Emprog ThunderBench™

Linker Script Guide

Version 1.2

1 - Linker Script guide

2 -Arm Specific Options

1 The Linker Scripts

Every link is controlled by a linker script. This script is written in the linker command language.

The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. Most linker scripts do nothing more than this. However, when necessary, the linker script can also direct the linker to perform many other operations, using the commands described below. The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled into the linker executable. You can use the '--verbose' command line option to display the default linker script. Certain command line options, such as '-r' or '-N', will affect the default linker script.

You may supply your own linker script by using the '-T' command line option. When you do this, your linker script will replace the default linker script.
You may also use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked.

1.1 Basic Linker Script Concepts

We need to define some basic concepts and vocabulary in order to describe the linker script language.

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an object file format. Each file is called an object file. The output file is often called an executable, but for our purposes we will also call it an object file. Each object file has, among other things, a list of sections. We sometimes refer to a section in an input file as an input section; similarly, a section in the output file is an output section.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section may be marked as loadable, which mean that the contents should be loaded into memory when the output file is run. A section with no contents may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section which is neither loadable nor allocatable typically contains some sort of debugging information.

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address would be the LMA, and the RAM address would be the VMA.
You can see the sections in an object file by using the objdump program with the '-h' option.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

You can see the symbols in an object file by using the nm program, or by using the objdump program with the '-t' option.

1.2 Linker Script Format -Linker scripts are text files.

You write a linker script as a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. You may separate commands using semicolons. Whitespace is generally ignored.

Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma which would otherwise serve to separate file names, you may put the file name in double quotes. There is no way to use a double quote character in a file name.

You may include comments in linker scripts just as in C, delimited by '/*' and '*/'. As in C, comments are syntactically equivalent to whitespace.

1.3 Simple Linker Script Example

Many linker scripts are fairly simple. The simplest possible linker script has just one command: 'SECTIONS'. You use the 'SECTIONS' command to describe the memory layout of the output file. The 'SECTIONS' command is a powerful command. Here we will describe a simple use of it. Let's assume your program consists only of code, initialized data, and uninitialized data. These will be in the '.text', '.data', and '.bss' sections, respectively. Let's assume further that these are the only sections which appear in your input files.

For this example, let's say that the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

```
SECTIONS
{
. = 0x10000;
.text : { *(.text) }
. = 0x8000000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

You write the 'SECTIONS' command as the keyword 'SECTIONS', followed by a series of symbol assignments and output section descriptions enclosed in curly braces.
The first line inside the 'SECTIONS' command of the above example sets the value of the special symbol '.', which is the location counter. If you do not specify the address of an

output section in some other way (other ways are described later), the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the 'SECTIONS' command, the location counter has the value '0'.

The second line defines an output section, '.text'. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections which should be placed into this output section. The '*' is a wildcard which matches any file name. The expression '*(.text)' means all '.text' input sections in all input files.

Since the location counter is '0x10000' when the output section '.text' is defined, the linker will set the address of the '.text' section in the output file to be '0x10000'.
The remaining lines define the '.data' and '.bss' sections in the output file. The linker will place the '.data' output section at address '0x8000000'. After the linker places the '.data' output section, the value of the location counter will be '0x8000000' plus the size of the '.data' output section. The effect is that the linker will place the '.bss' output section immediately after the '.data' output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the '.text' and '.data' sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the '.data' and '.bss' sections. That's it! That's a simple and complete linker script.

1.4 Simple Linker Script Commands

In this section we describe the simple linker script commands.

1.4.1 Setting the Entry Point

The first instruction to execute in a program is called the entry point. You can use the ENTRY linker script command to set the entry point. The argument is a symbol name:
ENTRY(symbol)
There are several ways to set the entry point. The linker will set the entry point by trying each of the following methods in order, and stopping when one of them succeeds:
_ the '-e' entry command-line option;
_ the ENTRY(symbol) command in a linker script;
_ the value of the symbol start, if defined;
_ the address of the first byte of the '.text' section, if present;
_ The address 0.

1.4.2 Commands Dealing with Files

Several linker script commands deal with files.

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.

You can place INCLUDE directives at the top level, in MEMORY or SECTIONS commands, or in output section descriptions.

INPUT(file, file, ...)
INPUT(file file ...)

The INPUT command directs the linker to include the named files in the link, as though they were named on the command line.

For example, if you always want to include ‘subr.o’ any time you do a link, but you can’t be bothered to put it on every link command line, then you can put ‘INPUT (subr.o)’ in your linker script.

In fact, if you like, you can list all of your input files in the linker script, and then invoke the linker with nothing but a ‘-T’ option.

In case a sysroot prefix is configured, and the filename starts with the ‘/’ character, and the script being processed was located inside the sysroot prefix, the filename will be looked for in the sysroot prefix. Otherwise, the linker will try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of ‘-L’.

If you use ‘INPUT (-lfile)’, ld will transform the name to libfile.a, as with the command line argument ‘-l’.

When you use the INPUT command in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

GROUP(file, file, ...)
GROUP(file file ...)

The GROUP 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.

AS_NEEDED(file, file, ...)
AS_NEEDED(file file ...)

This construct can appear only inside of the INPUT or GROUP commands, among other filenames. The files listed will be handled as if they appear directly in the INPUT or GROUP commands, with the exception of ELF shared libraries, that will be added only when they are actually needed. This construct essentially enables ‘—as-needed’ option for all the files listed inside of it and restores previous ‘—as-needed’ resp. ‘—no-as-needed’ setting afterwards.

OUTPUT(filename)

The OUTPUT command names the output file. Using OUTPUT(filename) in the linker script is exactly like using '-o filename' on the command line. If both are used, the command line option takes precedence.

You can use the OUTPUT command to define a default name for the output file other than the usual default of 'a.out'.

SEARCH_DIR(path)
The SEARCH_DIR command adds path to the list of paths where ld looks for archive libraries. Using SEARCH_DIR(path) is exactly like using '-L path' on the command line (If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

STARTUP(filename)
The STARTUP command is just like the INPUT command, except that filename will become the first input file to be linked, as though it were specified first on the command line. This may be useful when using a system in which the entry point is always the start of the first file.

1.4.3 Commands Dealing with Object File Formats

A couple of linker script commands deal with object file formats.

OUTPUT_FORMAT(bfdname)
OUTPUT_FORMAT(default, big, little)
The OUTPUT_FORMAT command names the BFD format to use for the output file.

Using OUTPUT_FORMAT(bfdname) is exactly
like using '—oformat bfdname' on the command line . If both are used, the command line option takes precedence.

You can use OUTPUT_FORMAT with three arguments to use different formats based on the '-EB' and '-EL' command line options. This permits the linker script to set the output format based on the desired endianness. If neither '-EB' nor '-EL' are used, then the output format will be the first argument, default. If '-EB' is used, the output format will be the second argument, big. If '-EL' is used, the output format will be the third argument, little.

For example, the default linker script for the MIPS ELF target uses this command:

OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips) This says that the default format for the output file is 'elf32-bigmips', but if the user uses the '-EL' command line option, the output file will be created in the 'elf32-littlemips' format.

TARGET(bfdname)
The TARGET command names the BFD format to use when reading input files. It affects subsequent INPUT and GROUP commands. This command is like using '-b bfdname' on the command line. If the TARGET command is used but OUTPUT_FORMAT is not, then the last TARGET command is also used to set the format for the output file.

1.4.4 Assign alias names to memory regions

Alias names can be added to existing memory regions created with the command. Each name corresponds to at most one memory region.

REGION_ALIAS(alias, region)

The REGION_ALIAS function creates an alias name alias for the memory region region. This allows a flexible mapping of output sections to memory regions. An example follows. Suppose we have an application for embedded systems which come with various memory storage devices. All have a general purpose, volatile memory RAM that allows code execution or data storage. Some may have a read-only, non-volatile memory ROM that allows code execution and read-only data access. The last variant is a read-only, non-volatile memory ROM2 with read-only data access and no code execution capability. We have four output sections:

_ .text program code;
_ .rodata read-only data;
_ .data read-write initialized data;
_ .bss read-write zero initialized data.

The goal is to provide a linker command file that contains a system independent part defining the output sections and a system dependent part mapping the output sections to the memory regions available on the system. Our embedded systems come with three different memory setups A, B and C:

Section Variant A Variant B Variant C
.text RAM ROM ROM
.rodata RAM ROM ROM2
.data RAM RAM/ROM RAM/ROM2
.bss RAM RAM RAM

The notation RAM/ROM or RAM/ROM2 means that this section is loaded into region ROM or ROM2 respectively. Please note that the load address of the .data section starts in all three variants at the end of the .rodata section.

The base linker script that deals with the output sections follows. It includes the system dependent linkcmds.memory file that describes the memory layout:

```
INCLUDE linkcmds.memory
SECTIONS
{
.text :
{
*(.text)
} > REGION_TEXT
.rodata :
{
*(.rodata)
```

```
rodata_end = .;
} > REGION_RODATA
.data : AT (rodata_end)
{
data_start = .;
*(.data)
} > REGION_DATA
data_size = SIZEOF(.data);
data_load_start = LOADADDR(.data);
.bss :
{
*(.bss)
} > REGION_BSS
}
```

Now we need three different linkcmds.memory files to define memory regions and alias names. The content of linkcmds.memory for the three variants A, B and C:
A Here everything goes into the RAM.

```
MEMORY
{
RAM : ORIGIN = 0, LENGTH = 4M
}
REGION_ALIAS("REGION_TEXT", RAM);
REGION_ALIAS("REGION_RODATA", RAM);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

B Program code and read-only data go into the ROM. Read-write data goes into the RAM. An image of the initialized data is loaded into the ROM and will be copied during system start into the RAM.

```
MEMORY
{
ROM : ORIGIN = 0, LENGTH = 3M
RAM : ORIGIN = 0x10000000, LENGTH = 1M
}
REGION_ALIAS("REGION_TEXT", ROM);
REGION_ALIAS("REGION_RODATA", ROM);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

C Program code goes into the ROM. Read-only data goes into the ROM2. Readwrite data goes into the RAM. An image of the initialized data is loaded into the ROM2 and will be copied during system start into the RAM.

```
MEMORY
{
```

```
ROM : ORIGIN = 0, LENGTH = 2M
ROM2 : ORIGIN = 0x10000000, LENGTH = 1M
RAM : ORIGIN = 0x20000000, LENGTH = 1M
}
REGION_ALIAS("REGION_TEXT", ROM);
REGION_ALIAS("REGION_RODATA", ROM2);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

It is possible to write a common system initialization routine to copy the .data section from ROM or ROM2 into the RAM if necessary:

```
#include <string.h>
extern char data_start [];
extern char data_size [];
extern char data_load_start [];
void copy_data(void)
{
if (data_start != data_load_start)
{
memcpy(data_start, data_load_start, (size_t) data_size);
}
}
```

1.4.5 Other Linker Script Commands

There are a few other linker scripts commands.

ASSERT(exp, message)
Ensure that exp is non-zero. If it is zero, then exit the linker with an error code, and print message.

EXTERN(symbol 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. You may list several symbols for each EXTERN, and you may use EXTERN multiple times. This command has the same effect as the '-u' command-line option.

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

INHIBIT_COMMON_ALLOCATION
This command has the same effect as the '—no-define-common' command-line option: to make ld omit the assignment of addresses to common symbols even for a non-relocatable output file.

INSERT [AFTER | BEFORE] output_section
This command is typically used in a script specified by '-T' to augment the default SECTIONS with, for example, overlays. It inserts all prior linker script statements after (or before) output section, and also causes '-T' to not override the default linker script. The exact insertion point is as for

orphan sections. The insertion happens after the linker has mapped input sections to output sections. Prior to the insertion, since '-T' scripts are parsed before the default linker script, statements in the '-T' script occur before the default linker script statements in the internal linker representation of the script. In particular, input section assignments will be made to '-T' output sections before those in the default script. Here is an example of how a '-T' script using INSERT might look:

```
SECTIONS
{
OVERLAY :
{
.ov1 { ov1*(.text) }
.ov2 { ov2*(.text) }
}
}
INSERT AFTER .text;
```

NOCROSSREFS(section section ...)
This command may be used to tell ld to issue an error about any references among certain output sections.

In certain types of programs, particularly on embedded systems when using overlays, 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 output section names. If ld detects any cross references between the sections, it reports an error and returns a non-zero exit status. Note that the NOCROSSREFS command uses output section names, not input section names.

OUTPUT_ARCH(bfdarch)
Specify a particular output machine architecture. The argument is one of the names used by the BFD library. You can see the architecture of an object file by using the objdump program with the '-f' option.

1.5 Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol and place it into the symbol table with a global scope.

1.5.1 Simple Assignments

You may assign to a symbol using any of the C assignment operators:

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

```
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

The first case will define symbol to the value of expression. In the other cases, symbol must already be defined, and the value will be adjusted accordingly. The special symbol name '.' indicates the location counter. You may only use this within a SECTIONS command. The semicolon after expression is required.

Expressions are defined below. You may write symbol assignments as commands in their own right, or as statements within a SECTIONS command, or as part of an output section description in a SECTIONS command.

The section of the symbol will be set from the section of the expression; for more information. Here is an example showing the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
.text :
{
*(.text)
_etext = .;
}
_bdata = (. + 3) & ~ 3;
.data : { *(.data) }
}
```

In this example, the symbol 'floating_point' will be defined as zero. The symbol '_etext' will be defined as the address following the last '.text' input section. The symbol '_bdata' will be defined as the address following the '.text' output section aligned upward to a 4 byte boundary.

1.5.2 PROVIDE

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and 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).

Here is an example of using PROVIDE to define 'etext':

```
SECTIONS
{
```

```
.text :
{
*(.text)
_etext = .;
PROVIDE(etext = .);
}
}
```

In this example, if the program defines '_etext' (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines 'etext' (with no leading underscore), the linker will silently use the definition in the program. If the program references 'etext' but does not define it, the linker will use the definition in the linker script.

1.5.3 PROVIDE HIDDEN

Similar to PROVIDE. For ELF targeted ports, the symbol will be hidden and won't be exported.

1.5.4 Source Code Reference

Accessing a linker script defined variable from source code is not intuitive. In particular a linker script symbol is not equivalent to a variable declaration in a high level language, it is instead a symbol that does not have a value.
Before going further, it is important to note that compilers often transform names in the source code into different names when they are stored in the symbol table. For example, Fortran compilers commonly prepend or append an underscore, and C++ performs extensive 'name mangling'. Therefore there might be a discrepancy between the name of a variable as it is used in source code and the name of the same variable as it is defined in a linker script. For example in C a linker script variable might be referred to as:

```
extern int foo;
```

But in the linker script it might be defined as:

```
_foo = 1000;
```

In the remaining examples however it is assumed that no name transformation has taken place.

When a symbol is declared in a high level language such as C, two things happen. The first is that the compiler reserves enough space in the program's memory to hold the value of the symbol. The second is that the compiler creates an entry in the program's symbol table which holds the symbol's address. ie the symbol table contains the address of the block of memory holding the symbol's value. So for example the following C declaration, at file scope:

```
int foo = 1000;
```

creates a entry called 'foo' in the symbol table. This entry holds the address of an 'int' sized block of memory where the number 1000 is initially stored.

When a program references a symbol the compiler generates code that first accesses the symbol table to find the address of the symbol's memory block and then code to read the value from that memory block. So:

foo = 1;
looks up the symbol ‘foo’ in the symbol table, gets the address associated with this symbol and then writes the value 1 into that address. Whereas:
int * a = & foo;
looks up the symbol ‘foo’ in the symbol table, gets it address and then copies this address into the block of memory associated with the variable ‘a’.
Linker scripts symbol declarations, by contrast, create an entry in the symbol table but do not assign any memory to them. Thus they are an address without a value. So for example the linker script definition:
foo = 1000;
creates an entry in the symbol table called ‘foo’ which holds the address of memory location 1000, but nothing special is stored at address 1000. This means that you cannot access the value of a linker script defined symbol - it has no value - all you can do is access the address of a linker script defined symbol.
Hence when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt to use its value. For example suppose you want to copy the contents of a section of memory called .ROM into a section called .FLASH and the linker script contains these declarations:
start_of_ROM = .ROM;
end_of_ROM = .ROM + sizeof (.ROM) - 1;
start_of_FLASH = .FLASH;
Then the C source code to perform the copy would be:
extern char start_of_ROM, end_of_ROM, start_of_FLASH;
memcpy (& start_of_FLASH, & start_of_ROM, & end_of_ROM - & start_of_ROM);
Note the use of the ‘&’ operators. These are correct.

1.6 SECTIONS Command

The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory.
The format of the SECTIONS command is:
SECTIONS
{
sections-command
sections-command
...
}
Each sections-command may of be one of the following:
_ an ENTRY command
_ a symbol assignment
_ an output section description
_ an overlay description
The ENTRY command and symbol assignments are permitted inside the SECTIONS command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because you can use those commands at meaningful points in the layout of the output file.
Output section descriptions and overlay descriptions are described below.

If you do not use a SECTIONS command in your linker script, the linker will place 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. The first section will be at address zero.

1.6.1 Output Section Description

The full description of an output section looks like this:

```
section [address] [(type)] :
[AT(lma)]
[ALIGN(section_align)]
[SUBALIGN(subsection_align)]
[constraint]
{
output-section-command
output-section-command
...
} [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

Most output sections do not use most of the optional section attributes.
The whitespace around section is required, so that the section name is unambiguous. The colon and the curly braces are also required. The line breaks and other white space are optional.

Each output-section-command may be one of the following:
_ a symbol assignment
_ an input section description
_ data values to include directly
_ a special output section keyword

1.6.2 Output Section Name

The name of the output section is section. section must meet the constraints of your output format. In formats which 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 a name which contains any unusual characters such as commas must be quoted.
The output section name '/DISCARD/' is special;

1.6.3 Output Section Address

The address is an expression for the VMA (the virtual memory address) of the output section. If you do not provide address, the linker will set it based on region if present, or otherwise based on the current value of the location counter.
If you provide address, the address of the output section will be set to precisely that. If you provide neither address nor region, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the

output section. The alignment requirement of the output section is the strictest alignment of any input section contained within the output section.
For example,
.text . : { *(.text) }
and
.text : { *(.text) }
are subtly different. The first will set the address of the '.text' output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a '.text' input section. The address may be an arbitrary expression. For example, if you want to align the section on a 0x10 byte boundary, so that the lowest four bits of the section address are zero, you could do something like this:

.text ALIGN(0x10) : { *(.text) }
This works because ALIGN returns the current location counter aligned upward to the specified value.
Specifying address for a section will change the value of the location counter, provided that the section is non-empty. (Empty sections are ignored).

1.6.4 Input Section Description

The most common output section command is an input section description.
The input section description is the most basic linker script operation. You use output sections to tell the linker how to lay out your program in memory. You use input section descriptions to tell the linker how to map the input files into your memory layout.

1.6.4.1 Input Section Basics

An input section description consists of a file name optionally followed by a list of section names in parentheses.
The file name and the section name may be wildcard patterns, which we describe further below.

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input '.text' sections, you would write:

*(.text)
Here the '*' is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, EXCLUDE FILE may be used to match all files except the ones specified in the EXCLUDE FILE list. For example:
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
will cause all .ctors sections from all files except 'crtend.o' and 'otherfile.o' to be included.
There are two ways to include more than one section:
*(.text .rdata)
*(.text) *(.rdata)
The difference between these is the order in which the '.text' and '.rdata' input sections will appear in the output section. In the first example, they will be intermingled, appearing in the same order as they are found in the linker input. In the second example, all '.text'

input sections will appear first, followed by all '.rdata' input sections.

You can specify a file name to include sections from a particular file. You would do this if one or more of your files contain special data that needs to be at a particular location in memory. For example:
data.o(.data)
You can also specify files within archives by writing a pattern matching the archive, a colon, then the pattern matching the file, with no whitespace around the colon.
'archive:file'
matches file within archive
'archive:'
matches the whole archive
':file' matches file but not one in an archive
Either one or both of 'archive' and 'file' can contain shell wildcards. On DOS based file systems, the linker will assume that a single letter followed by a colon is a drive specifier, so 'c:myfile.o' is a simple file specification, not 'myfile.o' within an archive called 'c'. 'archive:file' filespecs may also be used within an EXCLUDE_FILE list, but may not appear in other linker script contexts. For instance, you cannot extract a file from an archive by using 'archive:file' in an INPUT command.

If you use a file name without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may by useful on occasion. For example:
data.o
When you use a file name which is not an 'archive:file' specifier and does not contain any wild card characters, the linker will first see if you also specified the file name on the linker command line or in an INPUT command. If you did not, the linker will attempt to open the file as an input file, as though it appeared on the command line. Note that this differs from an INPUT command, because the linker will not search for the file in the archive search path.

1.6.4.2 Input Section Wildcard Patterns

In an input section description, either the file name or the section name or both may be wildcard patterns.

The file name of '*' seen in many examples is a simple wildcard pattern for the file name. The wildcard patterns are like those used by the Unix shell.
'*' matches any number of characters
'?' matches any single character
'[chars]' matches 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
'\' quotes the following character
When a file name is matched with a wildcard, the wildcard 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, whether it contains a '/' or not. In a section name, the wildcard characters will match a '/' character.

File name wildcard patterns only match files which are explicitly specified on the command line or in an INPUT command. The linker does not search directories to expand wildcards.
If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the 'data.o' rule will not be used:
.data : { *(.data) }
.data1 : { data.o(.data) }
Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the SORT_BY_NAME keyword, which appears before a wildcard pattern in parentheses (e.g., SORT_BY_NAME(.text*)).
When the SORT_BY_NAME keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.
SORT_BY_ALIGNMENT is very similar to SORT_BY_NAME. The difference is SORT_BY_ALIGNMENT will sort sections into ascending order by alignment before placing them in the output file.
SORT is an alias for SORT_BY_NAME.
When there are nested section sorting commands in linker script, there can be at most 1 level of nesting for section sorting commands.

1. SORT_BY_NAME (SORT_BY_ALIGNMENT (wildcard section pattern)). It will sort the input sections by name first, then by alignment if 2 sections have the same name.
2. SORT_BY_ALIGNMENT (SORT_BY_NAME (wildcard section pattern)). It will sort the input sections by alignment first, then by name if 2 sections have the same alignment.
3. SORT_BY_NAME (SORT_BY_NAME (wildcard section pattern)) is treated the same as SORT_ BY_NAME (wildcard section pattern).
4. SORT_BY_ALIGNMENT (SORT_BY_ALIGNMENT (wildcard section pattern)) is treated the same as SORT_BY_ALIGNMENT (wildcard section pattern).
5. All other nested section sorting commands are invalid.

When both command line section sorting option and linker script section sorting command are used, section sorting command always takes precedence over the command line option.
If the section sorting command in linker script isn't nested, the command line option will make the section sorting command to be treated as nested sorting command.

1. SORT_BY_NAME (wildcard section pattern) with '--sort-sections alignment' is equivalent to SORT_BY_NAME (SORT_BY_ALIGNMENT (wildcard section pattern)).
2. SORT_BY_ALIGNMENT (wildcard section pattern) with '--sort-section name' is equivalent to SORT_BY_ALIGNMENT (SORT_BY_NAME (wildcard section pattern)).

If the section sorting command in linker script is nested, the command line option will be ignored.
If you ever get confused about where input sections are going, use the '-M' linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.
This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all '.text' sections in '.text' and all '.bss' sections in '.bss'. The linker will place the '.data' section from all files beginning with an upper case character in '.DATA'; for all other files, the linker will place the '.data' section in '.data'.

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

1.6.4.3 Input Section for Common Symbols

A special notation is needed for common symbols, because in many object file formats common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named 'COMMON'.

You may use file names with the 'COMMON' section just as with any other input sections. You can use this to place common symbols from a particular input file in one section while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the '.bss' section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

Some object file formats have more than one type of common symbol. For example, the MIPS ELF object file format distinguishes standard common symbols and small common symbols. In this case, the linker will use a different special section name for other types of common symbols. In the case of MIPS ELF, the linker uses 'COMMON' for standard common symbols and '.scommon' for small common symbols. This permits you to map the different types of common symbols into memory at different locations.

You will sometimes see '[COMMON]' in old linker scripts. This notation is now considered obsolete. It is equivalent to '*(COMMON)'.

1.6.4.4 Input Section and Garbage Collection

When link-time garbage collection is in use ('--gc-sections'), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT_BY_NAME(*)(.ctors)).

1.6.4.5 Input Section Example

The following example is a complete linker script. It tells the linker to read all of the sections from file 'all.o' and place 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)
}
}
```

1.6.5 Output Section Data

You can include explicit bytes of data in an output section by using BYTE, SHORT, LONG, QUAD, or SQUAD as an output section command. Each keyword is followed by an expression in parentheses providing the value to store The value of the expression is stored at the current value of the location counter.

The BYTE, SHORT, LONG, and QUAD commands store one, two, four, and eight bytes (respectively). After storing the bytes, the location counter is incremented by the number of bytes stored. For example, this will store the byte 1 followed by the four byte value of the symbol 'addr':

```
BYTE(1)
LONG(addr)
```

When using a 64 bit host or target, QUAD and SQUAD are the same; they both store an 8 byte, or 64 bit, value. When both host and target are 32 bits, an expression is computed as 32 bits. In this case QUAD stores a 32 bit value zero extended to 64 bits, and SQUAD stores a 32 bit value sign extended to 64 bits.

If the object file format of the output file has an explicit endianness, which is the normal case, the value will be stored in that endianness. When the object file format does not have an explicit endianness, as is true of, for example, S-records, the value will be stored in the endianness of the first input object file.

Note—these commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

You may use the FILL command to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the value of the expression, repeated as necessary. A FILL statement covers memory locations after the point at which 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.

This example shows how to fill unspecified regions of memory with the value '0x90':

```
FILL(0x90909090)
```

The FILL command is similar to the '=fillexp' output section attribute, but it only affects the part of the section following the FILL command, rather than the entire section. If both are used, the FILL command takes precedence.

1.6.6 Output Section Keywords

There are a couple of keywords which can appear as output section commands.
CREATE_OBJECT_SYMBOLS
The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the CREATE_OBJECT_SYMBOLS command appears.
This is conventional for the a.out object file format. It is not normally used for any other object file format.
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 which 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 to place constructor information in the output section where the CONSTRUCTORS command appears. The CONSTRUCTORS command is ignored for other object file formats.

The symbol __CTOR_LIST__ marks the start of the global constructors, and the symbol __CTOR_END__ marks the end. Similarly, __DTOR_LIST__ and __DTOR_END__ mark the start and end of the global destructors. 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 gnu C++ normally calls constructors from a subroutine __main; a call to __main is automatically inserted into the startup code for main. gnu C++ normally runs destructors either by using atexit, or directly from the function exit.
For object file formats such as COFF or ELF which support arbitrary section names, gnu 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 which the gnu 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__ = .;
```

If you are using the gnu C++ support for initialization priority, which provides some control over the order in which global constructors are run, you must sort the constructors at link time to ensure that they are

executed in the correct order. When using the CONSTRUCTORS command, use 'SORT_BY_NAME(CONSTRUCTORS)' instead. When using the .ctors and .dtors sections, use '*(SORT_BY_NAME(.ctors))' and '*(SORT_BY_NAME(.dtors))' instead of just '*(.ctors)' and '*(.dtors)'.
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.

1.6.7 Output Section Discarding
The linker will not create output sections with no contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. 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, and if the input sections are not all empty. Other link script directives that allocate space in an output section will also create the output section.
The linker will ignore address assignments on discarded output sections, except when the linker script defines symbols in the output section. In that case the linker will obey the address assignments, possibly advancing dot even though the section is discarded.

The special output section name '/DISCARD/' may be used to discard input sections. Any input sections which are assigned to an output section named '/DISCARD/' are not included in the output file.

1.6.8 Output Section Attributes
We showed above that the full description of an output section looked like this:
section [address] [(type)] :
[AT(lma)]
[ALIGN(section_align)]
[SUBALIGN(subsection_align)]
[constraint]
{
output-section-command
output-section-command
...
} [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
We've already described section, address, and output-section-command. In this section we will describe the remaining section attributes.
1.6.8.1 Output Section Type
Each output section may have a type. The type is a keyword in parentheses. The following types are defined:
NOLOAD The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.
DSECT
COPY
INFO

OVERLAY These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.
The linker normally sets the attributes of an output section based on the input sections which map into it. You can override this by using the section type. For example, in the script sample below, the 'ROM' section is addressed at memory location '0' and does not need to be loaded when the program is run. The contents of the 'ROM' section will appear in the linker output file as usual.

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

1.6.8.2 Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA); The address expression which may appear in an output section description sets the VMA . The expression lma that follows the AT keyword specifies the load address of the section.

Alternatively, with 'AT>lma_region' expression, you may specify a memory region for the section's load address. Note that if the section has not had a VMA assigned to it then the linker will use the lma region as the VMA region as well.

If neither AT nor AT> is specified for an allocatable section, the linker will set the LMA such that the difference between VMA and LMA for the section is the same as the preceding output section in the same region. If there is no preceding output section or the section is not allocatable, the linker will set the LMA equal to the VMA.

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called '.text', which starts at 0x1000, one called '.mdata', which is loaded at the end of the '.text' section even though its VMA is 0x2000, and one called '.bss' to hold uninitialized data at address 0x3000. The symbol _data is defined with the value 0x2000, which shows that the location counter holds the VMA value, not the LMA value.

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

The run-time initialization code for use with a program generated with this linker script would include something like the following, to copy the initialized data from the ROM image to its runtime address. Notice how this code takes advantage of the symbols defined by the

linker script.
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;
/* ROM has data at end of text; copy it. */
while (dst < &_edata) {
*dst++ = *src++;
}
/* Zero bss */
for (dst = &_bstart; dst< &_bend; dst++)
*dst = 0;

1.6.8.3 Forced Output Alignment

You can increase an output section's alignment by using ALIGN.

1.6.8.4 Forced Input Alignment

You can force input section alignment within an output section by using SUBALIGN. The value specified overrides any alignment given by input sections, whether larger or smaller.

1.6.8.5 Output Section Constraint

You can specify that an output section should only be created if all of its input sections are read-only or all of its input sections are read-write by using the keyword ONLY_IF_RO and ONLY_IF_RW respectively.

1.6.8.6 Output Section Region

You can assign a section to a previously defined region of memory by using '>region'.
Here is a simple example:
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }

1.6.8.7 Output Section Phdr

You can assign a section to a previously defined program segment by using ':phdr'. If a section is assigned to one or more segments, then all subsequent allocated sections will be assigned to those segments as well, unless they use an explicitly :phdr modifier. You can use :NONE to tell the linker to not put the section in any segment at all. Here is a simple example:

PHDRS { text PT_LOAD ; }
SECTIONS { .text : { *(.text) } :text }

1.6.8.8 Output Section Fill

You can set the fill pattern for an entire section by using '=fillexp'. fillexp is an expression Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the value, repeated as necessary. If the fill expression is a simple hex number, ie. a string of hex digit starting with '0x' and without a trailing 'k' or 'M', then an arbitrarily long sequence of hex digits can be used to specify the fill

pattern; Leading zeros become part of the pattern too. For all other cases, including extra parentheses or a unary +, the fill pattern is the four least significant bytes of the value of the expression. In all cases, the number is big-endian.

You can also change the fill value with a FILL command in the output section commands;

Here is a simple example:
SECTIONS { .text : { *(.text) } =0x90909090 }

1.6.9 Overlay Description

An overlay description provides an easy way to describe sections which 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. Overlays are described using the OVERLAY command. The OVERLAY command is used within a SECTIONS command, like an output section description. The full syntax of the OVERLAY command is as follows:

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
{
secname1
{
output-section-command
output-section-command
...
} [:phdr...] [=fill]
secname2
{
output-section-command
output-section-command
...
} [: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 contruct, 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 the current value of the location counter).
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.

For each section within the OVERLAY, the linker automatically provides 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 which 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 the location counter 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 if referenced: __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) }
PROVIDE (__load_start_text0 = LOADADDR (.text0));
PROVIDE (__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0));
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
PROVIDE (__load_start_text1 = LOADADDR (.text1));
PROVIDE (__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1));
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

1.7 MEMORY Command

The linker's default configuration permits allocation of all available memory. You can override this by using the MEMORY command.
The MEMORY command describes the location and size of blocks of memory in the target. You can use it to describe which memory regions may be used by the linker, and which memory regions it must avoid. You can then assign sections to particular memory regions. The linker will set section addresses based on the memory regions, and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.
A linker script 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
...
}
The name is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name within the MEMORY command. However you can add later alias names to existing memory regions command.

The attr string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script. As described in, if you do not specify an output section forsome input section, the linker will create an output section with the same name as the input section. If you define region attributes, the linker will use them to select the memory region for the output section that it creates.

The attr string must consist only of the following characters:
‘R’ Read-only section
‘W’ Read/write section
‘X’ Executable section
Chapter 3: Linker Scripts 61
‘A’ Allocatable section
‘I’ Initialized section
‘L’ Same as ‘I’
‘!’ Invert the sense of any of the preceding attributes
If a unmapped section matches any of the listed attributes other than ‘!’, it will be placed in the memory region. The ‘!’ attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.
The origin is an numerical expression for the start address of the memory region. The expression must evaluate to a constant and it cannot involve any symbols. The keyword ORIGIN may be abbreviated to org or o (but not, for example, ORG).
The len is an expression for the size in bytes of the memory region. As with the origin expression, the expression must be numerical only and must evaluate to a constant. The keyword LENGTH may be abbreviated to len or l.
In the following example, we specify that there are two memory regions available for allocation: one starting at ‘0’ for 256 kilobytes, and the other starting at ‘0x40000000’ for four megabytes. The linker will place into the ‘rom’ memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the ‘ram’ memory region.
MEMORY
{
rom (rx) : ORIGIN = 0, LENGTH = 256K
ram (!rx) : org = 0x40000000, l = 4M
}
Once you define a memory region, you can direct the linker to place specific output sections

into that memory region by using the '>region' output section attribute. For example, if you have a memory region named 'mem', you would use '>mem' in the output section definition. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message. It is possible to access the origin and length of a memory in an expression via the ORIGIN(memory) and LENGTH(memory) functions:
_fstack = ORIGIN(ram) + LENGTH(ram) - 4;

1.8 PHDRS Command

The ELF object file format uses program headers, also knows as segments. The program headers describe how the program should be loaded into memory. You can print them out by using the objdump program with the '-p' option.
When you run an ELF program on a native ELF system, the system loader reads the program headers in order to figure out how to load the program. This will only work if the program headers are set correctly. This manual does not describe the details of how the system loader interprets program headers; for more information, see the ELF ABI.

The linker will create reasonable program headers by default. However, in some cases, you may need to specify the program headers more precisely. You may use the PHDRS command for this purpose. When the linker sees the PHDRS command in the linker script, it will not create any program headers other than the ones specified.
The linker only pays attention to the PHDRS command when generating an ELF output file. In other cases, the linker will simply ignore PHDRS.
This is the syntax of the PHDRS command. The words PHDRS, FILEHDR, AT, and FLAGS are keywords.

```
PHDRS
{
name type [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
[ FLAGS ( flags ) ] ;
}
```

The name is used only for reference in the SECTIONS command of the linker script. It is not put into the output file. Program header names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each program header must have a distinct name. The headers are processed in order and it is usual for them to map to sections in ascending load address order.
Certain program header types describe segments of memory which the system loader will load from the file. In the linker script, you specify the contents of these segments by placing allocatable output sections in the segments. You use the ':phdr' output section attribute to place a section in a particular segment.

It is normal to put certain sections in more than one segment. This merely implies that one segment of memory contains another. You may repeat ':phdr', using it once for each segment which should contain the section.

If you place a section in one or more segments using ':phdr', then the linker will place all

subsequent allocatable sections which do not specify ':phdr' in the same segments. This is for convenience, since generally a whole set of contiguous sections will be placed in a single segment. You can use :NONE to override the default segment and tell the linker to not put the section in any segment at all.

You may use the FILEHDR and PHDRS keywords after the program header type to further describe the contents of the segment. The FILEHDR keyword means that the segment should include the ELF file header. The PHDRS keyword means that the segment should include the ELF program headers themselves. If applied to a loadable segment (PT_LOAD), it must be the first loadable segment.

The type may be one of the following. The numbers indicate the value of the keyword.
PT_NULL (0)
Indicates an unused program header.
PT_LOAD (1)
Indicates that this program header describes a segment to be loaded from the file.
PT_DYNAMIC (2)
Indicates a segment where dynamic linking information can be found.
PT_INTERP (3)
Indicates a segment where the name of the program interpreter may be found.
Chapter 3: Linker Scripts 63
PT_NOTE (4)
Indicates a segment holding note information.
PT_SHLIB (5)
A reserved program header type, defined but not specified by the ELF ABI.
PT_PHDR (6)
Indicates a segment where the program headers may be found.
expression An expression giving the numeric type of the program header. This may be used for types not defined above.
You can specify that a segment should be loaded at a particular address in memory by using an AT expression. This is identical to the AT command used as an output section attribute . The AT command for a program header overrides the output section attribute.
The linker will normally set the segment flags based on the sections which comprise the segment. You may use the FLAGS keyword to explicitly specify the segment flags. The value of flags must be an integer. It is used to set the p_flags field of the program header.
Here is an example of PHDRS. This shows a typical set of program headers used on a native ELF system.

```
PHDRS
{
headers PT_PHDR PHDRS ;
interp PT_INTERP ;
text PT_LOAD FILEHDR PHDRS ;
data PT_LOAD ;
dynamic PT_DYNAMIC ;
}
```

```
SECTIONS
{
. = SIZEOF_HEADERS;
.interp : { *(.interp) } :text :interp
.text : { *(.text) } :text
.rodata : { *(.rodata) } /* defaults to :text */
...
. = . + 0x1000; /* move to a new page in memory */
.data : { *(.data) } :data
.dynamic : { *(.dynamic) } :data :dynamic
...
}
```

1.9 VERSION Command

The linker supports symbol versions when using ELF. Symbol versions are only useful when using shared libraries. The dynamic linker can use symbol versions to select a specific version of a function when it runs a program that may have been linked against an earlier version of the shared library.

You can include a version script directly in the main linker script, or you can supply the version script as an implicit linker script. You can also use the '--version-script' linker option.

The syntax of the VERSION command is simply

VERSION { version-script-commands }

The format of the version script commands is identical to that used by Sun's linker in Solaris 2.5. The version script defines a tree of version nodes. You specify the node names and interdependencies in the version script. You can specify which symbols are bound to which version nodes, and you 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;
extern "C++" {
ns::*;
"int f(int, double)";
```

}
} VERS_1.2;
This example version script defines three version nodes. The first version node defined is 'VERS_1.1'; it has no other dependencies. The script binds the symbol 'foo1' to 'VERS_1.1'. It reduces a number of symbols to local scope so that they are not visible outside of the shared library; this is done using wildcard patterns, so that any symbol whose name begins with 'old', 'original', or 'new' is matched. The wildcard patterns available are the same as those used in the shell when matching filenames (also known as "globbing"). However, if you specify the symbol name inside double quotes, then the name is treated as literal, rather than as a glob pattern.

Next, the version script defines node 'VERS_1.2'. This node depends upon 'VERS_1.1'. The script binds the symbol 'foo2' to the version node 'VERS_1.2'.
Finally, the version script defines node 'VERS_2.0'. This node depends upon 'VERS_1.2'. The scripts binds the symbols 'bar1' and 'bar2' are bound to the version node 'VERS_2.0'. When the linker finds a symbol defined in a library which is not specifically bound to a version node, it will effectively bind it to an unspecified base version of the library.

You can bind all otherwise unspecified symbols to a given version node by using 'global: *;' somewhere in the version script. Note that it's slightly crazy to use wildcards in a global spec except on the last version node. Global wildcards elsewhere run the risk of accidentally adding symbols to the set exported for an old version. That's wrong since older versions ought to have a fixed set of symbols.

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. Node name can be omitted, provided it is the only version node in the version script. Such version script doesn't assign any versions to symbols, only selects which symbols will be globally visible out and which won't.
{ global: foo; bar; local: *; };
When you link an application against a shared library that has versioned 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 SunOS 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. You can do this by putting something like:

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

in the C source file. This renames 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. A '.symver' directive takes precedence over a version script.

The second GNU extension is to allow multiple versions of the same function to appear in a given shared library. In this way you can make an incompatible change to an interface without increasing the major version number of the shared library, while still allowing applications linked against the old interface to continue to function.
To do this, you must use multiple '.symver' directives in the source file. Here is an example:
__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. You can do this with the 'foo@@VERS_2.0' type of '.symver' directive. You can only declare one version of a symbol as the 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 (i.e., 'old_foo'), or you can use the '.symver' directive to specifically bind to an external version of the function in question.
You can also specify the language in the version script:
VERSION extern "lang" { version-script-commands }
The supported 'lang's are 'C', 'C++', and 'Java'. The linker will iterate over the list of symbols at the link time and demangle them according to 'lang' before matching them to the patterns specified in 'version-script-commands'.
Demangled names may contains spaces and other special characters. As described above, you can use a glob pattern to match demangled names, or you can use a double-quoted string to match the string exactly. In the latter case, be aware that minor differences (such as differing whitespace) between the version script and the demangler output will cause a mismatch. As the exact string generated by the demangler might change in the future, even if the mangled name does not, you should check that all of your version directives are behaving as you expect when you upgrade.

1.10 Expressions in Linker Scripts

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as integers. All expressions are evaluated in the same size, which is 32 bits if both the host and target are 32 bits, and is otherwise 64 bits.
You can use and set symbol values in expressions.
The linker defines several special purpose builtin functions for use in expressions.

1.10.1 Constants

All constants are integers.
As in C, the linker considers an integer beginning with ‘0’ to be octal, and an integer beginning with ‘0x’ or ‘0X’ to be hexadecimal. Alternatively the linker accepts suffixes of ‘h’ or ‘H’ for hexadeciaml, ‘o’ or ‘O’ for octal, ‘b’ or ‘B’ for binary and ‘d’ or ‘D’ for decimal. Any integer value without a prefix or a suffix is considered to be decimal.
In addition, you can use the suffixes K and M to scale a constant by 1024 or 10242 respectively. For example, the following all refer to the same quantity:
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
_fourk_4 = 10000o;
Note - the K and M suffixes cannot be used in conjunction with the base suffixes mentioned above.

1.10.2 Symbolic Constants

It is possible to refer to target specific constants via the use of the CONSTANT(name) operator, where name is one of:
MAXPAGESIZE
The target’s maximum page size.
COMMONPAGESIZE
The target’s default page size.
So for example:
.text ALIGN (CONSTANT (MAXPAGESIZE)) : { *(.text) }
will create a text section aligned to the largest page boundary supported by the target.

1.10.3 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which 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.

1.10.4 Orphan Sections

Orphan sections are sections present in the input files which are not explicitly placed into the output file by the linker script. The linker will still copy these sections into the output file, but it has to guess as to where they should be placed. The linker uses a simple heuristic to do this. It attempts to place orphan sections after non-orphan sections of the same attribute, such as code vs data, loadable vs non-loadable, etc. If there is not enough room to do this then it places at the end of the file.
For ELF targets, the attribute of the section includes section type as well as section flag.
If an orphaned section's name is representable as a C identifier then the linker will automatically two symbols: start SECNAME and end SECNAME, where SECNAME is the name of the section. These indicate the start address and end address of the orphaned section respectively. Note: most section names are not representable as C identifiers because they contain a '.' character.

1.10.5 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 may only appear in an expression within a SECTIONS command. The . symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to . will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may not be moved backwards inside an output section, and may not be moved backwards outside of an output section if so doing creates areas with overlapping LMAs.

```
SECTIONS
{
output :
{
file1(.text)
. = . + 1000;
file2(.text)
. += 1000;
file3(.text)
} = 0x12345678;
}
```

In the previous example, the '.text' section from 'file1' is located at the beginning of the output section 'output'. It is followed by a 1000 byte gap. Then the '.text' section from 'file2' appears, also with a 1000 byte gap following before the '.text' section from 'file3'. The notation '= 0x12345678' specifies what data to write in the gaps.

Note: . actually refers to the byte offset from the start of the current containing object. Normally this is the SECTIONS statement, whose start address is 0, hence . can be used as an absolute address. If . is used inside a section description however, it refers to the byte offset from the start of that section, not an absolute address. Thus in a script like this:

```
SECTIONS
{
. = 0x100
```

```
.text: {
*(.text)
. = 0x200
}
. = 0x500
.data: {
*(.data)
. += 0x600
}
}
```

The '.text' section will be assigned a starting address of 0x100 and a size of exactly 0x200 bytes, even if there is not enough data in the '.text' input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move . backwards). The '.data' section will start at 0x500 and it will have an extra 0x600 bytes worth of space after the end of the values from the '.data' input sections and before the end of the '.data' output section itself.

Setting symbols to the value of the location counter outside of an output section statement can result in unexpected values if the linker needs to place orphan sections. For example, given the following:

```
SECTIONS
{
start_of_text = . ;
.text: { *(.text) }
end_of_text = . ;
start_of_data = . ;
.data: { *(.data) }
end_of_data = . ;
}
```

If the linker needs to place some input section, e.g. .rodata, not mentioned in the script, it might choose to place that section between .text and .data. You might think the linker should place .rodata on the blank line in the above script, but blank lines are of no particular significance to the linker. As well, the linker doesn't associate the above symbol names with their sections. Instead, it assumes that all assignments or other statements belong to the previous output section, except for the special case of an assignment to .. I.e., the linker will place the orphan .rodata section as if the script was written as follows:

```
SECTIONS
{
start_of_text = . ;
.text: { *(.text) }
end_of_text = . ;
start_of_data = . ;
.rodata: { *(.rodata) }
.data: { *(.data) }
end_of_data = . ;
}
```

This may or may not be the script author's intention for the value of start_of_data. One way to influence the orphan section placement is to assign the location counter to itself, as the linker assumes that an assignment to . is setting the start address of a following output section and thus should be grouped with that section. So you could write:

```
SECTIONS
{
start_of_text = . ;
.text: { *(.text) }
end_of_text = . ;
. = . ;
start_of_data = . ;
.data: { *(.data) }
end_of_data = . ;
}
```

Now, the orphan .rodata section will be placed between end_of_text and start_of_data.

1.10.6 Operators

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

Precedence Associativity Operators
highest
1 left - ~ ! y
2 left * / %
3 left + -
4 left >> <<
5 left == != > < <= >=
6 left &
7 left |
8 left &&
9 left ||
10 right ? :
11 right &= += -= *= /= z
Lowest y Prefix operators.

1.10.7 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and 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 linker script.

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.

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 '.', must be evaluated during section 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 :
{ *(.text) }
}
```

will cause the error message 'non constant expression for initial address'.

1.10.8 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.
The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the '-r' option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.
A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.
You can use the builtin function ABSOLUTE to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section '.data':

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

If 'ABSOLUTE' were not used, '_edata' would be relative to the '.data' section.

1.10.9 Builtin Functions

The linker script language includes a number of builtin functions for use in linker 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 (the VMA) 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;
}
... }
```

ALIGN(align)
ALIGN(exp,align)
Return the location counter (.) or arbitrary expression aligned to the next align boundary. The single operand ALIGN doesn't change the value of the location counter—it just does arithmetic on it. The two operand ALIGN allows an arbitrary expression to be aligned upwards (ALIGN(align) is equivalent to ALIGN(., align)).

Here is an example which aligns the output .data section to the next 0x2000 byte boundary after the preceding section and sets 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 address attribute of a section definition. The second use of ALIGN is used to defines the value of a symbol.

The builtin function NEXT is closely related to ALIGN.
ALIGNOF(section)
Return the alignment in bytes of the named section, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, the alignment of the .output section is stored as the first value in that section.

```
SECTIONS{ ...
.output {
LONG (ALIGNOF (.output))
...
}
... }
```

BLOCK(exp)
This is a synonym for ALIGN, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.
DATA_SEGMENT_ALIGN(maxpagesize, commonpagesize)
This is equivalent to either
(ALIGN(maxpagesize) + (. & (maxpagesize - 1)))
or
(ALIGN(maxpagesize) + (. & (maxpagesize - commonpagesize)))
depending on whether the latter uses fewer commonpagesize sized pages for the data segment (area between the result of this expression and DATA_SEGMENT_ END) than the former or not. If the latter form is used, it means commonpagesize bytes of runtime memory will be saved at the expense of up to commonpagesize wasted bytes in the on-disk file.

This expression can only be used directly in SECTIONS commands, not in any output section descriptions and only once in the linker script. commonpagesize should be less or equal to maxpagesize and should be the system page size the object wants to be optimized for (while still working on system page sizes up to maxpagesize).

Example:
. = DATA_SEGMENT_ALIGN(0x10000, 0x2000);
DATA_SEGMENT_END(exp)
This defines the end of data segment for DATA_SEGMENT_ALIGN evaluation purposes.
. = DATA_SEGMENT_END(.);

DATA_SEGMENT_RELRO_END(offset, exp)
This defines the end of the PT_GNU_RELRO segment when ‘-z relro’ option is used. Second argument is returned. When ‘-z relro’ option is not present, DATA_SEGMENT_RELRO_END does nothing, otherwise DATA_SEGMENT_ALIGN is padded so that exp + offset is aligned to the most commonly used page boundary for particular target. If present in the linker script, it must always come in between DATA_SEGMENT_ALIGN and DATA_SEGMENT_END.

. = DATA_SEGMENT_RELRO_END(24, .);
DEFINED(symbol)
Return 1 if symbol is in the linker global symbol table and is defined before the statement using DEFINED in the script, otherwise return 0. You can use this function to provide default values for symbols. For example, the following script 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 : . ;
...
}
...
}
LENGTH(memory)

Return the length of the memory region named memory.
LOADADDR(section)
Return the absolute LMA of the named section. This is normally the same as ADDR, but it may be different if the AT attribute is used in the output section definition .
MAX(exp1, exp2)
Returns the maximum of exp1 and exp2.
MIN(exp1, exp2)
Returns the minimum of exp1 and exp2.
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. ORIGIN(memory)

Return the origin of the memory region named memory.

SEGMENT_START(segment, default)

Return the base address of the named segment. If an explicit value has been given for this segment (with a command-line '-T' option) that value will be returned; otherwise the value will be default. At present, the '-T' commandline option can only be used to set the base address for the "text", "data", and "bss" sections, but you use SEGMENT_START with any segment name.

SIZEOF(section)
Return the size in bytes of the named section, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. 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 returns the size in bytes of the output file's headers. This is information which appears at the start of the output file. You can use this number when setting the start address of the first section, if you choose, to facilitate paging. When producing an ELF output file, if the linker script uses the SIZEOF_ HEADERS builtin function, the linker must compute the number of program headers before it has determined all the section addresses and sizes. If the linker later discovers

that it needs additional program headers, it will report an error ‘not enough room for program headers’. To avoid this error, you must avoid using the SIZEOF_HEADERS function, or you must rework your linker script to avoid forcing the linker to use additional program headers, or you must define the program headers yourself using the PHDRS command.

1.11 Implicit Linker Scripts

If you specify a linker input file which the linker cannot recognize as an object file or an archive file, it will try to read the file as a linker script. If the file can not be parsed as a linker script, the linker will report an error.

An implicit linker script will not replace the default linker script. Typically an implicit linker script would contain only symbol assignments, or the INPUT, GROUP, or VERSION commands.

Any input files read because of an implicit linker script will be read at the position in the

command line where the implicit linker script was read. This can affect archive searching

2 The Linker and the ARM family

For the ARM, ld will generate code stubs to allow functions calls between ARM and Thumb code. These stubs only work with code that has been compiled and assembled with the ‘-mthumb-interwork’ command line option. If it is necessary to link with old ARM object files or libraries, which have not been compiled with the -mthumb-interwork option then the ‘--support-old-code’ command line switch should be given to the linker. This will make it generate larger stub functions which will work with non-interworking aware ARM code.

Note, however, the linker does not support generating stubs for function calls to non-interworking aware Thumb code.

The ‘--thumb-entry’ switch is a duplicate of the generic ‘--entry’ switch, in that it sets the program’s starting address. But it also sets the bottom bit of the address, so that it can be branched to using a BX instruction, and the program will start executing in Thumb mode straight away.

The ‘--use-nul-prefixed-import-tables’ switch is specifying, that the import tables idata4 and idata5 have to be generated with a zero element prefix for import libraries. This is the old style to generate import tables. By default this option is turned off.

The ‘--be8’ switch instructs ld to generate BE8 format executable. This option is only valid when linking big-endian objects. The resulting image will contain big-endian data and little-endian code.

The ‘R_ARM_TARGET1’ relocation is typically used for entries in the ‘.init_array’ section.
It is interpreted as either ‘R_ARM_REL32’ or ‘R_ARM_ABS32’, depending on the target. The ‘--target1-rel’ and ‘--target1-abs’ switches override the default.

The ‘--target2=type’ switch overrides the default definition of the ‘R_ARM_TARGET2’ relocation. Valid values for ‘type’, their meanings, and target defaults are as follows:

‘rel’ ‘R_ARM_REL32’ (arm*-*-elf, arm*-*-eabi)
‘abs’ ‘R_ARM_ABS32’ (arm*-*-symbianelf)
‘got-rel’ ‘R_ARM_GOT_PREL’ (arm*-*-linux, arm*-*-*bsd)

The ‘R_ARM_V4BX’ relocation (defined by the ARM AAELF specification) enables objects compiled for the ARMv4 architecture to be interworking-safe when linked with other objects compiled for ARMv4t, but also allows pure ARMv4 binaries to be built from the same ARMv4 objects.

In the latter case, the switch ‘--fix-v4bx’ must be passed to the linker, which causes v4t BX rM instructions to be rewritten as MOV PC,rM, since v4 processors do not have a BX instruction.

In the former case, the switch should not be used, and ‘R_ARM_V4BX’ relocations are ignored. Replace BX rM instructions identified by ‘R_ARM_V4BX’ relocations with a branch to the following veneer:

TST rM, #1
MOVEQ PC, rM
BX Rn

This allows generation of libraries/applications that work on ARMv4 cores and are still interworking safe. Note that the above veneer clobbers the condition flags, so may cause incorrect progrm behavior in rare cases.

The ‘--use-blx’ switch enables the linker to use ARM/Thumb BLX instructions (available on ARMv5t and above) in various situations. Currently it is used to perform calls via the PLT from Thumb code using BLX rather than using BX and a mode-switching stub before each PLT entry. This should lead to such calls executing slightly faster.
This option is enabled implicitly for SymbianOS, so there is no need to specify it if you are using that target.

The ‘--vfp11-denorm-fix’ switch enables a link-time workaround for a bug in certain VFP11 coprocessor hardware, which sometimes allows instructions with denorm operands (which must be handled by support code) to have those operands overwritten by subsequent instructions before the support code can read the intended values.

The bug may be avoided in scalar mode if you allow at least one intervening instruction between a VFP11 instruction which uses a register and another instruction which writes to

the same register, or at least two intervening instructions if vector mode is in use. The bug only affects full-compliance floating-point mode: you do not need this workaround if you are using "runfast" mode. Please contact ARM for further details.

This workaround is enabled for scalar code by default for pre-ARMv7 architectures, but disabled by default for later architectures. If you know you are not using buggy VFP11 hardware, you can disable the workaround by specifying the linker option '--vfp-denorm-fix=none'. If you are using VFP vector mode, you should specify '--vfp-denorm-fix=vector'.

If the workaround is enabled, instructions are scanned for potentially-troublesome sequences, and a veneer is created for each such sequence which may trigger the erratum. The veneer consists of the first instruction of the sequence and a branch back to the subsequent instruction. The original instruction is then replaced with a branch to the veneer. The extra cycles required to call and return from the veneer are sufficient to avoid the erratum in both the scalar and vector cases.

The '--no-enum-size-warning' switch prevents the linker from warning when linking object files that specify incompatible EABI enumeration size attributes. For example, with this switch enabled, linking of an object file using 32-bit enumeration values with another using enumeration values fitted into the smallest possible space will not be diagnosed.

The '--no-wchar-size-warning' switch prevents the linker from warning when linking object files that specify incompatible EABI wchar_t size attributes. For example, with this switch enabled, linking of an object file using 32-bit wchar_t values with another using 16-bit wchar_t values will not be diagnosed.

The '--pic-veneer' switch makes the linker use PIC sequences for ARM/Thumb interworking veneers, even if the rest of the binary is not PIC. This avoids problems on uClinux targets where '--emit-relocs' is used to generate relocatable binaries.

The linker will automatically generate and insert small sequences of code into a linked ARM ELF executable whenever an attempt is made to perform a function call to a symbol that is too far away. The placement of these sequences of instructions - called stubs - is controlled by the command line option '--stub-group-size=N'. The placement is important because a poor choice can create a need for duplicate stubs, increasing the code sizw. The linker will try to group stubs together in order to reduce interruptions to the flow of code, but it needs guidance as to how big these groups should be and where they should be placed. The value of 'N', the parameter to the '--stub-group-size=' option controls where the stub groups are placed. If it is negative then all stubs are placed after the first branch that needs them. If it is positive then the stubs can be placed either before or after the branches that need them. If the value of 'N' is 1 (either +1 or -1) then the linker will choose exactly where to place groups of stubs, using its built in heuristics. A value of 'N' greater than 1 (or smaller than -1) tells the linker that a single group of stubs can service at most 'N' bytes from the input sections.

The default, if '--stub-group-size=' is not specified, is 'N = +1'.
Far calls stubs insertion is fully supported for the ARM-EABI target only, because it relies on object files properties not present otherwise.

The '--fix-cortex-a8' switch enables a link-time workaround for an erratum in certain Cortex-A8 processors. The workaround is enabled by default if you are targeting the ARM v7-A architecture profile. It can be enabled otherwise by specifying '--fix-cortex-a8', or disabled unconditionally by specifying '--no-fix-cortex-a8'.
The erratum only affects Thumb-2 code. Please contact ARM for further details.