ELF Linker and Utilities

User's Guide
for ARC®

0139-033

ELF Linker and Utilities User’s Guide for ARC®

ARC International

North America	Europe
3590 N. First Street, Suite 200	Verulam Point
San Jose, CA 95134 USA	Station Way
Tel. +1-408-437-3400	St Albans, AL1 5HE
Fax +1-408-437-3401	UK
	Tel.+44 (0) 20-8236-2800
	Fax +44 (0) 20-8236-2801

www.ARC.com

0139-033 August 2007

Contents

List of Examples

Chapter 1 — Before You Begin

In This Chapter

Technical Support

We encourage customers to visit our Technical Support site for technical information before submitting support requests via telephone. The Technical Support site is richly populated with the following:

ARCSolve — Frequently asked questions

ARCSpeak — Glossary of terms

White papers — Technical documents

Datasheets — ARC product flyers

Known Issues — Known bugs and workarounds.

Training — Future ARC Training sessions

Support — Entry point to sending in a query

Downloads — Access to useful scripts and tools

You can access this site via our corporate website at `http://www.ARC.com`, or you can access the site directly: `http://support.ARC.com`.

Note that you must register before you can submit a support request via our Extranet site.

About This Book

This *ELF Linker and Utilities User's Guide* describes how to use the 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.

This book contains the following topics:

- Using the ELF Linker
- Linker Options
- Linker Command Files
- SVR3 Command Files
- SVR4 Command Files
- Topics and Techniques
- Linker Error Messages
- Using the Utilities
- PROMs and Hex Files

Where to Go for More Information

The main readme file or release note describes any special files and provides information that was not available at the time the manuals were published.

The "Where to go for more information" section in the *MetaWare C/C++ Programmer's Guide* describes the following:

- Documents in the MetaWare Development Toolkit
- C and C++ Programming Documents
- Processor-Specific Documents
- Specifications and ABI Documents

For more information on SVR3-style command file formats, see the following

- *29K™ Family Assembler, Linker, and Librarian Reference Manual*
  Advanced Micro Devices, 1995
- *UNIX System V Release 3.2 Programmer's Guide*
  Intel Corporation, 1990

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

Document Conventions

Notes

Notes point out important information.

NOTE Names of command-line options are case-sensitive.

Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **`#if`** or **`#end`**, C comment delimiters, or invalid C tokens.

Terminological Conventions

This document observes the following terminological conventions.

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

Section and segment
A *segment* is a group of sections loaded as a unit.

Chapter 2 — Using the ELF Linker

In This Chapter

Invoking the Linker

If your toolkit includes an IDE, you can invoke the linker directly from the IDE; see the IDE online help for more information.

You can also invoke the linker from the command line in two ways:

- with the other compilation tools using the MetaWare C/C++ driver command
- by itself using the linker command

Invoking the linker using the driver command is the recommended method.

Invoking the Linker Using the Driver Command

The driver program is named **mcc**.

The driver automatically invokes the linker on files it recognizes as linker input files. The command line shown in Example 1 Invoking the Linker Using the Driver Command invokes the linker to link together object files `file1.o` and `file2.o` and produce an executable output file:

Example 1 Invoking the Linker Using the Driver Command

`mcc` *Processor_options* `file1.o file2.o`

Processor_options are the driver options to select the correct libraries for your processor configuration, as documented in the *MetaWare C/C++ Programmer's Guide for ARC*.

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, a secondary executable, or a command file or input map file). The driver passes such files directly to the linker.

If you include one or more executable files, the public symbols in them are treated as weak absolute symbols, but prevent corresponding symbols from being pulled out of a library. You may wish to do this if you would like the executable to provide its public functions for use by the secondary executable being linked rather than the secondary executable having its own copies of those functions, for example. See also Providing a Symbol Table for a Secondary Executable on page 71.

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

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**. The command line shown in Example 2 Passing an Option That Conflicts with a Compiler Option passes option **-V** (also a compiler option) to the linker:

Example 2 Passing an Option That Conflicts with a Compiler Option

`mcc` *Processor_options* `file1.c file2.c -Hldopt=-V`

See the *MetaWare C/C++ Programmer's Guide* for more information about using option **-Hldopt**.

Processor_options are the driver options to select the correct libraries for your processor configuration, as documented in the *MetaWare C/C++ Programmer's Guide for ARC*.

To determine whether a linker option conflicts with a compiler option, see the listing for the specific linker option in Linker-Options Reference on page 18.

Invoking the Linker Using the Linker Command

The linker program is named **ld***target*, where the suffix *target* identifies the target platform for which you are linking.

In this guide, we use **ld** to generically represent the linker command. See Table 1 Linker Program Names for the exact linker program name for your target.

Table 1 Linker Program Names

Processor Family	Target Suffix	Linker Program Name and Command
ARCtangent-A4	arc	**ldarc**
ARCtangent-A5 and later	ac	**ldac**

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 Linker Options on page 17 for information about linker options.)
- *input_file* is one or more object files (relocatable input files), archive files, executable files, or a 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 How the Linker Processes Archive Libraries on page 83 and option **-Bgrouplib** for more information on this topic.)

  If you include one or more executable files, the public symbols in them are treated as weak absolute symbols, but prevent corresponding symbols from being pulled out of a library. You may wish to do this if you would like the executable to provide its public functions for use by the secondary executable being linked rather than the secondary executable having its own copies of those functions, for example. See also Providing a Symbol Table for a Secondary Executable on page 71.
- *arg_file* is the name of an optional argument file. (See Specifying Command-Line Arguments in a File for information about argument files.)

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 **mcc** driver command line, use driver option **-Hldopt**, and precede the argument-file name with the “at” symbol (@); for example:

mcc *Processor_options* -Hldopt=@*arg_file* file.c

See the *MetaWare C/C++ Programmer’s Guide* for information about driver option **-Hldopt**.

Processor_options are the driver options to select the correct libraries for your processor configuration, as documented in the *MetaWare C/C++ Programmer’s Guide for ARC*.

Specifying an Argument File on the Linker Command Line

To specify an argument file on the **ld** 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 work around line-length limitations.

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 Linker Command Files on page 35.

Linking Run-Time Library Files

The linker links the MetaWare 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**.

The MetaWare C Run-Time Libraries are static libraries whose names are of the form `libname.a`.

Related Topic

- Selecting the Correct Libraries

Selecting the Correct Libraries

When you invoke the linker using the **mcc** driver, you must specify the options to select the correct libraries for your target processor configuration, just as when you use the driver to compile C or C++ files:

- Processor series and core version (such as **-a6 -core1**)
- Extensions (such as **-Xlib**)
- Configuration options (such as **-HB** or **-Hccm**)

For a complete list of options, see the *MetaWare C/C++ Programmer’s Guide for ARC*.

Chapter 3 — Linker Options

In This Chapter

Linker-Options Reference

You can specify most 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**).

This section provides detailed information about each linker option. See Invoking the Linker Using the Driver Command on page 14 for information about using linker options with the **mcc** driver command. See Invoking the Linker Using the Linker Command on page 15 for information about using linker options with the **ld** linker command.

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

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.

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.

For more information, see the following:

- Generating Relocation Fix-Ups versus Local Copies on page 73
- Rewiring Shared Calls through the PLT on page 76

-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

For incremental linking only

Option **-Ballocatecommon** forces the allocation of common data when you specify option **-r** for incremental linking. See Linking Incrementally on page 72 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

Option **-Bbase** specifies the origin of the `.text` section. The linker uses *0xaddress* 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 *0xaddress*, 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

Option **-Bcopydata** directs the linker to create an **INITDATA** entry to initialize from "initdata" tables all writable data sections at startup. For more information, see the listing for **INITDATA** in SVR3-Style Command Reference on page 43.

CAUTION If you use option **-Bcopydata** with your own linker command file, be sure that your linker command file does not prevent `.initdat` from being defined as an output section. Otherwise, the linker cannot create the label `_einitdat` (end of the `.initdat` section) and **_initcopy()** cannot copy initialized data.

-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 SVR3-Style Command Reference on page 43 for information about the **Assignment** command.)

-Bdynamic — Search for DLL libname when processing option -l

For 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**.

-Belfpflag=*0xval* — Add flags for non-MetaWare debuggers

Some debuggers (but not the MetaWare debugger) refuse to operate without certain flags in the ELF program headers. When you specify option **-Belfpflag**, the linker uses a bitwise "or" to add the hexadecimal value you specify as a flag into every ELF program header.

-Bexe — Build executable during incremental link

Instructs the linker to generate an executable even if **-r** is specified. When you use this option in conjunction with **-r**, the generated executable contains static relocation tables (increasing its size).

For more information, see Linking Incrementally on page 72.

-Bforce_{`big_`|`little_`}endian — Force endianness of generated tables

In certain situations it is desirable to mix endianness, for example on targets that boot big endian but are capable of running little-endian code. Options **-Bforce_big_endian** and **-Bforce_little_endian** disable the linker's endianness consistency checking and force the linker to generate its tables (such as the procedure linkage table or PLT) in the specified endianness.

CAUTION Because these options disable consistency checking, your application might switch endianness inadvertently. Be sure to generate a link map (option **-m**) and verify that your code links against libraries of the correct endianness.

-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

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.

-Bhardwired=*sym1*,*sym2*,[...] — Bind references to specified symbols within a DLL to symbol definitions within the link

For dynamic linking only

Normally, any global symbol referenced by a DLL that is defined within the DLL can be overridden at run time. Option **-Bhardwired** "hard wires" or binds the symbols you specify (`sym1,sym2,...`) to their link-time definitions. To hard wire all symbols in a DLL, use option **-Bsymbolic**.

-Bhelp — Display information about -B options designed for embedded development

Option **-Bhelp** displays a summary screen of **-B*** options designed specifically for developing embedded applications.

CAUTION These special **-B*** options should not be used to develop non-embedded applications, such as an application that runs on UNIX or Linux. An executable produced with these options might 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

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

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 Generating Relocation Fix-Ups versus Local Copies on page 73 for a more detailed discussion of this topic.

-Bnodemandload — Ignore boundary issues when mapping sections

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

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 Rewiring Shared Calls through the PLT on page 76.

-Bnozerobss — Do not zero BSS sections at run time

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**.

-Boverlay_num=*N* —Specify overlay package number

When you use the ARC Automated Overlay Manager, you can use this option to specify the overlay package number for the generated executable to use when generating its overlay offset table. For information in using overlays, see the *Automated Overlay Manager User's Guide*.

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

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

For targets that support position-independent code, 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 startup. These tables are processed by startup 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 startup 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

For 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

For 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.

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

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

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

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.)

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

Option **-Bstart_addr** specifies the origin of the `.text` section. The linker uses *0xaddress* 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 *0xaddress*, 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 libname when processing -l name

For 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

For dynamic linking only

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

Normally, references to global symbols within DLLs can be overridden at load time by like symbols being exported in the executable or in preceding DLLs. Option **-Bsymbolic** prevents such overriding. To bind specific symbol references, see option **-Bhardwired**.

-Bsymin=*path* — Read symbol definitions file

Read a symbol file generated using **-Bsymout**. You can use linker options **-Bsymin** and **-Bsymout** to obtain a list of public symbol definitions from a “base” or primary executable and then link a secondary executable against the primary executable. The linker resolves undefined symbols from the specified file as well as from the collection of objects and libraries presented on the command line. The symbols in the specified file are treated as weak absolute symbols, but prevent corresponding symbols from being pulled out of a library.

Related Topics

- -Bsymout=path — Generate a list of public symbol definitions
- Providing a Symbol Table for a Secondary Executable on page 71

-Bsymout=*path* — Generate a list of public symbol definitions

Generate a symbol file from a primary executable. You can use linker options **-Bsymout** and **-Bsymin** to obtain a list of public symbol definitions from a “base” or primary executable and then link a secondary executable against the primary executable.

Related Topics

- -Bsymin=path — Read symbol definitions file
- Providing a Symbol Table for a Secondary Executable on page 71

-Bzerobss — Zero BSS sections at run time instead of load time

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

NOTE Startup code must call the C run-time function **_initcopy()** to actually zero the `.bss` section.

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.

CAUTION Always use option **-Hldopt** when passing option **-C** using the **mcc** driver. See Resolving Conflicts in Linker and Compiler Option Names on page 14 for more information.

listing_type can be any of the following predefined values:

Table 2 listing_type Values

Value	Definition
`crossref`	Displays a cross-reference listing of global symbols.
`globals`	Displays a sorted list of global symbols with their mappings.
`output=`*file*	Directs output to *file* (default is `stdout`). The listing includes the memory map if you specify **-C** `sections` or **-m** on the same command line.
`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.
`size`	Used with `sections`, adds a table summarizing segment sizes (compare option **-zsize**, which displays the table only and only to `stdout`)
`symbols`	Displays a listing of all symbols in the generated output symbol table.
`tables`	Summarizes linker-generated tables in the map listing.
`unmangle`	Use 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`. You can include the memory map output by using **-C** `sections` or **-m** in your output file, for example:

```
ld -C crossref -C page=50 -m -C output=file.txt file.o
```

-d {*y*|*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 (if references require DLLs).
- **-d** *Y* forces dynamic linking (even if references do not require DLLs).

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

If the linker does not need to refer to DLLs to resolve external references, it generates a statically linked executable, even if you do not specify **-d** *n*. If your application explicitly loads DLLs, specify **-d** *Y* (note capitalization) to force generation of a dynamically linked executable.

-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.

Option **-e** also implicitly specifies **-u** *entry_name*, which enters the symbol *entry_name* into the symbol table as undefined. This ensures that the linker looks for *entry_name* in an archive if required to link successfully.

-G — Generate a DLL

For dynamic linking only

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

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

For 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* can be 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 Resolving Conflicts in Linker and Compiler Option Names on page 14 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 Resolving Conflicts in Linker and Compiler Option Names on page 14 for more information.

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

For 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 **-l** is also implemented as a compiler option. See Resolving Conflicts in Linker and Compiler Option Names on page 14 for more information.

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

For 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 How the Linker Processes Archive Libraries on page 83 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 and Linux hosts, or by semicolons (;) on Windows hosts.

How the Linker Determines the Library Search Paths

The linker detects whether it is cross-linking or host-linking, and uses the appropriate strategy below:

In cross linking, the linker searches for libraries in the following directory locations, in the order given. No environment variables are required.

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

In 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 and Linux hosts, or by semicolons (;) on Windows 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 and Linux 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.

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

Table 3 Default Host-System Library Directories

Host	Directory
UNIX hosts:	`/usr/ccs/lib:/usr/lib:/usr/local/lib`
Linux hosts:	`/lib:/usr/lib:/usr/local/lib`
Windows hosts:	`\lib`

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

Option **-m** generates a memory-map listing file with a table of section sizes 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`* **-C***`size`*.

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

For a table of segment sizes only and only to `stdout`, use option **-zsize**.

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

For 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 SVR4 Command Files on page 63 for more information.

CAUTION Option **-M** is also implemented as a compiler option. See Resolving Conflicts in Linker and Compiler Option Names on page 14 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

For 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. To produce an executable that has full relocation information during the incremental link, use **-Bexe** with **-r**. Used alone, option **-r** does not increase executable size (**-Bexe** does, by adding static relocation tables).

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.

NOTE Using **-r** for relocatable code in embedded development requires an elaborate linker-loader to resolve references at load time.

For more information, see Linking Incrementally on page 72.

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

For 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 and Linux targets, or by semicolons (;) for Windows 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

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.

-w — Suppress all warning messages

Option **-w** suppresses all linker warning messages. See Linker Error Messages on page 87 for more information about linker warning and error messages.

-x[*attribute*] — Generate a hex file

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 PROMs and Hex Files on page 109 for detailed information about hex-file formats.
>
> 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 4 Hex Conversion Characteristics lists valid values for *attribute*.

Table 4 Hex Conversion Characteristics

Attribute	Conversion Characteristic	elf2hex Option
`B`	Specify raw binary output	-B
`c list`	Specify section types to be converted	**-c**
`f fill`	Use with `B` or `Mx` only to specify fill value for padding between segments	**-f**
`help`	Display help about the **-x** option and its attributes	**-h**
`I`	Specify a hex file in Intel hex format	**-I**
`i bank`	Select the number of interleaved banks	**-i**
`l limit`	Use with `B` or `Mx` only to specify padding limit	**-l**
`m`	Specify a hex file in Motorola S3 Record format	**-m**
`Ms`	Specify Synopsis MIF format	**-Ms**
`Mx`	Specify Xilinx MIF format	**-Mx**
`n`	Specify width of the memory device	**-n**
`o`	Specify the name of the generated hex file	**-o**
`P` *address*	Assign load addresses starting from *address*	**-p**
`p` *sect*`:`*addr* ...	Assign sections to load addresses	**-p**
`q`	Specify a hex file in Mentor QUICKSIM Modelfile format	**-q**
`s`	Identify size of memory device	**-s**
`t`	Specify a hex file in Extended Tektronix Hex Record format	**-t**
`V`	Specify a file in Verilog **$readmemh** format	**-V**

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 elf2hex Option Reference on page 98 for more information.

-Xcompress_stats — Display compression statistics for each section

By default, the linker compresses data in the output sections that **INITDATA** specifies for initialization at run time. Option **-Xcompress_stats** prints a report of how much each section was compressed to `stdout` (note that some sections actually expand).

For example, the command line **ld -Xcompress_stats** `file.o` returns statistics such as the following:

```
[INITDATA] section .sdata compressed -66.6%
[INITDATA] section .data compressed 72.1%
[INITDATA] section .tls compressed -100.0%
[INITDATA] section .rodata compressed 49.7%
[INITDATA] section .text compressed 39.5%
[INITDATA] section .init compressed 10.5%
[INITDATA] section .fini compressed -12.5%
[INITDATA] Total overall compression: 42.0%
```

For more information on **INITDATA**, see the listing in SVR3 Command Files on page 39.

-Xnocompress — Suppress compression of .initdat data

Option **-Xnocompress** suppresses compression of data in the sections that **INITDATA** specifies for initialization at run time. For more information on **INITDATA**, see the listing for **INITDATA** in SVR3 Command Files on page 39.

-YP,*path* — Specify 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 and Linux hosts, or by semicolons (;) on Windows 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. If you are using an older linker, specify this option to be sure to receive error messages if undefined symbols remain.

See also **-znodefs** and **-znodefswarn**.

-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.

See also **-zdefs** and **-znodefswarn**.

-znodefswarn — Allow undefined symbols but issue warning

Option **-znodefswarn** allows undefined symbols, but reports them in a warning message.

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

See also **-zdefs** and **-znodefs**.

-zoffwarn=*n1*, *n2*, [...] — Suppress numbered warnings

Option **-zoffwarn** suppresses output of the warning or warnings you specify (*n1*, *n2*, . . .). To determine the numbers of the warnings, link once and record the numbers of any warnings you wish to suppress. Warnings not prefixed with a number cannot be suppressed.

For more information about errors and warnings, see Linker Error Messages on page 87.

-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.

You can use option **-zpurge** with compiler toggle `Each_function_in_own_section` to transitively eliminate unreferenced functions. See the *MetaWare C/C++ 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 (such as `rcsid`).

You can use option **-zpurgetext** with compiler toggle `Each_function_in_own_section` to transitively eliminate unreferenced functions. See the *MetaWare C/C++ Programmer's Guide* for information about this toggle.

-zsize — Display segment sizes to stdout

Option **-zsize** displays a table of segment sizes to standard output. Compare ***-C****size*, which allows the output to be redirected and is used with ***-C****sections*.

Related Topics

- Option **-C***size*
- Option **-m**

-zstdout — Write errors to standard output instead of standard error

Normally the linker writes errors and warnings to standard error. Use option **-zstdout** to redirect errors and warnings to standard output. For more information about errors and warnings, see Linker Error Messages on page 87.

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

For 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.

Chapter 4 — Linker Command Files

In This Chapter

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.

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**

The linker supports Diab- and AMD-style command files as SVR3-style command files by default.

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:

- SVR3 Command Files on page 39
- SVR4 Command Files on page 63

Sections in Executable Files

The ELF format allows many sections, which are classified according to type. Exact section names vary by target processor; while some information on section names generated is available in the *MetaWare C/C++ Programmer's Guide*, it is best to consult your processor's application binary interface specification for the exact section names valid on your processor.

- A section of type text is read-only and contains executable code. Typically this section is called `.text`. Sections `.init` and `.fini` are also of type text.
- A section of type lit is read-only but contains data; for example, string constants and **const** variables. Typical names for lit sections are `.lit`, `.initdat`, `.rodata`, `.rdata`, or `.rosdata` ('s' indicates "small," as in "read-only small data").
- A section of type data contains writable data. Typical names for data sections are `.data`, `.tls` (thread-local storage) and `.sdata` ('s' indicates "small").

- The BSS section is a writable data section that is initialized to zeros when the program is loaded. The BSS section does not occupy space in the object file. Typical names for BSS sections are `.bss` and `.sbss` (the initial ‘s’ indicates “small”).
- A section of type info is not memory-mapped. Its contents reside in the generated executable, but it is not loaded during execution.

For a tool for dumping information from ELF object and executable files, see Using the Binary Dump Utility on page 102.

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/Linux regular-expression characters, and are subject to the following requirements:

- Any use of wildcards must be quoted: `"file*.o"`
- Any reference to a library member must be quoted: `"libc.a(file.o)"`
- Any reference to a file in a location other than the directory specified in the linker command must include the full path.

Wildcards

*** — 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 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*.

\ — Escape other wildcard characters

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

Chapter 5 — SVR3 Command Files

In This Chapter

SVR3 Command-File Format

The linker supports an extended SVR3 command syntax that includes Diab- and AMD-style command syntax. You can 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.

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

Specifying an SVR3-Style Command File

By default, the linker assumes that any file on the command line that it does not recognize as either a relocatable object 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.

Sample Command File

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

NOTE Any reference within a command file to a file in a location other than the current working directory or the directory specified in the linker command must include the full path.

Example 3 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):
/*
Any alignments greater than 4 in the .text section override
the ALIGN(4) declared here.
BLOCK ensures the size of the section is a multiple
of 4, but does not set alignment by iteslf
*/
      {
         // 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);
```

NOTE The linker aligns output sections to the largest alignment value in any source module. If the linker command file contains an **ALIGN** specification smaller than the largest alignment in a source module, the linker issues a warning and overrides the **ALIGN** specification.

Example 3 Sample SVR3-Style Command File on page 40 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 startup 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.

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 *MetaWare C/C++ Programmer’s Guide* for more details about thread-local storage.

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).
- A semicolon at the end of each command line is optional. You can use it if you wish to make parsing less ambiguous.

- Numbers must be specified as C constants (for example, 123 or 0x89ABC).
  - — To express kilobytes, suffix a number with ‘K’ (for example, 24K). The linker automatically multiplies the number by 1024.
  - — To express megabytes, suffix a number with ‘M’ (for example, 16M) The linker automatically multiplies the number by 1024 * 1024.
- Any reference within a command file to a file in a location other than the current working directory or the directory specified in the linker command must include the full path.
- 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 Using Wildcards in File-Name References on page 37 for more information.

Expressions

The **Assignment** statement and 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.

SVR3-Style Command Reference

Table 5 SVR3-Style Command Types and Commands lists the command types and commands for SVR3-style command files.

Table 5 SVR3-Style Command Types and Commands

Command Type — Definition
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

Table 6 SVR3-Style Commands

Command — Definition
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

Table 6 SVR3-Style Commands

Command — Definition
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, bold punctuation characters such as " **[** " and " **{** " are meta-characters, which indicate choices of arguments, repeating arguments, and so on. Do not enter meta-characters in your command files.

Argument — Specify any command-line argument recognized by the linker

Syntax: **{** *argument*... **}**

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 Linker Options on page 17 for a description of linker command-line options.)

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:

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

NOTE We recommend 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 you copy files from one project to another, or from one host operating system to another. For more information, see Specifying Command-Line Arguments in a File on page 15.

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 */`
- `identifier @type assignment_op expression;`

assignment_op is one of the following assignment operators:

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

type is one of the following symbol types:

`data`

`func`

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 (available only within a section). This symbol for the program counter is useful for creating a hole (an empty range of memory) in the output section. Example 7 SVR3 SECTIONS Command for LOADING, ALIGNING, and Hole on page 55 shows how to create a hole. (See the **SECTIONS** section for information about statements.)

type specifications are case-insensitive. They set the ELF symbol type for the associated symbol in the generated symbol table as follows:

- *`data`* to `STT_OBJECT`
- *`func`* to `STT_FUNC`

The following alternate spellings are permitted:

- *`object`* for *`data`*
- *`function`* for *`func`*

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

```
_text1 = .;
```

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

```
. += 2000;
```

This example declares a global symbol to a constant value:

```
_ZERO = 0;
```

This example alerts the linker to the presence of a Thumb function not actually in the link (possibly in a ROM or Flash device), so that the linker can supply not only distance thunks but also mode thunks:

```
rtos_entry_function0 @thumb = ADDRESS(.rtos_function_table)+0x10;
```

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*`)`

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 4 Using SVR3 FILL Datum to Generate Fill Characters

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

Example 5 Using SVR3 LONG Datum to Initialize Program Memory 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.)

Example 5 Using SVR3 LONG Datum to Initialize Program Memory

```
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.

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
```

This example **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:

Table 7 Forms of Section

Section	Definition
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`.

The **INITDATA** command specifies an output section to be initialized at run time. This capability is required for programs that must run in ROM, or that need to place pre-initialized data in a read-only section such as ROM from which startup function **_initcopy()** can copy it into RAM. Ordinarily, the **INITDATA** command stores its output section in compressed form. See Linker-Options Reference on page 18 for compression-related options. For an example command file that compresses both code and pre-initialized data, see Compressing Code and Data on page 77.

NOTE The **INITDATA** statement no longer includes sections that are flagged as **NOLOAD**.

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.

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

NOTE You can apply the **INITDATA** command to text sections, but you must take care *not* to map to `.initdat` any code executed before `_initcopy()` finishes. 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`.

CAUTION If using your own linker command file, be sure that your linker command file does not prevent `.initdat` from being defined as a linker output section. Otherwise, the linker cannot create the label `_einitdat` (end of the `.initdat` section) and **_initcopy()** cannot copy initialized data.

For example, an output specification such as

```
.rodata : { * {type lit} }
```

places all of the lit input into a single `.rodata` section. Because the linker does not define a `.initdat` section, it cannot create the label `_einitdat` and **_initcopy()** fails.

- 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 MetaWare C/C++ standard library, and in source format in the MetaWare 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. Never map to `.initdat` any code that executes before **_initcopy()** completes execution.

For a discussion of how to use **INITDATA** and the `_initcopy()` function, see Initializing RAM from a Program in ROM on page 76.

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
```

The following **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.

The following **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`. Note that the example specifies only specific `.bss` sections so as not to zero any heap sections.

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

A section specification in **INITDATA** can now be prefixed with ~, which means “exclude this section”. Example:

```
INITDATA !data, ~.mydata
```

This means: add all sections of type *data* to the `.initdat` table, except the section named mydata.

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.

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 We recommend 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 Specifying Command-Line Arguments in a File on page 15 for another way to pass input file names to the linker.)

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 How the Linker Processes Archive Libraries on page 83 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* [,]

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 6 Using the SVR3 MEMORY Command to Specify Ranges of Memory

The following 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.

```
MEMORY {
   range_1 : ORIGIN = 0x1000, LENGTH = 0x8000,
   range_2 : ORIGIN = 0xa000, LENGTH = 0xc000
   }
SECTIONS {
   .text : {} > range_2
   .data : {} > range_1
   .bss  : {} > range_1
   }
```

The **SECTIONS** command in the preceding 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**

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*)

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* ...] : [AT (*expression*)] { [*stmnt* ...] }

> **NOTE** **AT** (*expression*) is identical to the qualifier `LOAD(expression)` (explained below), but must appear after the colon.

    [= *2_byte_pattern*]
    [> *memory_block*]

output_sspec has the following syntax:

{*sect_name*} [?] | {* TYPE *sect_type*}

The optional '?' after *sect_name* instructs the linker to omit the section if the section is empty.

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*)

  Aligns the section to the number of bytes to which *expression* evaluates.

- **LOAD**`(`*expression*`)`

  Loads the image at the address to which *expression* evaluates.

- **NOLOAD**

  Do not load section at run time. The linker processes the section to the output file, but marks it so that a loader does not load it into memory at run time. Note that the section still occupies a range of memory; it simply remains uninitialized during program load. For an example, see Marking a Section to Not Be Loaded on page 74.

NOTE The MetaWare debugger does not load sections marked **NOLOAD**.

- **SIZE**`(`*expression*`)`

  SIZE is the precomputed group or section size.

- **BLOCK**`(`*espression*`)`

  Ensures the byte size of the section is a multiple of the number to which *expression* evaluates, but does not by itself set alignment.

A section (or a group that contains one or more sections) can contain one or more *stmnt* items. *stmnt* 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:

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

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.

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:

Table 8 Pseudo Functions

Function	Definition
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.

You can use the **DEFINED** operator to conditionally reference symbols whose existence isn't known until link time; for example:

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

Sorting using "$":

The linker maps any input section *x*$*y* to section *x*, sorted by *y*. For example, `.text$func` maps to section `.text`, sorted by `func`. You can override this behavior by remapping the sections in the linker command file.

Binding to an address:

You can also specify the address of an output section or group using the **BIND** keyword followed by an expression that can contain the following pseudo functions:

Table 9 Pseudo Functions for Addresses

Function	Definition
ADDR(*mem_area***)**	Address of a memory area
ADDR(*sect_name***)**	Address of a previously defined section
HEADERSZ	The file offset following the ELF program header table (this is typically where the data of the first output section is placed)
Function	**Definition**
NEXT(*value***)**	The first multiple of *value* that falls in unallocated memory (*value* is typically a page size)
SIZEOF(*memory_area***)**	Size of a memory area defined using a **MEMORY** command
SIZEOF(*section_name***)**	Size of an output section

TYPE *section_type*	Type of an input or output section. *section_ type* is one of the following:
	`bss`
	`data`
	`lit`
	`text`
	`info`
	See Sections in Executable Files on page 36 for explanations.

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.

If you do not list any input sections, the linker accepts as input only input sections with the same name as the output section.

The components of an *input_sspec* are as follows:

Table 10 input_sspec Components

Component	Definition
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:

Table 11 sect_spec Definition

Component	Definition
sect_name	The section name
TYPE *sect_name*	The section type

NOTE To parse correctly, any file name that contains parentheses (denoting archive members), 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.

You can make assignments inside a statement. For information, see the listing for the **Assignment** command in SVR3-Style Command Reference on page 43.

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:

Table 12 Wild-Card Tokens

Token	Definition
?	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 7 SVR3 SECTIONS Command for LOADING, ALIGNING, and Hole

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

This example does the following:

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 8 SVR3 SECTIONS Command for Combining Input and Grouping Output

```
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 9 SVR3 SECTIONS Command for Creating Empty Section and Symbol

```
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 10 SVR3 SECTIONS Command for Combining Parts of Three Files

ld file.cmd mod1.o mod2.o mod3.o

```
# file.cmd:
SECTIONS {
#
# The group starts at address 0x400.
#
   GROUP BIND 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}
   }
```

This example links three object files to produce an output file with its `.data` sections combined into `outdata`, followed by a combined `.bss` section and two separate code sections, `usertext` and `.text`. 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 11 SVR3 SECTIONS Command for Grouping All Sections of a Type

```
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 12 SVR3 SECTIONS Command That Sets a Pointer to Free Memory

```
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 13 SVR3 SECTIONS Command for Mapping Input to Named Output

ld -u _ForceUndef file.cmd file.o

```
# file.cmd:
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:

Table 13 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 `memory3`, 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`.

Example 14 SVR3 SECTIONS Command for Locating Sections in ROM and RAM

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

This example specifies the locations for output sections as follows:

1. The first **GROUP** directive tells the linker to locate the executable code (`.text`) and read-only data (`.lit`) together at the single address 0x200000 (arbitrary address as an example).
2. The second **GROUP** directive tells the linker to locate the initialized data (`.data`) and uninitialized data (`.bss`) together at the single address 0x300000 (arbitrary address as an example).

START — Specify program entry point

Syntax:

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

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.

symbol is the name of an exported symbol. *value* is an address; it must be a C integer constant

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.

The following **START** command sets the entry point to the address of the symbol `begin`:

```
START begin
```

The following **START** command sets the entry point to address 800:

```
START 0x800
```

Additional Keywords

Because the linker's SVR3 command syntax has been extended to support AMD- and Diab-style command files, you can also use the keywords listed below if you have experience with AMD- or Diab-style command files.

Table 14 Additional Keywords

ABSOLUTE	**OUTPUT_FORMAT**
AT	**PUBLIC**
FILL	**RESADD**
FLOAT	**RESNUM**
FORCE_COMMON_ALLOCATION	**SEARCH_DIR**
GROUP	**SIZEOF_HEADERS**
NAME	**STARTOF**
NOFLOAT	**STARTUP**
ORDER	**TARGET**
OUTPUT_ARCH	

Chapter 6 — SVR4 Command Files

In This Chapter

SVR4 Command-File Format

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:

You can 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*.

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.

Sample Command File

This section explains the file shown in Example 15 Sample SVR4-Style Command File.

Example 15 Sample SVR4-Style Command File

```
text=LOAD ?RX;
text:$PROGBITS ?A!W;
data=LOAD ?RWX;
data:$PROGBITS ?AW;
data:$NOBITS ?AW;
my_note=NOTE;
my_note:$NOTE
```

The preceding command 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.

SVR4-Style Command Reference

You can use the following types of directives in an SVR4-style command file:

Table 15 SVR4-Style Directives

Type of Directive — Definition
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 Earlier versions of the linker supported extensions to the SVR4 command syntax that allowed users to define output sections by name. This support is now available with either AMD-style or Diab-style SVR3 command syntax, and is no longer supported by SVR4-style commands.

NOTE In the following syntax descriptions, bold punctuation characters such as "**[**" and "**{**" are meta-characters that indicate choices of arguments, repeating arguments, and so on. 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:

Table 16 SVR4 Optional Attribute Arguments

Argument	Description	Valid Values
seg_type	Loaded by the loader at load time	**LOAD**
	Read by the loader at load time	**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*.

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 the preceding 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 Using Wildcards in File-Name References on page 37).
- The attribute arguments can be one or more of the following:

Table 17 Mapping directive attribute arguments

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*.

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 16 SVR4 file for Mapping Input Sections to an Output Segment

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

In Example 16 SVR4 file for Mapping Input Sections to an Output Segment, 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*

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 C or C++ identifier.

Example 17 SVR4 file for Assigning a Section-Size Name to a Segment assigns the size name `Protected_Data_Section_Size` to segment `seg_1`.

Example 17 SVR4 file for Assigning a Section-Size Name to a Segment

```
seg_1 @ Protected_Data_Section_Size
```

Chapter 7 — Topics and Techniques

In This Chapter

Special Linker-Defined Symbols

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

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

Naming Conventions

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

Table 18 Symbol Naming Conventions

Symbol Name	Definition
`_f`*section_name*	Set to the address of the start of *section_name*
_esection_name	Set to the first byte following *section_name*

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

Linker-defined symbols may vary by target processor. The following are example linker-defined symbols for an ARM processor, where `Base` indicates the start of a section, and `Limit` indicates the end:

```
Image$$RW$$Base
              global  .data      020959ac  <linker defined>
Image$$RW$$Limit
              global  .bss       0215cea0  <linker defined>
Image$$ZI$$Base
              global  .bss       020a3538  <linker defined>
Image$$ZI$$Limit
              global  .bss       0215cea0  <linker defined>
```

Providing Symbols for a Symbol Table

You can provide an external symbol list that the linker can use to determine which symbols should be inserted into the symbol table of an executable. This allows functions in one executable of an application can call functions in another.

Examples:

- Your target system maintains a symbol table on the embedded device for run-time fixups.
- You intend to load more than one executable (at disjointed addresses).
- You want the linker to see a file of defined symbols early in the link to avoid pulling in library code if not required.

Related Topics

- Providing a Symbol Table for a Secondary Executable

Providing a Symbol Table for a Secondary Executable

To generate and read an external symbol file, link in two steps, using the options described in this section. This method allows subsequently linked executables to reference symbols within the first executable.

1. Link initially using linker option **-Bsymout** to obtain a list of public symbol definitions that can be subsequently used in another link.

   Syntax: **-Bsymout**=*path*

NOTE You can also use option **-J** to limit the symbols emitted.

2. Link the final executable using linker option **-Bsymin** with the file you specified to **-Bsymout**.

   Syntax: **-Bsymin**=*path*

   This option defines symbols that are exported from another executable that is assumed to be resident when the generated executable is loaded. The linker resolves undefined symbols from the **-Bsymin** file as well as from the collection of objects and libraries presented on the command line. The symbols in the **-Bsymin** file are treated as weak absolute symbols, but prevent corresponding symbols from being pulled out of a library.

Example 18 shows a build script that uses the driver to obtain a list of symbols from a main executable, then define selected ones for a secondary executable.

Example 18 Providing Symbols for the Symbol Table

```
mcc -Bbase=0x1000 -g main.c -Hhostlink -o main.out
-Bsymout=symlist.txt
grep -v __x symlist.txt > symlist1.txt
mcc -c -g queens.c
mcc "$@" -g -Bbase=0x100000 -Hnocrt demo.c queens.o -Hhostlink -o
demo.out -Bsymin=symlist1.txt
```

Related Topics

- Linking Incrementally on page 72
- **-Bsymout** in Linker-Options Reference on page 18
- **-Bsymin** in Linker-Options Reference on page 18
- **-J** in Linker-Options Reference on page 18

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. The code fragment in Example 19 Finding Section Size from C illustrates how to access the address of the `.data` section and determine its size:

Example 19 Finding Section Size from C

```
/* 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);
   }
```

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

Zeroing and Copying Sections without Using INITDATA

The linker provides a mechanism for zeroing and copying sections using the **INITDATA** command and the C library function **_initcopy()**. If you wish to avoid use of **INITDATA** (for example, to maintain compatibility with other toolsets), you can zero and copy sections manually using the linker-defined symbols.

The C fragment in Example 20 Zeroing a Section from C zeroes the `.heap` section:

Example 20 Zeroing a Section from C

```
.extern char _fheap[], _eheap[];

void zero_heap() {
   memset(_fheap, 0, _eheap - _fheap);
}
```

The code in Example 21 Storing a Copy of the Data Section from C stores a copy of the data section for reinitialization on warm starts using a reserved area of RAM:

Example 21 Storing a Copy of the Data Section from C

```
extern char _fdata[], _edata[];
static int is_warmstart = 0;
#define RESERVED (void*) 0x8000000
void warmstart() {
   if (is_warmstart)
      memcpy(_fdata,RESERVED,_edata-_fdata);
   else {
      is_warmstart = 1;
      memcpy(RESERVED,_fdata,_edata-_fdata);     }
      }
```

Linking Incrementally

Incremental linking is the process of combining two or more relocatable object files into a partially linked relocatable object file. You can use the resulting object file as input to additional linker invocations, and to your final executable. 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`.

Example 22 Linking Incrementally

```
ld -r -o t123.o t1.o t2.o t3.o // Combines t1.o..t3.o into t123.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. To produce an executable with full relocation information during an incremental link, use **-Bexe** with **-r**. The executable is larger because it contains fixup tables (static relocation tables), but the size of the executable code is not affected.

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

When to Use Incremental Linking

You might wish to link incrementally to prevent conflict between duplicate labels, or to obtain a partially linked executable (a library) that can be added to later with less chance of location problems like cache thrashing.

- Linking incrementally with **-r** and building an executable later does not cause changes to the final executable unless you use **-Bexe**.
- Linking incrementally with **-r** and including the output with other object files in an executable later does not affect the size of the executable unless you use **-Bexe**.
- Using **-Bexe** to produce an executable *during* the incremental link changes the executable by adding fixup tables, which increases its size.

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**.

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 Linker-Options Reference on page 18 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, 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. If your target supports position-independent code, see the *MetaWare C/C++ Programmer's Guide* for information on how to compile position-independently.

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.

Marking a Section to Not Be Loaded

To mark a section to not be loaded into memory at run time, use the **NOLOAD** qualifier within the **SECTIONS** command in an SVR3 command file.

In Example 23 Marking a Section to Not Be Loaded the ROM section is addressed at memory location 0 and does not need to be loaded when the program executes.

Example 23 Marking a Section to Not Be Loaded

```
SECTIONS      {
   ROM           (NOLOAD)  : { ... }
...
}
```

The linker processes the ROM section normally and its contents appear in the linker output file, but marked so that a loader knows not to load them at run time. The MetaWare debugger does not load sections marked **NOLOAD**, but utilities such as **elfdump** and **elf2hex** handle them normally.

Reserving Space in an Output Section

Your application might require you to reserve space within a single output section for special memory or variables from another application.

It is not possible to explicitly reserve space in the middle of an output section. Instead, split the output section into two output sections with different names and then locate the second one at a fixed address past the reserved area using a new **GROUP** directive. If the original output section is a default output section like `.data`, the new output section must have its data emitted into a named data section. Example 24 Reserving Space by Using a Second Section with a New Name (SVR3) on page 75 shows how to do this using C/C++ and an SVR3 command file. Depending on the size of your variables, you might need to use this method several times and do some tinkering to fill the default section without overflowing into the reserved section.

Example 24 Reserving Space by Using a Second Section with a New Name (SVR3)

C source

```
   ...
   #pragma data("data_section_above_reserved_area")
      int x =3;
   #pragam data()
   ...
```

Linker Command File

```
MEMORY
{
   vtable:            origin = 0x00000000, length = 0x00000100
   romem:             origin = 0x00000800, length = 0x00080000
   rwmem:             origin = 0x00080800, length = 0x00020000
   HIGH_DATA:         origin = 0x000A0800, length = 0x00010000
   kernel_data_ram:   origin = 0x000E0000, length = 0x00100000
   flash:             origin = 0x00200000, length = 0x00200000
}

SECTIONS
{

   GROUP : {
      .vectors: {}
   } > vtable

   GROUP : {
      .text   : {}
      .init   : {}
      .fini   : {}
      .rodata : {}
   } > romem

   GROUP : {
      .data : {}
      .mcount? : {}
      .tls? : {}
   } > rwmem

   GROUP : {
      .sdata : {}
      .sbss : {}
   } > rwmem

GROUP  {
     data_section_above_reserved_area : {}
        } > HIGH_DATA


   GROUP : {
      .bss   : {}
      .heap? SIZE(DEFINED _HEAPSIZE?_HEAPSIZE:0): {}
      .stack SIZE(DEFINED _STACKSIZE?_STACKSIZE:4096): {}
   } >rwmem
}
```

Rewiring Shared 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/Linux-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 Linker-Options Reference on page 18 for more information.

Initializing RAM from a Program in ROM

Applications in ROM must have their writable data sections initialized in RAM at run time. You can accomplish this using the command **INITDATA** in an SVR3 linker command file.

The **INITDATA** command places pre-initialized data in a read-only `.initdat` section from which the startup code can copy it for placement in RAM. The usual practice is to store the `.initdat` section in non-volatile read-only memory such as ROM, EPROM, or flash. When the embedded system is reset, the program starts executing in the non-volatile memory and function **_initcopy()** reinitializes data to its correct initial value.

Note that the data is compressed by default when stored in the `.initdat` section. If your code runs faster in volatile memory (RAM) than it does in non-volatile memory, you can store the executable code in compressed form in the `.initdat` section as well, and copy it to RAM during startup. However, it is essential to leave a small amount of startup code uncompressed so that execution can proceed to the **_initcopy()** function prior to decompression. You can do this by editing your linker command file to isolate the minimum startup code modules required.

When you run your application using the MetaWare debugger, **INITDATA** is not usually necessary, because the debugger downloads the data by default every time you restart. When you run your application without downloading to the debugger, **INITDATA** effectively “downloads” the data whenever the embedded system is reset.

NOTE Compressing code is not compatible with closely coupled memory (a Harvard architecture).

For details about the syntax and usage of **INITDATA**, see the **INITDATA** listing in SVR3-Style Command Reference on page 43.

Compressing Code and Data

Example 25 SVR3 Linker Command File to Compress Code and Data uses **INITDATA** to compress code and pre-initialized data. This simple example is sufficient for demo programs such as the queens demo on your distribution. Although written for an ARC processor, the example applies analogously to other processors.

Example 25 SVR3 Linker Command File to Compress Code and Data

```
INITDATA !data, !bss, !lit, .text, .init, .fini
SECTIONS {
   GROUP: {
      .start: {
         crt1.o(.text)

//Asterisk so libraries can be found without full path

         "*libct.a(arc_main.o)"(.text)
         "*libct.a(initcopy.o)"(.text)
         "*libct.a(lzdecompress.o)"(.text)
         "*libct.a(memcpy.o)"(.text)
         "*libct.a(f_memset.o)"(.text)
         }
      .initdat (LIT):
      }
   GROUP: {
      * (TEXT): {}
      * (LIT): {}
      }
   GROUP: {
      .sdata?: {}
      .sbss?: {}
      * (DATA): {}
      * (BSS): {}
      .stack ALIGN(4) SIZE(DEFINED _STACKSIZE?_STACKSIZE
                  :65536): {}
      .heap? ALIGN(4) SIZE(DEFINED _HEAPSIZE?_HEAPSIZE:0): {}
      }
   }
```

Linking for Closely Coupled Memory (a Harvard Architecture)

You must create a special linker command file to link programs for closely coupled memory, taking care to ensure that *all* data sections, whether read-only or read-write, are linked in the data space and only executable instructions are linked in the instruction space. Specifically, the lit and text sections must be separate, which is not the default case. Example 26 Sample Linker Command File for Closely Coupled Memory (Harvard) is a linker command file that links for closely coupled memory using the queens demo included on your distribution.

NOTE Avoid overlapping code and data addresses in your linker command file, as doing so can make it impossible to verify address validity using the debugger and simulator. The best practice is to continue using different addresses for code and data spaces even though the hardware permits overlapping addresses.

Example 26 Sample Linker Command File for Closely Coupled Memory (Harvard)

```
memory {
   CODE_ROM: ORIGIN = 0x00000000 LENGTH = 1M
   DATA_ROM: ORIGIN = 0x00100000 LENGTH = 1M
   DATA_RAM: ORIGIN = 0x00200000 LENGTH = 1M
   }

SECTIONS {
   GROUP: {
      .interrupt_vector_section?: {}
      * (TEXT): {}
      } > CODE_ROM

   GROUP: {
      * (LIT): {}
      } > DATA_ROM

   GROUP: {
      .sdata?: {}
      .sbss?: {}
      * (DATA): {}
      * (BSS): {}
      .stack ALIGN(4) SIZE(DEFINED _STACKSIZE?_STACKSIZE:65536):
        {}
      .heap? ALIGN(4) SIZE(DEFINED _HEAPSIZE?_HEAPSIZE:0): {}
      } > DATA_RAM
   }
```

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 MetaWare C/C++ distribution. This function initializes the appropriate sections from a table constructed by the linker in section `.initdat`.

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

Calling _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 `.initdat` 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.

Binding in ROM and Executing in RAM

SVR3 linker command files provide great flexibility in where you bind, load, and execute sections. This flexibility allows embedded systems to bind code and data in one location, start up, then copy to a faster memory device for execution.

Example 27 SVR3 Command File for Remapping Code from ROM to RAM is an SVR3 linker command file that binds boot code and read-only data in ROM and the rest of the code and the read-write data at their execution addresses in RAM. It uses the **INITDATA** command to create a section `.initdat` with a compressed copy in ROM of the non-boot code and read-write data from RAM. On booting, the startup function **_initdat()** uses a table in `.initdat` to unpack and decompress the copy of non-boot code and data from `.initdat` to their execution locations in RAM.

Example 27 SVR3 Command File for Remapping Code from ROM to RAM

```
// NOTE: This is a generalized example and does not
// correspond to any actual ABI
MEMORY  {
   FLASH: ORIGIN = 0x04000000 LENGTH = 512K
   SSRAM: ORIGIN = 0x00000000 LENGTH = 128K
   SRAM: ORIGIN = 0x000020000 LENGTH = 512K
   }
INITDATA !data, !bss, .text
// Store r/w data initial values in lit with .text
// and zero .bss (except any .stack)
SECTIONS {
   GROUP: {
      boot_section: {
      // boot_section runs from flash only

//Files assumed to be in current working directory

         crt1.o (.text)
         // crt1.o supplied for C run time does not init
         // all hardware
         crti.o(.text)
         crtn.o(.text)
         "libmw.a(initcopy.o)"(.text)
         // initcopy() unpacks .initdat
         "libc.a(lzdecompress.o)"(.text)
         // Linker compresses if it finds lzdecompress
         "libc.a(memset.o)"(.text)
         // Other target-specific modules might have to
         // remain in flash for startup
         "libmw.a(mem.o)"(.text)
         "libc.a(memcpy.o)"(.text)
         }
      * (LIT): {}
      // The INIDATA output section (.initdat) will be
      // stored here because it is of type LIT
      } > FLASH
   GROUP: {
      .text: {
```

```
      // output text section unpacked from flash and
      // run from SSRAM
      vector.o(.vector_section)
      // Here we assume that the vector table was moved
      * (TYPE text)
      // Place all other sections of type TEXT here
         }
       } > SSRAM
   GROUP : {
      * (DATA): {}
      .sdata?: {}
      // The small-data base register might need to
      // be set here, depending on the ABI
      .sbss?: {
         * {.sbss}
         }
      .heap? SIZE(DEFINED _HEAPSIZE?_HEAPSIZE
                  :0x2000): {}
      .stack SIZE(DEFINED _STACKSIZE?_STACKSIZE
                  :0x10000): {}
      } > SRAM
   }
```

Example 27 SVR3 Command File for Remapping Code from ROM to RAM on page 79 does the following:

1. Uses the **MEMORY** command to set up separate areas of memory named `FLASH`, `SSRAM`, and `SRAM`. If the linker reports that the sections are exceeded, increase the sizes or inspect the memory-map listing (produced using option **-m**) and determine the minimum sizes you require.
2. Uses the **INITDATA** command to have the linker store initialization values for all read-write data (`.data`) and all the code (output sections `.text`) in a read-only section `.initdat`. The **INITDATA** command also asks the linker to record the BSS (uninitialized data, `.bss`) section limits so that startup function **_initcopy()** can zero them during boot. You can omit this operation by removing the `!bss`.
3. Specifies the startup and boot code to be run only once, prior to remapping of code. The first **GROUP** places these startup modules in flash just prior to lit (read-only) data.
4. Scans all input modules for read-only data (input specification `* (LIT): {}`) and places them in output sections of the same names by default. The linker places read-only sections such as `.rodata` and `.lit` here.

   The linker also places the linker-generated section `.initdat` in LIT. Section `.initdat` contains the tables **_initcopy()** uses for copying or uncompressing on boot.
5. Locates the final run location of the reset vector area in fast static RAM starting at address 0. The second **GROUP** combines input section `.vector_section` with all other text modules (**TYPE** `text`) not already placed in the output section `boot_section` and places them in an output section named `.text` to match the standard. This code runs here but is loaded with the LIT data in `.initdat`. This code might be compressed if the decompression module `lzdecompress.o` is in the boot. If you do not want compression, use linker option **-Xnocompress**.

The application can then use the vector table to unpack data from flash into read-write memory and fast-access ram.

Example 28 Section Summary and Compression Statistics shows the section summary and compression statistics for a build with the following command line:

```
mcc -m -Hldopt=-Xcompress_stats -Hldopt=-Coutput=hello.map linker_cmd.txt vectors.s
hello.c
```

The command line uses -C`output=hello.map` to obtain a useful listing directed to a file and assumes a linker command file called `linker_cmd.txt`.

Example 28 Section Summary and Compression Statistics

```
SECTION SUMMARY
_______________

OUTPUT/   TYPE    START     END
 INPUT SECTION  ADDRESS   ADDRESS   LENGTH

.text     bss     00000000  0000763f  00007640
.data     bss     00020000  0002066b  0000066c
.tls      bss     0002066c  0002066f  00000004
.heap     bss     00020670  0002266f  00002000
.stack    bss     00022670  0003266f  00010000
.bss      bss     00032670  00032a33  000003c4
boot_section
          text    04000000  040009c7  000009c8
.initdat lit     040009c8  040062d3  0000590c
.rodata   lit     040062d4  040062d7  00000004
.rodata1 lit     040062d8  04006526  0000024f

START ADDRESS:  04000000
[INITDATA] section .data compressed 70.6%
[INITDATA] section .tls compressed -25.0%
[INITDATA] section .text compressed 26.5%
[INITDATA] Total overall compression: 28.6%
```

Example 29 Section Summary (without Compression) shows the section summary for a (non-compressed) build with the following command line:

```
mcc -m -Hldopt=-Xnocompress -Hldopt=-Coutput=hello.map linker_cmd.txt vectors.s
hello.c
```

The command line uses -C`output=hello.map` to obtain a useful listing directed to a file and assumes a linker command file called `linker_cmd.txt`.

Example 29 Section Summary (without Compression)

```
SECTION SUMMARY
_______________

OUTPUT/   TYPE    START     END
 INPUT SECTION  ADDRESS   ADDRESS   LENGTH

.text     bss     00000000  0000763f  00007640
.data     bss     00020000  0002066b  0000066c
.tls      bss     0002066c  0002066f  00000004
.heap     bss     00020670  0002266f  00002000
.stack    bss     00022670  0003266f  00010000
.bss      bss     00032670  00032a33  000003c4
boot_section
          text    04000000  040008ef  000008f0
.initdat lit     040008f0  040085d3  00007ce4
```

```
.rodata  lit    040085d4  040085d7  00000004
.rodata1 lit    040085d8  04008826  0000024f
```

Mapping Modules Directly

Depending on your processor configuration and software application, you can use pragmas and a linker command file to map heavily used data or code modules to fast RAM directly (rather than by section type) to optimize execution speed.

- Mapping Data Modules Directly
- Mapping Code Modules Directly

Mapping Data Modules Directly

Example 30 shows the pragma and linker command file for mapping data modules directly to RAM.

Example 30 Mapping Data Modules Directly (SVR3 File)

Source code for data module:

```
#pragma data("fast_data_section")
   int global_flag=1;
#pragma data();
```

Linker command file:

```
MEMORY
{
   ...
   ...
   LDST_RAM: origin = 0x00200000, length = 2K
   ...
   ...
}
   GROUP : {
      fast_data_section ? : {} // input section is select
                                // same as output section
      fast_data_modules_section ? : { // alternate form
                                      // with modules and
                                      // type specified
         file1.o (TYPE data)
         file2.o (TYPE data)
         }
   } > LDST_RAM
```

Mapping Code Modules Directly

Example 31 shows the pragma and linker command file for mapping code modules directly to RAM.

Example 31 Mapping Code Modules Directly (SVR3 File)

Source code for code module:

```
#pragma code("fast_text_section")
   void function2(void) {
   ...
   ...
#pragma code();
```

Linker command file:

```
MEMORY
{
   ...
   ...
   CODE_RAM: origin = 0x0000000, length = 2K
   ...
   ...
}
   GROUP : {
      fast_text_section ? : {} // input section is
                               // select same as
                                  // output section
      fast_text_modules_section ? : { // alternate form
                                      // with modules
                                      // and type specified
          file1.o (TYPE text)
          file2.o (TYPE text)
          }
   } > CODE_RAM
```

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**.

Default Convention for Processing Archive Libraries

If you do not specify option **-Bgrouplib**, the linker searches archive libraries in the order you specify them. When the linker encounters an archive library, it searches to resolve only those unresolved symbols that are pending when it encounters that archive library. In other words, the linker does not resolve backward references to archive libraries it has already processed. The linker searches the current archive library repeatedly, if required, to resolve references within that archive library.

To search an archive library more than once (for example, to resolve mutual references), either specify the archive library multiple times, or use linker option **-Bgrouplib**.

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 is always extracted.

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 re-scan 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.

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.

Locating Data from a Library Object

The linker can place specified data from a library object file in a specific output section or location.

Example 32 uses `libmw.a` from the MetaWare C runtime. but applies to any library. It uses the **GROUP** command to specify a new output section `LIBMW_MEMORY_LOCATION` with input data of type text from `libmw.a`.

- The leading ‘*’ is a wildcard for the leading path information.
- The trailing `(*)` indicates all objects in the library. You can also specify individual object files, such as `write.o`.
- You can replace `(TYPE` *`text`*`)` with other types, such as *`data`*, *`bss`*, *`lit`*, or `(*)` for “any.”
- You can use section names like `.data` or `.text` rather than `(TYPE text)`.

Example 32 Linking Library Data to a Specific Section

```
GROUP: {
   libmw_section: {
      "*libmw.a(*)"(TYPE text)
      // "*libmw.a(*)"(*) Results in warnings
      // because all section types are concatenated.
      // "*libmw.a(write.o)"(.text)  Specifies section name
   }  > LIBMW_MEMORY_LOCATION
```

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.

Locating Uncached Data in an Executable

Compiler Toggle `Uncached_in_own_section` places all uncached variables (**volatile** variables with toggle `Volatile_cache_bypass` left on or **_Uncached** variables) in the special ELF section `.ucdata`. You can then locate the uncached variables in your executable using a linker command file.

For more information on using the **volatile** and **_Uncached** type qualifiers and toggles, and their implications on different ARC targets, see the “Embedded Development” chapter in the *MetaWare C/C++ Programmer’s Guide for ARC*. This section presents a sample linker command file for locating data placed in the `.ucdata` section.

NOTE If you are placing your uncached data (`.ucdata`) with a linker command file and manually ensuring enough space between it and other data, you can save space by turning off toggle `Align_uncached_section`.

Example 33 Locating Uncached Data is a simplified example of a linker command file for mapping uncached data to a specific address in a target system with shared memory.

Example 33 Locating Uncached Data

```
MEMORY  {
                    RAM: ORIGIN = 0x0020000 LENGTH = 512K
                    CONTROL_REG_RAM: ORIGIN = 0x80000000 LENGTH = 64K
                    }

SECTIONS {

        GROUP: {
              *(TEXT) : {}
              *(LIT)  : {}
                    } > RAM
        GROUP: {
              .sdata?: {}
              .sbss?:  {}
              *(DATA): {}
```

```
            *(BSS):  {}
            .stack ALIGN(4) SIZE(DEFINED _STACKSIZE?_STACKSIZE:65536): {}
            .heap? ALIGN(4) SIZE(DEFINED _HEAPSIZE?_HEAPSIZE:0): {}
                     }  > RAM

      GROUP ALIGN(256): {
            // uncached data in control registers mapped to 0x80000000
            .ucdata NOLOAD ?: {} // such data is typically not loaded or zeroed
                  } > CONTROL_REG_RAM
         }
```

Chapter 8 — Linker Error Messages

In This Chapter

Error-Message Severity Levels

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

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.

You can suppress output of warnings that are prefixed by a number using option **-zoffwarn**.

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.

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.

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. To redirect errors to standard output, use option **-zstdout**.

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.

Chapter 9 — Using the Utilities

In This Chapter

Using the Archiver

This section describes the archiver, **ar**. The archiver is actually named **ar***target*, where the suffix *target* identifies the target platform. In this section, we use **ar** to generically represent the archiver command. See Table 19 Target Suffixes and Archiver Utility Names for the exact archiver name for your target.

Table 19 Target Suffixes and Archiver Utility Names

Processor Family	Target Suffix	Utility Name and Command
ARCtangent-A4	arc	**ararc**
ARCtangent-A5 and later	ac	**arac**

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

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 21 Archiver Command-Line Options.

NOTE In the following examples, bold punctuation characters such as "**[**" and "**{**" are meta-characters that indicate choices of arguments, repeating arguments, and so on. Do not enter meta-characters on the command line.

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

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

To delete members of an archive, use this syntax:

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

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*...]

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

ar -s[v] *archive*

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:

Table 20 Archiver Command-Line Arguments

Argument	Definition
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.

Archiver Options

Table 21 Archiver Command-Line Options summarizes the archiver options. A dash (-) before the option is not required. No spaces are allowed between multiple options.

Table 21 Archiver Command-Line Options

Option	Meaning
-d	Delete member(s) from an archive library.
-h	Display archiver command synopsis.
-m	Extract members from one archive library and insert them into another. If the members already exist in the second archive library, the archiver does not replace them.
-M	Extract members from one archive library and insert them into another. If the members already exist in the second archive library, the archiver overwrites them.
-r	Replace (or add) member(s) in an archive library.
-s	Reconstruct the symbol table of an archive library.
-S	Display the symbol table entries of an archive library. If you do not specify any archive members, the archiver assumes all members of the archive.
-t	Display names of members in an archive library. If you do not specify any archive members, the archiver assumes all members of the archive.
-x	Extract specified member(s) from an archive library. If you do not specify any archive members, the archiver assumes all members of the archive. Option **-x** does not alter the archive library.
-v	Display verbose output of archiver actions.

Specifying Archiver Command-Line Arguments in a File

You can place frequently used archiver command-line arguments in an argument file. To do so, 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.

Archiver Invocation Examples

This section presents several archiver invocation examples.

Example 34 Deleting a File from an Archive Library deletes object file `filename.o` from library `libx.a`. It provides verbose output:

Example 34 Deleting a File from an Archive Library

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

Example 35 Extracting Files from an Archive Library on page 92 extracts object files `survey.o` and `table.o` from library `liborgnl.a`. It does not provide verbose output:

Example 35 Extracting Files from an Archive Library

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

Example 36 Replacing Files in an Archive Library with New Files replaces the object files in library `libnew.a` with new object files named in the archiver argument file `newobjs.txt`. It provides verbose output:

Example 36 Replacing Files in an Archive Library with New Files

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

Using the Command-Line Profiling Utility

This section describes the **prof** statistical-profiling utility, which analyzes the run-time performance of a program compiled with a profiling option, then executed to collect data in a file. However, we recommend profiling using the MetaWare debugger or the GUI-based integrated profiling tool from ARC International instead of **prof**.

The profiling utility behaves similarly to the AT&T UNIX profiler.

The profiling utility is actually named **prof***target*, where the suffix *target* identifies the target platform. In this section, we use **prof** to generically represent the profiling-utility command. See Table 30 Target Suffixes and File-Size Utility Names for the exact name of the profiling utility for your target.

Table 22 Target Suffixes and Statistical-Profiling Utility Names

Processor Family	Target Suffix	Utility Name and Command
ARCtangent-A4	arc	**profarc**
ARCtangent-A5 and later	ac	**profac**

CAUTION Executables instrumented for profiling might require a sizeable heap. For information on providing a heap, see the *MetaWare C/C++ Programmer's Guide*.

Building for Statistical Profiling without Call Graphs

To produce an application that can generate the `mon.out` file that the statistical-profiling utility reads, you must link with driver option **-p** (to include call-graph information as well, link with driver option **-pg** as described in Building for Call-Graph Profiling on page 93). Documentation for the driver options is in the *MetaWare C/C++ Programmer's Guide*. If you need function-call counts, you must also compile individual source files with **-p**. Functions not compiled with option **-p** will show only execution-time information in the summary listing; they will not show call counts. When you compile and link your application with option **-p**, the compiler generates profiling instrumentation code as it translates your source code to assembly.

Building for Call-Graph Profiling

To produce an executable that will generate a `gmon.out` statistical-data file with call-graph information for the statistical-profiling utility to read, you must link with driver option **-pg** (see the *MetaWare C/C++ Programmer's Guide*).and then execute the instrumented program. (You can compile with **-p** or with **-pg**; the instrumentation results are the same with either option.)

Building the Statistical-Data File

To actually generate the `mon.out` or `gmon.out` file for the statistical-profiling utility to read, you must execute the instrumented application. As the instrumented application executes, the extra code collects function-call counts and performance timing information. At the end of execution, the application writes `mon.out` or `gmon.out` to the current working directory.

The statistical-profiling utility can read the `mon.out` or `gmon.out` file and the application and produce a detailed report summarizing the application's performance.

NOTE For some targets, you can view the profiling report using the MetaWare debugger when you are running the application, without the need for a `gmon.out` file. For more information, see the *MetaWare Debugger User's Guide*.

Invoking the Profiling Utility

This is the syntax for the profiling utility:

`prof` [*options*] [*exe_file*]

The command-line arguments have the following meanings:

Table 23 Profiling Command-Line Arguments

options	Display options (see Table 24 Command-Line Options for the Profiling Utility).
exe_file	Executable file for which you want to display the profiling information. The default is `a.out`.

Table 24 Command-Line Options for the Profiling Utility

Option	Meaning
Sorting options (mutually exclusive)	
-a	Sort by increasing symbol address.
-c	Sort by decreasing number of calls.

Table 24 Command-Line Options for the Profiling Utility

Option	Meaning
-n	Sort lexically by symbol name.
-t	Sort by decreasing percentage of total time (default).
Generic options (work with files linked with driver option **-p** or option **-pg**)	
-C	Unmangle C++ symbol names before printing them out.
-d	Dump raw call count and `pc` sampling tables for low-level analysis.
-e	Write errors to standard output instead of standard error.
-g	Include static (non-global) function symbols (default).
-h	Suppress the heading on the report.
-i	Show integer count rather than time information.
-l	Do not include static (non-global) function symbols.
-m *file*	Use statistical data file *file* instead of the default.
-p *digits*	Specify *digits* as the number of fraction digits for times in seconds.
-q	Suppress the copyright message.
-2	Show possible second symbol contributing to an entry.
Call-graph-specific options (work only with files linked with option **-pg**)	
-D	Visualize call graph using AT&T graphs.
-G	Visualize the call graph using GNU VCG graphs.
-L	Visualize the call graph in “landscape” orientation (requires AT&T’s **dot** and **lefty**).
-P	Generate a PostScript file for the call graph (requires AT&T’s **dot** and **lefty**).
-s	Suppress the call-graph summary.
-T	Generate a tiled call graph (requires AT&T’s **dot** and **lefty**).

NOTE Options **--L**, **-P**, and **-T** require you to first install **dot** and **lefty**, which are available from www.graphviz.org

Viewing and Interpreting Profiling Output

This section discusses how to interpret profiling results for applications compiled with the different profiling options.

Viewing Results from a Flat Profile Listing Linked with -p

If you compiled or linked with option **-p**, you can view profiling information for your program overall, and for functions within your program. Table 25 Interpreting the -p-Linked Flat-Profile Listing defines each column heading in the flat profile listing in terms of how it relates to the named function. The functions in a flat profile listing are not listed in the order in which they were called.

Table 25 Interpreting the -p-Linked Flat-Profile Listing

Column Heading	Meaning
`% time`	The percentage of the program’s total running time used by this function.
`cumulative seconds`	A running sum of the number of seconds used by this function and those listed above it.
`self seconds`	The number of seconds used by this function alone. This is the major sort for this listing.
`num calls`	The number of times this function was invoked, if this function is profiled. If the function is not profiled, this field is blank.

Table 25 Interpreting the -p-Linked Flat-Profile Listing

Column Heading	Meaning
`ms/call`	The average number of milliseconds used by this function per call, if this function is profiled. If the function is not profiled, this field is blank.
`address`	The address of the function.
`name`	The name of the function. This is the minor sort for this listing.

Viewing Results from a Flat-Profile Listing Linked with -pg

If you compiled or linked with option **-pg**, you can view information for functions that have been profiled. Table 26 Interpreting the -pg-Linked Flat-Profile Listing shows a portion of the flat profile listing generated from a program linked with option **-pg**. The functions in a flat profile listing are not listed in the order in which they were called.

Table 26 Interpreting the -pg-Linked Flat-Profile Listing

Column Heading	Meaning
`% time`	The percentage of the program's total running time used by this function.
`cumulative seconds`	A running sum of the number of seconds used by this function and those listed above it.
`self seconds`	The number of seconds used by this function alone. This is the major sort for this listing.
`num calls`	The number of times this function was invoked, if this function is profiled. If the function is not profiled, this field is blank.
`self ms/call`	The average number of milliseconds used by this function per call, if this function is profiled. If the function is not profiled, this field is blank.
`total ms/call`	The average number of milliseconds used by this function and its descendents per call, if this function is profiled. If the function is not profiled, this field is blank.
`address`	The address of the function.
`name`	The name of the function. This is the minor sort for this listing.

Using the ELF-to-Hex Conversion Utility

This section describes the ELF-to-hex converter, **elf2hex**, a stand-alone utility for converting an executable ELF file to any of several output formats.

Generating an ELF executable, then invoking **elf2hex** to convert it to hex files is the same as using the linker with option **-x**. Attributes applied to linker option **-x** have the same name, functionality, and default as the corresponding **elf2hex** options.

For background information on hex files and PROM devices, see Appendix A — PROMs and Hex Files.

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

- Motorola S3 Record format (the default)
- Memory Initialization File format
  in Synopsis (**-Ms**) and Xilinx (**-Mx**) variants

- Extended Tektronix Hex format (see option **-t**)
- Intel hex format (see option **-I**)
- Binary (see option **-B**)
- Mentor QUICKSIM Modelfile format (see option **-q**)
- Verilog `$readmemh` format (see option **-V**)

For a discussion of the device model assumed by the converter, see PROM-Device Model on page 110.

For a discussion of the supported formats, see Hex-File Formats on page 113.

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

Table 27 elf2hex Command-Line Arguments

Argument	Definition
option	Conversion options, separated by spaces. See elf2hex Option Reference on page 98 for a 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 the default option, **-m**. If you specify an option that does not exist, **elf2hex** writes an error message to `stderr`.

The dash before each option is not required.

elf2hex Invocation Examples

This section presents **elf2hex** invocation examples.

The command shown in Example 37 Converting an ELF file to Motorola S3 Format 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.

Example 37 Converting an ELF file to Motorola S3 Format

```
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 1 Example of File Partitioning for ROM on page 97.

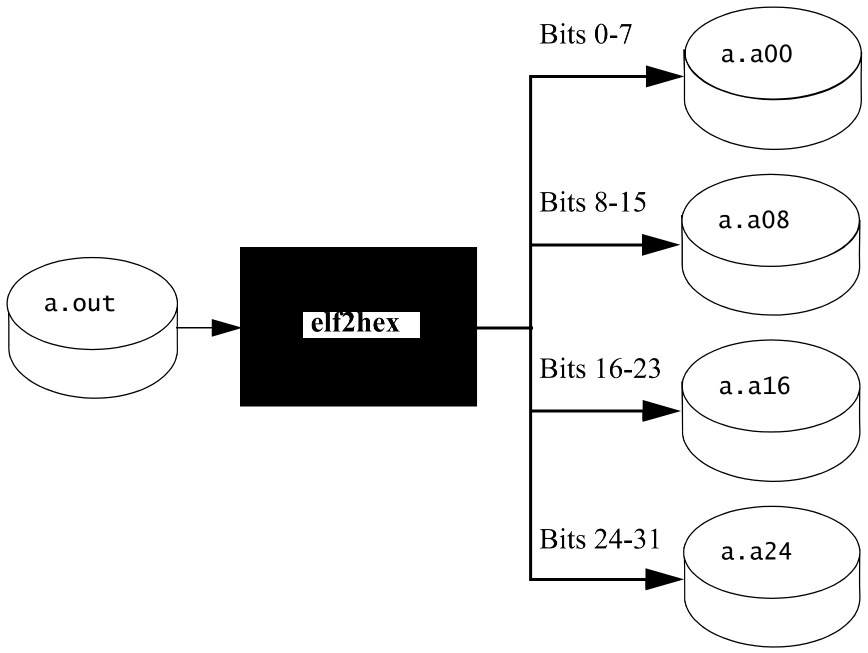

Figure 1 Example of File Partitioning for ROM

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 Hex Output-File Naming Conventions on page 112 for a discussion of hex-file naming conventions.

The command shown in Example 37 Converting an ELF file to Motorola S3 Format 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.

Example 38 Converting an ELF file to Extended Tektronix Hex Format

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

If the input file is relocatable, **elf2hex** issues a warning message but continues with the translation, ignoring the relocation information. In this example, the output file appears as follows:

```
%4A6CE44000250101105E40017E158101180340837002008300030082400301792172450101
%4A6C6440200340839002008300030082410301792172450101030179047245010115836000
%4A6414404001FF82FF160061601560600461616100A4FF61FD15828201A800801970400101
%4A65F440600300790172450101A0FF00FE7040010124797E01247F7F798379790225797901
%4A68344080CE0087793E00807FC000007A157E0100030279009279797FCE0081792479817F
%4A691440A0147E7E798379790225797901CE0087793600007FC000007A157F8100257D7D68
%4A63E440C0030076008179760214777 97D030078000300790014797977 1E00787915767601
%32605440E04179760AACFF79F870400101C0000080157D7D68
%0A81A44000
```

Figure 2 Sample Output of Extended Tektronix Hex Format

You can partition the translated object file into a set of files, each suitable for programming one of the read-only memories in a multi-PROM system. For details about hex file formats, see Hex-File Formats on page 113.

The command shown in Example 39 Converting Text and Lit Sections to Binary Format converts single, contiguous text and lit sections into binary format for loading by a debugger or flash programmer:

Example 39 Converting Text and Lit Sections to Binary Format

```
elf2hex -B -ctl -o file.bin  file.elf
```

The command shown in Example 40 Converting Text Section Only to Binary Format on page 98 converts only the text (code) section into binary format. Use **-ctld** if data has not been copied into a lit section by **-Bcopydata** or the linker command file:

Example 40 Converting Text Section Only to Binary Format

```
elf2hex -B -ct -o text.bin file.elf
```

The command shown in Example 41 Converting Lit Section Only to Binary Format converts only the lit (code) section into raw binary format:

Example 41 Converting Lit Section Only to Binary Format

```
elf2hex -B -cl -o rodata.bin file.elf
```

elf2hex Option Reference

This section lists the command-line options for **elf2hex**.

NOTE Options that specify an output format (**-B**, **-I**, **-m**, **-Ms**, **-Mx**, **-q**, **-t**, **-V**) are mutually exclusive.

-B — Specify raw binary output

Option**-B** generates an output file in binary format (essentially raw binary without tables). You can use a binary file containing the text and lit sections (or text, data, and lit sections) as input to a flash loader or debugger loading mechanism. (To specify sections to be output, see **elf2hex** option **-c**.)

If you intend to produce binary output, be sure the loadable segments of the ELF file are not addressed too far apart (you can modify the addresses in your linker command file). The **elf2hex** utility fills gaps between loadable segments with zeros (or the data you specify using option **-f**). The default limit for fill space is 1 M, which you can change using option **-l**. You can also extract the loadable segments separately using two **elf2hex** invocations, for example if one segment is text and another data.

Default = `-f 0 -l 1M`

NOTE **elf2hex** rejects load segments that are not in load order. If this happens, edit your linker command file to reorder the segments. (You can obtain a listing of segments in load order using the **elfdump** utility with option **-h**.)

-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:

Table 28 Section Types to Be Converted

Type	Definition
b	bss sections
d	data sections

Table 28 Section Types to Be Converted

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.

-f *fill* — Specify fill value for padding between segments

With option **-B** or **-Mx**, you can use additional option **-f** *fill* to specify a 32-bit value to use as fill when padding between segments is required.

Default = `0` (zero)

Use option **-l** to specify the maximum gap to fill.

-h — Display synopsis of elf2hex options

Displays a synopsis of options for **elf2hex**.

-I — Specify a hex file in Intel format

Directs the linker to generate an output file in Intel hex format. See Intel-Hex-Record Format on page 116 for more information.

-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 = 1

-l *limit* — Specify padding limit

With option **-B** or **-Mx**, you can use additional option **-l** *limit* to specify the maximum gap to be filled with fill data. Argument *limit* is a decimal or hex number optionally followed by *`K`* or *`M`* to indicate kilobytes or megabytes.

To disable fill (padding) by **elf2hex**, use this option with a limit of 0 (zero).

To specify the value used in the fill data, use option **-f**.

Default = *`1M`*

-m — Specify a hex file in Motorola S3 Record format (default option)

Directs the linker to generate a hex output file in Motorola S3 Record format. See Motorola S3 Record Format on page 113 for more information.

-Ms — Specify Synopsis MIF format

The Synopsis MIF (Memory Image File) format resembles the Mentor QUICKSIM format but with different byte ordering of 32-bit words. For more information about the Mentor QUICKSIM format, see Mentor QUICKSIM Modelfile Format on page 120.

-Mx — Specify Xilinx MIF format

The Xilinx MIF (Memory Initialization File) format is essentially the same as binary format, with the binary data represented as ASCII zeros and ones (and newlines).

For fill (padding), see also options **-f** and **-l**.

-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, and 32.

Default = 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 Hex Output-File Naming Conventions on page 112 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 Mentor QUICKSIM Modelfile Format on page 120 for more information.

-s *size* — Identify size of memory device

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

Table 29 Memory Device Size Parameters

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

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

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

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

Directs the linker to generate a hex output file in Extended Tektronix Hex Record format. See Extended Tektronix Hex-Record Format on page 115 for more information.

-V — Specify Verilog $readmemh format

Directs the linker to generate an output file in the format read by the Verilog **$readmemh** system call.

Using the File-Size Utility

This section describes the **size** file-size utility, which displays the size in bytes of one or more object files, executables, or archive libraries (including **strip**ped files). The file-size utility is actually named **size***target*, where the suffix *target* identifies the target platform. In this section, we use **size** to generically represent the file-size utility command. See Table 30 Target Suffixes and File-Size Utility Names for the exact file-size utility name for your target.

Table 30 Target Suffixes and File-Size Utility Names

Processor Family	Target Suffix	Utility Name and Command
ARCtangent-A4	arc	**sizearc**
ARCtangent-A5 and later	ac	**sizeac**

Invoking the File-Size Utility

You invoke **size** as follows:

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

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

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

size prints out the following information for each *filename*:

- The number of bytes occupied by text, read-only data, data, and BSS
- The total number of bytes occupied by text, read-only data, and BSS combined

If you specify an archive library, **size** outputs the preceding information for every object file in the archive. It then totals the bytes occupied by text, data, read-only data, and BSS, and the bytes in all the object files.

NOTE For executables, **size** uses information from the program headers, where text and read-only data are often combined.

Unless you specify otherwise (see Command-Line Options for size on page 102), sizes are displayed in decimal.

Command-Line Options for size

Table 31 size Command-Line Options lists the command-line options available for **size**.

Table 31 size Command-Line Options

Option	Meaning
-c	Display size of common data **NOTE** If you use option **-c** on multiple object files at the same time, it counts the size of a given common symbol only once and assigns that value to the first object file containing the symbol.
-d	Display sizes in decimal (the default)
-f	Display size of each allocable section by name
-F	Display size of each loadable segment (`PT_LOAD` program header)
-g	Display sizes in GNU format (read-only data as text)
-o	Display sizes in octal
-q	Do not display the copyright message
-r	Treat read-only data as writable, pre-initialized data
-u	Display sizes in UNIX/Linux format (read-only data as text)
-x	Display sizes in hexadecimal

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. When used with an archive library, **elfdump** displays the contents of each member file.

The **elfdump** utility is actually named **elfdump***target*, where the suffix *target* identifies the target platform. In this section, we use **elfdump** to generically represent the **elfdump** utility command. See Table 32 Target Suffixes and elfdump Utility Names for the exact **elfdump** utility name for your target.

Table 32 Target Suffixes and elfdump Utility Names

Processor Family	Target Suffix	Utility Name and Command
ARCtangent-A4	arc	**elfdumparc**
ARCtangent-A5 and later	ac	**elfdumpac**

Object files compiled with MetaWare 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 3 Structure Dumps of ELF Object and Executable Files shows some typical **elfdump** outputs.

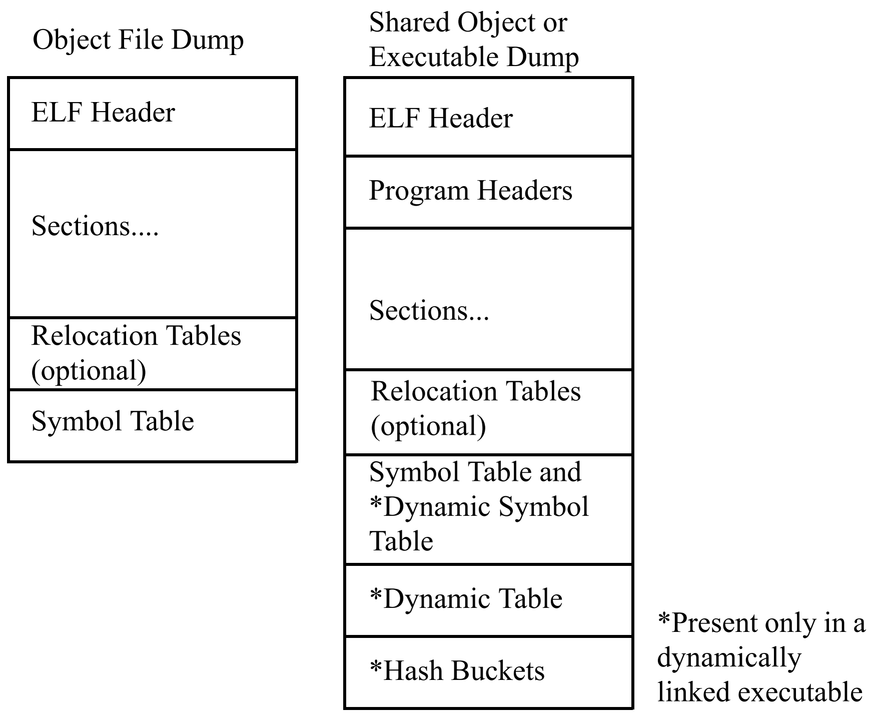
Object File Dump
ELF Header
Sections....
Relocation Tables (optional)
Symbol Table
Shared Object or Executable Dump
ELF Header
Program Headers
Sections...
Relocation Tables (optional)
Symbol Table and *Dynamic Symbol Table
*Dynamic Table
*Hash Buckets
*Present only in a dynamically linked executable

Figure 3 Structure Dumps of ELF Object and Executable Files

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

Invoking elfdump

You invoke **elfdump** as follows:

elfdump [*options*] *filename* [*filename* ...]

These are the command-line arguments:

Table 33 elfdump 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 34 elfdump command-Line Options for a list of options.

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 34 elfdump command-Line Options lists the command-line options available for **elfdump**.

Table 34 elfdump command-Line Options

Option	Information Dumped
-d	Section content (without disassembly)
-D	Dynamic table
-g	DWARF 2 stack-frame and line-table information
-G	All DWARF 2 debug information
-h	ELF header
-H	Hash table contents
-n	Contents of `.note` sections
-p	Program header
-r	Relocation tables
-s	Section headers
-t	Symbol table or tables
-T	Disassembled text sections
-v	DLL version information (if any)
-z	Section contents; bytes of any section for which a disassembler exists are disassembled

Default Options for elfdump

The default option setting is **-DhHprstv**. 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.

Using the Symbol-List Utility

The object-file symbol-list utility, **nm**, displays the symbol-table contents of object files and archives. The symbol-list utility is actually named **nm***target*, where the suffix *target* identifies the target platform. In this section, we use **nm** to generically represent the symbol-list utility command. See

Table 35 Target Suffixes and Symbol-List Utility Names for the exact symbol-list utility name for your target.

Table 35 Target Suffixes and Symbol-List Utility Names

Processor Family	Target Suffix	Utility Name and Command
ARCtangent-A4	Arc	**nmarc**
ARCtangent-A5 and later	ac	**nmac**

Invoking the Symbol-List Utility

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

nm [*options*] *object_file*[, *object_file* ...]

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

Table 36 nm Command-Line Arguments

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

Command-Line Options for nm

Table 37 nm Command-Line Options lists the command-line options for **nm**.

Table 37 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.
-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.

Using the Name unmangler

The MetaWare C++ compiler converts the name of each function to a *mangled name* that encodes the function type in the text of the name. This encoding is necessary in C++ to support overloading multiple functions with the same name. The compiler also mangles variable names when generating assembly output. The assembler and linker then communicate the names to the final executable.

Name mangling makes type-safe linkage possible. This means you can safely call a function that is defined in another module, and not worry whether that function is declared with the same parameter types in the other module.

NOTE For information about correctly mangling the names of enumeration types nested in C++ classes, see the listing for toggle `Nested_enum_mangling` in the *MetaWare C/C++ Programmer's Guide*.

Converting Mangled Names

The MetaWare name unmangler, **unmangle**, converts the mangled function and variable names generated by the C++ compiler into a form similar to the C++ declaration syntax. You can redirect linker output or assembler output to **unmangle** to use it as a filter. (The MetaWare debugger automatically unmangles mangled names when debugging.)

unmangle looks for certain sequences in a name to determine whether the name is a mangled name.

NOTE **unmangle** does not try to convert names that do not match the format of a mangled name; it simply echoes such names without modification.

For example, names containing embedded `__of` or `@` strings are treated as mangled names. Other key sequences also exist, and vary from system to system.

CAUTION **unmangle** expects mangled names to be correctly constructed. If the mangled name contains an error, **unmangle** might produce unexpected output. Thus a typographical error on the command line might result in an incorrect unmangled name.

Invoking the Name Unmangler

You invoke the name unmangler as follows:

`unmangle` [*mangled_string* **...**]

If you invoke unmangle with a mangled string (such as **unmangle** *_fnc__FiT1*), it unmangles the string and prints the result to `stdout`. You can supply any number of mangled strings; **unmangle** prints the unmangled form of each string to `stdout`, each on a separate line. You can also redirect a file to `stdin` to unmangle all mangled strings in that file.

If you invoke **unmangle** without feeding it a mangled string, it reads data from **stdin**, parses the data into words (delimited by whitespace — spaces, tabs, or newlines), and attempts to unmangle each word. This technique allows you to use **unmangle** as a filter.

Examples of Name Unmangling

Example 42 Unmangling C++ Function Names shows how function names are mangled and unmangled.

Example 42 Unmangling C++ Function Names

```
//C++ function declarations:
int fnc( int i1, int i2 );
void fnc( short i1, short i2 );

//Mangled function names
_fnc__FiT1
_fnc__FsT1

//unmangle invoked on first mangled name:
unmangle _fnc__FiT1
//output
fnc(int,int)
//unmangle invoked on second mangled name:
unmangle _fnc__FsT1
//output
fnc(short,short)
```

Unmangled Names and Class Membership

Unmangled names are fully qualified with class membership scope. For example, this command:

```
unmangle _j__1XFi
```

writes this to `stdout`:

```
X::j(int)
```

Using the strip Utility

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

The **strip** utility is similar to the UNIX/Linux **strip** command. It accepts relocatable object files and ELF executable files and makes them more compact by stripping out unnecessary information. Using **strip** also makes it more difficult for others to decompile executables.

NOTE The MetaWare debugger requires symbol information. Do not strip the symbol table (option **-s**) if you intend to debug using the MetaWare debugger, or any symbolic debugger.

Invoking strip

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

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

These are the command-line arguments:

Table 38 strip Command-Line Arguments

options	Options that specify the type of file to be stripped. See Table 39 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.

Command-Line Options for strip

Table 39 strip Command-Line Options lists the command-line options for **strip**

NOTE The MetaWare debugger requires symbol information. Do not strip the symbol table (option **-s**) if you intend to debug using the MetaWare debugger, or any symbolic debugger.

Table 39 strip Command-Line Options

Option	Meaning
-c	Strips comment information. This option is useful for reducing the size of libraries.
-d	Strips debugging information but leaves line numbers.
-h	Strips section headers and unallocated sections.
-l	Strips line numbers and debugging information.
-L	Strips locals from the symbol table.
-q	Suppresses copyright message.
-s	Strips the symbol table
-u	Strips unallocated sections.
-x	Strips line numbers and debugging information (same as **-l**).

Default Settings for strip

For ELF executables, the **strip** utility deletes symbol tables, line-number information, and debugging information by default (default options **-ls**).

For object files, the **strip** utility deletes line-number and debugging information and strips the symbol table of all local symbols (default options **-lL**).

Appendix A — PROMs and Hex Files

In This Appendix

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.

For information about using linker option **-x** to generate hex files, see Linker-Options Reference on page 18. For information about the ELF-to-hex conversion utility, **elf2hex**, see Using the Command-Line Profiling Utility on page 92.

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.

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 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 4 A Single Bank of PROM Devices on page 111.

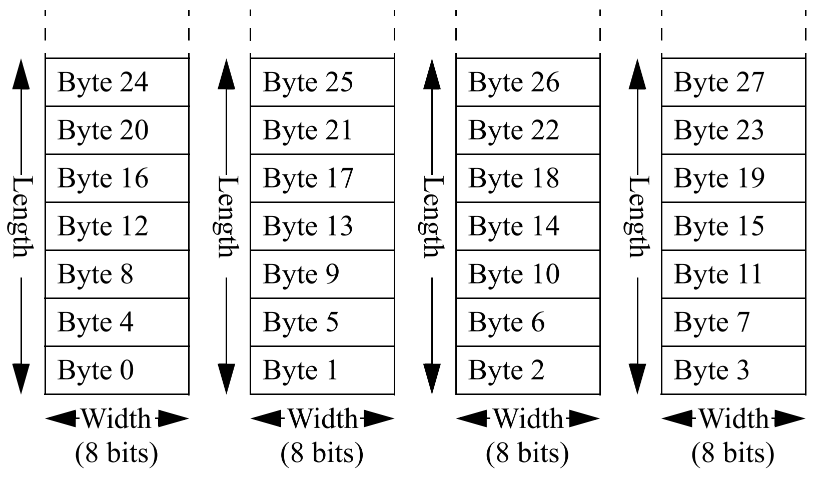

Figure 4 A Single Bank of PROM Devices

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 Hex Output-File Naming Conventions on page 112.

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 5 two Banks of PROM Devices (0,1,2,3 and 4,5,6,7) on page 111

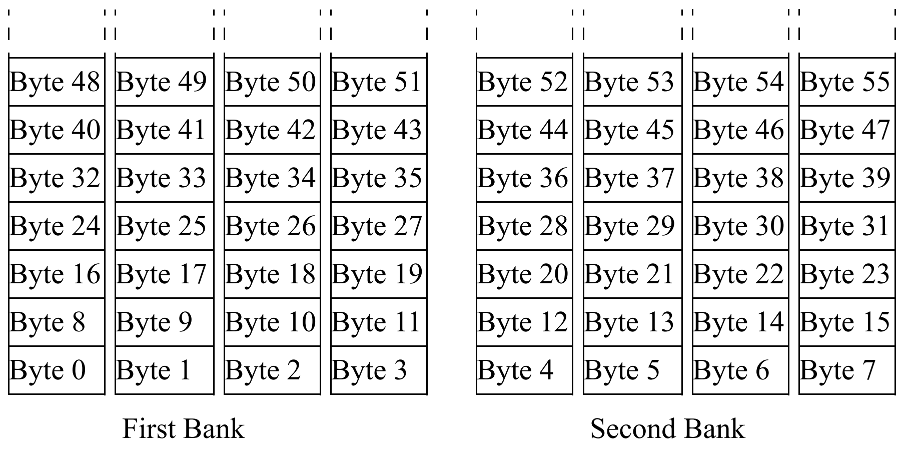

Figure 5 two Banks of PROM Devices (0,1,2,3 and 4,5,6,7)

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.

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 Using the Command-Line Profiling Utility on page 92.) This section describes naming conventions for the hex output file(s).

For a discussion of PROMs and related terminology, see PROM-Device Model on page 110.

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 attribute **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.

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.

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:

Table 40 Suffix Elements of Two Characters

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 **-ts** 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 **-tn** 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).

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:

Table 41 Suffix Elements of Three Characters

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 **-ts** 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 **-tn** 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 file-name conventions.

Hex-File Formats

The linker and **elf2hex** utility support the following formats:

- Motorola S3 Record Format
- Extended Tektronix Hex-Record Format
- Intel-Hex-Record Format
- Binary Output Format (see **elf2hex** option **-B**)
- Mentor QUICKSIM Modelfile Format
- Verilog $readmemh Format

Motorola S3 Record Format

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

Table 42 S-record 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.*

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 6 Data Record in Motorola S3 Record Format illustrates a Motorola S3 data record.

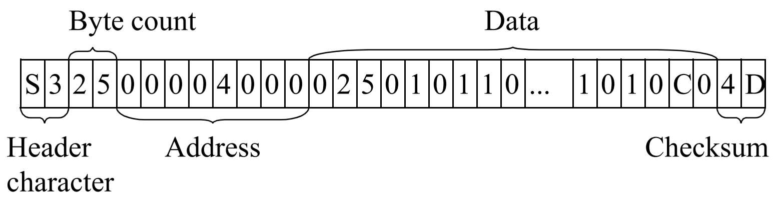
Byte count
Data
S 3 2 5 0 0 0 0 4 0 0 0 0 2 5 0 1 0 1 1 0... 1 0 1 0 C 0 4 D
Header character
Address
Checksum

Figure 6 Data Record in Motorola S3 Record Format

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 43 Start and EOF Characters in Motorola S3 Record Format on page 114 lists the address field length specified by each pair of start characters and the associated end-of-file character(s).

Table 43 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.

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 43 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.

Extended Tektronix Hex-Record Format

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

Table 44 Tektronix Hex Record Format

Format	Definition
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

Data Record

Extended Tektronix Hex Format data records contain the following components:

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

Figure 7 Data record in Extended Tektronix Hex Format illustrates the format of an Extended Tektronix Hex Format data record.

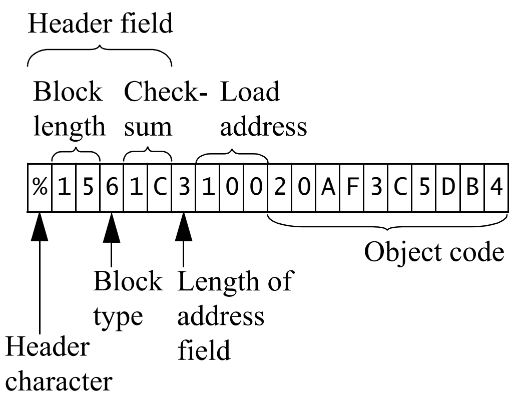

Figure 7 Data record in Extended Tektronix Hex Format

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

Table 45 Header field Components in Extended Tektronix Hex-Format Data Record 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 45 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.

Intel-Hex-Record Format

This section provides a brief overview of the Intel hex record format. For complete information, see the *Intel Hexadecimal Object File Format Specification*, Revision A, 1/6/88, from Intel Corporation.

Format of Intel Hex Records

Intel hex records of all types share a common format. Figure 8 Intel Hex Record illustrates the basic format of an Intel hex record.

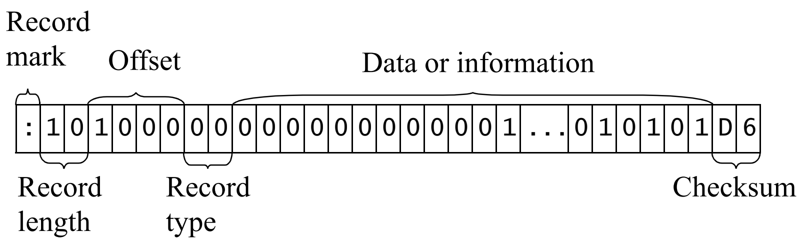

Figure 8 Intel Hex Record

The record-mark field (the first byte of a record) always contains a colon (:), which serves to mark records as Intel hex records.

The record-length field is one byte long and indicates the number of bytes of data or information between the record-type and checksum fields.

The offset field is two bytes long. In data records, it specifies the 16-bit starting load offset used to calculate the location for placement of the data bytes. In records of other types, it always contains 0000.

The record-type field is one byte long and specifies the record type. The following values are possible:

Table 46 Intel-Hex-Record Field Values

00	Data record	03	Start segment address record
01	End-of-file record	04	Extended linear address record
02	Extended segment address record	05	Start linear address record

For an overview of each of the record types, see Summary of Record Types on page 117.

The data or information field is variable in length, and contains information of the type specified in the record-type field. The data or information consists of zero or more bytes encoded as pairs of hexadecimal digits.

The checksum field is one byte in length. The checksum is calculated by taking the two's complement of the sum of all the data bytes in the record, excluding the colon and the checksum itself. The sum of all bytes plus the checksum is 00.

Summary of Record Types

The following are the permissible Intel hex record types.

Type 00: Data Record

Data records contain (in their data field) the hexadecimal digits of the ASCII representation of part of a memory image. They calculate the absolute address to place that data based on the contents of the offset field and depending on whether the file contains an extended linear address record or an extended segment address record.

Type 01: End-of-File Record

End-of-file records are the last record in an Intel hex file. Their length is always zero because they do not contain data or information bytes. The load-offset field is not used and contains 0000. The checksum for end-of-file records is always ASCII FF (hexadecimal 04646H).

Type 02: Extended Segment Address Record

Extended segment address records contain (in their data fields) the segment base address used by data records to find the absolute memory address for data bytes. The length of extended segment address records is 02. The load-offset field in extended segment address records is not used and contains 0000.

Type 03: Start Segment Address Record

Start segment address records specify the execution start address of the associated object file. Their data field contains the 20-bit segment address for the CS and IP registers of 8086 and 80186 processors. The load-offset field in start segment address records is not used and contains 0000.

Type 04: Extended Linear Address Record

The data field of extended linear address records contains the linear base address used by data records to find the absolute memory address for data bytes. The length of extended linear address records is 02. The load-offset field in extended linear address records is not used and contains 0000.

Type 05: Start Linear Address Record

Start linear address records specify the execution start address of the associated object file. The data field contains the 32-bit linear address of the EIP register of the 80386 processor. (Code that starts execution in the 80386 real mode should use a start segment address record instead, because it specifies both the CS and IP register contents required by real mode.) The load-offset field in start linear address records is not used and contains 0000.

Example Intel Hex Records

Example 43 Intel Hex File is an Intel hex file for a small program. The 04 in the record-type field of the first record indicates that it is an extended linear address record. Its data contains the segment base address referenced by the offset field in the subsequent data records to calculate the absolute memory address for content bytes. The 01 in the record-type field of the last record indicates that it is an end-of-file record.

Example 43 Intel Hex File

```
:020000040001F9
:1000000000FE1F6080FE9F67007C3F600000AAAA80
:10001000007C9F600010010080050030020062003B
:1000200001FAE157FFFFFF7F0203002000804040FC
:10003000000400120020600800007E157FFFFFF7F67
:1000400002020020017E004000FA1E6000001F38FE
:10005000164000001FA1E60007C1F60FF010000D6
:1000600000001F12FF01000000201F08FF01000018
:10007000007C1F600002000000001F120002000050
:1000800000201F08000200000180BF1FFFFFFF7F4C
:08009000FFFFFF7FFFFFFF7F70
:1010000000000000000001010101010101010101D6
:1010100001010001010101010101010101010101C1
:1010200001000001010101010101010101010101B2
:1010300000000000000001010000000000010101AB
:10104000000000010101010101010101010101010193
:1010500001010101010101010101010101010101010180
:1010600001010101010101010101010101010101010170
:10107000010101000000000000000000000000000000006D
:00000001FF
```

Example 44 ELF Dump of Intel Hex File is the **elfdump** output of Example 43 Intel Hex File. Note the contents of the `.text` and `.data` sections, and their addresses (0x10000 and 0x11000 respectively).

Example 44 ELF Dump of Intel Hex File

```
ELF header
    ident=<CLASS32,DATA2LSB,EI_VERSION=1>, type=EXEC
    machine=ARC, version=1, entry=0x10000, flags=0
    ehsize=52, phoff=0x34, phentsize=32, phnum=2
    shoff=0x948, shentsize=40, shnum=11, string_section=10
Program header #0
    type=LOAD, vaddr=0x10000, off=0x74
    paddr=0x0, filesz=0x98, memsz=0x98, align=4096, flags=<X,R>
Program header #1
    type=LOAD, vaddr=0x11000, off=0x10c
    paddr=0x0, filesz=0x80, memsz=0x10080, align=4096, flags=<W,R>

Section headers
Section #1: .text, type=PROGBITS, addr=0x10000, off=0x74
       size=152(0x98), link=0, info=0, align=4, entsize=1
       flags=<ALLOC,EXECINSTR>
Section #2: .sdata, type=NOBITS, addr=0x11000, off=0x10c
       size=0(0x0), link=0, info=0, align=1, entsize=1
       flags=<WRITE,ALLOC>
Section #3: .data, type=PROGBITS, addr=0x11000, off=0x10c
       size=128(0x80), link=0, info=0, align=1, entsize=1
       flags=<WRITE,ALLOC>
Section #4: .stack, type=NOBITS, addr=0x11080, off=0x18c
       size=65536(0x10000), link=0, info=0, align=1,
          entsize=1
       flags=<WRITE,ALLOC>
Section #5: .comment, type=PROGBITS, addr=0x0, off=0x18c
       size=32(0x20), link=0, info=0, align=1, entsize=1
       flags=0
Section #6: .arcextmap, type=PROGBITS, addr=0x0,
          off=0x1ac
       size=1562(0x61a), link=0, info=0, align=1,
          entsize=1
       flags=0
Section #7: .arcextmapdummyrelo, type=PROGBITS, addr=0x0,
          off=0x7c6
       size=4(0x4), link=0, info=0, align=1, entsize=1
       flags=0
Section #8: .strtab, type=STRTAB, addr=0x0, off=0x7ca
       size=76(0x4c), link=0, info=0, align=1, entsize=1
       flags=0
Section #9: .symtab, type=SYMTAB, addr=0x0, off=0x818
       size=208(0xd0), link=8, info=10, align=4,
         entsize=16
       flags=0
Section #10: .shstrtab, type=STRTAB, addr=0x0, off=0x8e8
       size=93(0x5d), link=0, info=0, align=1, entsize=1
       flags=0

Symbol table ".symtab"
     Value      Size Bind Type Sect     Name
     -----      ---- ---- ---- ----     ----
  1  0x10000     152 LOC  SECT .text
  2  0x11000     128 LOC  SECT .data
  3  0x11080   65536 LOC  SECT .stack
  4  0x0           0 LOC  FILE ABS      t_aux.s
  5  0x11000       0 LOC  NULL .data    ok_aux
  6  0x10048       0 LOC  NULL .text    done_ok
  7  0x10044       0 LOC  NULL .text    next_loop
  8  0x10054       0 LOC  NULL .text    done_err
  9  0x10058       0 LOC  NULL .text    done
 10  0x10000       0 GLOB NULL .text    _start
 11  0x0           0 GLOB NULL ABS      _HEAPSIZE
 12  0x11000       0 GLOB NULL .sdata   _SDA_BASE_
Content of section .text (#1) size=0x98 start=0x74
_start:
10000: 601ffe00                 mov          %r0,0
10004: 679ffe80                 mov          %lp_count,128
10008: 603f7c00 aaaa0000        mov          %r1,0xaaaa_0000
 ; 0xaaaa0000
10010: 609f7c00 00011000        mov          %r4,0x1_1000                  ;
ok_aux
10018: 30000580                 lp       done_ok
1001c: 00620002                 ldb          %r3,[%r4,%r0]
10020: 57e1fa01                 sub.f        0,%r3,1
10024: 7fffffff                 nop
10028: 20000302                 bnz          next_loop
1002c: 40408000                 add          %r2,%r1,%r0
10030: 12000400                 sr       %r2,[%r0]
10034: 08602000                 lr       %r3,[%r0]
```

```
10038: 57e10700              sub.f          0,%r2,%r3
1003c: 7fffffff              nop
10040: 20000202              bnz            done_err
next_loop:
10044: 40007e01              add            %r0,%r0,1
done_ok:
10048: 601efa00              mov.f          %r0,0
1004c: 381f0000 00004016     j         done                    ; 0x4016
done_err:
10054: 601efa01              mov.f          %r0,1
done:
10058: 601f7c00 000001ff     mov            %r0,0x1ff                    ;
0x1ff = _HEAPSIZE+0x1ff
10060: 121f0000 000001ff     sr        %r0,[%aux1ff]                     ;
0x1ff = _HEAPSIZE+0x1ff
10068: 081f2000 000001ff     lr        %r0,[%aux1ff]                     ;
0x1ff = _HEAPSIZE+0x1ff
10070: 601f7c00 00000200     mov            %r0,0x200                    ;
0x200 = _HEAPSIZE+0x200
10078: 121f0000 00000200     sr        %r0,[%aux200]                     ;
0x200 = _HEAPSIZE+0x200
10080: 081f2000 00000200     lr        %r0,[%aux200]                     ;
0x200 = _HEAPSIZE+0x200
10088: 1fbf8001              flag           1
1008c: 7fffffff              nop
10090: 7fffffff              nop
10094: 7fffffff              nop
Content of section .data (#3) size=0x80 start=0x10c
ok_aux:
 11000:   00 00 00 00 00 00 01 01   01 01 01 01 01 01 01 01
'................'
 11010:   01 01 00 01 01 01 01 01   01 01 01 01 01 01 01 01
'................'
 11020:   01 00 00 01 01 01 01 01   01 01 01 01 01 01 01 01
'................'
 11030:   00 00 00 00 00 00 01 01   00 00 00 00 00 01 01 01
'................'
 11040:   00 00 00 01 01 01 01 01   01 01 01 01 01 01 01 01
'................'
 11050:   01 01 01 01 01 01 01 01   01 01 01 01 01 01 01 01
'................'
 11060:   01 01 01 01 01 01 01 01   01 01 01 01 01 01 01 01
'................'
 11070:   01 01 01 00 00 00 00 00   00 00 00 00 00 00 00 00
'................'
```

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 45 Portion of a QUICKSIM File shows a portion of a QUICKSIM file.

Example 45 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.

Verilog $readmemh Format

The Verilog **$readmemh** format is the format read by the Verilog **$readmemh** system call.

The format consists of three elements:

- White space (including carriage returns)
- Comments (preceded by “//” as in C++)
- Hexadecimal numbers representing data words

Each number is separated by white space. When the system task is executed, each number in the file is assigned sequentially to an address in memory, starting at zero unless otherwise specified. Addresses are word addresses, where the definition of *word* is system-dependent. Addresses can be specified as @*address_in_hexadecimal*. In that case, the next data word is loaded into the specified address, and subsequent data words are loaded into addresses sequential from there. The hexadecimal data may include an underscore as a separator to enhance readability.

Example 46 $readmemh File shows a simple **$readmemh** file. Reading the first line into an 8-bit memory loads 0xef into address 0x00; reading the first line into a 64-bit memory loads 0x0000_0000_dead_beef into address 0x0000_0000_0000_0000.

Example 46 $readmemh File

```
dead_beef    // Loaded into word 0x00000000 by default
abad_cafe    // Loaded into word 0x00000001
@55          // Next word address
5555_aaaa    // Loaded into word 0x00000055
aaaa_5555    // Loaded into word 0x00000056
```

Appendix B — Quick Options List

For a complete listing of linker options, see Linker-Options Reference on page 18.

-b	Do not do any special processing of shared symbols
-Ball_archive	Extract all members of an archive library
-Ballocatecommon	Force allocation of common data
-Bbase=0x*address*[:0x*address*]	Specify the origin address in hexadecimal
-Bcopydata	Create an **INITDATA** entry for all writable data
-Bdefine:*sym*=*expr*	Define a public symbol
-Bdynamic	Search for DLL libname when processing option **-l**
-Belfpflag=*0xval*	Add flags for non-MetaWare debuggers
-Bexe	Build executable during incremental link
-Bforce_{`big_`|`little_`}**endian**	Force endianness of generated tables
-Bgrouplib	Scan archives as a group
-Bhardalign	Force each output segment to align on a page boundary
-Bhardwired=*sym1*,*sym2*,[...]	Bind references to specified symbols within a DLL to symbol definitions within the link
-Bhelp	Display information about **-B** options designed for embedded development
-Blstrip	Strip local symbols from symbol table
-Bmovable	Make dynamic executable file movable
-Bnoallocdyn	Do not map dynamic tables in virtual memory
-Bnocopies	Do not make local copies of shared variables; insert relocation fix-ups
-Bnodemandload	Ignore boundary issues when mapping sections
-Bnoheader	Do not include ELF header in loadable segments
-Bnoplt	Do not implicitly map symbols into the PLT of the executable file
-Bnozerobss	Do not zero bss sections at run time
-Boverlay_num	Specify overlay package number
-Bpagesize=*size*	Specify page size of target processor in bytes
-Bpictable[=[`text`|`data`][,*name*]]	Generate run time fix-up table
-Bpurgedynsym	Export only those symbols referenced by DLLs being linked against
-BpurgeNEEDED	Include only explicitly referenced DLLs in the “needed” list of a generated module
-Brogot	Place the `.got` section in a read-only section
-Brogotplt	Place the `.got` and `.plt` sections in read-only sections
-Broplt	Place the `.plt` section in a read-only section
-Bstart_addr=*0xaddress*[:*0xaddress*]	Specify origin address in hexadecimal

-Bstatic	Search for static library libname when processing **-l** *name*
-Bsymbolic	Bind intra-module global symbol references to symbol definitions within the link
-Bsymin	Read symbol definitions file
-Bsymout	Generate a list of public symbol definitions
-Bzerobss	Zero BSS sections at run time instead of load time
-C *listing_type*	Display listing of specified type
-d {*y*|*Y*|*n*}	Generate a dynamically or statically linked executable
-e *entry_name*	Specify program entry point
-G	Generate a DLL
-h *name*	Use name to refer to the generated DLL
-H	Display linker command-line syntax help screen
-I *name*	Write name into the program header as the path name of the dynamic loader
-J *file*	Export only the symbols listed in file when generating a DLL
-l *name*	Search for library whose name contains name
-L *paths*	Specify search path to resolve **-l** *name* specifications
-m	Write a memory-map listing file to standard output
-M *cmd_file*	Process an SVR4-style command file (input map file)
-o *out_file*	Specify name of the output file
-q	Do not display copyright message
-Q {*y*|*n*}	Specify whether version information appears in output file
-r	Generate relocatable object file for incremental linking
-R *pathlist*	Define the search path used to resolve references to DLLs at run time
-s	Strip symbols and debugging information from the output file's symbol table
-t	Suppress warnings about multiply defined common symbols of unequal sizes
-u *ext_name*	Create undefined symbol
-w	Suppress all warning messages
-x[*attribute*]	Generate a hex file
-Xcompress_stats	Display compression statistics for each section
-Xnocompress	Suppress compression of `.initdat` data
-YP,*path*	Specify default search path to resolve subsequent **-l** *name* specifications
-zdefs	Do not allow undefined symbols; force fatal error
-zdup	Permit duplicate symbols
-zlistunref	Diagnose unreferenced files and input sections
-znodefs	Allow undefined symbols
-znodefswarn	Allow undefined symbols but issue warning
-zoffwarn=*n1*, *n2*, [...]	Suppress numbered warnings

-zpurge	Omit unreferenced input sections
-zpurgetext	Omit unreferenced executable input sections
-zsize	Display segment sizes to `stdout`
-zstdout	Write errors to standard output instead of standard error
-ztext	Do not allow output relocations against read-only sections

Index