XGC Users Guide

Using the Assembler Linker and Utilities

www.xgc.com

XGC Users Guide

Using the Assembler Linker and Utilities

Order Number: XGC-ALU-110101

XGC Technology

London
UK
Web: `<www.xgc.com>`

XGC Users Guide: Using the Assembler Linker and Utilities

by Free Software Foundation and XGC Technology

Publication date January 2011

Acknowledgments

This guide is based on documentation distributed by the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

License

Contents

About this Guide

This guide contains detailed information about the assembler, linker and object code utilities included with all XGC compilation systems. The command line examples given show the native form of the commands. For cross compilation systems, each command must be prefixed with the prefix name given in the appropriate product-specific documentation.

This guide does not contain details of the target computer instruction set or any additional assembler directives. These are documented in the relevant volumes of the target computer vendor's documentation. For additional command line options see the Getting Started guide that ships with the compilation system.

1. Audience

This guide is written for the experienced programmer who is already familiar with high-order programming languages and with embedded systems programming in general. Users of the assembler will require a detailed understanding of the target computer's instruction set.

2. Related Documents

Getting Started a product-specific guide.

The Compiler User Guide.

XGC User Guide Part V, which describes the simulator and debugger.

XGC Libraries, which documents the library functions available with all XGC compilers.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC user manuals.

You can send your comments in the following ways:

- Internet electronic mail: readers_comments@xgc.com

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

Technical support enquiries should be directed to the XGC web site [http://www.xgc.com/] or by email to support@xgc.com.

4. Documentation Conventions

This guide uses the following typographic conventions:

`%, $`
A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bash shell.

`#`
: A number sign represents the superuser prompt.

`$`**`vi hello.c`**
: Boldface type in interactive examples indicates typed user input.

`file`
: Italic or slanted type indicates variable values, place-holders, and function argument names.

[|], { | }
: In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...
: In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated.

cat(1)
: A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Mb/s
: This symbol indicates megabits per second.

MB/s
: This symbol indicates megabytes per second.

Ctrl+**x**
: This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows. In examples, this key combination is printed in bold type (for example, **Ctrl**+**C**).

I

Using the Macro Assembler

Chapter 1

Summary

Here is a brief summary of how to invoke the assembler. For details, see Chapter 3, *Assembler Command-Line Options* [11] .

as

prefix-as

Synopsis

`prefix-as` [[-[adhlns]]] [[=file]] [[-D]] [[--defsym `sym=val`]] [[-f]] [[--help]] [[-I `dir`]] [[-J]] [[-K]] [--size-sort] [[-L]] [[--version]] [[-W]] [[-w]] [[-x]] [[-Z]] [[[--] | [`files` ...]]]

Options

-a[dhlns]
Turn on listings, in any of a variety of ways:

-ad
omit debugging directives

-ah
: include high-level source

-al
: include assembly

-an
: omit forms processing

-as
: include symbols

`=file`
: set the name of the listing file

You may combine these options; for example, use **-aln** for assembly listing without forms processing. The “ `=file` ” option, if used, must be the last one. By itself, **-a** defaults to **-ahls** that is, all listings turned on.

-D
: Ignored. This option is accepted for script compatibility with calls to other assemblers.

--defsym *`sym`*=*`value`*
: Define the symbol *`sym`* to be *`value`* before assembling the input file. *`value`* must be an integer constant. As in C, a leading “ `0x` ” indicates a hexadecimal value, and a leading “ `0` ” indicates an octal value.

-f
: "fast"--skip white-space and comment preprocessing (assume source is compiler output).

--help
: Print a summary of the command line options and exit.

-I *`dir`*
: Add directory *`dir`* to the search list for **.include** directives.

-J
: Don't warn about signed overflow.

-K
: Issue warnings when difference tables altered for long displacements.

-L
: Keep (in the symbol table) local symbols, starting with “ `L` ” .

.o *`objfile`*
: Name the object-file output from the assembler *`objfile`*.

-R
: Fold the data section into the text section.

--statistics
: Print the maximum space (in bytes) and total time (in seconds) used by assembly.

-v , **-version**
: Print the `as` version.

--version
: Print the `as` version and exit.

-W
: Suppress warning messages.

-w
: Ignored.

-x
: Ignored.

-Z
: Generate an object file even after errors.

`-- |` *`files`* `...`
: Standard input, or source files to assemble.

Chapter 2 *Overview*

The GNU assembler is really a family of assemblers. If you have used the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

The assembler is primarily intended to assemble the output of the compiler for use by the linker. Nevertheless, we've tried to make the assembler assemble correctly everything that other assemblers for the same machine would assemble.

Unlike older assemblers, the assembler is designed to assemble a source program in one pass of the source file. This has a subtle impact on the **.org** directive (see .org [61]).

2.1. Object File Formats

The assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See Section 6.5, “Symbol Attributes” [33] .

2.2. Command Line

After the program name `prefix-as` ,the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

-- (two hyphens) by itself names the standard input file explicitly, as one of the files for the assembler to assemble.

Except for -- any command line argument that begins with a hyphen (-) is an option. Each option changes the behavior of the assembler. No option changes the way another option works. An option is a - followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
prefix-as -o my-object-file.o mumble.s
prefix-as -omy-object-file.o mumble.s
```

2.3. Input Files

We use the phrase *source program*, abbreviated *source*, to describe the program input to one run of the assembler. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run the assembler it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give the assembler a command line that has zero or more input file names. The input files are read (from left file name to right).

A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give the assembler no file names it attempts to read one input file from the the assembler standard input, which is normally your terminal. You may have to type **Ctrl-D** to tell the assembler there is no more program to assemble.

Use **--** if you need to explicitly name the standard input file in your command line.

If the source is empty, the assembler produces a small, empty object file.

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a “logical” file. See Section 2.5, “Error and Warning Messages” [10] .

Physical files are those files named in the command line given to the assembler .

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when the assembler source is itself synthesized from other files. See .app-file [41] .

2.4. Output (Object) File

Every time you run the assembler it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`. You can give it another name by using the **-o** option. Conventionally, object file names end with **.o**. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the `a.out` format.)

The object file is meant for input to the linker .It contains assembled program code, information to help the linker integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

2.5. Error and Warning Messages

The assembler may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs the assembler automatically. Warnings report an assumption made so that the assembler could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where NNN is a line number). If a logical file name has been given (see **.app-file**: .app-file [41]) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see .line [55])then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand UNIX tradition).

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

Chapter 3

Assembler Command-Line Options

If you are invoking the assembler via the compiler, you can use the **-Wa** option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the **-Wa**) by commas. For example:

```
$ prefix-gcc -c -g -O -Wa,-alhd,-L file.c
```

emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this **-Wa** mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the **-v** option to see precisely what options it passes to each compilation pass, including the assembler.)

Enable Listings: **-a[cdhlns]**

These options enable listing output from the assembler. By itself, **-a** requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: **-ah** requests a high-level language listing, **-al** requests an output-program assembly listing, and **-as** requests a symbol table listing. High-level listings require that a compiler debugging option like **-g** be used, and that assembly listings (**-al**) be requested also.

Use the **-ac** option to omit false conditionals from a listing. Any lines that are not assembled because of a false **.if** (or **.ifdef**, or any other conditional), or a true **.if** followed by an **.else**, will be omitted from the listing.

Use the **-ad** option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives **.list**, **.nolist**, **.psize**, **.eject**, **.term**, and **.sbttl**. The **-an** option turns off all forms processing. If you do not request listing output with one of the **-a** options, the listing-control directives have no effect.

The letters after **-a** may be combined into one option, *e.g.*, **-aln**.

-D

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with the assembler.

Work Faster: **-f**

-f should only be used when assembling programs written by a (trusted) compiler. **-f** stops the assembler from doing white-space and comment preprocessing on the input file(s) before assembling them. See Section 4.1, “Preprocessing” [17]
.

Warning. If you use **-f** when the files actually need to be pre-processed (if they contain comments, for example), the assembler does not work correctly.

.include search path: **-I** *`path`*

Use this option to add a *`path`* to the list of directories the assembler searches for files specified in **.include** directives (see .include [52]). You may use **-I** as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, the assembler searches any **-I** directories in the same order as they were specified (left to right) on the command line.

Difference Tables: **-K**

The assembler sometimes alters the code emitted for directives of the form “ `.word` *`sym1-sym2`* ” ; see **.word**. You can use the **-K** option if you want a warning issued when this is done.

Include Local Labels: **-L**

Labels beginning with “ `L` ” (upper case only) are called *local labels*. See Section 6.3, “Symbol Names” [32] .Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both the assembler and the linker discard such labels, so you do not normally debug with them.

This option tells the assembler to retain those “ `L...` ” symbols in the object file. Usually if you do this you also tell the linker to preserve symbols whose names begin with “ `L` ” .

By default, a local label is any label beginning with “ `L` ” ,but each target is allowed to redefine the local label prefix.

Name the Object File: **-o**

There is always one object file output when you run the assembler. By default it has the name `a.out`. You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, the assembler overwrites any existing file of the same name.

Join Data and Text Sections: **-R**

> **-R** tells the assembler to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See Chapter 5, *Sections and Relocation* [25] .)
>
> When you specify **-R** it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of the assembler. In future, **-R** may work this way.
>
> When the assembler is configured for COFF output, this option is only useful if you use sections named “ **.text** ” and “ **.data** ” .

Display Assembly Statistics: **--statistics**

> Use **--statistics** to display two statistics about the resources used by the assembler :the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in `cpu` seconds).

Announce Version: **-v**

> You can find out what version of the assembler is running by including the option **-v** (which you can also spell as **--version**) on the command line.

Suppress Warnings: **-W**

> The assembler should never give a warning or error message when assembling compiler output. But programs written by people often cause the assembler to give a warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how the assembler assembles your file. Errors, which stop the assembly, are still reported.

Generate Object File in Spite of Errors: **-Z**

After an error message, the assembler normally produces no output. If for some reason you are interested in object file output even after the assembler gives an error message on your program, use the **-Z** option. If there are any errors, the assembler continues anyway, and writes an object file after a final warning message of the form:

```
n errors, m warnings, generating bad object file.
```

Chapter 4

Assembler Syntax

This chapter describes the machine-independent syntax allowed in a source file. The assembler syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler.

4.1. Preprocessing

The assembler internal preprocessor:

- adjusts and removes extra white-space. It leaves one space or tab before the keywords on a line, and turns any other white-space on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the **.include** directive (see .include [52]). You can use the GNU C compiler driver to get other “CPP” style preprocessing, by giving the input file a “ **.S** ” suffix.

Excess white-space, comments, and character constants cannot be used in the portions of the input text that are not pre-processed.

If the first line of an input file is `#NO_APP` or if you use the **-f** option, white-space and comments are not removed from the input file. Within an input file, you can ask for white-space and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain white-space or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and white-space.

4.2. White-space

White-space is one or more blanks or tabs, in any order. White-space is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see Section 4.6.1, “Character Constants” [21]), any white-space means the same as exactly one space.

4.3. Comments

There are two ways of rendering comments to the assembler .In both cases the comment is equivalent to one space.

Anything from “ `/*` ” through the next “ `*/` ” is a comment. This means you may not nest these comments.

```
/*
                        The only way to include a newline ('\n') in a comment
                        is to use this sort of comment.
                        */

                        /* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline is considered a comment and is ignored. The line comment character is see Chapter 2, *Overview* [7] .

On some machines there are two different line comment characters. One character only begins a comment if it is the first non-white-space character on a line, while the other always begins a comment.

To be compatible with past assemblers, lines that begin with “ # ” have a special interpretation. Following the “ # ” should be an absolute expression (see Chapter 7, *Assembler Expressions* [35] .): the logical line number of the *next* line. Then a string (see Section 4.6.1.1, “Strings” [21] .) is allowed: if present it is a new logical file name. The rest of the line, if any, should be white-space.

If the first non-white-space characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
  # This is an ordinary comment.
# 42-6 "new_file_name"    # New logical file name
# This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of the assembler.

4.4. Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters “ _.$ ” .On most machines, you can also use $ in symbol names; exceptions are noted in Chapter 2, *Overview* [7] .No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See Chapter 6, *Assembler Symbols* [31] .

4.5. Statements

A *statement* ends at a newline character (“ \n ”)or line separator character. (The line separator is usually “ ; ” , unless this conflicts with the comment character; see Chapter 2, *Overview* [7] ..) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When the assembler reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include white-space. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol that determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot "`.`" then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction. Different versions of the assembler for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (`:`). White-space before a label or after a colon is permitted, but you may not have white-space between a label's symbol and its colon. See Section 6.1, "Labels" [31] .

```
label:     .directive    followed by something
                        another_label:          # This is an empty statement
                        instruction   operand_1, operand_2, ...
```

4.6. Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
                        .ascii "Ring the bell\7"                 # A string
                        .octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
                        .float 0f-314159265358979323846264338327\
                95028841971.693993751E-40             # - pi, a flonum
```

4.6.1. Character Constants

There are two kinds of character constants. A *character* stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

4.6.1.1. Strings

A *string* is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to *escape* these characters: precede them with a backslash “ \ ” character. For example “ \\ ” represents one backslash: the first \ is an escape that tells the assembler to interpret the second character literally as a backslash (which prevents the assembler from recognizing the second \ as an escape character). The complete list of escapes follows.

`\b`

Mnemonic for backspace; for ASCII this is octal code 010.

`\f`

Mnemonic for FormFeed; for ASCII this is octal code 014.

`\n`

Mnemonic for newline; for ASCII this is octal code 012.

`\r`

Mnemonic for carriage-Return; for ASCII this is octal code 015.

`\t`

Mnemonic for horizontal Tab; for ASCII this is octal code 011.

`.` *`digit digit digit`*

An octal character code. The numeric code is 3 octal digits. For compatibility with other UNIX systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.

`\x` *`hex-digits...`*

A hex character code. All trailing hex digits are combined. Either upper or lower case `x` works.

`\\`

Represents one “ `\` ” character.

`\"`

Represents one “ `"` ” character. Needed in strings to represent this character, because an un-escaped “ `"` ” would end the string.

`.` *`anything-else`*

Any other character when escaped by `\` gives a warning, but assembles as if the “ `\` ” was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However, the assembler has no other interpretation, so the assembler knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

4.6.1.2. Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write `'\\` where the first `\` escapes the second `\`. As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. the assembler assumes your character code is ASCII: `'A` means 65, `'B` means 66, and so on.

4.6.2. Number Constants

The assembler distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an `int` in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers, described below.

4.6.2.1. Integers

A binary integer is “ `0b` ” or “ `0B` ” followed by zero or more of the binary digits “ `01` ” .

An octal integer is “ `0` ” followed by zero or more of the octal digits (“ `01234567` ”).

A decimal integer starts with a non-zero digit followed by zero or more decimal digits (“ `0123456789` ”).

A hexadecimal integer is “ `0x` ” or “ `0X` ” followed by one or more hexadecimal digits chosen from “ `0123456789abcdefABCDEF` ” .

To denote a negative integer, use the prefix operator - discussed under expressions (see Section 7.2.3, “Prefix Operator” [36]).

4.6.2.2. Bignums

A *bignum* has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

4.6.2.3. Flonums

A *flonum* represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by the assembler to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of the assembler specialized to that computer.

A flonum is written by writing (in order)

- The digit “ `0` ” .
- A letter, to tell the assembler the rest of the number is a flonum. `e` is recommended. Case is not important.
- An optional sign: either “ `+` ” or `-`.
- An optional *integer part*: zero or more decimal digits.
- An optional *fractional part*: “ `.` ” followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An “ `E` ” or “ `e` ” .
  - Optional sign: either “ `+` ” or `-`.
  - One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

The assembler does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running the assembler .

Chapter 5 *Sections and Relocation*

5.1. Background

Roughly, a section is a range of addresses, with no gaps; all data between those addresses is treated the same for some particular purpose. For example, there may be a *read only* section.

The linker reads many object files (partial programs) and combines their contents to form a runnable program. When the assembler emits an object file, the partial program is assumed to start at address `0x00000000`. The linker assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how the assembler uses sections.

The linker moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by the assembler has at least three sections, any of which may be empty. These are named *text*, *data* and *bss* sections.

When it generates COFF output, the assembler can also generate whatever other named sections you specify using the " **.section** " directive (see .section [65]). If you do not use any directives that place output in the " **.text** " or " **.data** " sections, these sections still exist, but are empty.

Within the object file, the text section starts at address `0x00000000`, the data section follows, and the bss section follows the data section.

To let the linker know which data changes when the sections are relocated, and how to change that data, the assembler also writes to the object file details of the relocation needed. To perform relocation the linker must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of

```
                                (address)  (start-address of section)?
```

- Is the reference to an address "Program-Counter relative"?

In fact, every address the assembler ever uses is expressed as

```
                        (section) + (offset into section)
```

Further, most expressions the assembler computes have this section-relative nature.

In this manual we use the notation {*secname* *N*} to mean "offset *N* into section *secname*."

Apart from text, data and bss sections you need to know about the *absolute* section. When the linker mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is relocated to run-time address 0 by the linker .Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

The idea of sections is extended to the *undefined* section. Any address whose section is unknown at assembly time is by definition rendered {undefined *`U`*} where *`U`* is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy, the word *section* is used to describe groups of sections in the linked program. The linker puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by the linker ;others are invented for use of the assembler and have no meaning except during assembly.

5.2. Linker Sections

The linker deals with just four kinds of sections, summarized below.

named sections

> These sections hold your program. the assembler and the linker treat them as separate but equal sections. Anything you can say of one section is true another.

bss section

> This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common

storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always relocated to runtime address 0. This is useful if you want to refer to an address that the linker must not change when relocating. In this sense we speak of absolute addresses being *un-relocatable*: they do not change during relocation.

undefined section

This section is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The example uses the traditional section names “ **.text** ” and “ **.data** ” .Memory addresses are on the horizontal axis.

5.3. Sub-Sections

You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. The assembler allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a “ **.text 0** ” before each section of code being output, and a “ **.text 1** ” before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes.

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; the linker and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a " `.text` *`expression`* " or a " **.data** *`expression`* " statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: " **.section** *`name`*, *`expression`* " . *`Expression`* should be an absolute expression. (See Chapter 7, *Assembler Expressions* [35] .) If you just say " **.text** " then " `.text 0` " is assumed. Likewise " **.data** " means " `.data 0` " .Assembly begins in `text 0`. For instance:

```
.text 0     # The default subsection is text 0 anyway.
                        .ascii "This lives in the first text subsection. *"
                        .text 1
                        .ascii "But this lives in the second text subsection."
                        .data 0
                        .ascii "This lives in the data section,"
                        .ascii "in the first data subsection."
                        .text 0
                        .ascii "This lives in the first text section,"
                        .ascii "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to the assembler there is no concept of a subsection location counter. There is no way to directly manipulate a location counter but the **.align** directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

5.4. bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

The **.lcomm** directive defines a symbol in the bss section; see .lcomm [55] .

The **.comm** directive may be used to declare a common symbol, which is another form of uninitialized symbol; see .comm [43] .

When assembling for a target that supports multiple sections, such as ELF or COFF, you may switch into the **.bss** section and define symbols as usual; see .section [65] . You may only assemble zero values into the section. Typically the section will only contain symbol definitions and **.skip** directives (see .skip [68]).

Chapter 6 Assembler Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

6.1. Labels

A *label* is written as a symbol immediately followed by a colon “ : ” . The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

6.2. Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign “ = ” , followed by an expression (see Chapter 7, *Assembler Expressions* [35]). This is equivalent to using the **.set** directive. See .set [67] .

6.3. Symbol Names

Symbol names begin with a letter or with one of “ **._** ” .On most machines, you can also use “ `$` ” in symbol names; exceptions are noted in See Chapter 2, *Overview* [7] .That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 2, *Overview* [7]), and underscores.

Case of letters is significant: `foo` is a different symbol name than `Foo`.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names “ `0` ” “ `1` ” ... “ `9` ” .To define a local symbol, write a label of the form “ `N:` ” (where N represents any digit). To refer to the most recent previous definition of that symbol write “ `Nb` ” ,using the same digit as when you defined the label. To refer to the next definition of a local label, write “ `Nf` ” where N gives you a choice of 10 forward references. The “ `b` ” stands for "backwards" and the “ `f` ” stands for "forwards".

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L`

All local labels begin with “ `L` ” .Normally both the assembler and the linker forget symbols that start with “ `L` ” .These labels are used for symbols you are never intended to see. If you use the **-L** option then the assembler retains these symbols in the object file. If you also instruct the linker to retain these symbols, you may use them in debugging.

`digit`

If the label is written “ `0:` ” then the digit is “ `0` ” .If the label is written “ `1:` ” then the digit is “ `1` ” .And so on up through “ `9:` ” .

`A`

This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value “ `\001` ” .

`ordinal number`

This is a serial number to keep the labels distinct. The first “ `0:` ” gets the number “ `1` ” ;The 15th “ `0:` ” gets the number “ `15` ” ;*etc.*. Likewise for the other labels “ `1:` ” through “ `9:` ” .

For instance, the first `1:` is named `L1C-A1`, the 44th `3:` is named `L3C-A44`.

6.4. The Special Dot Symbol

The special symbol “ **.** ” refers to the current address that the assembler is assembling into. Thus, the expression “ `melvin: .long .` ” defines `melvin` to contain its own address. Assigning a value to **.** is treated the same as a **.org** directive. Thus, the expression “ **.=.+4** ” is the same as saying “ **.space 4** ” .

6.5. Symbol Attributes

Every symbol has, as well as its name, the attributes “Value” and “Type”. Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, the assembler assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

6.5.1. Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as the linker changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and the linker tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a **.comm** common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

6.5.2. Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

6.5.3. Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between **.def** and **.endef** directives.

Primary Attributes. The symbol name is set with **.def**; the value and type, respectively, with **.val** and **.type**.

Auxiliary Attributes. The the assembler directives **.dim**, **.line**, **.scl**, **.size**, and **.tag** can generate auxiliary symbol table information for COFF.

Chapter 7 Assembler Expressions

An *expression* specifies an address or numeric value. White-space may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when the assembler sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression but the second pass is currently not implemented. The assembler aborts with an error message in this situation.

7.1. Empty Expressions

An empty expression has no value: it is just white-space or null. Wherever an absolute expression is required, you may omit the expression, and the assembler assumes a value of (absolute) 0. This is compatible with other assemblers.

7.2. Integer Expressions

An *integer expression* is one or more *arguments* delimited by *operators*.

7.2.1. Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called *arithmetic operands*. In this manual, to avoid confusing them with the “instruction operands” of the machine language, we use the term “argument” to refer to parts of expressions only, reserving the word “operand” to refer only to machine instruction operands.

Symbols are evaluated to yield {*`section`* *`NNN`*} where *`section`* is one of text, data, bss, absolute, or undefined. *`NNN`* is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and the assembler pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis “ `(` ” followed by an integer expression, followed by a right parenthesis “ `)` ” ;or a prefix operator followed by an argument.

7.2.2. Operators

Operators are arithmetic functions, like `+` or `%`. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by white-space.

7.2.3. Prefix Operator

The assembler has the following *prefix operators*. They each take one argument, which must be absolute.

-

  Negation. Two's complement negation.

~

  Complementation. Bitwise not.

7.2.4. Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

   `*`
   : *Multiplication*.

   `/`
   : *Division*. Truncation is the same as the C operator “ / ”

   `%`
   : *Remainder*.

   `<` , `<<`
   : *Shift Left*. Same as the C operator “ << ” .

   `>` , `>>`
   : *Shift Right*. Same as the C operator “ >> ” .

2. Intermediate precedence

   `|`
   : *Bitwise Inclusive Or*.

   `&`
   : *Bitwise And*.

   `^`
   : *Bitwise Exclusive Or*.

   `!`
   : *Bitwise Or Not*.

3. Lowest Precedence

   `+`
   : *Addition*. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

-

Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

Chapter 8

Assembler Directives

All assembler directives have names that begin with a period (“ **.** ”). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 2, *Overview* [7] .

.abort

abort

Synopsis

```
.abort
```

Description

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells the assembler to quit also. One day **.abort** will not be supported.

.ABORT

ABORT

Synopsis

```
.ABORT
```

Description

When producing COFF output, the assembler accepts this directive as a synonym for “ **.abort** ” .

.align

align

Synopsis

`.align` *`abs-expr`*, *`abs-expr`*, *`abs-expr`*

Description

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The assembler also provides **.balign** and **.p2align** directives, described later, which have a consistent behavior across all architectures.

.app-file

app-file

Synopsis

`.app-file` *`string`*

Description

.app-file (which may also be spelled “ **.file** ”)tells the assembler that we are about to start a new logical file. *`string`* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes “ `"` ” ;but if you wish to specify an empty file name is permitted, you must give the quotes-`""`. This statement may go away in future: it is only recognized to be compatible with old the assembler programs.

.ascii

ascii

Synopsis

`.ascii` "*`string`*"...

Description

.ascii expects zero or more string literals (see Section 4.6.1.1, “Strings” [21])separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz

asciz

Synopsis

`.asciz` "*string*"...

Description

.asciz is just like **.ascii**, but each string is followed by a zero byte. The "z" in “ **.asciz** ” stands for "zero".

.balign[wl]

balign[wl]

Synopsis

`.balign[wl]` *abs-expr*, *abs-expr*, *abs-expr*

Description

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example, “ `.balign 8` ” advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The **.balignw** and **.balignl** directives are variants of the **.balign** directive. The **.balignw** directive treats the fill pattern as a two byte word value. The **.balignl** directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

.byte

byte

Synopsis

`.byte` *`expressions`*

Description

.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

.comm

comm

Synopsis

`.comm` *`symbol`*, *`length`*

Description

.comm declares a named common area in the bss section. Normally the linker reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Use **.comm** to tell the linker that it must be at least *`length`* bytes long. The linker allocates space for each **.comm** symbol that is at least as long as the longest **.comm** request in any of the partial programs linked. *`length`* is an absolute expression.

.data

data

Synopsis

`.data` *`subsection`*

Description

.data tells the assembler to assemble the following statements onto the end of the data subsection numbered *`subsection`* (which is an absolute expression). If *`subsection`* is omitted, it defaults to zero.

.def

def

Synopsis

`.def` *`name`*

Description

Begin defining debugging information for a symbol `name`; the definition extends until the **.endef** directive is encountered.

.dim

dim

Synopsis

```
.dim
```

Description

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside **.def**/**.endef** pairs.

.double

double

Synopsis

```
.double flonums
```

Description

.double expects zero or more flonums, separated by commas. It assembles floating point numbers. See Chapter 2, *Overview* [7] .

.eject

eject

Synopsis

```
.eject
```

Description

Force a page break at this point, when generating assembly listings.

.else

else

Synopsis

```
.else
```

Description

.else is part of the the assembler support for conditional assembly; see .if [51] .It marks the beginning of a section of code to be assembled if the condition for the preceding **.if** was false.

.endef

endef

Synopsis

```
.endef
```

Description

This directive flags the end of a symbol definition begun with **.def**.

.endif

endif

Synopsis

```
.endif
```

Description

.endif is part of the the assembler support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See .if [51] .

.equ

equ

Synopsis

`.equ` *`symbol`*, *`expression`*

Description

This directive sets the value of *`symbol`* to *`expression`*. It is synonymous with " **.set** " ;see .set [67] .

.equiv

equiv

Synopsis

`.equiv` *`symbol`*, *`expression`*

Description

The **.equiv** directive is like **.equ** and **.set**, except that the assembler will signal an error if *`symbol`* is already defined.

Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
                                .err
                                .endif
                                .equ SYM,VAL
```

.err

err

Synopsis

```
.err
```

Description

If the assembler assembles a **.err** directive, it will print an error message and, unless the **-Z** option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

.extern

extern

Synopsis

```
.extern
```

Description

.extern is accepted in the source program for compatibility with other assemblers but it is ignored. the assembler treats all undefined symbols as external.

.file

file

Synopsis

```
.file string
```

Description

.file (which may also be spelled “ **.app-file** ”)tells the assembler that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes “ " ” ;but if you wish to specify an empty file name, you must give the quotes-"". This statement may go away in future: it is only recognized to be compatible with old the assembler programs.

.fill

fill

Synopsis

```
.fill repeat, size, value
```

Description

result, *size* and *value* are absolute expressions. This emits *repeat* copies of *size* bytes. *Repeat* may be zero or more. *Size* may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *repeat* bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are *value* rendered in the byte-order of an integer on the computer the assembler is assembling for. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and *value* are optional. If the second comma and *value* are absent, *value* is assumed zero. If the first comma and following tokens are absent, *size* is assumed to be 1.

.float

float

Synopsis

`.float` *`flonums`*

Description

This directive assembles zero or more flonums, separated by commas. It has the same effect as **.single**. The exact kind of floating point numbers emitted depends on how the assembler is configured. See Chapter 2, *Overview* [7] .

.global

global

Synopsis

`.global` *`symbol`* `.globl` *`symbol`*

Description

.global makes the symbol visible to the linker. If you define *`symbol`* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *`symbol`* takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings (“ **.globl** ” and “ **.global** ”)are accepted, for compatibility with other assemblers.

.hword

hword

Synopsis

```
.hword expressions
```

Description

This expects zero or more `expressions`, and emits a 16 bit number for each.

This directive is a synonym for " **.short** " ;depending on the target architecture, it may also be a synonym for " **.word** " .

.ident

ident

Synopsis

```
.ident
```

Description

This directive is used by some assemblers to place tags in object files. the assembler simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

.if

if

Synopsis

```
.if absolute expression
```

Description

The **.if** marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an `absolute expression`)is non-zero. The end of the conditional section of code must be marked by **.endif** (see .endif [46]); optionally, you may include code for the alternative condition, flagged by **.else** (see .else [46]).

The following variants of **.if** are also supported:

.ifdef *`symbol`*

Assembles the following section of code if the specified *`symbol`* has been defined.

.ifndef *`symbol`* , **.ifnotdef** *`symbol`*

Assembles the following section of code if the specified *`symbol`* has not been defined. Both spelling variants are equivalent.

.include

include

Synopsis

```
.include "file"
```

Description

This directive provides a way to include supporting files at specified points in your source program. The code from *`file`* is assembled as if it followed the point of the **.include**; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the **-I** command-line option (see Chapter 3, *Assembler Command-Line Options* [11]). Quotation marks are required around *`file`*.

.int

int

Synopsis

`.int` *`expressions`*

Description

Expect zero or more *`expressions`*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

.irp

irp

Synopsis

`.irp` *`symbol`*, *`values`*...

Description

Evaluate a sequence of statements assigning different values to *`symbol`*. The sequence of statements starts at the **.irp** directive, and is terminated by an **.endr** directive. For each *`value`*, *`symbol`* is set to *`value`*, and the sequence of statements is assembled. If no *`value`* is listed, the sequence of statements is assembled once, with *`symbol`* set to the null string. To refer to *`symbol`* within the sequence of statements, use `\`*`symbol`*.

For example, assembling

```
        .irp    param,1,2,3
                           move    d\param,sp@-
                           .endr
```

is equivalent to assembling

```
        move    d1,sp@-
                        move    d2,sp@-
                        move    d3,sp@-
```

.irpc

irpc

Synopsis

`.irpc` *`symbol`*,*`values`*...

Description

Evaluate a sequence of statements assigning different values to *`symbol`*. The sequence of statements starts at the **.irpc** directive, and is terminated by an **.endr** directive. For each character in *`value`*, *`symbol`* is set to the character, and the sequence of statements is assembled. If no *`value`* is listed, the sequence of statements is assembled once, with *`symbol`* set to the null string. To refer to *`symbol`* within the sequence of statements, use `\`*`symbol`*.

For example, assembling

```
        .irpc    param,123
                        move    d\param,sp@-
                        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
                        move    d2,sp@-
                        move    d3,sp@-
```

.lcomm

lcomm

Synopsis

`.lcomm` *`symbol`*, *`length`*

Description

Reserve *`length`* (an absolute expression) bytes for a local common denoted by *`symbol`*. The section and value of *`symbol`* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *`symbol`* is not declared global (see .global [50]), so is normally not visible to the linker.

.lflags

lflags

Synopsis

`.lflags`

Description

The assembler accepts this directive, for compatibility with other assemblers, but ignores it.

.line

line

Synopsis

`.line` *`line-number`*

Description

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, the assembler still recognizes it when producing COFF output, and treats “ **.line** ” as though it were the COFF “ **.ln** ” *if* it is found outside a **.def**/**.endef** pair.

Inside a **.def**, “ **.line** ” is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

.linkonce

linkonce

Synopsis

`.linkonce [`*`type`*`]`

Description

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The **.linkonce** directive must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The *`type`* argument is optional. If specified, it must be one of the following strings. For example:

```
.linkonce same_size
```

Not all types may be supported on all object file formats.

discard
  Silently discard duplicate sections. This is the default.

one_only
Warn if there are duplicate sections, but still keep only one copy.

same_size
Warn if any of the duplicates have different sizes.

same_contents
Warn if any of the duplicates do not have exactly the same contents.

.ln

ln

Synopsis

```
.ln line-number
```

Description

“ **.ln** ” is a synonym for “ **.line** ” .

.list

list

Synopsis

```
.list
```

Description

Control (in conjunction with the **.nolist** directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). **.list** increments the counter, and **.nolist** decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the **-a** command line option; see Chapter 3, *Assembler Command-Line Options* [11]), the initial value of the listing counter is one.

.long

long

Synopsis

`.long` *`expressions`*

Description

.long is the same as “ **.int** ” ,see .int [53] .

.macro

macro

Synopsis

`.macro`

Description

The commands **.macro** and **.endm** allow you to define macros that generate assembly output. For example, this definition specifies a macro `sum` that puts a sequence of numbers into memory:

```
          .macro  sum from=0, to=5
                          .long   \from
                          .if     \to-\from
                          sum     "(\from+1)",\to
                          .endif
                          .endm
```

With that definition, “ `SUM 0,5` ” is equivalent to this assembly input:

```
        .long   0
                            .long   1
                            .long   2
                            .long   3
                            .long   4
                            .long   5
```

.macro *`macname`* , **.macro** *`macname macargs`* ...

Begin the definition of a macro called *`macname`*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with “ `=`*`deflt`* ” . For example, these are all valid **.macro** statements:

.macro comm
Begin the definition of a macro called `comm`, which takes no arguments.

.macro plus1 p, p1 , **.macro plus1 p p1**
Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write “ `\p` ” or “ `\p1` ” to evaluate the arguments.

.macro reserve_str p1=0 p2
Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as “ `reserve_str` *`a,b`* ” (with “ `\p1` ” evaluating to *`a`* and “ `\p2` ” evaluating to *`b`*), or as “ `reserve_str ,`*`b`* ” (with “ `\p1` ” evaluating as the default, in this case “ `0` ” ,and “ `\p2` ” evaluating to *`b`*).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, “ `sum 9,17` ” is equivalent to “ `sum to=17, from=9` ” .

.endm
Mark the end of a macro definition.

.exitm
Exit early from the current macro definition.

`\@`
The assembler maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with “ `\@` ” ,but *only within a macro definition* .

.nolist

nolist

Synopsis

```
.nolist
```

Description

Control (in conjunction with the **.list** directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). **.list** increments the counter, and **.nolist** decrements it. Assembly listings are generated whenever the counter is greater than zero.

.octa

octa

Synopsis

`.octa` *`bignums`*

Description

This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term “octa” comes from contexts in which a “word” is two bytes; hence *octa*-word for 16 bytes.

.org

org

Synopsis

`.org` *`new-lc`*, *`fill`*

Description

Advance the location counter of the current section to *`new-lc`*. *`new-lc`* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use **.org** to cross sections: if *`new-lc`* has the wrong section, the **.org** directive is ignored. To be compatible with former assemblers, if the section of *`new-lc`* is absolute, the assembler issues a warning, then pretends the section of *`new-lc`* is the same as the current subsection.

.org may only increase the location counter, or leave it unchanged; you cannot use **.org** to move the location counter backwards.

Because the assembler tries to assemble programs in one pass, *`new-lc`* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *`fill`* which should be an absolute expression. If the comma and *`fill`* are omitted, *`fill`* defaults to zero.

.p2align[wl]

p2align[wl]

Synopsis

`.p2align[wl]` *`abs-expr`*, *`abs-expr`*, *`abs-expr`*

Description

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example " **.p2align 3** " advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The **.p2alignw** and **.p2alignl** directives are variants of the **.p2align** directive. The **.p2alignw** directive treats the fill pattern as a two byte word value. The **.p2alignl** directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

.psize

psize

Synopsis

```
.psize lines, columns
```

Description

Use this directive to declare the number of lines and, optionally, the number of columns to use for each page, when generating listings.

If you do not use **.psize**, listings use a default line-count of 60. You may omit the comma and `columns` specification; the default width is 200 columns.

The assembler generates form feeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using **.eject**).

If you specify `lines` as `0`, no form feeds are generated save those explicitly specified with **.eject**.

.quad

quad

Synopsis

```
.quad bignums
```

Description

.quad expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum. The term "quad" comes from contexts in which a "word" is two bytes; hence *quad*-word for 8 bytes.

.rept

rept

Synopsis

```
.rept count
```

Description

Repeat the sequence of lines between the **.rept** directive and the next **.endr** directive *count* times.

For example, assembling

```
        .rept   3
                        .long   0
                        .endr
```

is equivalent to assembling

```
        .long   0
                        .long   0
                        .long   0
```

.sbttl

sbttl

Synopsis

`.sbttl "sub heading"`

Description

Use *sub heading* as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.scl

scl

Synopsis

`.scl` *`class`*

Description

Set the storage-class value for a symbol. This directive may only be used inside a **.def**/**.endef** pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

.section

section

Synopsis

`.section` *`name`*, *`subsection`*

Description

Assemble the following code into end of subsection numbered *`subsection`* in the COFF named section *`name`*. If you omit *`subsection`*, the assembler uses subsection number zero. “ **.section .text** ” is equivalent to the **.text** directive; “ `.section .data` ” is equivalent to the **.data** directive.

For COFF targets, the **.section** directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b`
: bss section (uninitialized data)

`n`
: section is not loaded

`d , w`
: writable section

`d`
: data section

`r`
: read-only section

`x`
: executable section

For ELF targets, the **.section** directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`a`
: allocate only

`w`
: writable section

`d`
: data section (same as >`w`)

`r`
: read-only section

`x`
: executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the **.section** directive is not quoted, it is taken as a subsegment number (see Section 5.3, “Sub-Sections” [28]).

.set

set

Synopsis

`.set` *`symbol`*, *`expression`*

Description

Set the value of *`symbol`* to *`expression`*. This changes *`symbol`*'s value and type to conform to *`expression`*. If *`symbol`* was flagged as external, it remains flagged. (See Section 6.5, “Symbol Attributes” [33] .)

You may **.set** a symbol many times in the same assembly.

If you **.set** a global symbol, the value stored in the object file is the last value stored into it.

.short

short

Synopsis

`.short` *`expressions`*

Description

.short is normally the same as “ **.word** ” .See .word [73] .

In some configurations, however, **.short** and **.word** generate numbers of different lengths; see Chapter 2, *Overview* [7] .

.single

single

Synopsis

```
.single flonums
```

Description

This directive assembles zero or more flonums, separated by commas. It has the same effect as **.float**. See Chapter 2, *Overview* [7] .

.size

size

Synopsis

```
.size
```

Description

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside **.def**/**.endef** pairs.

.skip

skip

Synopsis

```
.skip size, fill
```

Description

This directive emits *`size`* bytes, each of value *`fill`*. Both *`size`* and *`fill`* are absolute expressions. If the comma and *`fill`* are omitted, *`fill`* is assumed to be zero. This is the same as " **.space** " .

.space

space

Synopsis

```
.space size, fill
```

Description

This directive emits *`size`* bytes, each of value *`fill`*. Both *`size`* and *`fill`* are absolute expressions. If the comma and *`fill`* are omitted, *`fill`* is assumed to be zero. This is the same as " **.skip** " .

.stabd, .stabn, .stabs

.stabd, .stabn, .stabs

Synopsis

```
.stabd, .stabn, .stabs
```

Description

There are three directives that begin " **.stab** " .All emit symbols (see Chapter 6, *Assembler Symbols* [31]), for use by symbolic debuggers. The symbols are not entered in the the assembler hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

`string`

This is the symbol's name. It may contain any character except " `\000` " ,so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type
: An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but the linker and debuggers choke on silly bit patterns.

other
: An absolute expression. The symbol's “other” attribute is set to the low 8 bits of this expression.

desc
: An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value
: An absolute expression which becomes the symbol's value.

If a warning is detected while reading a **.stabd**, **.stabn**, or **.stabs** statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

.stabd *type*, *other*, *desc*
: The “name” of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

  The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the **.stabd** was assembled.

.stabn *type*, *other*, *desc*, *value*
: The name of the symbol is set to the empty string `""`.

.stabs *string* , *type*, *other*, *desc*, *value*
: All five fields are specified.

.string

string

Synopsis

```
.string "str"
```

Description

Copy the characters in `str` to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a zero byte. You can use any of the escape sequences described in Section 4.6.1.1, “Strings” [21] .

.tag

tag

Synopsis

```
.tag structname
```

Description

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside **.def**/**.endef** pairs. Tags are used to link structure definitions in the symbol table with instances of those structures.

.text

text

Synopsis

```
.text subsection
```

Description

Tells the assembler to assemble the following statements onto the end of the text subsection numbered *`subsection`*, which is an absolute expression. If *`subsection`* is omitted, subsection number zero is used.

.title

title

Synopsis

```
.title "heading"
```

Description

Use *`heading`* as the title (second line, immediately after the source file name and page number) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.type

type

Synopsis

```
.type int
```

Description

This directive, permitted only within **.def**/**.endef** pairs, records the integer *`int`* as the type attribute of a symbol table entry.

.val

val

Synopsis

`.val` *`addr`*

Description

This directive, permitted only within **.def** / **.endef** pairs, records the address *`addr`* as the value attribute of a symbol table entry.

.word

word

Synopsis

`.word` *`expressions`*

Description

This directive expects zero or more *`expressions`*, of any section, separated by commas.

For each expression, the assembler emits a 16 or 32-bit number, according to the target word size.

II

Using the Linker

Chapter 9

Linker Overview

The linker combines a number of object and archive files, relocates their data and ties up symbol references. It is usually the last step in a compilation.

The linker accepts *Linker Command Language* files written in a superset of AT&T's Link Editor Command Language syntax, to provide explicit and total control over the linking process.

The linker uses the general-purpose BFD libraries to operate on object files. This allows the linker to read, combine, and write object files in many different formats for example, COFF or `Intel Hex`. Different formats may be linked together to produce any available kind of object file. See Appendix A, *BFD* [171] ,for more information.

Aside from its flexibility, the linker is more helpful than other linkers in providing diagnostic information. Many linkers abandon execution immediately upon encountering an error; whenever possible, the linker continues executing, allowing you to identify other errors (or, in some cases, to get an output file in spite of the error).

Chapter 10 Linker Invocation

The linker is meant to cover a broad range of situations, and to be as compatible as possible with other linkers. As a result, you have many choices to control its behavior.

10.1. Command Line Options

The linker supports a plethora of command-line options, but in practice few of them are used in any particular context. For example, to link a file `hello.o`:

```
$ prefix-ld -o output crt0.o hello.o -lc
```

This tells the linker to produce a file called *`output`* as the result of linking the file `crt0.o` with `hello.o` and the library `libc.a`, which will come from the standard search directories. (See the discussion of the **-l** option below.)

The command-line options to the linker may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options that may be meaningfully specified more than once are noted in the descriptions below.

Non-option arguments are objects files that are to be linked together. They may follow, precede, or be mixed in with command-line options, except that an object file argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using **-l**, **-R**, and the script command language. If *no* binary input files at all are specified, the linker does not produce any output, and issues the message “ `No input files` ” .

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way *augments* the main linker script used for the link (either the default linker script or the one specified by using **-T**). This feature permits the linker to link against a file that appears to be an object or an archive, but actually merely defines some symbol values, or uses **INPUT** or **GROUP** to load other objects. Note that specifying a script in this way should only be used to augment the main linker script; if you want to use some command that logically can only appear once, such as the **SECTIONS** or **MEMORY** command, you must replace the default linker script using the **-T** option. See Chapter 11, *Linker Command Language* [97] .

For options whose names are a single letter, option arguments either must follow the option letter without intervening white-space, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, **--oformat** and **--oformat** are equivalent. Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, **--oformat srec** and **--oformat=srec** are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

-b *`input-format`* , **--format=** *`input-format`*

The linker is configured to support more than one kind of object file. You can use the **-b** option to specify the binary format for input object files that follow this option on the command line. You don't usually need to specify this, as the linker should be configured to expect as a default input format the most usual format on each machine. *`input-format`* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with **objdump -i** .) See Appendix A, *BFD* [171] .

You may want to use this option if you are linking files with an unusual binary format. You can also use **-b** to switch formats explicitly (when linking object files of different formats), by including " **-b** *`input-format`* " before each group of object files in a particular format.

The default format is taken from the environment variable GNUPREFIX. See Section 10.2, "Environment Variables" [95] .You can also define the input format from a script, using the command `PREFIX`; see Section 11.7, "Option Commands" [125] .

-d , **-dc** , **-dp**

These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with **-r**). The script command **FORCE_COMMON_ALLOCATION** has the same effect. See Section 11.7, "Option Commands" [125] .

-e *`entry`* , **--entry=** *`entry`*

Use *`entry`* as the explicit symbol for beginning execution of your program, rather than the default entry point. See Section 11.5, "The Entry Point" [120] ,for a discussion of defaults and other ways of specifying the entry point.

-E , **--export-dynamic**

When creating a dynamically linked executable, add all symbols to the dynamic symbol table. Normally, the dynamic symbol table contains only symbols that are used by a dynamic object. This option is needed for some uses of `dlopen`.

-f , **--auxiliary** *name*

When creating an ELF shared object, set the internal DT_AUXILIARY field to the specified name. This tells the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object *name*.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT_AUXILIARY field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object *name*. If there is one, it will be used instead of the definition in the filter object. The shared object *name* need not exist. Thus the shared object *name* may be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine specific performance.

This option may be specified more than once. The DT_AUXILIARY entries will be created in the order in which they appear on the command line.

-F *name* , **--filter** *name*

When creating an ELF shared object, set the internal DT_FILTER field to the specified name. This tells the dynamic linker that the symbol table of the shared object that is being created should be used as a filter on the symbol table of the shared object *name*.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT_FILTER field. The dynamic linker will resolve symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object *name*. Thus the filter object can be used to select a subset of the symbols provided by the object *name*.

Some older linkers used the -F option throughout a compilation tool chain for specifying object-file format for both input and output object files. The XGC linker uses other mechanisms for this purpose: the -b, --format, --oformat options, the PREFIX command in linker scripts, and the GNUPREFIX environment variable. The linker will ignore the -F option when not creating an ELF shared object.

--force-exe-suffix

Make sure that an output file has a `.exe` suffix.

If a successfully built fully linked output file does not have a `.exe` or `.dll` suffix, this option forces the linker to copy the output file to one of the same name with a `.exe` suffix. This option is useful when using unmodified UNIX makefiles on a Microsoft Windows host, since some versions of Windows won't run an image unless it ends in a `.exe` suffix.

-g

Ignored. Provided for compatibility with other tools.

-G *`value`* , **--gpsize=** *`value`*

Set the maximum size of objects to be optimized using the GP register to *`size`*. This is only meaningful for object file formats such as ECOFF, which supports putting large and small objects into different sections. This is ignored for other object file formats.

-h *`name`* , **-soname=** *`name`*

When creating an ELF shared object, set the internal DT_SOGNAT field to the specified name. When an executable is linked with a shared object that has a DT_SOGNAT field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the DT_SOGNAT field rather than the using the file name given to the linker.

-i

Perform an incremental link (same as option **-r**).

-l *`archive`* , **--library=** *`archive`*

Add archive file *`archive`* to the list of files to link. This option may be used any number of times. The linker will search its path-list for occurrences of `lib`*`archive`*`.a` for every *`archive`* specified.

On systems that support shared libraries, the linker may also search for libraries with extensions other than `.a`. Specifically, on ELF and Sun-OS systems, the linker will search a directory for a library with an extension of `.so` before searching for one with an extension of `.a`. By convention, a `.so` extension indicates a shared library.

The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol that was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

See the **-(** option for a way to force the linker to search archives multiple times.

You may list the same archive multiple times on the command line.

-L `searchdir` , **--library-path=** `searchdir`
Add path `searchdir` to the list of paths that the linker will search for archive libraries and the linker control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All **-L** options apply to all **-l** options, regardless of the order in which the options appear.

The default set of paths searched (without being specified with **-L**) depends on which emulation mode the linker is using, and in some cases also on how it was configured. See Section 10.2, “Environment Variables” [95] .

The paths can also be specified in a link script with the `SEARCH_DIR` command. Directories specified this way are searched at the point in which the linker script appears in the command line.

-m `emulation`
Emulate the `emulation` linker. You can list the available emulations with the **--verbose** or **-V** options. The default depends on how the linker was configured.

-M , **--print-map**
Print (to the standard output) a link map diagnostic information about where symbols are mapped by the linker ,and information on global common storage allocation.

-n , **--nmagic**
Set the text segment to be read only, and mark the output as `NMAGIC` if possible.

-N , **--omagic**
Set the text and data sections to be readable and writable. Also, do not page-align the data segment. If the output format supports UNIX style magic numbers, mark the output as `OMAGIC`.

-o *`output`* , **--output=** *`output`*
Use *`output`* as the name for the program produced by the linker; if this option is not specified, the name `a.out` is used by default. The script command `OUTPUT` can also specify the output file name.

-r , **--relocatable**
Generate relocatable output that is, generate an output file that can in turn serve as input to the linker .This is often called *partial linking* .As a side effect, in environments that support standard UNIX magic numbers, this option also sets the output file's magic number to `OMAGIC`. If this option is not specified, an absolute file is produced. When linking C++ programs, this option *will not* resolve references to constructors; to do that, use **-Ur**.

This option does the same thing as **-i**.

-R *`filename`* , **--just-symbols=** *`filename`*
Read symbol names and their addresses from *`filename`*, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs. You may use this option more than once.

For compatibility with other ELF linkers, if the **-R** option is followed by a directory name, rather than a file name, it is treated as the **-rpath** option.

-s , **--strip-all**
Omit all symbol information from the output file.

-S , **--strip-debug**
Omit debugger symbol information (but not all symbols) from the output file.

-t , **--trace**
: Print the names of the input files as the linker processes them.

-T *`commandfile`* , **--script=** *`commandfile`*
: Read link commands from the file *`commandfile`*. These commands replace the linker's default link script (rather than adding to it), so *`commandfile`* must specify everything necessary to describe the target format. You must use this option if you want to use a command that can only appear once in a linker script, such as the `SECTIONS` or `MEMORY` command. See Chapter 11, *Linker Command Language* [97] .If *`commandfile`* does not exist, the linker looks for it in the directories specified by any preceding **-L** options. Multiple **-T** options accumulate.

-u *`symbol`* , **--undefined=** *`symbol`*
: Force *`symbol`* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. **-u** may be repeated with different option arguments to enter additional undefined symbols.

-v , **--version** , **-V**
: Display the version number for the linker .The **-V** option also lists the supported emulations.

-x , **--discard-all**
: Delete all local symbols.

-X , **--discard-locals**
: Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with “ `L` ” .

-y *`symbol`* , **--trace-symbol=** *`symbol`*
: Print the name of each linked file in which *`symbol`* appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore.

  This option is useful when you have an undefined symbol in your link but don't know where the reference is coming from.

-Y *`path`*
: Add *`path`* to the default library search path. This option exists for Solaris compatibility.

-z *`keyword`*
: This option is ignored for Solaris compatibility.

`-( archives -) , --start-group archives --end-group`

The *`archives`* should be a list of archive files. They may be either explicit file names, or **-l** options.

The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.

Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

-assert *`keyword`*

This option is ignored for Sun-OS compatibility.

-Bdynamic , **-dy** , **-call_shared**

Link against dynamic libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for **-l** options that follow it.

-Bstatic , **-dn** , **-non_shared** , **-static**

Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for **-l** options that follow it.

-Bsymbolic

When creating a shared library, bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library. This option is only meaningful on ELF platforms that support shared libraries.

--cref

Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output.

The format of the table is intentionally simple, so that it may be easily processed by a script if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`--defsym` *`symbol=expression`*

Create a global symbol in the output file, containing the absolute address given by *`expression`*. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *`expression`* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use + and - to add or subtract hexadecimal constants or symbols. If you need more elaborate expressions, consider using the linker command language from a script (see Section 11.2.6, “Assignment: Defining Symbols” [101]). *Note:* there should be no white space between *`symbol`*, the equals sign (=), and *`expression`*.

--dynamic-linker *`file`*

Set the name of the dynamic linker. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing.

-EB

Link big-endian objects. This affects the default output format.

-EL

Link little-endian objects. This affects the default output format.

--help

Print a summary of the command-line options on the standard output and exit.

-Map *`mapfile`*

Print to the file *`mapfile`* a link map -- diagnostic information about where symbols are mapped by the linker, and information on global common storage allocation.

--no-keep-memory
The linker normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells the linker to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if the linker runs out of memory space while linking a large executable.

--no-whole-archive
Turn off the effect of the **--whole-archive** option for subsequent archive files.

--noinhibit-exec
Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

--oformat *`output-format`*
The linker may be configured to support more than one kind of object file. If the linker is configured this way, you can use the **--oformat** option to specify the binary format for the output object file. Even when the linker is configured to support alternative object formats, you don't usually need to specify this, as the linker should be configured to produce as a default output format the most usual format on each machine. *`output-format`* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with “ `objdump -i` ” .) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See Appendix A, *BFD* [171] .

-qmagic
This option is ignored for Linux compatibility.

-Qy
This option is ignored for SVR4 compatibility.

--relax
An option with machine dependent effects.

On some platforms, the **--relax** option performs global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes and synthesizing new instructions in the output object file.

--retain-symbols-file *`filename`*

Retain *only* the symbols listed in the file *`filename`*, discarding all others. *`filename`* is simply a flat file, with one symbol name per line. This option is especially useful in environments where a large global symbol table is accumulated gradually, to conserve run-time memory.

--retain-symbols-file does *not* discard undefined symbols, or symbols needed for relocations.

You may only specify **--retain-symbols-file** once in the command line. It overrides **-s** and **-S**.

--sort-common

This option tells the linker to sort the common symbols by size when it places them in the appropriate output sections. First come all the one byte symbols, then all the two bytes, then all the four bytes, and then everything else. This is to prevent gaps between symbols due to alignment constraints.

--split-by-file

Similar to **--split-by-reloc** but creates a new output section for each input file.

--split-by-reloc *`count`*

Tries to creates extra sections in the output file so that no single output section in the file contains more than *`count`* relocations. This is useful when generating huge relocatable for down-loading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section. Note that this will fail to work with object file formats that do not support arbitrary sections. The linker will not split up individual input sections for redistribution, so if a single input section contains more than *`count`* relocations one output section will contain that many relocations.

--stats

Compute and display statistics about the operation of the linker, such as execution time and memory usage.

--traditional-format

For some targets, the output of the linker is different in some ways from the output of some existing linker. This switch requests the linker to use the traditional format instead.

For example, on Sun-OS, the linker combines duplicate entries in the symbol string table. This can reduce the size of an output file with full debugging information by over 30 percent. Unfortunately, the Sun-OS `dbx` program can not read the resulting program (`gdb` has no trouble). The **--traditional-format** switch tells the linker to not combine duplicate entries.

-Tbss *`org`* , **-Tdata** *`org`* , **-Ttext** *`org`*

Use *`org`* as the starting address for--respectively--the `bss`, `data`, or the `text` segment of the output file. *`org`* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading “ `0x` ” usually associated with hexadecimal values.

-Ur

For anything other than C++ programs, this option is equivalent to **-r**: it generates relocatable output that is, an output file that can in turn serve as input to the linker. When linking C++ programs, **-Ur** *does* resolve references to constructors, unlike **-r**. It does not work to use **-Ur** on files that were themselves linked with **-Ur**; once the constructor table has been built, it cannot be added to. Use **-Ur** only for the last partial link, and **-r** for the others.

--verbose

Display the version number for the linker and list the linker emulations supported. Display which input files can and cannot be opened. Display the linker script if using a default built in script.

--version-script= *`version-scriptfile`*

Specify the name of a version script to the linker. This is typically used when creating shared libraries to specify additional information about the version hierarchy for the library being created. This option is only meaningful on ELF platforms that support shared libraries. See Section 11.6, “Version Script” [121] .

--warn-common

Warn when a common symbol is combined with another common symbol or with a symbol definition. UNIX linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols.

Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.

There are three kinds of global symbols, illustrated here by C examples:

“ `int i = 1;` ”
: A definition, which goes in the initialized data section of the output file.

“ `extern int i;` ”
: An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

“ `int i;` ”
: A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

The **--warn-common** option can produce five kinds of warnings. Each warning consists of a pair of lines: the first describes the symbol just encountered, and the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.

1. Turning a common symbol into a reference, because there is already a definition for the symbol.

```
file(section): warning: common of
overridden by definition
file(section): warning: defined h
```

2. Turning a common symbol into a reference, because a later definition for the symbol is encountered. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section):
warning: definition of `symbol' overri
common file(section):
warning: common is here
```

3. Merging a common symbol with a previous same-sized common symbol.

```
file(section):
warning: multiple common of `symbol'
file(section): warning:
previous common is here
```

4. Merging a common symbol with a previous larger common symbol.

```
file(section):
warning: common of `symbol' overridden
common file(section):
warning: larger common is here
```

5. Merging a common symbol with a previous smaller common symbol. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: common of `sym
overriding smaller common
file(section): warning: smaller common
```

--warn-constructors
: Warn if any global constructors are used. This is only useful for a few object file formats. For formats like COFF or ELF, the linker can not detect the use of global constructors.

--warn-multiple-gp
: Warn if multiple global pointer values are required in the output file. This is only meaningful for certain processors, such as the Alpha. Specifically, some processors put large-valued constants in a special section. A special register (the global pointer) points into the middle of this section, so that constants can be loaded efficiently via a base-register relative addressing mode. Since the offset in base-register relative mode is fixed and relatively small (e.g., 16 bits), this limits the maximum size of the constant pool. Thus, in large programs, it is often necessary to use multiple global pointer values in order to be able to address all possible constants. This option causes a warning to be issued whenever this case occurs.

--warn-once
: Only warn once for each undefined symbol, rather than once per module that refers to it.

--warn-section-align
: Warn if the address of an output section is changed because of alignment. Typically, the alignment will be set by an input section. The address will only be changed if it not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section (see Section 11.4, “Specifying Output Sections” [108]).

--whole-archive
: For each archive mentioned on the command line after the **--whole-archive** option, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library. This option may be used more than once.

--wrap *`symbol`*
: Use a wrapper function for *`symbol`*. Any undefined reference to *`symbol`* will be resolved to `__wrap_`*`symbol`* .Any undefined reference to `__real_`*`symbol`* will be resolved to *`symbol`*.

This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_`*`symbol`* .If it wishes to call the system function, it should call `__real_`*`symbol`* .

Here is a trivial example:

```
void *
                                    __wrap_malloc (int c)
                                    {
                                    printf ("malloc called with %ld\n", c);
                                    return __real_malloc (c);
                                    }
```

If you link other code with this file using `--wrap malloc`, then all calls to `malloc` will call the function `__wrap_malloc` instead. The call to `__real_malloc` in `__wrap_malloc` will call the real `malloc` function.

You may wish to provide a `__real_malloc` function as well, so that links without the **--wrap** option will succeed. If you do this, you should not put the definition of `__real_malloc` in the same file as `__wrap_malloc`; if you do, the assembler may resolve the call before the linker has a chance to wrap it to `malloc`.

10.2. Environment Variables

You can change the behavior of the linker with the environment variable `GNUPREFIX`.

`GNUPREFIX` determines the input-file object format if you don't use **-b** (or its synonym **--format**). Its value should be one of the BFD names for an input format (see Appendix A, *BFD* [171]). If there is no `GNUPREFIX` in the environment, the linker uses the natural format of the target. If `GNUPREFIX` is set to `default` then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of ensuring that the magic number used to specify object-file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search-list, so ambiguities are resolved in favor of convention.

Chapter 11

Linker Command Language

The command language provides explicit control over the link process, allowing complete specification of the mapping between the linker's input files and its output. It controls:

- input files
- file formats
- output file layout
- addresses of sections
- placement of common blocks

You may supply a command file (also known as a linker script) to the linker either explicitly through the **-T** option, or implicitly as an ordinary file. Normally you should use the **-T** option. An implicit linker script should only be used when you want to augment, rather than replace, the default linker script; typically an implicit linker script would consist only of `INPUT` or `GROUP` commands.

If the linker opens a file that it cannot recognize as a supported object or archive format, nor as a linker script, it reports an error.

11.1. Linker Scripts

The linker command language is a collection of statements; some are simple keywords setting a particular option, some are used to select and group input files or name output files; and two statement types have a fundamental and pervasive impact on the linking process.

The most fundamental command of the linker command language is the `SECTIONS` command (see Section 11.4, “Specifying Output Sections” [108]). Every meaningful command script must have a `SECTIONS` command: it specifies a "picture" of the output file's layout, in varying degrees of detail. No other command is required in all cases.

The `MEMORY` command complements `SECTIONS` by describing the available memory in the target architecture. This command is optional; if you don't use a `MEMORY` command, the linker assumes sufficient memory is available in a contiguous block for all output. See Section 11.3, “Memory Layout” [107] .

You may include comments in linker scripts just as in C: delimited by “ `/*` ” and “ `*/` ” .As in C, comments are syntactically equivalent to white-space.

11.2. Expressions

Many useful commands involve arithmetic expressions. The syntax for expressions in the command language is identical to that of C expressions, with the following features:

- All expressions evaluated as integers and are of “long” or “unsigned long” type.
- All constants are integers.
- All of the C arithmetic operators are provided.
- You may reference, define, and create global variables.
- You may call special purpose built-in functions.

11.2.1. Integers

An octal integer is “ `0` ” followed by zero or more of the octal digits (“ `01234567` ”).

```
_as_octal = 0157255;
```

A decimal integer starts with a non-zero digit followed by zero or more digits (“ `0123456789` ”).

```
_as_decimal = 57005;
```

A hexadecimal integer is “ `0x` ” or “ `0X` ” followed by one or more hexadecimal digits chosen from “ `0123456789abcdefABCDEF` ” .

```
_as_hex = 0xdead;
```

To write a negative integer, use the prefix operator **-** (see Section 11.2.4, “Operators” [101]).

```
_as_neg = -57005;
```

Additionally the suffixes `K` and `M` may be used to scale a constant by respectively. For example, the following all refer to the same quantity:

```
        _fourk_1 = 4K;
                            _fourk_2 = 4096;
                            _fourk_3 = 0x1000;
```

11.2.2. Symbol Names

Unless quoted, symbol names start with a letter, underscore, or point and may include any letters, underscores, digits, points, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol that contains odd characters or has the same name as a keyword, by surrounding the symbol name in double quotes:

```
          "SECTION" = 9;
                              "with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, “ `A-B` ” is one symbol, whereas “ `A- B` ” is an expression involving subtraction.

11.2.3. The Location Counter

The special linker variable *dot* “ `.` ” always contains the current output location counter. Since the `.` always refers to a location in an output section, it must always appear in an expression within a `SECTIONS` command. The `.` symbol may appear anywhere that an ordinary symbol is allowed in an expression, but its assignments have a side effect. Assigning a value to the `.` symbol will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
                            {
                            output :
                            {
                            file1(.text)
                            . = . + 1000;
                            file2(.text)
                            . += 1000;
                            file3(.text)
                            } = 0x1234;
                            }
```

In the previous example, `file1` is located at the beginning of the output section, then there is a 1000 byte gap. Then `file2` appears, also with a 1000 byte gap following before `file3` is loaded. The notation “ `=0x1234` ” specifies what data to write in the gaps (see Section 11.4.4, “Optional Section Attributes” [116]).

11.2.4. Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

Precedence	Associativity	Operators
(Highest)		
1	left	`!- ~`[a]
2	left	`*/ %`
3	left	`+-`
4	left	`>> <<`
5	left	`== != > < <= >=`
6	left	`&`
7	left	`|`
8	left	`&&`
9	left	`||`
10	right	`?:`
11	right	`&= += -= *= /=`[b]
(Lowest)		

[a]Prefix operators

[b] See Section 11.2.6, “Assignment: Defining Symbols” [101]

11.2.5. Evaluation

The linker uses *lazy evaluation* for expressions; it only calculates an expression when absolutely necessary. The linker needs the value of the start address, and the lengths of memory regions, in order to do any linking at all; these values are computed as soon as possible when the linker reads in the command file. However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

11.2.6. Assignment: Defining Symbols

You may create global symbols, and assign values (addresses) to global symbols, using any of the C assignment operators:

```
symbol = expression ;
symbol &= expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
```

Two things distinguish assignment from other operators in linker expressions.

- Assignment may only be used at the root of an expression; “ `a=b+3;` ” is allowed, but “ `a+b=3;` ” is an error.
- You must place a trailing semicolon (`;`) at the end of an assignment statement.

Assignment statements may appear:

- as commands in their own right in the linker script; or
- as independent statements within a `SECTIONS` command; or
- as part of the contents of a section definition in a `SECTIONS` command.

The first two cases are equivalent in effect both define a symbol with an absolute address. The last case defines a symbol whose address is relative to a particular section (see Section 11.4, “Specifying Output Sections” [108]).

When a linker expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file; a relocatable expression type is one in which the value is expressed as a fixed offset from the base of a section.

The type of the expression is controlled by its position in the script file. A symbol assigned within a section definition is created relative to the base of the section; a symbol assigned in any other place is created as an absolute symbol. Since a symbol created within a section definition is relative to the base of the section, it will remain relocatable if relocatable output is requested. A symbol may be created with an absolute value even when assigned to within a section definition by using the absolute assignment function

`ABSOLUTE`. For example, to create an absolute symbol whose address is the last byte of an output section named `.data`:

```
SECTIONS{ ...
                          .data :
                          {
                          *(.data)
                          _edata = ABSOLUTE(.) ;
                          }
                          ... }
```

The linker tries to put off the evaluation of an assignment until all the terms in the source expression are known (see Section 11.2.5, “Evaluation” [101]). For instance, the sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation. Some expressions, such as those depending upon the location counter *dot*, “ `.` ” must be evaluated during allocation. If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following

```
SECTIONS { ...
                          text 9+this_isnt_constant :
                          { ...
                          }
                          ... }
```

will cause the error message “Non constant expression for initial address ” .

In some cases, it is desirable for a linker script to define a symbol only if it is referenced, and only if it is not defined by any object included in the link. For example, traditional linkers defined the symbol “ `etext` ” .However, ANSI C requires that the user be able to use “ `etext` ” as a function name without encountering an error. The `PROVIDE` keyword may be used to define a symbol, such as “ `etext` ” ,only if it is referenced but not defined. The syntax is `PROVIDE(`*`symbol = expression`*`)` .

11.2.7. Arithmetic Functions

The command language includes a number of built-in functions for use in link script expressions.

`ABSOLUTE(`*`exp`*`)`
: Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *`exp`*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section-relative.

`ADDR(`*`section`*`)`
: Return the absolute address of the named *`section`*. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
                                        .output1 :
                                        {
                                        start_of_output_1 = ABSOLUTE(.);
                                        ...
                                        }
                                        .output :
                                        {
                                        symbol_1 = ADDR(.output1);
                                        symbol_2 = start_of_output_1;
                                        }
                                        ... }
```

`LOADADDR(`*`section`*`)`
: Return the absolute load address of the named *`section`*. This is normally the same as `ADDR`, but it may be different if the `AT` keyword is used in the section definition (see Section 11.4.4, “Optional Section Attributes” [116]).

`ALIGN(`*`exp`*`)`
: Return the result of the current location counter (`.`) aligned to the next *`exp`* boundary. *`exp`* must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

ALIGN doesn't change the value of the location counter it just does arithmetic on it. As an example, to align the output .data section to the next 0x2000 byte boundary after the preceding section and to set a variable within the section to the next 0x8000 boundary after the input sections:

```
SECTIONS{ ...
.data ALIGN(0x2000): {
*(.data)
variable = ALIGN(0x8000);
}
... }
```

The first use of ALIGN in this example specifies the location of a section because it is used as the optional *start* attribute of a section definition (see Section 11.4.4, “Optional Section Attributes” [116]). The second use simply defines the value of a variable.

The built-in NEXT is closely related to ALIGN.

DEFINED(*symbol*)

Return 1 if *symbol* is in the linker global symbol table and is defined, otherwise return 0. You can use this function to provide default values for symbols. For example, the following command-file fragment shows how to set a global symbol begin to the first location in the .text section but if a symbol called begin already existed, its value is preserved:

```
SECTIONS{ ...
.text : {
begin = DEFINED(begin) ? begin : . ;
...
}
... }
```

`NEXT(`*`exp`*`)`
: Return the next unallocated address that is a multiple of *`exp`*. This function is closely related to `ALIGN(`*`exp`*`)` ; unless you use the `MEMORY` command to define discontinuous memory for the output file, the two functions are equivalent.

`SIZEOF(`*`section`*`)`
: Return the size in bytes of the named *`section`*, if that section has been allocated. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
                                        .output {
                                        .start = . ;
                                        ...
                                        .end = . ;
                                        }
                                        symbol_1 = .end - .start ;
                                        symbol_2 = SIZEOF(.output);
                                        ... }
```

`SIZEOF_HEADERS` , `sizeof_headers`
: Return the size in bytes of the output file's headers. You can use this number as the start address of the first section, if you choose, to facilitate paging.

`MAX(`*`exp1`*`, `*`exp2`*`)`
: Returns the maximum of *`exp1`* and *`exp2`*.

`MIN(`*`exp1`*`, `*`exp2`*`)`
: Returns the minimum of *`exp1`* and *`exp2`*.

11.2.8. Semicolons

Semicolons (`;`) are required in the following places. In all other places they can appear for esthetic reasons but are otherwise ignored.

`Assignment`
: Semicolons must appear at the end of assignment expressions. See Section 11.2.6, “Assignment: Defining Symbols” [101] .

11.3. Memory Layout

The linker's default configuration permits allocation of all available memory. You can override this configuration by using the `MEMORY` command. The `MEMORY` command describes the location and size of blocks of memory in the target. By using it carefully, you can describe which memory regions may be used by the linker, and which memory regions it must avoid. The linker does not shuffle sections to fit into the available regions, but does move the requested sections into the correct regions and issue errors when the regions become too full.

A command file may contain at most one use of the `MEMORY` command; however, you can define as many blocks of memory within it as you wish. The syntax is:

```
MEMORY
                        {
                        name (attr) : ORIGIN = origin, LENGTH = len
                        ...
                        }
```

`name`
: is a name used internally by the linker to refer to the region. Any symbol name may be used. The region names are stored in a separate name space, and will not conflict with symbols, file names or section names. Use distinct names to specify multiple regions.

`(`*`attr`*`)`
: is an optional list of attributes, permitted for compatibility with the AT&T linker but not used by the linker beyond checking that the attribute list is valid. Valid attribute lists must be made up of the characters "`LIRWX`". If you omit the attribute list, you may omit the parentheses around it as well.

`origin`
: is the start address of the region in physical memory. It is an expression that must evaluate to a constant before memory allocation is performed. The keyword `ORIGIN` may be abbreviated to `org` or `o` (but not, for example, “ `ORG` ”).

len
: is the size in bytes of the region (an expression). The keyword `LENGTH` may be abbreviated to `len` or `l`.

For example, to specify that memory has two regions available for allocation one starting at 0 for 256 kilobytes, and the other starting at `0x40000000` for four megabytes:

```
MEMORY
                        {
                        rom : ORIGIN = 0, LENGTH = 256K
                        ram : org = 0x40000000, l = 4M
                        }
```

Once you have defined a region of memory named *mem*, you can direct specific output sections there by using a command ending in " *>mem* " within the `SECTIONS` command (see Section 11.4.4, "Optional Section Attributes" [116]). If the combined output sections directed to a region are too big for the region, the linker will issue an error message.

11.4. Specifying Output Sections

The `SECTIONS` command controls exactly where input sections are placed into output sections, their order in the output file, and to which output sections they are allocated.

You may use at most one `SECTIONS` command in a script file, but you can have as many statements within it as you wish. Statements within the `SECTIONS` command can do one of three things:

- define the entry point;
- assign a value to a symbol;
- describe the placement of a named output section, and which input sections go into it.

You can also use the first two operations defining the entry point and defining symbols outside the `SECTIONS` command: see Section 11.5, "The Entry Point" [120] and Section 11.2.6, "Assignment: Defining Symbols" [101] .They are permitted here as well for your convenience in reading the script, so that symbols and the entry point can be defined at meaningful points in your output-file layout.

If you do not use a `SECTIONS` command, the linker places each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file.

11.4.1. Section Definitions

The most frequently used statement in the `SECTIONS` command is the *section definition* ,which specifies the properties of an output section: its location, alignment, contents, fill pattern, and target memory region. Most of these specifications are optional; the simplest form of a section definition is

```
SECTIONS { ...
                            secname : {
                            contents
                            }
                            ... }
```

`secname` is the name of the output section, and *`contents`* a specification of what goes there for example, a list of input files or sections of input files (see Section 11.4.2, “Section Placement” [110]). As you might assume, the white-space shown is optional. You do need the colon “ `:` ” and the braces “ `{}` ” ,however.

`secname` must meet the constraints of your output format. In formats that only support a limited number of sections, such as `a.out`, the name must be one of the names supported by the format (`a.out`, for example, allows only `.text`, `.data` or `.bss`). If the output format supports any number of sections, but with numbers and not names (as is the case for Oasys), the name should be supplied as a quoted numeric string. A section name may consist of any sequence of characters, but any name that does not conform to the standard linker symbol name syntax must be quoted. See Section 6.3, “Symbol Names” [32] ,Symbols.

The special *secname* “ `/DISCARD/` ” may be used to discard input sections. Any sections that are assigned to an output section named “ `/DISCARD/` ” are not included in the final link output.

The linker will not create output sections that do not have any contents. This is for convenience when referring to input sections that may or may not exist. For example,

```
.foo { *(.foo) }
```

will only create a “ `.foo` ” section in the output file if there is a “ `.foo` ” section in at least one input file.

11.4.2. Section Placement

In a section definition, you can specify the contents of an output section by listing particular input files, by listing particular input-file sections, or by a combination of the two. You can also place arbitrary data in the section, and define symbols relative to the beginning of the section.

The *`contents`* of a section definition may include any of the following kinds of statement. You can include as many of these as you like in a single section definition, separated from one another by white-space.

`filename`

You may simply name a particular input file to be placed in the current output section; *all* sections from that file are placed in the current section definition. If the file name has already been mentioned in another section definition, with an explicit section name list, then only those sections that have not yet been allocated are used.

To specify a list of particular files by name:

```
.data : { afile.o bfile.o cfile.o }
```

The example also illustrates that multiple statements can be included in the contents of a section definition, since each file name is a separate statement.

`filename( section )` , `filename( section , section, ... )` , `filename( section section ... )`

You can name one or more sections from your input files, for insertion in the current output section. If you wish to specify a list of input-file sections inside the parentheses, you may separate the section names by either commas or white-space.

`* (section)` , `*(section, section, ...)` , `*(section section ...)`

Instead of explicitly naming particular input files in a link control script, you can refer to *all* files from the linker command line: use “ `*` ” instead of a particular file name before the parenthesized input-file section list.

If you have already explicitly included some files by name, “ `*` ” refers to all *remaining* files those whose places in the output file have not yet been defined.

For example, to copy sections `1` through `4` from an Oasys file into the `.text` section of an `a.out` file, and sections `13` and `14` into the `.data` section:

```
SECTIONS {
                                        .text :{
                                        *("1" "2" "3" "4")
                                        }

                                        .data :{
                                        *("13" "14")
                                        }
                                        }
```

“ `[section ... ]` ” used to be accepted as an alternate way to specify named sections from all unallocated input files. Because some operating systems (VMS) allow brackets in file names, that notation is no longer supported.

`filename( COMMON )` , `*( COMMON )`

Specify where in your output file to place uninitialized data with this notation. `*(COMMON)` by itself refers to all uninitialized data from all input files (so far as it is not yet allocated); `filename(COMMON)` refers to uninitialized data from a particular file. Both are special cases of the general mechanisms for specifying where to place input-file sections: the linker permits you to refer to uninitialized data as if it were in an input-file section named `COMMON`, regardless of the input file's format.

In any place where you may use a specific file or section name, you may also use a wild-card pattern. The linker handles wild-cards much as the UNIX shell does. A “ `*` ” character matches any number of characters. A “ `?` ” character matches any single character. The sequence “ `[`*`chars`*`]` ” will match a single instance of any of the *`chars`*; the - character may be used to specify a range of characters, as in “ `[a-z]` ” to match any lower case letter. A “ `\` ” character may be used to quote the following character.

When a file name is matched with a wild-card, the wild-card characters will not match a “ `/` ” character (used to separate directory names on UNIX). A pattern consisting of a single “ `*` ” character is an exception; it will always match any file name. In a section name, the wild-card characters will match a “ `/` ” character.

Wild-cards only match files that are explicitly specified on the command line. The linker does not search directories to expand wild-cards. However, if you specify a simple file name a name with no wild-card characters in a linker script, and the file name is not also specified on the command line, the linker will attempt to open the file as though it appeared on the command line.

In the following example, the command script arranges the output file into three consecutive sections, named `.text`, `.data`, and `.bss`, taking the input for each from the correspondingly named sections of all the input files:

```
SECTIONS {
                            .text : { *(.text) }
                            .data : { *(.data) }
                            .bss :  { *(.bss)  *(COMMON) }
                            }
```

The following example reads all of the sections from file `all.o` and places them at the start of output section `outputa`, which starts at location `0x10000`. All of section `.input1` from file `foo.o` follows immediately, in the same output section. All of section `.input2` from `foo.o` goes into output section `outputb`, followed by section `.input1` from `foo1.o`. All of the remaining `.input1` and `.input2` sections from any files are written to output section `outputc`.

```
SECTIONS {
                            outputa 0x10000 :
                            {
                            all.o
                            foo.o (.input1)
                            }
                            outputb :
                            {
                            foo.o (.input2)
                            foo1.o (.input1)
                            }
                            outputc :
                            {
                            *(.input1)
                            *(.input2)
                            }
                            }
```

This example shows how wild-card patterns might be used to partition files. All `.text` sections are placed in `.text`, and all `.bss` sections are placed in `.bss`. For all files beginning with an upper case character, the `.data` section is placed into `.DATA`; for all other files, the `.data` section is placed into `.data`.

```
SECTIONS {
                            .text : { *(.text) }
                            .DATA : { [A-Z]*(.data) }
                            .data : { *(.data) }
                            .bss : { *(.bss) }
                            }
```

11.4.3. Section Data Expressions

The foregoing statements arrange, in your output file, data originating from your input files. You can also place data directly in an output section from the link command script. Most of these additional statements involve expressions (see Expressions, Section 11.2, “Expressions” [98]). Although these statements are shown separately here for ease of presentation, no such segregation is needed within a section definition in the `SECTIONS` command; you can intermix them freely with any of the statements we've just described.

`CREATE_OBJECT_SYMBOLS`
Create a symbol for each input file in the current section, set to the address of the first byte of data written from that input file. For instance, with `a.out` files it is conventional to have a symbol for each input file. You can accomplish this by defining the output `.text` section as follows:

```
SECTIONS {
                                        .text 0x2020 :
                                        {
                                        CREATE_OBJECT_SYMBOLS
                                        *(.text)
                                        _etext = ALIGN(0x2000);
                                        }
                                        ...
                                        }
```

If `sample.ld` is a file containing this script, and `a.o`, `b.o`, `c.o`, and `d.o` are four input files with contents like the following--

```
/* a.c */

                                        afunction() { }
                                        int adata=1;
                                        int abss;
```

" `ld -M -T sample.ld a.o b.o c.o d.o` " would create a map like this, containing symbols matching the object file names:

```
00000000 A __DYNAMIC
                                        00004020 B _abss
                                        00004000 D _adata
                                        00002020 T _afunction
                                        00004024 B _bbss
                                        00004008 D _bdata
                                        00002038 T _bfunction
                                        00004028 B _cbss
                                        00004010 D _cdata
                                        00002050 T _cfunction
                                        0000402c B _dbss
                                        00004018 D _ddata
                                        00002068 T _dfunction
                                        00004020 D _edata
```

```
00004030 B _end
00004000 T _etext
00002020 t a.o
00002038 t b.o
00002050 t c.o
00002068 t d.o
```

symbol = *expression* ; , *symbol f*= *expression* ;

symbol is any symbol name (see Section 11.2.2, “Symbol Names” [99]). "*f*=" refers to any of the operators `&= += -= *= /=` which combine arithmetic and assignment.

When you assign a value to a symbol within a particular section definition, the value is relative to the beginning of the section (see Section 11.2.6, “Assignment: Defining Symbols” [101]). If you write

```
SECTIONS {
                                        abs = 14 ;
                                        ...
                                        .data : { ... rel = 14 ; ... }
                                        abs2 = 14 + ADDR(.data);
                                        ...
                                        }
```

`abs` and `rel` do not have the same value; `rel` has the same value as `abs2`.

BYTE(*expression*) , SHORT(*expression*) , LONG(*expression*) , QUAD(*expression*)

By including one of these four statements in a section definition, you can explicitly place one, two, four, or eight bytes (respectively) at the current address of that section. `QUAD` is only supported when using a 64-bit host or target.

Multiple-byte quantities are represented in whatever byte order is appropriate for the output file format (see Appendix A, *BFD* [171]).

FILL(*expression*)

Specify the “fill pattern” for the current section. Any otherwise unspecified regions of memory within the section (for example, regions you skip over by assigning a new value to the location counter “ . ”)are filled with the two least significant bytes from the *expression* argument. A **FILL** statement covers memory locations *after* the point it occurs in the section definition; by including more than one **FILL** statement, you can have different fill patterns in different parts of an output section.

11.4.4. Optional Section Attributes

Here is the full syntax of a section definition, including all the optional portions:

```
SECTIONS {
                              ...
                              secname start BLOCK(align) (NOLOAD) : AT ( ldadr
                              { contents } >region :phdr =fill
                              ...
                              }
```

secname and *contents* are required. See Section 11.4.1, “Section Definitions” [109] ,and Section 11.4.2, “Section Placement” [110] ,for details on *contents*. The remaining elements *start*, `BLOCK(`*align*`)` , `(NOLOAD)`, `AT (` *ldadr* `)` , `>`*region* , `:`*phdr* ,and `=`*fill* are all optional.

start
: You can force the output section to be loaded at a specified address by specifying *start* immediately following the section name. *start* can be represented as any expression. The following example generates section *output* at location `0x40000000`:

```
SECTIONS {
                                          ...
                                          output 0x40000000: {
                                          ...
                                          }
                                          ...
                                          }
```

`BLOCK(`*`align`*`)`

You can include **BLOCK()** specification to advance the location counter `.` prior to the beginning of the section, so that the section will begin at the specified alignment. *`align`* is an expression.

`(NOLOAD)`

Use " **(NOLOAD)** " to prevent a section from being loaded into memory each time it is accessed. For example, in the script sample below, the **ROM** segment is addressed at memory location " `0` " and does not need to be loaded into each object file:

```
SECTIONS {
                                        ROM  0  (NOLOAD)  : { ... }
                                        ...
                                        }
```

`AT (` *`ldadr`* `)`

The expression *`ldadr`* that follows the **AT** keyword specifies the load address of the section. The default (if you do not use the **AT** keyword) is to make the load address the same as the relocation address. This feature is designed to make it easy to build a ROM image. For example, this **SECTIONS** definition creates two output sections: one called " `.text` " , which starts at `0x1000`, and one called " `.mdata` " ,which is loaded at the end of the " `.text` " section even though its relocation address is `0x2000`. The symbol `_data` is defined with the value `0x2000`:

```
SECTIONS
                                        {
                                        .text 0x1000 : { *(.text) _etext = . ; }
                                        .mdata 0x2000 :
                                        AT ( ADDR(.text) + SIZEOF ( .text ) )
                                        { _data = . ; *(.data); _edata = . ;  }
                                        .bss 0x3000 :
                                        { _bstart = . ;  *(.bss) *(COMMON) ; _benc
                                        }
```

The run-time initialization code (for C programs, usually `crt0`) for use with a ROM generated this way has to include something like the following, to copy the initialized data from the ROM image to its runtime address:

```
char *src = _etext;
                                        char *dst = _data;

                                        /* ROM has data at end of text; copy
                                        while (dst < _edata) {
                                        *dst++ = *src++;
                                        }

                                        /* Zero bss */
                                        for (dst = _bstart; dst< _bend; dst++
                                        *dst = 0;
```

`>`*`region`*
: Assign this section to a previously defined region of memory. See Section 11.3, “Memory Layout” [107] .

`=`*`fill`*
: Including `=`*`fill`* in a section definition specifies the initial fill value for that section. You may use any expression to specify *`fill`*. Any unallocated holes in the current output section when written to the output file will be filled with the two least significant bytes of the value, repeated as necessary. You can also change the fill value with a **FILL** statement in the *`contents`* of a section definition.

11.4.5. Overlays

The **OVERLAY** command provides an easy way to describe sections that are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the runtime memory address as required, perhaps by simply manipulating addressing bits. This approach can be useful, for example, when a certain region of memory is faster than another.

The **OVERLAY** command is used within a **SECTIONS** command. It appears as follows:

```
OVERLAY start : [ NOCROSSREFS ] AT ( ldaddr )
{
secname1 { contents } :phdr =fill
secname2 { contents } :phdr =fill
...
} >region :phdr =fill
```

Everything is optional except **OVERLAY** (a keyword), and each section must have a name (*secname1* and *secname2* above). The section definitions within the **OVERLAY** construct are identical to those within the general **SECTIONS** construct (see Section 11.4, “Specifying Output Sections” [108]), except that no addresses and no memory regions may be defined for sections within an **OVERLAY**.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the **OVERLAY** as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to `.`).

If the **NOCROSSREFS** keyword is used, and there any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another. See Section 11.7, “Option Commands” [125] .

For each section within the `OVERLAY`, the linker automatically defines two symbols. The symbol `__load_start_`*secname* is defined as the starting load address of the section. The symbol `__load_stop_`*secname* is defined as the final load address of the section. Any characters within *secname* that are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of `.` is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a `SECTIONS` construct.

```
OVERLAY 0x1000 : AT (0x4000)
                          {
                          .text0 { o1/*.o(.text) }
                          .text1 { o2/*.o(.text) }
                          }
```

This will define both `.text0` and `.text1` to start at address 0x1000. `.text0` will be loaded at address 0x4000, and `.text1` will be loaded immediately after `.text0`. The following symbols will be defined: `__load_start_text0`, `__load_stop_text0`, `__load_start_text1`, `__load_stop_text1`.

C code to copy overlay `.text1` into the overlay area might look like the following.

```
extern char __load_start_text1, __load_stop_text1;
                          memcpy ((char *) 0x1000, &__load_start_text1,
                          &__load_stop_text1 - &__load_start_text1);
```

Note that the **OVERLAY** command is just syntactic sugar, since everything it does can be done using the more basic commands. The above example could have been written identically as follows.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
                          __load_start_text0 = LOADADDR (.text0);
                          __load_stop_text0 = LOADADDR (.text0) + SIZEOF (.
                          .text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o
                          __load_start_text1 = LOADADDR (.text1);
                          __load_stop_text1 = LOADADDR (.text1) + SIZEOF (.
                          . = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1
```

11.5. The Entry Point

The linker command language includes a command specifically for defining the first executable instruction in an output file (its *entry point*). Its argument is a symbol name:

```
ENTRY(symbol)
```

Like symbol assignments, the **ENTRY** command may be placed either as an independent command in the command file, or among the section definitions within the **SECTIONS** command whatever makes the most sense for your layout.

ENTRY is only one of several ways of choosing the entry point. You may indicate it in any of the following ways (shown in descending order of priority: methods higher in the list override methods lower down).

- the **-e** `entry` command-line option;
- the `ENTRY(symbol)` command in a linker control script;
- the value of the symbol `start`, if present;
- the address of the first byte of the `.text` section, if present;
- The address `0x00000000`.

For example, you can use these rules to generate an entry point with an assignment statement: if no symbol start is defined within your input files, you can simply define it, assigning it an appropriate value:

```
start = 0x2020;
```

The example shows an absolute address, but you can use any expression. For example, if your input object files use some other symbol-name convention for the entry point, you can just assign the value of whatever symbol contains the start address to start:

```
start = other_symbol ;
```

11.6. Version Script

The linker command script includes a command specifically for specifying a version script, and is only meaningful for ELF platforms that support shared libraries. A version script can be build directly into the linker script that you are using, or you can supply the version script as just another input file to the linker at the time that you link. The command script syntax is:

```
VERSION { version script contents }
```

The version script can also be specified to the linker by means of the **--version-script** linker command line option. Version scripts are only meaningful when creating shared libraries.

The format of the version script itself is identical to that used by Sun's linker in Solaris 2.5. Versioning is done by defining a tree of version nodes with the names and interdependencies specified in the version script. The version script can specify which symbols are bound to which version nodes, and it can reduce a specified set of symbols to local scope so that they are not globally visible outside of the shared library.

The easiest way to demonstrate the version script language is with a few examples.

```
VERS_1.1 {
                        global:
                        foo1;
                        local:
                        old*;
                        original*;
                        new*;
                        };

                        VERS_1.2 {
                        foo2;
                        } VERS_1.1;

                        VERS_2.0 {
                        bar1; bar2;
                        } VERS_1.2;
```

In this example, three version nodes are defined. VERS_1.1 is the first version node defined, and has no other dependencies. The symbol foo1 is bound to this version node, and a number of symbols that have appeared within various object files are reduced in scope to local so that they are not visible outside of the shared library.

Next, the node VERS_1.2 is defined. It depends upon VERS_1.1. The symbol foo2 is bound to this version node.

Finally, the node VERS_2.0 is defined. It depends upon VERS_1.2. The symbols bar1 and bar2 are bound to this version node.

Symbols defined in the library that aren't specifically bound to a version node are effectively bound to an unspecified base version of the library. It is possible to bind all otherwise unspecified symbols to a given version node using global: * somewhere in the version script.

Lexically the names of the version nodes have no specific meaning other than what they might suggest to the person reading them. The 2.0 version could just as well have appeared in between 1.1 and 1.2. However, this would be a confusing way to write a version script.

When you link an application against a shared library that has version-ed symbols, the application itself knows which version of each symbol it requires, and it also knows which version nodes it needs from each shared library it is linked against. Thus at runtime, the dynamic loader can make a quick check to make sure that the libraries you have linked against do in fact supply all of the version nodes that the application will need to resolve all of the dynamic symbols. In this way it is possible for the dynamic linker to know with certainty that all external symbols that it needs will be resolvable without having to search for each symbol reference.

The symbol versioning is in effect a much more sophisticated way of doing minor version checking that Sun-OS does. The fundamental problem that is being addressed here is that typically references to external functions are bound on an as-needed basis, and are not all bound when the application starts up. If a shared library is out of date, a required interface may be missing; when the application tries to use that interface, it may suddenly and unexpectedly fail. With symbol versioning, the user will get a warning when they start their program if the libraries being used with the application are too old.

There are several GNU extensions to Sun's versioning approach. The first of these is the ability to bind a symbol to a version node in the source file where the symbol is defined instead of in the versioning script. This was done mainly to reduce the burden on the library maintainer. This can be done by putting something like:

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

in the C source file. This renamed the function original_foo to be an alias for foo bound to the version node VERS_1.1. The local: directive can be used to prevent the symbol original_foo from being exported.

The second GNU extension is to allow multiple versions of the same function to appear in a given shared library. In this way an incompatible change to an interface can take place without increasing the major version number of the shared library, while still allowing applications linked against the old interface to continue to function.

This can only be accomplished by using multiple .symver directives in the assembler. An example of this would be:

```
__asm__(".symver original_foo,foo@");
                        __asm__(".symver old_foo,foo@VERS_1.1");
                        __asm__(".symver old_foo1,foo@VERS_1.2");
                        __asm__(".symver new_foo,foo@@VERS_2.0");
```

In this example, foo@ represents the symbol foo bound to the unspecified base version of the symbol. The source file that contains this example would define 4 C functions: original_foo, old_foo, old_foo1, and new_foo.

When you have multiple definitions of a given symbol, there needs to be some way to specify a default version to which external references to this symbol will be bound. This can be accomplished with the foo@@VERS_2.0 type of .symver directive. Only one version of a symbol can be declared 'default' in this manner - otherwise you would effectively have multiple definitions of the same symbol.

If you wish to bind a reference to a specific version of the symbol within the shared library, you can use the aliases of convenience (that is old_foo), or you can use the .symver directive to specifically bind to an external version of the function in question.

11.7. Option Commands

The command language includes a number of other commands that you can use for specialized purposes. They are similar in purpose to command-line options.

CONSTRUCTORS

When linking using the `a.out` object file format, the linker uses an unusual set construct to support C++ global constructors and destructors. When linking object file formats that do not support arbitrary sections, such as **ECOFF** and **XCOFF**, the linker will automatically recognize C++ global constructors and destructors by name. For these object file formats, the **CONSTRUCTORS** command tells the linker where this information should be placed. The **CONSTRUCTORS** command is ignored for other object file formats.

The symbol __CTOR_LIST__ marks the start of the global constructors, and the symbol __DTOR_LIST marks the end. The first word in the list is the number of entries, followed by the address of each constructor or destructor, followed by a zero word. The compiler must arrange to actually run the code. For these object file formats *xgc* C++ calls constructors from a subroutine __main; a call to __main is automatically inserted into the startup code for main. *xgc* C++ runs destructors either by using atexit, or directly from the function exit.

For object file formats such as COFF or ELF that support multiple sections, XGC C++ will normally arrange to put the addresses of global constructors and destructors into the .ctors and .dtors sections. Placing the following sequence into your linker script will build the sort of table that the C++ runtime code expects to see.

```
      __CTOR_LIST__ = .;
                              LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
                              *(.ctors)
                              LONG(0)
                              __CTOR_END__ = .;
                              __DTOR_LIST__ = .;
                              LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
                              *(.dtors)
                              LONG(0)
```

```
                                        __DTOR_END__ = .;
```

Normally the compiler and linker will handle these issues automatically, and you will not need to concern yourself with them. However, you may need to consider this if you are using C++ and writing your own linker scripts.

FLOAT , **NOFLOAT**
These keywords were used in some older linkers to request a particular math subroutine library. The linker doesn't use the keywords, assuming instead that any necessary subroutines are in libraries specified using the general mechanisms for linking to archives; but to permit the use of scripts that were written for the older linkers, the keywords **FLOAT** and **NOFLOAT** are accepted and ignored.

FORCE_COMMON_ALLOCATION
This command has the same effect as the **-d** command-line option: to make the linker assign space to common symbols even if a relocatable output file is specified (**-r**).

INCLUDE *`filename`*
Include the linker script *`filename`* at this point. The file will be searched for in the current directory, and in any directory specified with the **-L** option. You can nest calls to **INCLUDE** up to 10 levels deep.

INPUT (*`file`*, *`file`*, ...) , **INPUT** (*`file`* *`file`* ...)
Use this command to include binary input files in the link, without including them in a particular section definition. Specify the full name for each *`file`*, including “ `.a` ” if required.

The linker searches for each *`file`* through the archive-library search path, just as for files you specify on the command line. See the description of **-L** in Section 10.1, “Command Line Options” [79] .

If you use “ **-l** *`file`* ” ,the linker will transform the name to `lib`*`file`*`.a` as with the command line argument **-l**.

GROUP (*`file`*, *`file`*, ...) , **GROUP** (*`file`* *`file`* ...)
This command is like **INPUT**, except that the named files should all be archives, and they are searched repeatedly until

no new undefined references are created. See the description of **-(** in Section 10.1, “Command Line Options” [79] .

OUTPUT (`filename`)
Use this command to name the link output file `filename`. The effect of **OUTPUT(`filename`)** is identical to the effect of “ `-o filename` ” ,which overrides it. You can use this command to supply a default output-file name other than `a.out`.

OUTPUT_ARCH (`bfdname`)
Specify a particular output machine architecture, with one of the names used by the BFD back-end routines (see Appendix A, *BFD* [171]). This command is often unnecessary; the architecture is most often set implicitly by either the system BFD configuration or as a side effect of the **OUTPUT_FORMAT** command.

OUTPUT_FORMAT (`bfdname`)
When the linker is configured to support multiple object code formats, you can use this command to specify a particular output format. `bfdname` is one of the names used by the BFD back-end routines (see Appendix A, *BFD* [171]). The effect is identical to the effect of the **--oformat** command-line option. This selection affects only the output file; the related command **PREFIX** affects primarily input files.

SEARCH_DIR (`path`)
Add `path` to the list of paths where the linker looks for archive libraries. **SEARCH_DIR(`path`)** has the same effect as “ **-L** `path` ” on the command line.

STARTUP (`filename`)
Ensure that `filename` is the first input file used in the link process.

PREFIX (`format`)
When the linker is configured to support multiple object code formats, you can use this command to change the input-file object code format (like the command-line option **-b** or its synonym **--format**). The argument `format` is one of the strings used by BFD to name binary formats. If **PREFIX** is specified but **OUTPUT_FORMAT** is not, the last **PREFIX** argument is also used as the default format for the the linker output file. See Appendix A, *BFD* [171] .

If you don't use the **PREFIX** command, the linker uses the value of the environment variable GNUPREFIX, if available, to select the output file format. If that variable is also absent, the linker uses the default format configured for your machine in the BFD libraries.

NOCROSSREFS (***`section section`*** **...)**

This command may be used to tell the linker to issue an error about any references among certain sections.

In certain types of programs, particularly on embedded systems, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors. For example, it would be an error if code in one section called a function defined in the other section.

The **NOCROSSREFS** command takes a list of section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. The **NOCROSSREFS** command uses output section names, defined in the **SECTIONS** command. It does not use the names of input sections.

III

Using the Object Code Utilities

Chapter 12 The Utilities

12.1. ar

```
prefix-ar [-]p[mod [relpos]] archive [member...]
prefix-ar -M [ <mri-script> ]
```

The **ar** program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), time-stamp, owner, and group are preserved in the archive, and can be restored on extraction.

ar can maintain archives whose members have names of any length; however, depending on how **ar** is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to a.out) or 16 characters (typical of formats related to COFF).

ar is considered a binary utility because archives of this sort are most often used as *libraries* holding commonly needed subroutines.

ar creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier “ **s** ” .Once created, this index is updated in the archive whenever **ar** makes a change to its contents (save for the “ **q** ” update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use “ **nm -s** ” or “ **nm --print-armap** ” to list this index table. If an archive lacks the table, another form of **ar** called **ranlib** can be used to add just the table.

ar is designed to be compatible with two different facilities. You can control its activity using command-line options, like the different varieties of **ar** on UNIX systems; or, if you specify the single command-line option “ **-M** ” ,you can control it with a script supplied via standard input, like the MRI "librarian" program.

12.1.1. Controlling ar on the command line

```
prefix-ar [-]p[mod [relpos]] archive [member...]
```

When you use **ar** in the UNIX style, **ar** insists on at least two arguments to execute: one key letter specifying the *operation* (optionally accompanied by other key-letters specifying *modifiers*), and the archive name to act on.

Most operations can also accept further `member` arguments, specifying particular files to operate on.

ar allows you to mix the operation code `p` and modifier flags `mod` in any order, within the first command-line argument.

If you wish, you may begin the first command-line argument with a dash.

The `p` key-letter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

d

Delete modules from the archive. Specify the names of modules to be deleted as `member`...; the archive is untouched if you specify no files to delete.

If you specify the “ **v** ” modifier, **ar** lists each module as it is deleted.

m

Use this operation to *move* members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with **m**, any members you name in the `member` arguments are moved to the *end* of the archive; you can use the “ **a** ” , “ **b** ” ,or “ **i** ” modifiers to move them to a specified place instead.

p

Print the specified members of the archive, to the standard output file. If the “ **v** ” modifier is specified, show the member name before copying its contents to standard output.

If you specify no `member` arguments, all the files in the archive are printed.

q

Quick append; add the files `member...` to the end of `archive`, without checking for replacement.

The modifiers “ **a** ” , “ **b** ” ,and “ **i** ” do *not* affect this operation; new members are always placed at the end of the archive.

The modifier “ **v** ” makes **ar** list each file as it is appended.

Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use “ **ar s** ” or **ranlib** explicitly to update the symbol table index.

r

Insert the files `member`... into `archive` (with *replacement*). This operation differs from “ **q** ” in that any previously existing members are deleted if their names match those being added.

If one of the files named in `member`... does not exist, **ar** displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the file; but you may use one of the modifiers " **a** " , " **b** " ,or " **i** " to request placement relative to some existing member.

The modifier " **v** " used with this operation elicits a line of output for each file inserted, along with one of the letters " **a** " or " **r** " to indicate whether the file was appended (no old member deleted) or replaced.

t

Display a *table* listing the contents of `archive`, or those of the files listed in `member`... that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), times-tamp, owner, group, and size, you can request that by also specifying the " **v** " modifier.

If you do not specify a `member`, all files in the archive are listed.

If there is more than one file with the same name (say, " **fie** ")in an archive (say " **b.a** "), " **ar t b.a fie** " lists only the first instance; to see them all, you must ask for a complete listing in our example, " **ar t b.a** " .

x

Extract members (named `member`) from the archive. You can use the " **v** " modifier with this operation, to request that **ar** list each name as it extracts it.

If you do not specify a `member`, all files in the archive are extracted.

A number of modifiers (`mod`) may immediately follow the `p` key-letter, to specify variations on an operation's behavior:

a

Add new files *after* an existing member of the archive. If you use the modifier " **a** " ,the name of an existing archive member must be present as the `relpos` argument, before the `archive` specification.

b

Add new files *before* an existing member of the archive. If you use the modifier “ **b** ” ,the name of an existing archive member must be present as the `relpos` argument, before the `archive` specification. (same as “ **i** ”).

c

Create the archive. The specified `archive` is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

f

Truncate names in the archive. **ar** will normally permit file names of any length. This will cause it to create archives which are not compatible with the native **ar** program on some systems. If this is a concern, the “ **f** ” modifier may be used to truncate file names when putting them in the archive.

i

Insert new files *before* an existing member of the archive. If you use the modifier “ **i** ” ,the name of an existing archive member must be present as the `relpos` argument, before the `archive` specification. (same as “ **b** ”).

l

This modifier is accepted but not used.

o

Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

s

Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running “ **ar s** ” on an archive is equivalent to running “ **ranlib** ” on it.

u

Normally, “ **ar r** ” ... inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The “ **u** ” modifier is allowed only for the operation

“ **r** ” (replace). In particular, the combination “ **qu** ” is not allowed, since checking the time stamps would lose any speed advantage from the operation “ **q** ” .

v

This modifier requests the *verbose* version of an operation. Many operations display additional information, such as filenames processed, when the modifier “ **v** ” is appended.

V

This modifier shows the version number of **ar**.

12.1.2. Controlling ar with a script

```
prefix-ar -M [ <script ]
```

If you use the single command-line option “ **-M** ” with **ar**, you can control its operation with a rudimentary command language. This form of **ar** operates interactively if standard input is coming directly from a terminal. During interactive use, **ar** prompts for input (the prompt is “ **AR >** ”), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and **ar** abandons execution (with a nonzero exit code) on any error.

The **ar** command language is *not* designed to be equivalent to the command-line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to **ar** for developers who already have scripts written for the MRI "librarian" program.

The syntax for the **ar** command language is straightforward:

- commands are recognized in upper or lower case; for example, **LIST** is the same as **list**. In the following descriptions, commands are shown in upper case for clarity.
- a single command may appear on each line; it is the first word on the line.
- empty lines are allowed, and have no effect.

- comments are allowed; text after either of the characters “ ***** ” or “ **;** ” is ignored.
- Whenever you use a list of names as part of the argument to an **ar** command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- “ **+** ” is used as a line continuation character; if “ **+** ” appears at the end of a line, the text on the following line is considered part of the current command.

Here are the commands you can use in **ar** scripts, or when using **ar** interactively. Three of them have special significance:

OPEN or **CREATE** specify a *current archive*, which is a temporary file required for most of the other commands.

SAVE commits the changes so far specified by the script. Prior to **SAVE**, commands affect only the temporary copy of the current archive.

ADDLIB *`archive`* , **ADDLIB *`archive`* (*`module`*, *`module`*, ... *`module`*)**
: Add all the contents of *`archive`* (or, if specified, each named *`module`* from *`archive`*) to the current archive.

  Requires prior use of **OPEN** or **CREATE**.

ADDMOD *`member`*, *`member`*, ... *`member`*
: Add each named *`member`* as a module in the current archive.

  Requires prior use of **OPEN** or **CREATE**.

CLEAR
: Discard the contents of the current archive, canceling the effect of any operations since the last **SAVE**. May be executed (with no effect) even if no current archive is specified.

CREATE *`archive`*
: Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as *`archive`* until you use **SAVE**. You can overwrite existing archives; similarly, the contents of any existing file named *`archive`* will not be destroyed until **SAVE**.

DELETE ***`module`*, *`module`*, ... *`module`***

Delete each listed *`module`* from the current archive; equivalent to “ **ar -d *`archive`* *`module`* ... *`module`*** ” .

Requires prior use of **OPEN** or **CREATE**.

DIRECTORY *`archive`* (*`module`*, ... *`module`*) , **DIRECTORY *`archive`* (*`module`*, ... *`module`*) *`outputfile`***

List each named *`module`* present in *`archive`*. The separate command **VERBOSE** specifies the form of the output: when verbose output is off, output is like that of “ **ar -t *`archive`* *`module`*...** ” .When verbose output is on, the listing is like “ **ar -tv *`archive`* *`module`*...** ” .

Output normally goes to the standard output stream; however, if you specify *`outputfile`* as a final argument, **ar** directs the output to that file.

END

Exit from **ar**, with a **0** exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last **SAVE** command, those changes are lost.

EXTRACT *`module`*, *`module`*, ... *`module`*

Extract each named *`module`* from the current archive, writing them into the current directory as separate files. Equivalent to “ **ar -x *`archive`* *`module`*...** ” .

Requires prior use of **OPEN** or **CREATE**.

LIST

Display full contents of the current archive, in “verbose” style regardless of the state of **VERBOSE**. The effect is like “ **ar tv *`archive`*** ”).

Requires prior use of **OPEN** or **CREATE**.

OPEN *`archive`*

Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect *`archive`* until you next use **SAVE**.

REPLACE *module*, *module*, ... *module*

In the current archive, replace each existing *module* (named in the **REPLACE** arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist.

Requires prior use of **OPEN** or **CREATE**.

VERBOSE

Toggle an internal flag governing the output from **DIRECTORY**. When the flag is on, **DIRECTORY** output matches output from “ **ar -tv** ” ....

SAVE

Commit your changes to the current archive, and actually save it as a file with the name specified in the last **CREATE** or **OPEN** command.

Requires prior use of **OPEN** or **CREATE**.

12.2. nm

```
prefix-nm [ -a | --debug-syms ]  [ -g | --extern-only ]
[ -B ]  [ -C | --demangle ] [ -D | --dynamic ]
[ -s | --print-armap ]  [ -A | -o | --print-file-name ]
[ -n | -v | --numeric-sort ]  [ -p | --no-sort ]
[ -r | --reverse-sort ]  [ --size-sort ] [ -u | --undefine
[ -t radix | --radix=radix ] [ -P | --portability ]
[ --target=bfdname ] [ -f format | --format=format ]
[ --defined-only ] [-l | --line-numbers ]
[ --no-demangle ] [ -V | --version ]  [ --help ]  [ objfil
```

nm lists the symbols from object files *objfile*.... If no object files are listed as arguments, **nm** assumes **a.out**.

For each symbol, **nm** shows:

- The symbol value, in the radix selected by options (see below), or hexadecimal by default.

- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

  A
  : The symbol's value is absolute, and will not be changed by further linking.

  B
  : The symbol is in the uninitialized data section (known as BSS).

  C
  : The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references. For more details on common symbols, see the discussion of -warn-common in Section 10.1, “Command Line Options” [79] .

  D
  : The symbol is in the initialized data section.

  G
  : The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.

  I
  : The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.

  N
  : The symbol is a debugging symbol.

  R
  : The symbol is in a read only data section.

  S
  : The symbol is in an uninitialized data section for small objects.

T
The symbol is in the text (code) section.

U
The symbol is undefined.

W
The symbol is weak. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

-
The symbol is a stabs symbol in an a.out object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are used to hold debugging information; for more information, see *the stabs debug format* ,included with the debugger source files.

?
The symbol type is unknown, or object file format specific.

- The symbol name.

The long and short forms of options, shown here as alternatives, are equivalent.

-A , **-o** , **--print-file-name**
Precede each symbol by the name of the input file (or archive element) in which it was found, rather than identifying the input file once only, before all of its symbols.

-a , **--debug-syms**
Display all symbols, even debugger-only symbols; normally these are not listed.

-B
The same as “ **--format=bsd** ” (for compatibility with the MIPS **nm**).

-C , **--demangle**
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See

Section 12.9, “c++filt” [159] ,for more information on demangling.

--no-demangle
Do not demangle low-level symbol names. This is the default.

-D , **--dynamic**
Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

-f ***`format`*** , **--format=*`format`***
Use the output format *`format`*, which can be **bsd**, **sysv**, or **POSIX**. The default is **bsd**. Only the first character of *`format`* is significant; it can be either upper or lower case.

-g , **--extern-only**
Display only external symbols.

-l , **--line-numbers**
For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry which refers to the symbol. If line number information can be found, print it after the other symbol information.

-n , **-v** , **--numeric-sort**
Sort symbols numerically by their addresses, rather than alphabetically by their names.

-p , **--no-sort**
Do not bother to sort the symbols in any order; print them in the order encountered.

-P , **--portability**
Use the POSIX.2 standard output format instead of the default format. Equivalent to “ **-f posix** ” .

-s , **--print-armap**
When listing symbols from archive members, include the index: a mapping (stored in the archive by **ar** or **ranlib**) of which modules contain definitions for which names.

-r , **--reverse-sort**
Reverse the order of the sort (whether numeric or alphabetic); let the last come first.

--size-sort
Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.

-t ***radix*** , **--radix=*radix***
Use *radix* as the radix for printing the symbol values. It must be " **d** " for decimal, " **o** " for octal, or " **x** " for hexadecimal.

--target=*bfdname*
Specify an object code format other than your system's default format. See Section 13.1, "Target Selection" [163] ,for more information.

-u , **--undefined-only**
Display only undefined symbols (those external to each object file).

--defined-only
Display only defined symbols for each object file.

-V , **--version**
Show the version number of **nm** and exit.

--help
Show a summary of the options to **nm** and exit.

12.3. objcopy

```
prefix-objcopy [ -F bfdname | --target=bfdname ]
[ -I bfdname | --input-target=bfdname ]
[ -O bfdname | --output-target=bfdname ]
[ -S | --strip-all ]  [ -g | --strip-debug ]
[ -K symbolname | --keep-symbol=symbolname ]
[ -N symbolname | --strip-symbol=symbolname ]
[ -x | --discard-all ]  [ -X | --discard-locals ]
[ -b byte | --byte=byte ]
[ -i interleave | --interleave=interleave ]
[ -R sectionname | --remove-section=sectionname ]
[ -p | --preserve-dates ] [ --debugging ]
[ --gap-fill=val ] [ --pad-to=address ]
[ --set-start=val ] [ --adjust-start=incr ]
[ --adjust-vma=incr ]
[ --adjust-section-vma=section{=,+,-}val ]
[ --adjust-warnings ] [ --no-adjust-warnings ]
[ --set-section-flags=section=flags ]
[ --add-section=sectionname=filename ]
[ --change-leading-char ] [ --remove-leading-char ]
[ --weaken ]
[ -v | --verbose ] [ -V | --version ]  [ --help ]
infile [outfile]
```

The **objcopy** utility copies the contents of an object file to another. **objcopy** uses the BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of **objcopy** is controlled by command-line options.

objcopy creates temporary files to do its translations and deletes them afterward. **objcopy** uses `bfd` to do all its translation work; it has access to all the formats described in `bfd` and thus is able to recognize most formats without being told explicitly. See Appendix A, *BFD* [171] .

objcopy can be used to generate S-records by using an output target of “ **srec** ” (e.g., use “ **-O srec** ”).

objcopy can be used to generate a raw binary file by using an output target of “ **binary** ” (e.g., use “ **-O binary** ”). When **objcopy** generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the load address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use “ **-S** ” to remove sections containing debugging information. In some cases “ **-R** ” will be useful to remove sections which contain information which is not needed by the binary file.

infile **,** ***outfile***
: The source and output files, respectively. If you do not specify *`outfile`*, **objcopy** creates a temporary file and destructively renames the result with the name of *`infile`*.

-I *`bfdname`* , --input-target=*`bfdname`*
: Consider the source file's object format to be *`bfdname`*, rather than attempting to deduce it. See Section 13.1, “Target Selection” [163] ,for more information.

-O *`bfdname`* , --output-target=*`bfdname`*
: Write the output file using the object format *`bfdname`*. See Section 13.1, “Target Selection” [163] ,for more information.

-F *`bfdname`* , --target=*`bfdname`*
: Use *`bfdname`* as the object format for both the input and the output file; that is, simply transfer data from source to destination with no translation. See Section 13.1, “Target Selection” [163] ,for more information.

-R *`sectionname`* , --remove-section=*`sectionname`*
: Remove any section named *`sectionname`* from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.

-S , --strip-all
: Do not copy relocation and symbol information from the source file.

-g , --strip-debug
: Do not copy debugging symbols from the source file.

--strip-unneeded
: Strip all symbols that are not needed for relocation processing.

-K *`symbolname`* , --keep-symbol=*`symbolname`*
: Copy only symbol *`symbolname`* from the source file. This option may be given more than once.

-N *`symbolname`* , --strip-symbol=*`symbolname`*
: Do not copy symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than **-K**.

-x , --discard-all
: Do not copy non-global symbols from the source file.

-X , **--discard-locals**
Do not copy compiler-generated local symbols. (These usually start with “ **L** ” or “ **.** ” .)

-b ***byte*** , **--byte=*byte***
Keep only every *byte*th byte of the input file (header data is not affected). *byte* can be in the range from 0 to *interleave*-1, where *interleave* is given by the “ **-i** ” or “ **--interleave** ” option, or the default of 4. This option is useful for creating files to program *rom*. It is typically used with an **srec** output target.

-i ***interleave*** , **--interleave=*interleave***
Only copy one out of every *interleave* bytes. Select which byte to copy with the *-b* or “ **--byte** ” option. The default is 4. **objcopy** ignores this option if you do not specify either “ **-b** ” or “ **--byte** ” .

-p , **--preserve-dates**
Set the access and modification dates of the output file to be the same as those of the input file.

--debugging
Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

--gap-fill *val*
Fill gaps between sections with *val*. This is done by increasing the size of the section with the lower address, and filling in the extra space created with *val*.

--pad-to *address*
Pad the output file up to the virtual address *address*. This is done by increasing the size of the last section. The extra space is filled in with the value specified by “ **--gap-fill** ” (default zero).

--set-start *val*
Set the address of the new file to *val*. Not all object file formats support setting the start address.

--adjust-start *incr*
Adjust the start address by adding *incr*. Not all object file formats support setting the start address.

--adjust-vma *`incr`*
Adjust the address of all sections, as well as the start address, by adding *`incr`*. Some object file formats do not permit section addresses to be changed arbitrarily. Note that this does not relocate the sections; if the program expects sections to be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

--adjust-section-vma *`section`*{=,+,-}*`val`*
Set or adjust the address of the named *`section`*. If “ **=** ” is used, the section address is set to *`val`*. Otherwise, *`val`* is added to or subtracted from the section address. See the comments under “ **--adjust-vma** ” ,above. If *`section`* does not exist in the input file, a warning will be issued, unless “ **--no-adjust-warnings** ” is used.

--adjust-warnings
If “ **--adjust-section-vma** ” is used, and the named section does not exist, issue a warning. This is the default.

--no-adjust-warnings
Do not issue a warning if “ **--adjust-section-vma** ” is used, even if the named section does not exist.

--set-section-flags *`section`*=*`flags`*
Set the flags for the named section. The *`flags`* argument is a comma separated string of flag names. The recognized names are “ **alloc** ” , “ **load** ” , “ **readonly** ” , “ **code** ” , “ **data** ” ,and “ **rom** ” .Not all flags are meaningful for all object file formats.

--add-section *`sectionname`*=*`filename`*
Add a new section named *`sectionname`* while copying the file. The contents of the new section are taken from the file *`filename`*. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

--change-leading-char
Some object file formats use special characters at the start of symbols. The most common such character is underscore, which compilers often add before every symbol. This option tells **objcopy** to change the leading character of every symbol when it converts between object file formats. If the object file formats use the same leading character, this option has no

effect. Otherwise, it will add a character, or remove a character, or change a character, as appropriate.

--remove-leading-char
: If the first character of a global symbol is a special symbol leading character used by the object file format, remove the character. The most common symbol leading character is underscore. This option will remove a leading underscore from all global symbols. This can be useful if you want to link together objects of different file formats with different conventions for symbol names. This is different from **--change-leading-char** because it always changes the symbol name when appropriate, regardless of the object file format of the output file.

--weaken
: Change all global symbols in the file to be weak. This can be useful when building an object which will be linked against other objects using the **-R** option to the linker. This option is only effective when using an object file format which supports weak symbols.

-V , **--version**
: Show the version number of **objcopy**.

-v , **--verbose**
: Verbose output: list all object files modified. In the case of archives, " **objcopy -V** " lists all members of the archive.

--help
: Show a summary of the options to **objcopy**.

12.4. objdump

```
prefix-objdump [ -a | --archive-headers ]
[ -b bfdname | --target=bfdname ] [ --debugging ]
[ -C | --demangle ] [ -d | --disassemble ]
[ -D | --disassemble-all ] [ --disassemble-zeroes ]
[ -EB | -EL | --endian={big | little } ]
[ -f | --file-headers ]
[ -h | --section-headers | --headers ]  [ -i | --info
[ -j section | --section=section ]
[ -l | --line-numbers ] [ -S | --source ]
[ -m machine | --architecture=machine ]
[ -r | --reloc ] [ -R | --dynamic-reloc ]
[ -s | --full-contents ]  [ --stabs ]
[ -t | --syms ] [ -T | --dynamic-syms ] [ -x | --all-heade
[ -w | --wide ] [ --start-address=address ]
[ --stop-address=address ]
[ --prefix-addresses] [ --[no-]show-raw-insn ]
[ --adjust-vma=offset ]
[ --version ]  [ --help ]
objfile...
```

objdump displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

`objfile`... are the object files to be examined. When you specify archives, **objdump** shows information on each of the member object files.

The long and short forms of options, shown here as alternatives, are equivalent. At least one option besides “ **-l** ” must be given.

-a , **--archive-header**
: If any of the `objfile` files are archives, display the archive header information (in a format similar to “ **ls -l** ”). Besides the information you could list with “ **ar tv** ” , “ **objdump -a** ” shows the object file format of each archive member.

--adjust-vma=*`offset`*
: When dumping information, first add `offset` to all the section addresses. This is useful if the section addresses do not correspond to the symbol table, which can happen when putting sections at particular addresses when using a format which can not represent section addresses, such as a.out.

-b *`bfdname`* , **--target=*`bfdname`***
: Specify that the object-code format for the object files is `bfdname`. This option may not be necessary; `objdump` can automatically recognize many formats.

For example,

```
prefix-objdump -b oasys -m vax -h fu.o
```

displays summary information from the section headers (“ **-h** ”)of **fu.o**, which is explicitly identified (“ **-m** ”)as a VAX object file in the format produced by Oasys compilers. You can list the formats available with the “ **-i** ” option. See Section 13.1, “Target Selection” [163] ,for more information.

-C , **--demangle**
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See Section 12.9, “c++filt” [159] ,for more information on demangling.

--debugging
Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.

-d , **--disassemble**
Display the assembler mnemonics for the machine instructions from `objfile`. This option only disassembles those sections which are expected to contain instructions.

-D , **--disassemble-all**
Like “ **-d** ” ,but disassemble the contents of all sections, not just those expected to contain instructions.

--prefix-addresses
When disassembling, print the complete address on each line. This is the older disassembly format.

--disassemble-zeroes
Normally the disassembly output will skip blocks of zeroes. This option directs the disassembler to disassemble those blocks, just like any other data.

-EB , **-EL** , **--endian={big|little}**
Specify the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file

format which does not describe endianness information, such as S-records.

-f , **--file-header**
Display summary information from the overall header of each of the *`objfile`* files.

-h , **--section-header** , **--header**
Display summary information from the section headers of the object file.

File segments may be relocated to nonstandard addresses, for example by using the “ **-Ttext** ” , “ **-Tdata** ” ,or “ **-Tbss** ” options to **ld**. However, some object file formats, such as a.out, do not store the starting address of the file segments. In those situations, although **ld** relocates the sections correctly, using “ **objdump -h** ” to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

--help
Print a summary of the options to **objdump** and exit.

-i , **--info**
Display a list showing all architectures and object formats available for specification with “ **-b** ” or “ **-m** ” .

-j ***`name`*** , **--section=*`name`***
Display information only for section *`name`*.

-l , **--line-numbers**
Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with “ **-d** ” , “ **-D** ” ,or “ **-r** ” .

-m ***`machine`*** , **--architecture=*`machine`***
Specify the architecture to use when disassembling object files. This can be useful when disassembling object files which do not describe architecture information, such as S-records. You can list the available architectures with the “ **-i** ” option.

-r , **--reloc**
Print the relocation entries of the file. If used with “ **-d** ” or “ **-D** ” ,the relocations are printed interspersed with the disassembly.

-R , **--dynamic-reloc**
Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.

-s , **--full-contents**
Display the full contents of any sections requested.

-S , **--source**
Display source code intermixed with disassembly, if possible. Implies “ **-d** ” .

--show-raw-insn
When disassembling instructions, print the instruction in hex as well as in symbolic form. This is the default except when **--prefix-addresses** is used.

--no-show-raw-insn
When disassembling instructions, do not print the instruction bytes. This is the default when **--prefix-addresses** is used.

--stabs
Display the full contents of any sections requested. Display the contents of the .stab and .stab.index and .stab.excl sections from an ELF file. This is only useful on systems (such as Solaris 2.0) in which **.stab** debugging symbol-table entries are carried in an ELF section. In most other file formats, debugging symbol-table entries are interleaved with linkage symbols, and are visible in the “ **--syms** ” output. For more information on stabs symbols, see *the stabs debugging format* .

--start-address=*address*
Start displaying data at the specified address. This affects the output of the **-d**, **-r** and **-s** options.

--stop-address=*address*
Stop displaying data at the specified address. This affects the output of the **-d**, **-r** and **-s** options.

-t , **--syms**
Print the symbol table entries of the file. This is similar to the information provided by the “ **nm** ” program.

-T , **--dynamic-syms**
Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared

libraries. This is similar to the information provided by the “ **nm** ” program when given the “ **-D** ” (“ **--dynamic** ”)option.

--version
Print the version number of **objdump** and exit.

-x , **--all-header**
Display all available header information, including the symbol table and relocation entries. Using “ **-x** ” is equivalent to specifying all of “ **-a -f -h -r -t** ” .

-w

--wide
Format some lines for output devices that have more than 80 columns.

12.5. ranlib

```
prefix-ranlib [-vV] archive
```

ranlib generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

You may use “ **nm -s** ” or “ **nm --print-armap** ” to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

The **ranlib** program is another form of **ar**; running **ranlib** is completely equivalent to executing “ **ar -s** ” .See Section 12.1, “ar” [131] .

-v , **-V**
Show the version number of **ranlib**.

12.6. size

```
prefix-size [ -A | -B | --format=compatibility ]
[ --help ]  [ -d | -o | -x | --radix=number ]
[ --target=bfdname ]  [ -V | --version ]
objfile...
```

The **size** utility lists the section sizes and the total size for each of the object or archive files `objfile` in its argument list. By default, one line of output is generated for each object file or each module in an archive.

`objfile...` are the object files to be examined.

The command line options have the following meanings:

-A , **-B** , **--format=*compatibility***
: Using one of these options, you can choose whether the output from **size** resembles output from System V **size** (using “ **-A** ” ,or “ **--format=sysv** ”), or Berkeley **size** (using “ **-B** ” ,or “ **--format=berkeley** ”). The default is the one-line format similar to Berkeley's.

  Here is an example of the Berkeley (default) format of output from **size**:

```
prefix-size --format=Berkeley ranlib size
text    data    bss     dec     hex     f
294880  81920   11592   388392  5ed28   r
294880  81920   11888   388688  5ee50   s
```

  This is the same data, but displayed closer to System V conventions:

```
prefix-size --format=SysV ranlib size
ranlib  :
section         size         addr
.text         294880         8192
```

```
.data        81920      303104
.bss         11592      385024
Total       388392
size  :
section       size        addr
.text       294880        8192
.data        81920      303104
.bss         11888      385024
Total       388688
```

--help
Show a summary of acceptable arguments and options.

-d , **-o** , **-x** , **--radix=*`number`***
Using one of these options, you can control whether the size of each section is given in decimal (“ **-d** ” ,or “ **--radix=10** ”); octal (“ **-o** ” ,or “ **--radix=8** ”); or hexadecimal (“ **-x** ” ,or “ **--radix=16** ”). In “ **--radix=*`number`*** ” , only the three values (8, 10, 16) are supported. The total size is always given in two radices; decimal and hexadecimal for “ **-d** ” or “ **-x** ” output, or octal and hexadecimal if you're using “ **-o** ” .

--target=*`bfdname`*
Specify that the object-code format for *`objfile`* is *`bfdname`*. This option may not be necessary; **size** can automatically recognize many formats. See Section 13.1, “Target Selection” [163] ,for more information.

-V , **--version**
Display the version number of **size**.

12.7. strings

```
prefix-strings [-afov] [-min-len] [-n min-len] [-t radix]
[--all] [--print-file-name] [--bytes=min-len]
[--radix=radix] [--target=bfdname]
[--help] [--version] file...
```

For each *`file`* given, **strings** prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

strings is mainly useful for determining the contents of non-text files.

-a , **--all** , **-**
Do not scan only the initialized and loaded sections of object files; scan the whole files.

-f , **--print-file-name**
Print the name of the file before each string.

--help
Print a summary of the program usage on the standard output and exit.

-n ***`min-len`*** , ***`-min-len`*** , **--bytes=*`min-len`***
Print sequences of characters that are at least *`min-len`* characters long, instead of the default 4.

-o
Like " **-t o** " .Some other versions of **strings** have " **-o** " act like " **-t d** " instead. Since we can not be compatible with both ways, we simply chose one.

-t ***`radix`*** , **--radix=*`radix`***
Print the offset within the file before each string. The single character argument specifies the radix of the offset**--** " **o** " for octal, " **x** " for hexadecimal, or " **d** " for decimal.

--target=*`bfdname`*
Specify an object code format other than your system's default format. See Section 13.1, "Target Selection" [163] ,for more information.

-v , **--version**
Print the program version number on the standard output and exit.

12.8. strip

```
prefix-strip [ -F bfdname | --target=bfdname | --target=bf
[ -I bfdname | --input-target=bfdname ]
[ -O bfdname | --output-target=bfdname ]
[ -s | --strip-all ] [ -S | -g | --strip-debug ]
[ -K symbolname | --keep-symbol=symbolname ]
[ -N symbolname | --strip-symbol=symbolname ]
[ -x | --discard-all ] [ -X | --discard-locals ]
[ -R sectionname | --remove-section=sectionname ]
[ -o file ] [ -p | --preserve-dates ]
[ -v | --verbose ]  [ -V | --version ]  [ --help ]
objfile...
```

strip discards all symbols from object files `objfile`. The list of object files may include archives. At least one object file must be given.

strip modifies the files named in its argument, rather than writing modified copies under different names.

-F *bfdname* , --target=*bfdname*
: Treat the original `objfile` as a file with the object code format `bfdname`, and rewrite it in the same format. See Section 13.1, “Target Selection” [163] ,for more information.

--help
: Show a summary of the options to **strip** and exit.

-I *bfdname* , --input-target=*bfdname*
: Treat the original `objfile` as a file with the object code format `bfdname`. See Section 13.1, “Target Selection” [163] ,for more information.

-O *bfdname* , --output-target=*bfdname*
: Replace `objfile` with a file in the output format `bfdname`. See Section 13.1, “Target Selection” [163] ,for more information.

-R *sectionname* , --remove-section=*sectionname*
: Remove any section named `sectionname` from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.

-s , **--strip-all**
Remove all symbols.

-g , **-S** , **--strip-debug**
Remove debugging symbols only.

--strip-unneeded
Remove all symbols that are not needed for relocation processing.

-K ***`symbolname`*** , **--keep-symbol=*`symbolname`***
Keep only symbol *`symbolname`* from the source file. This option may be given more than once.

-N ***`symbolname`*** , **--strip-symbol=*`symbolname`***
Remove symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than **-K**.

-o ***`file`***
Put the stripped output in *`file`*, rather than replacing the existing file. When this argument is used, only one *`objfile`* argument may be specified.

-p , **--preserve-dates**
Preserve the access and modification dates of the file.

-x , **--discard-all**
Remove non-global symbols.

-X , **--discard-locals**
Remove compiler-generated local symbols. (These usually start with " **L** " or " **.** " .)

-V , **--version**
Show the version number for **strip**.

-v , **--verbose**
Verbose output: list all object files modified. In the case of archives, " **strip -v** " lists all members of the archive.

12.9. c++filt

```
prefix-c++filt [ -_ | --strip-underscores ]
[ -n | --no-strip-underscores ]
[ -s format | --format=format ]
[ --help ]  [ --version ]  [ symbol... ]
```

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as *mangling*). The **c++filt** program does the inverse mapping: it decodes (*demangles*) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use **c++filt** to decipher individual symbols:

```
prefix-c++filt symbol
```

If no `symbol` arguments are given, **c++filt** reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

-_ , **--strip-underscores**
: On some systems, both the C and C++ compilers put an underscore in front of every name. For example, the C name **foo** gets the low-level name **_foo**. This option removes the initial underscore. Whether **c++filt** removes the underscore by default is target dependent.

-n , **--no-strip-underscores**
: Do not remove the initial underscore.

-s ***format*** , **--format=*format***

nm can decode three different methods of mangling, used by different C++ compilers. The argument to this option selects which method it uses:

gnu

the one used by the compiler (the default method)

lucid

the one used by the Lucid compiler

arm

the one specified by the C++ Annotated Reference Manual

--help

Print a summary of the options to **c++filt** and exit.

--version

Print the version number of **c++filt** and exit.

12.10. addr2line

```
prefix-addr2line [ -b bfdname | --target=bfdname ]
[ -C | --demangle ]
[ -e filename | --exe=filename ]
[ -f | --functions ] [ -s | --basename ]
[ -H | --help ] [ -V | --version ]
[ addr addr ... ]
```

addr2line translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

The executable to use is specified with the **-e** option. The default is **a.out**.

addr2line has two modes of operation.

In the first, hexadecimal addresses are specified on the command line, and **addr2line** displays the file name and line number for each address.

In the second, **addr2line** reads hexadecimal addresses from standard input, and prints the file name and line number for each address on standard output. In this mode, **addr2line** may be used in a pipe to convert dynamically chosen addresses.

The format of the output is “ **FILENAME:LINENO** ” .The file name and line number for each address is printed on a separate line. If the **-f** option is used, then each “ **FILENAME:LINENO** ” line is preceded by a “ **FunctionNAME** ” line which is the name of the function containing the address.

If the file name or function name can not be determined, **addr2line** will print two question marks in their place. If the line number can not be determined, **addr2line** will print 0.

The long and short forms of options, shown here as alternatives, are equivalent.

-b ***`bfdname`*** , **--target=*`bfdname`***
: Specify that the object-code format for the object files is *`bfdname`*.

-C , **--demangle**
: Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See Section 12.9, “c++filt” [159] ,for more information on demangling.

-e ***`filename`*** , **--exe=*`filename`***
: Specify the name of the executable for which addresses should be translated. The default file is **a.out**.

-f , **--functions**
: Display function names as well as file and line number information.

-s , **--basenames**
: Display only the base of each file name.

Chapter 13 Selecting the target system

You can specify three aspects of the target system to the object code utilities, each in several ways:

- the target
- the architecture
- the linker emulation (which applies to the linker only)

In the following summaries, the lists of ways to specify values are in order of decreasing precedence. The ways listed first override those listed later.

13.1. Target Selection

A *target* is an object file format. A given target may be supported for multiple architectures (see Section 13.2, “Architecture selection” [165]). A target selection may also have variations for different operating systems or architectures.

The command to list valid target values is “ ***`prefix`*-objdump -i** ” (the first column of output contains the relevant information).

Some sample values are: **a.out-hp300bsd**, **ecoff-littlemips**, **a.out-sunos-big**.

You can also specify a target using a configuration triplet. This is the same sort of name that is passed to configure to specify a target. When you use a configuration triplet as an argument, it must be fully canonicalized. You can see the canonical version of a triplet by running the shell script **config.sub** which is included with the sources.

objdump Target

Ways to specify:

1. command line option: “ **-b** ” or “ **--target** ”
2. environment variable **GNUPREFIX**
3. deduced from the input file

objcopy and strip Input Target

Ways to specify:

1. command line options: “ **-I** ” or “ **--input-target** ” ,or “ **-F** ” or “ **--target** ”
2. environment variable **GNUPREFIX**
3. deduced from the input file

objcopy and strip Output Target

Ways to specify:

1. command line options: “ **-O** ” or “ **--output-target** ” ,or “ **-F** ” or “ **--target** ”
2. the input target (see "**objcopy** and **strip** Input Target" above)
3. environment variable **GNUPREFIX**
4. deduced from the input file

nm, size, and strings Target

Ways to specify:

1. command line option: “ **--target** ”
2. environment variable **GNUPREFIX**
3. deduced from the input file

Linker Input Target

Ways to specify:

1. command line option: “ **-b** ” or “ **--format** ” (see Section 10.1, “Command Line Options” [79] .)
2. script command **PREFIX** (see Section 11.7, “Option Commands” [125] .)
3. environment variable **GNUPREFIX** (see Section 10.2, “Environment Variables” [95] .)
4. the default target of the selected linker emulation (see Section 13.3, “Linker emulation selection” [166] .)

Linker Output Target

Ways to specify:

1. command line option: “ **-oformat** ” (see Section 10.1, “Command Line Options” [79] .)
2. script command **OUTPUT_FORMAT** (see Section 11.7, “Option Commands” [125] .)
3. the linker input target (see "Linker Input Target" above)

13.2. Architecture selection

An *architecture* is a type of `cpu` on which an object file is to run. Its name may contain a colon, separating the name of the processor family from the name of the particular `cpu`.

The command to list valid architecture values is “ **objdump -i** ” (the second column contains the relevant information).

Sample values: “ **m68k:68020** ” , “ **mips:3000** ” , “ **sparc** ” .

objdump Architecture

Ways to specify:

1. command line option: “ **-m** ” or “ **--architecture** ”
2. deduced from the input file

objcopy, nm, size, strings Architecture

Ways to specify:

1. deduced from the input file

Linker Input Architecture

Ways to specify:

1. deduced from the input file

Linker Output Architecture

Ways to specify:

1. script command **OUTPUT_ARCH** (see Section 11.7, “Option Commands” [125] .)
2. the default architecture from the linker output target (see Section 13.1, “Target Selection” [163] .)

13.3. Linker emulation selection

A linker *emulation* is a “personality” of the linker, which gives the linker default values for the other aspects of the target system. In particular, it consists of

- the linker script
- the target
- several “hook” functions that are run at certain stages of the linking process to do special things that some targets require

The command to list valid linker emulation values is “ ***prefix*-ld -V** ” .

Sample values: “ **coff_erc** ” , “ **coff_i186** ” , “ **coff_1750** ” , “ **coff_m68k** ” .

Ways to specify:

1. command line option: “ **-m** ” (see Section 10.1, “Command Line Options” [79] .)
2. environment variable **LDEMULATION**
3. compiled-in **DEFAULT_EMULATION** from **Makefile**, which comes from **EMUL** in **config/*target*.mt**

IV

Appendices

Appendix A BFD

The linker accesses object and archive files using the BFD libraries. These libraries allow the linker to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library. To conserve runtime memory, however, the linker and associated tools are usually configured to support only a subset of the object file formats available. You can use `objdump -i` (see Section 12.4, “objdump” [148])to list all the formats available for your configuration.

One minor artifact of the BFD solution that you should bear in mind is the potential for information loss. There are two places where useful information can be lost using the BFD mechanism: during conversion and during output. See Section A.1.1, “Information Loss” [172] .

A.1. How it Works: An Outline of BFD

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file's data structures.

As different information from the object files is required, BFD reads from different sections of the file and processes them. For example, a very common operation for the linker is processing symbol tables. Each BFD back end provides a routine for converting between the object file's representation of symbols and an internal canonical format. When the linker asks for the symbol table of an object file, it calls through a memory pointer to the routine from the relevant BFD back end which reads and converts the table into a canonical form. The linker then operates upon the canonical form. When the link is finished and the linker writes the output file's symbol table, another BFD back end routine is called to take the newly created symbol table and convert it into the chosen output format.

A.1.1. Information Loss

Information can be lost during output. The output formats supported by BFD do not provide identical facilities, and information which can be described in one form has nowhere to go in another format. One example of this is alignment information in `b.out`. There is nowhere in an `a.out` format file to store alignment information on the contained data, so when a file is linked from `b.out` and an `a.out` image is produced, alignment information will not propagate to the output file. (The linker will still use the alignment information internally, so the link is performed correctly).

Another example is COFF section names. COFF files may contain an unlimited number of sections, each one with a textual section name. If the target of the link is a format which does not have many sections (e.g., `a.out`) or has sections without names (e.g., the Oasys format), the link cannot be done simply. You can circumvent this problem by describing the desired input-to-output section mapping with the linker command language.

Information can be lost during canonicalization. The BFD internal canonical form of the external formats is not exhaustive; there are structures in input formats for which there is no direct representation internally. This means that the BFD back ends cannot maintain all possible data richness through the transformation between external to internal and back to external formats.

This limitation is only a problem when an application reads one format and writes another. Each BFD back end is responsible for maintaining as much data as possible, and the internal BFD canonical form has structures which are opaque to the BFD core, and exported only to the back ends. When a file is read in one format, the canonical form is generated for BFD and the application. At the same time, the back end saves away any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, there is no information lost when linking or copying big endian COFF to little endian COFF, or `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

A.1.2. The BFD canonical object-file format

The greatest potential for loss of information occurs when there is the least overlap between the information provided by the source format, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you understand which kinds of data you can count on preserving across conversions.

files
: Information stored on a per-file basis includes target machine architecture, particular implementation format type, a demand pageable bit, and a write protected bit. Information like UNIX magic numbers is not stored here -- only the magic numbers' meaning, so a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set. The byte order of the target is stored on a per-file basis, so that big- and little-endian object files may be used with one another.

sections
: Each section in the input file contains the name of the section, the section's original address in the object file, size and alignment information, various flags, and pointers into other BFD data structures.

symbols
: Each symbol contains a pointer to the information for the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, it relocates all symbols to make them relative to the base of the section where they were defined. Doing this ensures that each

symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. The linker can operate on a collection of symbols of wildly different formats without problems.

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

relocation level

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in, and a pointer to a relocation type descriptor. Relocation is performed by passing messages through the relocation type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a 68k COFF file, even though 68k COFF has no such relocation type.

line numbers

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows finding out the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format which can simply derive this information can pass it successfully between formats (COFF, IEEE and Oasys).

Index