing away unreferenced data sections that contain, for example, source-control information strings (for instance, `rcsid`).

Option **`-zpurgetext`** can be used with compiler toggle `Each_function_in_own_section` to transitively eliminate unreferenced functions. See the **Programmer's Guide** for information about this toggle.

`-ztext` — Do not allow output relocations against read-only sections

Dynamic linking only

Option **`-ztext`** forces an error diagnostic if any relocations against non-writable, allocatable sections remain at the end of the link. Use the **`-ztext`** option in dynamic mode only.

Using Linker Command Files 3

This chapter provides general information about linker command files, and explains how to use linker files; it contains the following sections:

Overview The linker automatically maps input sections from object files to output sections in executable files. If you are linking statically, you can change the default linker mapping by invoking a linker command file.

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.

The ELF linker can read a series of commands provided in a command file, which makes it possible for you to customize the mapping of sections and symbols in input files to output sections or segments.

3.1 Command-File Types Supported

The linker supports two popular styles of command file:

- SVR3-style command files, the default
- SVR4-style command files (also called AT&T-style map files), specified with command-line option **`-M`**

We recommend that you use the SVR3-style command-file format, because of its greater functionality and wider portability, and because the GNU and Diab Data linkers also support this format.

Note: Linker command-file formats are not interchangeable. For example, you cannot use AT&T map-file commands in an SVR3-style command file.

For information about a specific command file type, see the following:

- Chapter 4: *SVR3-Style Command Files*
- Chapter 5: *SVR4-Style Command Files*

Support for Diab-Style and AMD-Style Command Files

For backward compatibility, the linker supports Diab-style command files (specified with option **`-D`**) and AMD-style command files (specified with option **`-A`**).

However, the features in each of these formats have been merged into the new SVR3 command-file format, and these command-line options have been deprecated.

The linker processes Diab-style and AMD-style command files as SVR3-style command files.

3.2 Classification of Sections in Executable Files

Sections in executable files are classified according to type:

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

3.3 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**

- `abc*` matches any name that begins 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; for example, `/lib/libc.a(*)` matches any member of `/lib/libc.a`.

? — Match exactly one character

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

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

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

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

`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

`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

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

`\` — Escapes other wildcard characters

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

SVR3-Style Command Files 4

This chapter describes SVR3-style command files and how to use the commands with the MetaWare linker; it contains the following sections:

§4.1: *Specifying an SVR3-Style Command File*

§4.2: *Sample Command File*

§4.3: *Command-File Conventions*

§4.4: *SVR3-Style Command Reference*

Overview You use SVR3-style command files to do the following:

- Specify how input sections are mapped into output sections.
- Specify how output sections are grouped into segments.
- Define memory layout.
- Explicitly assign values to global symbols.

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

For a complete discussion of SVR3-style command files, see Appendix B: *Mapfile Option* in the **AT&T UNIX System V Release 3 Programmer's Guide: ANSI C and Programming Support Tools**.

4.1 Specifying an SVR3-Style Command File

By default, the linker assumes that any file on the command line that is not recognized as either a relocatable oject file or an archive library is a UNIX SVR3-style linker command file. To pass an SVR3-style command file to the linker, place it on the command line; for example:

```
ld file1.o cmd_file.cmd
```

You can specify multiple command files; the linker processes them as if they were concatenated.

4.2 Sample Command File

This sample command file is explained in detail in the following sections.

Example 4.1 Sample SVR3-Style Command File

```
#  This is a comment
/* This is also a comment */
// This is another comment

MEMORY {
   // These commands describe memory locations
   RAM:  ORIGIN = 0x00010000 LENGTH = 1M
   ROM:  ORIGIN = 0xFF800000 LENGTH = 512K
}

SECTIONS {
   GROUP : {
      .text ALIGN(4) BLOCK(4):
      {
         // Link code sections here
         * (.text)
         // Link C++ constructor and destructors here
         * (.init , '.init$*')
         * (.fini , '.fini$*')
      }
      .initdat ALIGN(4): {}
      .tls ALIGN(4): {}
      .rodata ALIGN(8) BLOCK(8):
      {
         * (TYPE lit)
      }
   } > ROM
   GROUP : {
      .data ALIGN(8) BLOCK(8):
      {
         . = ALIGN(8);
         _SDA_BASE_ = .;
         * (.rosdata, .sdata, .sbss)
         . = ALIGN(8);
         _SDA2_BASE_ = .;
         * (.rosdata2, .sdata2, .sbss2)
         * (TYPE data)
      }
      .bss:
      {
         * (TYPE bss)
      }
      .heap (BSS) ALIGN(8) BLOCK(8):
      {
         ___h1 = .;
         * (.heap)
         ___h2 = .;
         // Make the heap at least 8K (optional)
         . += (___h2 - ___h1 < 8K) ? (8K - (___h2 -
               ___h1)) : 0;
      }
      .stack (BSS) ALIGN(8) BLOCK(8):
      {
         // Use this space as the stack at startup.
         ___s1 = .;
         * (.stack)
         ___s2 = .;
         // Make the stack at least 8K (optional)
         . += (___s2 - ___s1 < 8K) ? (8K - (___s2 -
               ___s1)) : 0;
      }
   } > RAM
}

// Mark unused memory for alternate heap management pool
__FREE_MEM     = ADDR(.stack) + SIZEOF(.stack);
__FREE_MEM_END = ADDR(RAM) + SIZEOF(RAM);
```

Description of Previous Example

Example 4.1 declares two memory regions, ROM and RAM. The **SECTIONS** command groups non-writable program sections to ROM and writable sections to RAM. (This is not a requirement.)

External symbols __FREE_MEM and __FREE_MEM_END (at the end of the example) provide a way to describe those memory regions on your target board that are not allocated to parts of your program. Here is an example of C code that uses these symbols:

```
extern char __FREE_MEM[], __FREE_MEM_END[];
unsigned sizeof_free_mem() {
   return __FREE_MEM_END - __FREE_MEM;
   }
```

The run-time library uses .heap for dynamic memory allocation. You can access that memory by calling malloc(), sbrk() or the C++ **new** operator.

The default application start-up file crt1.o assigns the end of section .stack to the stack pointer of the main application thread.

You can configure the size of the heap and stack at link time by changing the linker command file. You can also declare arrays in your application that the linker will map into these sections. Here is an example of C code that uses an array to extend the stack by 512K:

```
#pragma Off(multiple_var_defs)
#pragma BSS(".stack")
char __addtostack[512*1024];
#pragma BSS
```

In the preceding code, the pragmas force the compiler to create array __addtostack in an uninitialized data section named .stack. The linker combines all sections named .stack together, extending the size of the output section in the application.

Embedded PowerPC targets only

On embedded PowerPC targets, the run-time library uses section .initdat to zero the .bss section and copy ROM to RAM, because the program loader does not perform this action by default.

The section .tls is used by multi-threaded applications. Variables such as errno are global and could be destroyed by several threads. The compiler uses a mechanism called “thread-local storage” to deal with this problem.

See the `readme` file and the **High C/C++ Programmer's Guide** for more details about thread-local storage and the `.tls` section.

4.3 Command-File Conventions

The following conventions apply to SVR3-style linker command files:

- Keywords are case sensitive. They must be UPPERCASE.
- Commands and their arguments can start in any column. Separate arguments from their respective commands by whitespace.
- Comments start with a pound sign (#) and end with a newline. You can also use C-style comment syntax (`/* ... */`) or C++-style syntax (`//` to end of line).
- Numbers must be specified as C constants (for example, `123` or `0x89ABC`).
  - To express kilobytes, suffix a number with `K` (for example, `24K`) — the number will automatically be multiplied by 1024.
  - To express megabytes, suffix a number with `M` (for example `16M`) — the number will automatically be multiplied by 1024 * 1024.
- You can use wildcard characters to reference file names in command files. Any name containing wildcard characters must be enclosed in single or double quotation marks. See §3.3: *Using Wildcards in File-Name References* for more information.

4.3.1 Expressions

The **`SECTIONS`** and **`MEMORY`** commands can contain expressions.

An *expression* is any combination of numbers and identifiers that contains one or more of the following C-language operators:

```
!    ~    -
*    /    %
+    -
>>   <<
==   !=    >    <    <=   >=
&
|
&&
||
?:
```

Operator precedence is the same as in the C language. You can use parentheses to alter operator precedence.

4.4 SVR3-Style Command Reference

Table 4.1 lists the commands and command types for SVR3-style command files.

Table 4.1 SVR3-Style Commands

Command or Command Type	Purpose
Argument	Specify any command-line argument recognized by the linker.
Assignment	Assign a value to an identifier.
Datum	Generate tables or write values in an output section
DEMANDLOAD	Specify an executable that can be demand-loaded from disk.
INITDATA	Specify output sections to initialize at run time.
LOAD	Read input files.
MEMORY	Specify a range of memory.
NODEMANDLOAD	Specify an executable that cannot be demand-loaded from disk.

Command or Command Type	Purpose
`OUTPUT`	Specify the name of the output file.
`SECTIONS`	Specify how to map input sections to output sections.
`START`	Specify program entry point

These commands and command types are covered in more detail in the following sections.

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.

Argument

Specify any command-line argument recognized by the linker

Syntax **{** *`argument`***...** **}**

Description *`argument`* is any command-line argument recognized by the linker, such as an option or an input-file name. The first argument on a line must be an option; that is, it must begin with a dash(**-**). (See Chapter 2: *Linker Command-Line Options* for a description of linker command-line options.)

Example

```
LOAD main.o,alpha.o,lib_1.a
-o test.out
-znodefs
```

This example instructs the linker to load object files `main.o` and `alpha.o`, scan archive file `lib_1.a`, and generate output file `test.out`, allowing undefined symbols.

Note: MetaWare recommends specifying command-line options in makefiles or project files instead of in linker command files. Command-line options are generally not portable to other linkers. Path names in linker command files can lead to problems when files are copied from one project to another, or from one host operating system to another. (See §1.3: *Specifying Command-Line Arguments in a File* for another way to specify command-line options.)

Assignment

Assign a value to an identifier

Syntax *`assignment`*

`assignment` is one of the following:

- *`identifier`* *`assignment_op`* *`expression`*`;`
- `.` *`assignment_op`* *`expression`*`; /* Only in sections */`

`assignment_op` is one of the following assignment operators:

```
=   +=  -=  *=  /=
```

Description You use an assignment to set or reset the value of an identifier.

If the assignment is in a statement, you can use a period (`.`) to represent the current program counter. This symbol for the program counter is useful for creating a hole (an empty range of memory) in the output section. Example 2 shows how to create a hole. (See the **`SECTIONS`** section for information about statements.)

Example 1 This example sets the identifier `_text1` to the current value of the program counter:

```
_text1 = .;
```

Example 2 This example creates a hole in the current output section by advancing the location pointer 2000 bytes:

```
. += 2000;
```

Example 3 This example declares a global symbol to a constant value:

```
_ZERO = 0;
```

Datum

Generate tables or write values in an output section

Syntax *`datum`*

`datum` is one of the following:

- **BYTE**(*number*[,*number*]...)
- **SHORT**(*number*[,*number*]...)
- **LONG**(*number*[,*number*]...)
- **QUAD**(*number*[,*number*]...)
- **FILL**(*number*)

Description Datum directives can be specified within the body of an output section specification. They allow you to put generated tables or write values at key locations within an output section.

Example 1

```
SECTIONS
{
   .rodata:
   {
      * (TYPE lit) ;
      BYTE(1,2,3,4);
      SHORT(5,6);
      LONG(0x1c8674);
      FILL(64); //Generate 64 bytes of fill characters
   }
}
```

Example 2 This example assumes input sections named `.text`, `.data`, and `.bss` only. The linker will add a third section named `.copytable` which contains a simple table that could be used to initialize memory at startup. This is a contrived example. (See **`INITDATA`** for an automated way to initialize your program memory at startup or reset.)

```
SECTIONS
{
   .text : {}
   .data : {}
   .bss : {}
   .copytable ALIGN(4) :
   {
      __COPY_TABLE_START = .;
      LONG(ADDR(.text), ADDR(.text) + SIZEOF(.text));
      LONG(ADDR(.data), ADDR(.data) + SIZEOF(.data));
      LONG(ADDR(.bss),  ADDR(.bss)  + SIZEOF(.bss));
      LONG(0);
      COPY_TABLE_END = .;
   }
}
```

DEMANDLOAD

Specify an executable that can be demand-loaded from disk

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

page_size is the page size (in bytes) of the target memory configuration; its value must be an integral power of 2. The default value of *page_size* 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:

```
offset(s) modulo page_size ==
address(s) modulo page_size
```

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

```
DEMANDLOAD 65536
```

INITDATA

Specify output sections to initialize at run time

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

`section` takes one of the following forms:

`sect_name`	Name of the output section to be initialized (can be wildcarded).
`!`*`sect_type`*	Type of section to be initialized. Valid types are `bss`, `data`, and `lit`.

Description The **`INITDATA`** command specifies an output section to be initialized at run time. This capability is required for programs that must run in ROM.

You can reference the output section by name, or by type. If you reference the section by type, all sections of the specified type are initialized.

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.)

Your program must call the library function `_initcopy()` at the start of program execution to initialize the control sections from the table in `.initdat`. (`_initcopy()` is available in object format in the High C/C++ standard library, and in source format in the High C/C++ `lib` directory.)

Caution: The application program must call `_initcopy()` early in the startup sequence. Any global variables read prior to `_initcopy()` might contain garbage. Any global variables written to prior to `_initcopy()` might be reinitialized.

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).

Caution: Do not specify your stack in the list of sections to be zeroed. If you do, _initcopy() will not be able to return to its caller because the return address on the stack will have been cleared.

The .stack section is not considered a bss section as far as the **INITDATA** mechanism is concerned; this prevents the stack from being zeroed when you use **INITDATA** to zero bss sections, as follows:

```
INITDATA !bss
```

Example 1 This **INITDATA** command directs 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 sections of type data, regardless of how they are named, and to zero sections named .bss and .sbss.

```
INITDATA !data, .bss, .sbss
```

LOAD

Read input files

Syntax

```
LOAD input_file[,input_file]...
INPUT(input_file[,input_file] ...
```

input_file is the name of an input object file, a library, or another SVR3 linker command file.

Note: If `input_file` is not in the current working directory, you must specify its full path name.

Description The **`LOAD`** command specifies input object files to be linked as if you specified them on the command line. The linker uses an input file's internal format to determine the nature of the file.

The linker reads the input files, regardless of whether you specify them on the command line or in a command file, in the order it encounters them. For example, given this command line:

```
ld file1.o file2.o 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`.

Note: MetaWare recommends specifying files on the command line instead of using the **`LOAD`** command. The **`LOAD`** command is generally not portable to other linkers. Path names in linker command files can lead to problems when files are copied from one project to another, or from one host operating system to another. (See §1.3: *Specifying Command-Line Arguments in a File* for another way to pass input file names to the linker.)

Note: The order of archive libraries on the command line is important in resolving external symbol references, because of the way the linker reads the files. (See §6.6: *How the Linker Processes Archive Libraries* and option **`-Bgrouplib`** for more information on this topic.)

MEMORY

Specify a range of memory

Syntax **`MEMORY`** `{` *`memory_specification`*`...` `}`

A *memory_specification* has the following syntax:

```
memory_range : ORIGIN = expression[ , ]
   LENGTH = expression[ , ]
```

Description A *memory_specification* names a range of memory (*memory_range*) and defines where it begins and how large it is.

You specify the starting address of the memory range with the keyword **ORIGIN**, followed by an assignment operator (=) and an expression that evaluates to the starting address. The expression can be followed by a comma.

You specify the size of the memory range with the keyword **LENGTH**, followed by an assignment operator (=) and an expression that evaluates to the size. The expression can be followed by a comma, to separate this memory specification from the next one.

Example

```
MEMORY {
   range_1 : ORIGIN = 0x1000, LENGTH = 0x8000,
   range_2 : ORIGIN = 0xa000, LENGTH = 0xc000
   }

SECTIONS {
   .text : {} > range_2
   .data : {} > range_1
   .bss  : {} > range_1
   }
```

This memory command specifies two memory ranges:

- `range_1` begins at address `0x1000` and is `0x8000` bytes in size.
- `range_2` begins at address `0xa000` and is `0xc000` bytes in size.

The **SECTIONS** command in this example allocates output section `.text` to `range_2` and output sections `.data` and `.bss` to `range_1`.

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

Syntax **NODEMANDLOAD**

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

For information about demand loading, see the entry for **DEMANDLOAD**.

OUTPUT

Specify the name of the output file

Syntax **`OUTPUT`**`(`*`filename`*`)`

Description The **`OUTPUT`** command specifies the name of the output file. The default output-file name is `a.out`.

SECTIONS

Specify how to map input sections to output sections

Note: Before you use the **`SECTIONS`** command, we recommend that you read this entire section, including the examples, to get a general idea of the possible forms the **`SECTIONS`** command can take. Doing so will help you understand this command's complex syntax.

Syntax **`SECTIONS`** `{` *`entry`*`...` `}`

An *`entry`* consists of a section or a group:

```
{ section | group }
```

`section` has the following syntax:

```
output_sspec
   [ qualifier... ] : { [ statement... ] }
   [ = 2_byte_pattern ]
   [ > memory_block ]
```

`group` has the following syntax:

```
GROUP
   [ qualifier... ] : { [ section... ] }
   [ > memory_block ]
```

`qualifier` can be any of the following, all optional:

- *`absolute_address`* or **`BIND`**`(`*`expression`*`)`
- **`ALIGN`**`(`*`expression`*`)`
- **`LOAD`**`(`*`expression`*`)`
- **`SIZE`**`(`*`expression`*`)`
- **`BLOCK`**`(`*`espression`*`)`

A section (or a group that contains one or more sections) can contain one or more *`statement`* items. *`statement`* is one of the following:

- *`file_name`* or *`regular_expression`*, followed by an optional list of input section specifications in parentheses: (*`input_sspec`*`...`). The *`input_sspec`* specifier uses the following syntax:

  ```
  input_file (sect_spec)
  "input_file (member)" (sect_spec)
  ```

- *`identifier assignment_op expression`*`;`
- `.` *`assignment_op expression`*`;` (only inside a section definition)
- **`STORE`**`(`*`value_to_store, size`*`)`
- **`TABLE`**`(`*`pattern`*, *`size`*`)`

`assignment_op` is one of the following assignment operators:

```
=   +=  -=  *=  /=
```

Description You use the **`SECTIONS`** command to map input sections (*`input_sspec`*) to an output section (*`output_sspec`*).

An output section contains all input sections you specify in the statement portions of the corresponding *`output_sspec`*. If you do not include any explicit input sections, the linker recognizes as input only those input sections having the same name as the output section, for example `.data` or `.text`.

Groups, specified with the keyword **`GROUP`**, enable you to map multiple output sections consecutively into one continuous memory block, or *segment*. Groups also provide a convenient way to specify that everything in a group goes to the same **`MEMORY`**. You can also set the **`FILL`** character for each output section in a group by specifying it for the group.

Expressions

An expression can consist of the following:

- Identifiers that denote a global symbol (which can be defined in the command file with an Assignment command)
- C-style infix and unary operators:

  ```
  - + / *
  ```

- The current relative program counter (.)
- C-style integer constants; for example:

  ```
  8
  0xFE4000
  ```

- A C-style conditional expression:

  ```
  x?y:z
  ```

- Any of the following pseudo functions:

ADDR(*mem_area*)	Address of a memory area
ADDR(*sect_name*)	Address of a previously defined section
DEFINED *symbol*	Evaluates to `1` (one) if *symbol* is defined; `0` (zero) otherwise; typically used in ternary expressions (`x?y:z`)
HEADERSZ	The file offset following the ELF program header table (this is typically where the data of the first output section is placed)
NEXT(*value*)	The first multiple of *value* that falls in unallocated memory (*value* is typically a page size)
SIZEOF(*mem_area*)	Size of a memory area defined with a **MEMORY** command
SIZEOF(*sect_name*)	Size of an output section

Note: A "`.`" symbol can appear only in an *output_sspec* or a statement.

An expression must evaluate to a value that is appropriate for its context. For example, a context that requires a size (for example, **SIZEOF**(`x`)) must have an expression that evaluates to an absolute value (as opposed to a section-relative value).

Two section-relative expressions can be subtracted to produce an absolute value, provided that the two expressions refer to the same section; for example:

```
A = ADDR(.text) + 100;
B = ADDR(.text) + SIZEOF(.text);
C = A - B
```

In section- or group-address specification, an expression cannot forward-reference the address of another section or group.

The **DEFINED** operator can be used to conditionally reference symbols whose existence isn’t known until link time; for example:

```
SSIZE = DEFINED STACK_SIZE ? STACK_SIZE : 0x10000;
```

Group and Output Section Qualifiers

A section or group can contain an address specification, an alignment specification, a load specification, a memory-block specification, and/or a fill specification. All of these specifications are optional.

- An *address specification* consists of the keyword **ADDR**, followed by an expression that evaluates to the address of the group or output section.
- An *alignment specification* consists of the keyword **ALIGN**, followed by an expression that evaluates to a power of 2.
- A *load specification* consists of the keyword **LOAD**, followed by an expression that, when evaluated, sets the physical-address field in the corresponding program header in which the section or group is mapped.

  Use the load specification when the physical address during loading is not the same as the logical address during execution; for example, if initialized data is to be loaded into ROM (physical address), but moved to RAM (logical address) during startup.
- A *memory-block specification* consists of a right angle bracket (>) followed by the name of the memory block (*memory_block*). A memory-block specification defines a memory block in which the section or group is to be stored. You must define *memory_block* in a **MEMORY** command before you can use it in a memory-block specification.
- A section can contain a *fill specification* for a two-byte fill pattern (*2_byte_pattern*) that the linker uses to fill holes. (A *hole* is a free block of memory of a designated size; it provides space used by the application when the program executes.) The fill specification consists of an assignment operator (=) followed by *2_byte_pattern*, which is a four-digit hexadecimal number (for example 0xc3c3).

Statements

You use a *statement* inside an *output_sspec* to specify the following:

- An input section specification (*input_sspec*)
- A symbol assignment
- A **STORE** specification
- A **TABLE** specification

Input Section Specification

An *input_sspec* specifies the input sections that are to be mapped into the enclosing output section.

An input section is denoted by the input file where it resides, and the section name. You can specify a section *type* rather than a section name to denote a class of sections.

Components of an input_sspec

The components of an *input_sspec* are as follows:

input_file	A file name or regular expression representing one or more input object files or archive libraries
(*member*)	An optional archive member of an input archive library (can also be a regular expression)
sect_spec	A section name or type

sect_spec can be either of the following:

sect_name	The section name
TYPE *sect_name*	The section type

Note: To parse correctly, any file name that contains parentheses (denoting archive members), square brackets (`[]`), or a question mark (`?`) must be enclosed in quotes.

If *`input_file`* is an archive library, and you specify a *`member`*, both *`input_file`* and *`member`* must be enclosed in quotes:

```
"input_file (member)" (sect_spec)
```

A particular input section can match the pattern of more than one *`input_sspec`* 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_sspec`* that is the most specific, where *most specific* is determined as follows:

- If a subset of the matching *`input_sspec`* 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` supersedes the wildcard character.)

- If one of the matching *`input_sspec`* 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` supersedes section type name **`TYPE`** `text`.)

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

Assignments

You can make assignments inside a statement. For information, see the listing for the Assignment command in §4.4: *SVR3-Style Command Reference*.

Reserving and Initializing Memory Space for Storage

You can also use the keywords **STORE** and **TABLE** in a statement to reserve and initialize memory space for storage:

- **STORE** stores data at the current address. The arguments to **STORE** are the value to be stored (*value_to_store*) and the byte size of the storage area (*size*). *size* is normally 4 for a 32-bit value.
- **TABLE** creates a table of identifiers of a uniform size. The arguments to **TABLE** are a wild-card string that specifies identifiers to be included in the table (*pattern*), and the byte size of each storage element (*size*). *pattern* can contain the following wild-card tokens:

?	Match any one character
*	Match any string of characters, including the empty string
[ ]	Match any of the characters listed between the square brackets

  Any pattern that contains these characters must be enclosed in double quotes.

Example 1

```
SECTIONS {
   .text : {}
   .data ALIGN(4) : {
      file1.o ( .data )
      _afile1 = .;
      . = . + 1000;
      * ( .data )
      } = 0xa0a0
   .bss : { TYPE bss }
   }
```

This example does the following things:

1. It loads the .text sections from all the input files into the .text output section.
2. It aligns the .data output section on the next four-byte boundary.

3. It loads the data section from input file `file1.o` into output section `.data`.
4. It sets the identifier `_afile1` to the value of the current program counter.
5. It creates a 1000-byte hole in output section `.data`, by advancing the program counter (`_afile1` now points to the starting address of the hole).
6. It loads the rest of the `.data` sections from the files on the command line into output section `.data`.
7. It fills the hole with the pattern `a0a0`.
8. It loads all `bss` input sections, regardless of their name, into the `.bss` output section.

Note: This example assumes that the linker has been invoked with a list of input-file names on the command line.

Example 2

```
SECTIONS {
   .text BIND((0x8000 + HEADERSZ + 15) & ~ 15) : {
      * ( .init )
      * ( .text )
      }
   GROUP BIND(NEXT(0x4000) +
      ((ADDR(.text) + SIZEOF(.text)) % 0x4000)) : {
      .data : {}
      .bss : {}
      }
   }
```

This example does the following:

1. It combines all input sections named `.init` and `.text` into the output section `.text`, which is allocated at the next 16-byte boundary after address `0x8000` plus the size of the combined headers (**`HEADERSZ`**).
2. It groups together the `.data` and `.bss` output sections and puts them at the next multiple of `0x4000` plus the remainder of the end address of the `.text` output section (**`ADDR`**`(.text) +` **`SIZEOF`**`(.text)`) divided by `0x4000`.

If the size of output section `.text` is `0x37490` and the value of **`HEADERSZ`** is `0xc4`:

```
NEXT(0x4000)                                    = 0x40000
ADDR(.text)                                     = 0x80d0
SIZEOF(.text)                                   = 0x37490
(ADDR(.text) + SIZEOF(.text)) % 0x4000 = 0x3560
ADDR(.data)                                     = 0x43560
```

Note: This example assumes that the linker has been invoked with a list of input-file names on the command line.

Example 3

```
SECTIONS {
   GROUP BIND(0x800000): {
      .text : {}
      .data : {}
      .tags : {}
      .test_section : {
         UNIQUE_SECTION = .;
         . = . + 1000;
         }
      .bss  :{}
      .sbss :{}
      }
   }
```

This example does the following:

1. It groups together the `.text`, `.data`, and `.tags` output sections and puts them at address `0x800000`.
2. It creates an empty output section called `.test_section`, of 1000 bytes, and defines the symbol `UNIQUE_SECTION` to reference `.test_section`'s starting address.
3. It appends the `.bss` and `.sbss` output sections to the group immediately after `.test_section`.

Note: This example assumes that the linker has been invoked with a list of input-file names on the command line.

Example 4

```
SECTIONS {
#
# The group starts at address 0x400.
#
   GROUP ADDRESS 0x400 : {
```

```
#
# The first output section is "outdata".
# All .data sections are combined in "outdata".
#
      outdata : {
         * {.data}
         }
#
# The second output section starts at the first
# 0x80-byte boundary after the "outdata" section,
# and contains all bss sections.
#
      .bss ALIGN(0x80) : {
         * {TYPE bss }
         }
#
# The third output section, named "usertext",
# contains section "mycode" of module mod1.o.
#
      usertext : {
         mod1.o {mycode}
         }
      }
#
# The fourth output section starts at address
# 0x4000 and contains all .text sections,
# followed by section "mycode" of module mod2.o.
#
   .text ADDRESS 0x4000 : {
      * {.text}
      mod2.o {mycode}
   }
```

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

1. `outdata` (located at `0x400`)
2. `.bss` (located at the next boundary of `0x80` after outdata)
3. `usertext` (immediately following .bss)
4. `.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 5

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

This **`SECTIONS`** command arranges the output sections in order as follows:

1. output section `.text` (located at `0xc0004000`)
2. any other output sections of type `text`
3. any output sections of type `lit`
4. any output sections of type `data`
5. 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 6

```
MEMORY {
   memory1: ORIGIN = 0x00060000 LENGTH = 0x200000
   }
SECTIONS {
   GROUP : {
      .text    : {}
      .init    : {}
      .fini    : {}
      .rodata  : {}
      .sdata2  : {}
      .rosdata : {}
      .data    : {}
      .sdata   : {}
      .sbss    : {}
      .bss     : {}
      } > memory1
   }
End_of_Image = ADDR(.bss)+SIZEOF(.bss);
```

This command file does the following:

1. It specifies a memory range named `memory1`, `0x200000` bytes in size, beginning at memory address `0x00060000`.
2. It maps all input sections named `.text`, `.init`, `.fini`, `.rodata`, `.sdata2`, `.rosdata`, `.data`, `.sdata`, `.sbss`, and `.bss` from the input files to output sections of the same name.
3. It groups these output sections one after the other in memory range `memory1`.
4. It sets a pointer `End_of_Image` to point to the free memory beyond the `.bss` output section.

Example 7

```
-u _ForceUndef
MEMORY {
   memory1: ORIGIN = 0x00000400 LENGTH = 0x400
   memory2: ORIGIN = 0x00000800 LENGTH = 0x4000
   memory3: ORIGIN = 0x00006000 LENGTH = 0x100000
   memory4: ORIGIN = 0xffff0100 LENGTH = 0xf00
   memory5: ORIGIN = 0xffff1000 LENGTH = 0x7000
   }
SECTIONS {
   .special1: {
      *(.sp1)
      } > memory1
   .special2: {
      *(.sp2)
      } > memory2
GROUP : {
   .init    : {}
   .fini    : {}
   .sdata   : {}
   .rosdata : {}
   .sbss    : {}
   .sdata2  : {}
   } > memory3
GROUP BIND (ADDR(.sdata2) + SIZEOF(.sdata2)) : {
   .data    : {}
   .rodata  : {}
   .bss     : {}
   } > memory3
.vectors: {
   *(.vect)
   } > memory4
.text: {
   *(.text)
   } > memory5
}
```

This example does the following:

1. With linker option **-u**, it creates an undefined symbol `_ForceUndef`.
2. It specifies five memory ranges:

Memory Range	Size In Bytes	Beginning Memory Address
memory1	0x400	0x00000400
memory2	0x4000	0x00000800
memory3	0x100000	0x000060
memory4	0xf00	0xffff0100
memory5	0x7000	0xffff1000

3. It maps all `.sp1` input sections from the input files to output section `.special1`, and stores `.special1` in `memory1`.
4. It maps all `.sp2` input sections from the input files to output section `.special2`, and stores `.special2` in `memory2`.
5. It maps all input sections named `.init`, `.fini`, `.sdata`, `.rosdata`, `.sbss`, and `.sdata2` from the input files to an output section of the

same name, and groups these output sections in the order specified in `memory3`.

6. It maps all input sections named `.data`, `.rodata`, and `.bss` from the input files to an output section of the same name, and groups these output sections in the order specified in `memory4`, starting at the first address of free memory after output section `.sdata2`.
7. It maps all input sections `.vect` from the input files to output section `.vectors`, and stores `.vectors` in `memory4`.
8. It maps all input sections `.text` from the input files to output section `.text`, and stores `.text` in `memory5`.

START

Specify program entry point

Syntax **`START`** **[***`symbol`***|***`value`***]**
`ENTRY` `(`*`symbol`*`)`

`symbol` is the name of an exported symbol. *`value`* is an address; it 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
```

SVR4-Style Command Files 5

This chapter describes SVR4-style command files (also called AT&T-style command files or input map files) and how to use the commands (also called directives) with the ELF linker. It contains the following sections:

Overview You use SVR4-style command files to do the following:

- Declare segments; specify values for attributes such as type, permissions, addresses, length, and alignment.
- Control mapping of input sections to segments.
- Declare a global-absolute symbol that can be referenced from object files.

For a complete discussion of SVR4-style command files, see Appendix B: *Mapfile Option* in the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools**.

5.1 Specifying an SVR4-Style Command File

You specify an SVR4-style command file with linker option **-M**, followed by the name of the file; for example:

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

You can specify multiple command files; the linker processes them as if they were concatenated.

5.2 Sample Command File

Example

```
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 segments (`text`, `data`, and `my_note`) and sets their permissions and attributes.

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

5.3 SVR4-Style Command Reference

You can include the following types of commands in an SVR4-style command file:

Type of Directive	**Purpose**
Segment Declaration	Create or modify segment in executables
Mapping Directive	Specify how to map input sections to segments
Size-Symbol Declaration	Define new symbol representing size of a segment

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

Note: Former versions of the linker supported extensions to the SVR4 command syntax that allow users to define output sections by name. This support is now available with either

AMD-style or Diab-style command syntax, and is no longer supported by SVR4-style commands.

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.

Segment Declaration

Create or modify segment in executables

Syntax `s_name`**[**`= attribute` **[...]]**`;`

`s_name` is an identifier, the name of a segment.

The optional *`attribute`* arguments can be one or more of the following:

Argument	Description	Valid Values
seg_type	Loaded by the loader at load time Read by the loader at load time	**LOAD** **NOTE**
permiss_flags	Any permutation of read, write, and execute	**?[R|W|X]**
virtual_addr	Hexadecimal address	**V***address*
physical_addr	Hexadecimal address	**P***address*
length	Integer up to 2^{32}	**L***address*
alignment	An integer power of 2	**A***integer*

Note: The *`attribute`* 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 A segment declaration creates a new segment in the executable file or changes the attribute values of an existing segment.

A *segment* is a group of consecutive output sections with like characteristics that are loaded as a unit via the ELF program header mechanism. Each program header entry of type `PT_LOAD` references a segment.

A segment name (*`s_name`*) does not appear in the output file; it is merely a logical designation.

Example `seg_1 = LOAD V0x30005000 L0x1000;`

In this example, segment `seg_1` is declared as follows:

- to be loaded at load time (*`seg_type`* = **`LOAD`**)
- starting at virtual address `30005000` (*`virtual_addr`* = **`V`**`0x30005000`)
- for a length of hexadecimal 1000 bytes (*`length`* = **`L`**`0x1000`)

Because `seg_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 segments

Syntax `{`*`s_name`*`} : [`*`attribute`*` ...][ : `*`file_name`*`...];`

- *`s_name`* is an identifier, the name of a segment.
- *`file_name`* is the name of an object file, which can be a regular expression (see §3.3: *Using Wildcards in File-Name References*).
- The *`attribute`* arguments can be one or more of the following:

Argument	Description	Valid Values
`s_type`	No bits section	**`$NOBITS`**
	Note section	**`$NOTE`**
	Ordinary data or text sections	**`$PROGBITS`**
`s_flags`	Any permutation of allocatable, writable, and executable	**`?[!][A|W|X]`**

Note: The *`attribute`* 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 segments.

The linker places input sections in output sections of the same name. Output sections are in turn mapped to the specified segment (*s_name*).

Example

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

In this example, output segment `segment_1` is mapped with the following:

- all input sections from the object files `file_1.o` and `file_2.o` that contain data (**`$PROGBITS`**) and are allocatable + executable (**`?AX`**)
- all `.rodata` input sections from `file_1.o` and `file_2.o`.

Size-Symbol Declaration

Define new symbol representing size of a segment

Syntax **{***sect_name* **|** *seg_name***}** @ *symbol_size_name*

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

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

Example

```
seg_1 @ Protected_Data_Section_Size
```

This example assigns the size name `Protected_Data_Section_Size` to segment `seg_1`.

Additional Linker Topics 6

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

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.

6.1.1 Naming Conventions

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

`_f`*`section_name`*	Set to the address of the start of *`section_name`*
`_e`*`section_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`.

6.1.2 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);
   }
```

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 object file as input to additional linker invocations. You specify incremental linking with linker option **`-r`**.

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
```

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.

6.2.1 Resolving Common Blocks in Incremental Linking

When you link incrementally, the linker by default 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 assigns appropriate addresses for common blocks. In incremental linking, common blocks are left to be resolved in the final link.

To direct the linker to resolve common blocks in incremental linking, specify linker option **`-Ballocatecommon`**.

6.3 Generating Relocation Fix-Ups Versus Local Copies

When linking an executable that references a variable within a 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 **`-Bnocopies`**. See §2.1: *Linker Options Reference* for more information.

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 DLL.

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 (with 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 **`-Bnocopies`** has no negative effect on resource utilization.

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 DLL to reference the local copy. Rewiring the references in the DLL this way is seldom a problem, because DLLs 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 UNIX-based systems, 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 DLL, the sharing becomes more complex. The operating system's dynamic loader must resolve every reference to a DLL symbol. Because a DLL can be mapped at arbitrary virtual addresses at load time, each process running the same program might map the associated DLLs 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 DLL 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 DLLs 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 **`-Bnoplt`**, the linker will not implicitly create PLT entries. See §2.1: *Linker Options Reference* 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`**.

For details about the syntax and usage of **`INITDATA`**, see the entry for this command in §4.4: *SVR3-Style Command Reference*.

6.5.1 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.

6.5.2 Calling _initcopy()

Typically, you call `_initcopy()` at program startup, either from the assembly language program entry point or shortly thereafter from C:

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

The return value is `0` (zero) if there were no errors, and non-zero if the format of the `.initdata` section appears to be incorrect. You can choose to ignore the return value if suitable diagnostics cannot be performed at such an early stage of program initialization.

6.6 How the Linker Processes Archive Libraries

The linker supports two methods of scanning archive libraries. Which method the linker uses depends on whether you specify linker option **`-Bgrouplib`**.

6.6.1 Default Convention for Processing Archive Libraries

If you do not specify option **`-Bgrouplib`**, 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 (for example, to resolve mutual references), either specify the library multiple times on the command line, or use linker option **`-Bgrouplib`**.

6.6.2 Grouped Convention for Processing Archive Libraries

To direct the linker to scan archives as a group, so that mutual dependencies between archives are resolved without requiring that an archive be specified multiple times, use linker option **`-Bgrouplib`**.

When you specify option **`-Bgrouplib`**, when the linker encounters an archive on the command line, it "merges" the contents of the archive with any preceding archives, then scans the result for unresolved references. This process is repeated for each subsequent archive the linker encounters.

Note: If a symbol is exported by more than one archive, the earliest one will always be extracted.

6.6.3 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, and you do not specify linker option **`-Bgrouplib`**.

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.

6.6.4 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, symbol `_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 linker 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 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 7

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 error 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 error.

Utilities 8

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

This section describes the MetaWare archiver, **`ar`**.

Note: The archiver is named **`ar`***`suffix`*, where *`suffix`* represents the target processor. See the **Installation Guide** for the exact name of the archiver for your target. In this manual, **`ar`** generically represents the 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*.

Note: In the following examples, 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 on the command line.

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:

archive	The name of an archive library.
input_archive	The name of an archive library.
file	A relocatable object file.
member	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.

Option	Meaning
-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.

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 ELF-to-Hex 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

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.

For a discussion of the device model assumed by the converter, see §A.2: *PROM Device Model*.

For a discussion of the supported formats, see §A.4: *Hex File Formats*.

8.2.1 Invoking the ELF-to-Hex Converter

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

```
elf2hex [options] input_file
```

These are the command-line arguments:

`option` Conversion options, separated by spaces. See §8.2.3: *elf2hex Option Reference* for a complete list of options.

`input_file` 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.

The dash before the option is not required.

8.2.2 elf2hex Invocation Examples

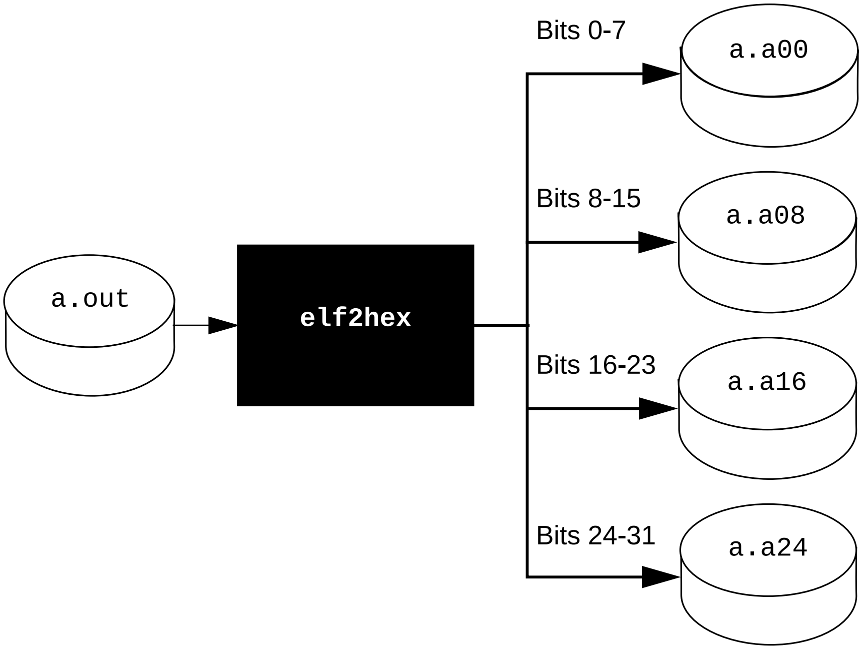
This section presents two **`elf2hex`** invocation examples.

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, **`elf2hex`** generates additional sets of files; their file-name extensions begin with the letter `c`, then `d`, and so on. 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:

Figure 8.2 Sample Output of Extended Tektronix Hex Format

```
%4A6CE44000250101105E40017E158101180340837002008300030082400301792172450101
%4A6C644020034083900200830003008241030179217245010103017904724501011583600
%4A6414404001FF82FF160061601560600461616100A4FF61FD15828201A800801970400101
%4A65F440600300790172450101A0FF00FE7040010124797E01247F7F798379790225797901
%4A68344080CE0087793E00807FC000007A157E0100030279009279797FCE0081792479817F
%4A691440A0147E7E798379790225797901CE0087793600007FC000007A157F8100257D7D68
%4A63E440C003007600817976021477797D03007800030079001479797 71E00787915767601
%32605440E04179760AACFF79F870400101C0000080157D7D68
%0A81A44000
```

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 about hex file formats, see §A.4: *Hex File Formats*.

8.2.3 elf2hex Option Reference

Note: Attributes applied to linker option **`-x`** have the same name, functionality, and default as the following **`elf2hex`** options.

Invoked with option **`-x`**, the linker generates a new ELF executable and the corresponding hex files as a single-step process; it is the equivalent of generating the ELF executable, then invoking **`elf2hex`** to convert it to hex files.

`-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:

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

Default = **`-c`** `dlt`.

If you do not specify option **`-c`**, **`elf2hex`** converts `data`, `lit`, and `text` 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.

`-p` *`address`* — **Assign load addresses starting from *`address`***

Specifies a load address that is different from the bind address of the sections in the executable. The first loadable section gets mapped to the load address specified by *`address`*. The load addresses of all other sections need to be mapped appropriately.

For example, this command causes the first section (normally `.text`) to be loaded at address `0x02800000`:

```
elf2hex -p 0x02800000 filename
```

All other sections will be mapped at addresses decided by the offset of the first section's bind address from `0x02800000`.

`-p` *`sect_1`*`:`*`addr_1`*[`,`*`sect_2`*[`:`*`addr_2`*]]`. . .` — **Assign specified section(s) to specified load address(es)**

Specifies a load address that is different from the bind address of the sections in the executable. This usage of option **`-p`** enumerates the load addresses for many sections; *`sect_1`* is loaded at *`addr_1`*, *`sect_2`* is loaded at *`addr_2`*, and so on.

For example, this command causes the `.data` section to be specifically loaded at address `0x02800000`:

```
elf2hex -p .data:0x02800000 flash.exe
```

For example, this command causes the `.text` section to be specifically loaded at address `0x02800000`, followed immediately by the `.data` section:

```
elf2hex -p .text:0x02800000,.data flash.exe
```

`-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 *`E`*`Kx`*`W`* or *`E`*`Mx`*`W`*, 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 (*`E`*`K`) or
`1` megabyte (*`E`*`M`)

`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.

8.3 Using the File-Size Utility

This section describes the **`size`** file-size utility, which displays the size of one or more object files or archive libraries.

Note: The file-size utility is named **`size`***`suffix`*, where *`suffix`* represents the target processor. See the **Installation Guide** for the exact name of the file-size utility for your target. In this manual, **`size`** generically represents the file-size utility.

`size` prints out the following for each of its object-file arguments:

- The number of bytes each in the text section, data section, and bss section
- The total number of bytes required by the text, data, and bss sections combined

If you specify an archive library, **`size`** outputs the preceding information for every object file in the archive. It then totals the bytes in all text sections, data sections, and bss sections, and the bytes in all the object files.

Unless you specify otherwise (see §8.3.2: *Command-Line Options for size*), sizes are displayed in decimal.

8.3.1 Invoking size

You invoke **`size`** as follows:

```
size [options][filename[, filename ...]]
```

`filename` is the name of an object file or archive library.

If you do not specify *`filename`*, **`size`** displays a listing of the **`size`** command-line options.

8.3.2 Command-Line Options for size

Table 8.2 lists the command-line options available for **`size`**.

Table 8.2 size Command-Line Options

Option	Meaning
`-d`	Display sizes in decimal (the default).
`-o`	Display sizes in octal.
`-q`	Do not display the copyright message.
`-x`	Display sizes in hexadecimal.

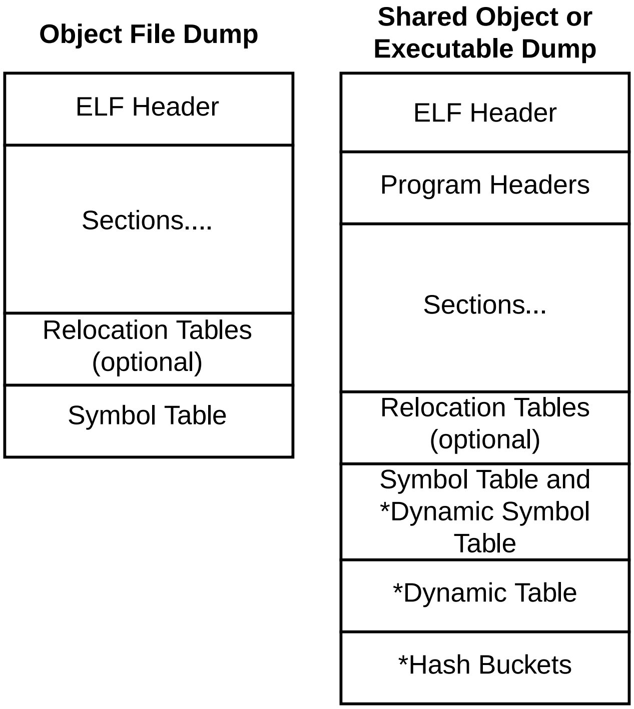
8.4 Using the Binary Dump Utility

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

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

Figure 8.3 Structure Dumps of ELF Object and Executable Files


Object File Dump
ELF Header
Sections....
Relocation Tables (optional)
Symbol Table
Shared Object or Executable Dump
ELF Header
Program Headers
Sections...
Relocation Tables (optional)
Symbol Table and *Dynamic Symbol Table
*Dynamic Table
*Hash Buckets

*Present only in a dynamically linked executable

Note: You cannot apply **`elfdump`** directly to archive libraries. You must first extract the objects with **`ar`**, then apply **`elfdump`** to them.

8.4.1 Invoking elfdump

You invoke **`elfdump`** as follows:

```
elfdump [options] filename[filename ...]
```

These are the command-line arguments:

options	Options that specify the category of information to be dumped.
filename	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. See Table 8.3, *elfdump Command-Line Options* for a complete list of options.

8.4.2 Command-Line Options for elfdump

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
-D	Dynamic table
-h	ELF header
-H	Hash table contents
-p	Program header
-r	Relocation header sections
-s	Section headers
-t	Symbol table(s)
-X	Section contents of debug/line sections
-z	Section contents; bytes of any section are disassembled for which a disassembler exists

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.

8.5 Using the Symbol-List Utility

The object-file symbol list utility, **`nm`**, displays the symbol-table contents of object files and archives.

Note: The symbol-list utility is named **`nm`***`suffix`*, where *`suffix`* represents the target processor. See the **Installation Guide** for the exact name of the symbol-list utility for your target. In this manual, **`nm`** generically represents the symbol-list utility command.

8.5.1 Invoking nm

You invoke **`nm`** with the following command:

```
nm [options] object_file[, object_file ...]
```

These are the **`nm`** command-line arguments:

`options`	One or more **`nm`** command-line options
`object_file`	Object file or archive for which you want to display the symbol table contents

8.5.2 Command-Line Options for nm

Table lists the command-line options for **`nm`**:

Table 8.4 nm Command-Line Options

Option	Meaning
`-A`	Append the full path name of the object file or archive to each displayed symbol.
`-C`	Unmangle C++ names.

Option	Meaning
-d	Display symbol values in decimal.
-g	Display only global symbols.
-h	Do not display header line before displaying symbols.
-n	Sort symbols by name.
-o	Display symbol values in octal.
-p	Produce easily parsable output where each symbol name is preceded by its value and a class character. Class characters are displayed in uppercase for global symbols and lowercase for static symbols. Class characters have the following meanings: A: Absolute symbol B: bss (uninitialized data space) symbol D: Data-object symbol F: File symbol S: Section T: Text symbol U: Undefined
-Q	Do not display copyright message.
-u	Display undefined symbols only.
-v	Sort symbols by value.
-x	Display symbol values in hexadecimal.

8.6 Using the Strip Utility

Note: The **strip** utility documented in this section applies to all ELF-based targets.

The MetaWare **strip** utility is similar to the UNIX **strip** command. The **strip** utility deletes symbol tables, line-number information, and debug information from ELF executables. The resulting executables are more compact.

strip can also be applied to object files, in which case the line-number and debug information is removed and the symbol table is stripped of all symbols except those required by the linker for fixups.

Note: Programs that have been stripped cannot be debugged with a symbolic debugger.

8.6.1 Invoking strip

You invoke the **strip** utility as follows:

```
strip [options] object_file[, object_file ...]
```

These are the command-line arguments:

options Options that specify the type of file to be stripped. See Table 8.5, *strip Command-Line Options* for a complete list of options.

object_file ELF object file, shared object file, or executable file. This is the only required argument.

Defaults By default, **strip** removes section headers and the `.symtab`, `.line`, and `.debug` sections.

8.6.2 Command-Line Options for strip

Table 8.5 lists the command-line options for **strip**.

Table 8.5 strip Command-Line Options

Option	Meaning
-h	Displays help text
-l	Removes the `.line` section only
-r	Does not remove section headers
-V	Produces verbose output
-x	Removes `.debug` and `.line` sections, but keeps `.symtab`

Working with PROMs and Hex Files A

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 §2.1: *Linker Options Reference*. For information about the ELF-to-hex conversion utility, **`elf2hex`**, see §8.2: *Using the ELF-to-Hex 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 the key characteristics of PROM devices:

- width
- size or length
- word width and banks
- multiple banks

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.

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).

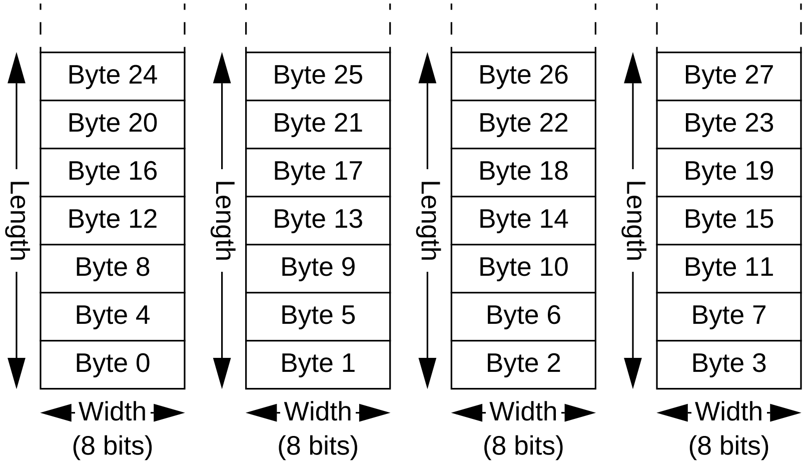
PROM devices can be combined into words. The word width can be 8, 16, or 32 bits. This value represents the number of bits accessible in a single read or write to the group (or bank) of devices. The default word size is 32 bits.

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

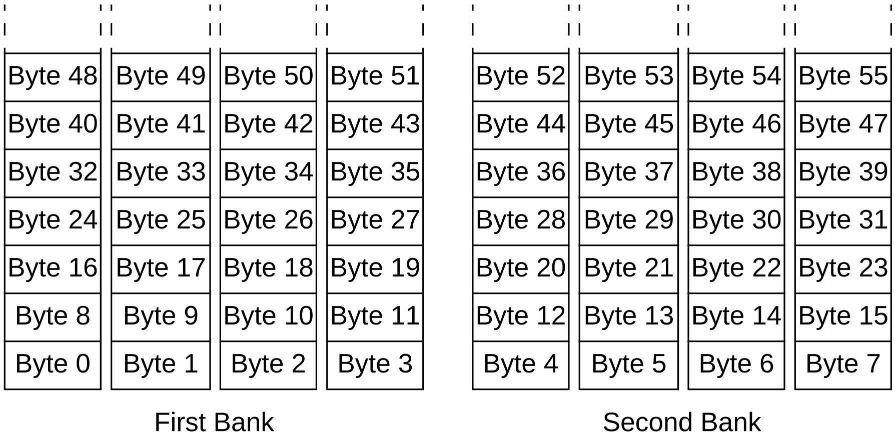


In the case of a single PROM bank (of 8-bit devices and 32-bit words), four hex files are generated — one file per device. The files are named according to the conventions described in §A.3: *Hex Output-File Naming Conventions*.

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 (0,1,2,3 and 4,5,6,7)



Byte 48	Byte 49	Byte 50	Byte 51	Byte 52	Byte 53	Byte 54	Byte 55
Byte 40	Byte 41	Byte 42	Byte 43	Byte 44	Byte 45	Byte 46	Byte 47
Byte 32	Byte 33	Byte 34	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
Byte 24	Byte 25	Byte 26	Byte 27	Byte 28	Byte 29	Byte 30	Byte 31
Byte 16	Byte 17	Byte 18	Byte 19	Byte 20	Byte 21	Byte 22	Byte 23
Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7

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. (For information about **`elf2hex`**, see §8.2: *Using the ELF-to-Hex Conversion Utility*.) 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 the following command, the linker produces the hex output file `file.hex`:

```
ld  -x file
```

Note: If you use linker option **`-x`** with a ttribute `o` 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:

Suffix Element	Description
`row`	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 **`-t`**`s` or **`coff2hex`** option **`-s`**).
`bit`	A two-digit decimal number that denotes a bit position, starting from the right-most byte of a word:
	`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 **`-t`**`n` 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:

Suffix Element	Description
`row`	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 specified device length (linker option **`-t`**`s` or **`elf2hex`** option **`-s`**).
`bank`	A single letter that denotes the bank. `a` denotes the first bank, `b` denotes the second bank, and so on.
`nybble`	The bit position divided by 4 (for example, 24 becomes 6, 16 becomes 4, and so on). This number is based on the specified device width (linker option **`-t`**`n` or **`elf2hex`** option **`-n`**). 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-name conventions.

A.4 Hex File Formats

This section summarizes the supported hex formats:

- Motorola Hex Record Format
- Extended Tektronix Hex Record Format
- Mentor QUICKSIM Modelfile Format

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*.

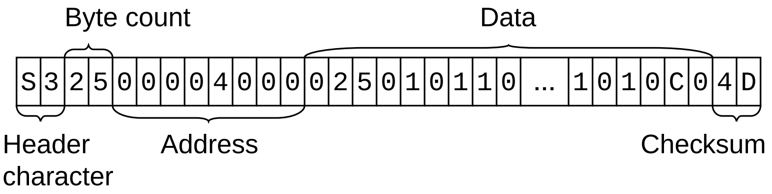
A.4.1.1 Data Records (Start Characters S1 — S3)

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


Byte count
Data
S 3 2 5 0 0 0 0 4 0 0 0 0 2 5 0 1 0 1 1 0 ... 1 0 1 0 C 0 4 D
Header character
Address
Checksum

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)

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.

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

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

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

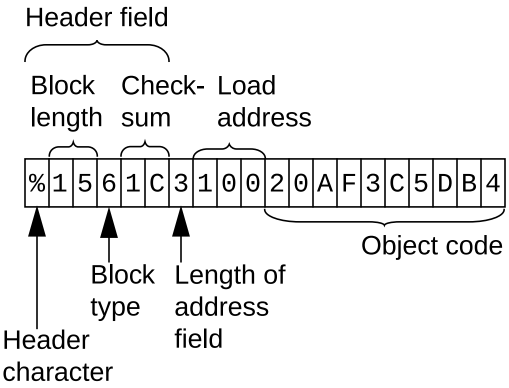
A.4.2.1 Data Record

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



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	The `%` character; indicates records in Extended Tektronix Hex format
Block length	2	Number of characters in record, minus the `%` character
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

The address 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.

The load address specifies where the object code will be located. This is a variable-length number that can contain up to 16 characters.

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 **`-x`** with attribute `s` or **`elf2hex`** option **`-s`**) is ignored for QUICKSIM records.

Index