CodeWarrior Development Studio for StarCore DSP Architectures

SC100 Linker User Guide

Revised: May 4, 2012

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, TX 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

This chapter provides an introduction to the StarCore linker. This chapter contains the following topics:

1.1 Actions of the StarCore Linker

The StarCore Linker generates an executable file by combining the object files and libraries specified in your project. Using a linker command file (LCF), you can instruct the linker to store portions of your executable in different areas of memory. The linker relocates and binds symbols to addresses according to the directives in your LCF.

NOTE The StarCore linker supports the SC1000 and SC3000 architectures, and all Freescale devices based on these architectures, including the SC140 and SC3400 and its variants.

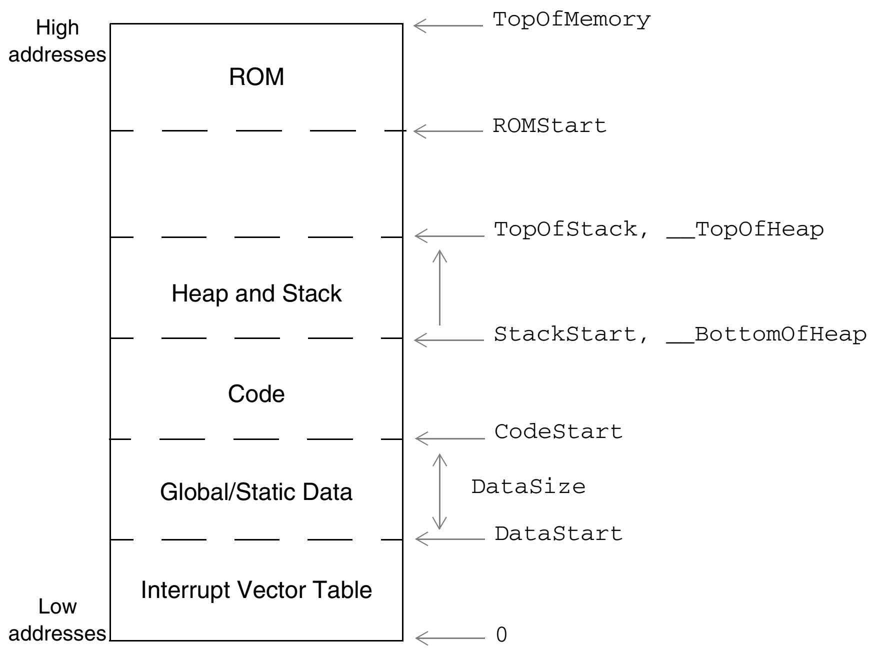
The highlights of the linking process are:

1. Build a map of available memory according to the .memory and .reserve directives defined in the linker command file.
2. Read the input object files and libraries. Combine like-named sections, build the global symbol table, and resolve undefined references.
3. Strip data never used; strip the code of functions never called.
4. Place segments in memory, according to `.firstfit`, `.org`, and `.segment` directives. Place all absolute sections before placing relocatable sections. If any sections are left over after processing the `.segment` directives, place those remaining sections in their own segments on a first-fit basis.
5. Generate any `LoadAddr_` symbols for overlays. Check each module's symbol table to ensure that there are no more undefined references.
6. Process the relocations to resolve external reference values.
7. Write the executable output file.

1.2 Memory Layout and Configuration for Single Core

The default memory layout (for single core, such as SC140, SC3400 etc.) is a single linear block, divided into data and code areas, as Figure 1.1 depicts.

Figure 1.1 Default Memory Layout



Note that the heap and stack use the same area of memory: this represents *dynamic* configuration, which is the default. (In a *static* configuration, the stack and heap use different areas of memory.)

All three memory models use the same default layout, but with different values that define the distribution of the memory areas. The definitions of these values are in the three linker command files provided in `$SC100_HOME/etc`, as Table 1.1 shows.

Table 1.1 Default Memory Values

Memory Model	Contents	From	To	Default
Unspecified	Interrupt vector table	0	0x1ff	
	Global & static variables	DataStart	DataStart+DataSize-1	0x0200
	Program code	CodeStart	StackStart-1	0x100000
	Stack and heap	StackStart	TopOfStack	0x200000
	ROM	ROMStart	TopOfMemory	0x300000
crtscsmm.cmd	Interrupt vector table	0	0x1ff	
	Global & static variables	DataStart	DataStart+DataSize-1	0x101ff
	Program code	CodeStart	StackStart-1	0x27fff
	Stack and heap	StackStart	TopOfStack	0x2fff00
	ROM	ROMStart	TopOfMemory	0x3ffff
crsctmm.cmd	Interrupt vector table	0	0x1ff	0x1ff
	Global & static variables	DataStart	DataStart+DataSize-1	0x81ff
	Program code	CodeStart	StackStart-1	0x27fff
	Stack and heap	StackStart	TopOfStack	0x2fff00
	ROM	ROMStart	TopOfMemory	0x3ffff
crtscbmm.cmd	Interrupt vector table	0	0x1ff	
	Global & static variables	DataStart	DataStart+DataSize-1	0xfffff
	Program code	CodeStart	StackStart-1	0x3ffff
	Stack and heap	StackStart	TopOfStack	0x2fff00
	ROM	ROMStart	TopOfMemory	0x3ffff

You can change these default values and configure the memory map to meet your specific requirements by modifying the appropriate linker command file. Alternatively, you can create your own linker command file. When configuring the memory map, keep these points in mind:

- Make sure that the code size and data size values do not overlap.
- Locating the stack and heap together, in one contiguous area of memory, tracks with dynamic configuration.
- You may split and distribute other sections of memory over non-contiguous parts of memory, as required.

Use the `-c` option to force the linker to use a command file other than the default `crtscsmm.cmd`.

1.3 Startup Environment

The linker startup code (specifically for MSC8144 and MSC8156) consists of these steps:

1. Initialize SR with the default settings
2. Initialize the temp stack pointer
3. Initialize the VBA to .invec's origin
4. Disable translation and protection
5. Initialize C variables (zero `.bss` sections)
6. First HOOK (function `__target_asm_start`)

   This hook is used to:

   - enable MMU
   - define the translation for stack and heap
   - disable memory protection

   The memory descriptor that contains the stack and MMU tables is defined using these symbols in the LCF:

   - `_LocalData_b`: memory descriptor virtual address
   - `_LocalData_size`: memory descriptor size
   - `_LocalData_Phys_b`: memory descriptor physical address

     This symbol can be replaced by `_LocalData_b` for 1:1 mappings. In addition, this address is used to compute the physical address for all the cores by using this expression:
     `_LocalData_Phys_b + core_id * _LocalData_size`

   Table 1.2 shows the First HOOK code in LCF.

Table 1.2 First HOOK Code in LCF

Code in LCF	Description
`.provide _ENABLE_MMU_TRANSLATION, 1`	Enables the MMU translation. If the value is 1, the Address Translation Enable (ATE) bit in the MMU Control Register will be set by the `__target_c_start` function.
`SYSTEM_DATA_MMU_DEF`	Specifies different attributes of MMU

The ATE bit enables or disables the address translation mechanism. If disabled, addresses are not translated (for example, from a virtual address to a physical address). The reset value is configured according to external DSP subsystem plug.

7. Initialize the stack pointer

   The code in LCF for initializing the stack pointer is:

   `_StackStart`

8. Second HOOK (function `__target_c_start()`)

   This hook is used to:

   - define the MMU translations
   - define the memory protection
   - set a system task and a user task

   This function is called after setting the stack pointer and before C/C++ initializations.

   Table shows the Second HOOK code in LCF.

Table 1.3 Second HOOK Code in LCF

Code in LCF	Description
`.provide _ENABLE_MMU_PROTECTION, 1`	Enables the MMU protection. If the value is 1, the Memory Protection Enable (MPE) bit in the MMU Control Register will be set by the `__target_c_start` function.
`.provide _ENABLE_MMU_TRANSLATION, 1`	Enables the MMU translation. If the value is 1, the ATE bit in the MMU Control Register will be set by the `__target_c_start` function.

Table 1.3 Second HOOK Code in LCF

Code in LCF	Description
`.provide _SYSTEM_TASK_ID, 0`	Specifies the system task identifier. All descriptors for this task are enabled in MATT. Used in function `__target_c_start` to test if the descriptor belongs to the system task.
`.provide _ENABLE_DEFAULT_TASK_ID, 1`	Specifies the user task identifier. The descriptors of this task are set by `__target_c_start` function. There is only one user task.

`.provide _MMU_HIGH_PRIORITY, 0x10000000`	Used in `.att_mmu_settings` to define the descriptor attribute bitmask. The descriptor attribute bitmask specifies the permission for descriptor ranges to overlap in the virtual memory. Based on this information, the linker can check the associated priority and the descriptor overlapping in virtual memory. This attribute can be set in `.att_mmu` directive in the attribute field:
	`.att_mmu "Data_private_mmu", \`
	`_VIRTUAL_PRIVATE_MEM_DATA_start, \`
	`_VIRTUAL_PRIVATE_MEM_DATA_end, \`
	`“descriptor__m2__cacheable_wb__sys__private__data”, \`
	`attribute: SYSTEM_DATA_MMU_DEF | _MMU_HIGH_PRIORITY, \`
	`after_physical_address: _M2_PRIVATE_start`
	The symbol is used in `__target_c_start` function to relatively sort the MATT entries so that two virtual memory overlapping descriptors preserve the priority that you specify.

NOTE The MATT descriptors priority mechanism allows memory regions to overlap. This mechanism is handled by dividing the MATT into separate pairs of descriptors. Each pair contains two sequential index descriptors. For example: descriptor 0 is coupled with descriptor 1, descriptor 2 is coupled with descriptor 3, and so on. In addition, each descriptor has a fixed priority only in the specific pair. The priority is determined by the index number of the descriptor. The higher index number has a higher priority. For example, descriptor 3 has a higher priority over descriptor 2, but no priority over other pairs of descriptors. The descriptor with _MMU_HIGH_PRIORITY attribute is indexed with an odd value and its pair is indexed with a lower even value.

MPE (Memory Protection Enable) bit is a central bit that enables or disables the protection-checking function in all the enabled segment descriptors. It also enables or disables the miss interrupt support on a miss access.

9. Third HOOK (function `__target_setting()`)

   This hook is used to perform target specific settings, and it is empty by default. This hook is called before enabling the interrupt system. At this level, you can write code in either C/C++ or Assembly language.

   The code in LCF for Third HOOK is:

   `.provide _ENABLE_CACHE, 1`

   Activates the cache (L1 and L2) on startup. To deactivate, set the value to "-1".

   For MSC8156 architecture, you specify M2 memory size using a rule set and a new symbol, `_M2_Setting`. The `_M2_Setting` symbol has replaced the `_L2_Setting` symbol.

   Listing 1.1 shows the rule set.

Listing 1.1 Rule Set for _M2_Setting Symbol Values

```
; --------------------------------
; _M2_Setting configuration for M2
; ------------------
; M2 size  Rule-set
; ------------------
;   0KB     0x00 - all used as L2Cache
;  64KB     0x01
; 128KB     0x03
; 192KB     0x07
; 256KB     0x0f
; 320KB     0x1f
; 384KB     0x3f
; 448KB     0x7f
; 512KB     0xff
.provide _M2_Setting,  0x03
```

   If you specify the value `0x00`, entire memory is used as L2 cache.

10. Set the `argv` and `argc` arguments

11. Enable the interrupts

12. Jump to `main()` function

2

Using the Linker

This chapter explains how to use the StarCore linker. Sections are:

- 2.1 Invoking the Linker
- 2.2 Command-Line Options

2.1 Invoking the Linker

There are two ways to invoke the StarCore linker:

- Launching the linker as part of the CodeWarrior IDE
- Entering this command on the command line:

```
sc100-ld [option ...] {file | -larchive} ...
```

where:

`option`

One or more of the Table 2.1 options. These options and their arguments are case sensitive. You must maintain the spacing shown; for example, there must be a space between the `-c` option and its argument, but the `-l` option must not have such a space.

`file`

One or more relocatable object files. The linker processes the files in their command-line order.

`-l archive`

The `-l` option and the name of an archive. The linker automatically appends the `.elb` extension to the archive name. (Unlike earlier linker releases, this linker release does not add the `lib` prefix to the archive name.)

You may specify `-l`*`archive`* multiple times on the command line. The linker processes the archives in their command-line order. The linker's default (standard) search path is `$SC100_HOME/lib`.

2.2 Command-Line Options

Table 2.1 summarizes command-line options available with the linker. Subsequent subsections explain these options in more detail.

NOTE All options of Table 2.1 are valid with the `-Xlnk` passthrough option of the `scc` command line.

Table 2.1 StarCore Linker Command-Line Options

Option	Description
`-allow-multiple-definition`	Allows the use of multiple definitions for a symbol. The first definition found is used. By default, the sc100-ld linker does not allow multiple definitions. To disable again, if enabled, use `-disable-allow-multiple-definition`.
`-arch` *`name`*	Specifies the target architecture. The *name* value can be sc110, sc120, sc140, sc140e, msc8101, starlite, sc1200, sc1400, sc2200, sc2400, sc3200, sc3400, or msc8144.
`-bsstab-load`	Tells linker to use load address (physical address) when it creates .bsstab section. (Changes default behavior for msc8144 architectures.)
`-bsstab-run`	Tells linker to use run address (virtual address) when it creates .bsstab section. (Changes default behavior for all architectures but msc8144.)
`-bsstable-file <output_file.txt>`	Lets you skip emitting SREC records for the `.bss` type sections, which are not placed at the end of the segment. When a .bss type section is not placed at the end of the segment, the section is converted to a data type section during the linking process. Therefore, no address and size information for such sections exist in the `__bss_table`. When you use the `-bsstable-file <output_file.txt>` command to be able to skip emitting SREC records for such sections, the linker generates a new output file that contains the required information. The new output file is a text file that contains a table with the two columns: • physical_address; represented by a 32-bit hexadecimal unsigned integer • size; in bytes; represented by a 32-bit hexadecimal unsigned integer Note that in case of multi-core msc8144 and msc8156 architectures, the linker generates a single output file for all cores. The syntax for running the `-bsstable-file` command is: `sc100-ld -bsstable-file bss_table_file.txt file1.eln file2.eln file3.eln lib1.elb -o file_out.eld`
`-c` *`commandfile`*	Specifies the name and location of the linker command file.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-check-mnemonics`	Checks for `bmtset.w` instructions; issues a warning at link time if such an instruction conflicts with starlite architecture. (Opposite option is `-disable-check-mnemonics`.)
`-check-mnemonics-errors`	Checks for `bmtset.w` instructions; issues an error message at link time if such an instruction conflicts with starlite architecture.
`-C`	Tells the linker to *not* use a linker command file. (Appropriate for any programs that explicitly place code and data in memory, for example, by means of assembly-language `.org` directives.)
`-enable-error-placing-section-on-first-fit-basis`	Tells the linker to generate an error message for each section when it is not explicitly mentioned in LCF. These sections are placed on first fit basis algorithm.
`-enable-warn-section-has-not-been-found`	Tells the linker to generate a warning message for each section not found.
`-debug-reloc-load`	Uses load address to compute debug-section relocation information. (Default option; new debugger support for overlay sections.)
`-debug-reloc-run`	Uses run address to compute debug-section relocation information. (Old debugger support for overlay sections; for backwards compatibility.)
`-disable-check-ELF`	Stops checking the ELF format for input files. (Opposite option is `-enable-check-ELF`, which is the default.)
`-disable-check-mnemonics`	Stops checking for `bmtset.w` instructions, suppressing associated warnings or error messages. (Opposite option is `-check-mnemonics`.)
`-disable-display-padding-info`	Tells the linker to *not* output information for padding in map file. (Default option. Opposite option is `-enable-display-padding-info`.)
`-disable-emit-att_mmu`	Overrides default creation of an `.att_mmu` section if overlay sections are involved with the address translation table. (Opposite option is `-enable-emit-att_mmu`, which is the default.)
`-disable-emit-bsstab`	Tells the linker to *not* create a `.bsstab` section. (Opposite option is `-enable-emit-bsstab`, which is the default.)
`-disable-emit-note`	Tells the linker to *not* create a `.note` section. (Opposite option is `-enable-emit-note`, which is the default.)

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-disable-emit-signature-info`	This option disables the storing of information in the `other/ovl_other` field from MMU table/overlay table if the signature for section is not set. This is default option.
`-disable-emit-ovltab`	Tells the linker to *not* create an `.ovltab` section. (Opposite option is `-enable-emit-ovltab`, which is the default.)
`-disable-error-placing-section-on-first-fit-basis`	Tells the linker to inhibit error message generation for each section when it is not explicitly mentioned in LCF. These sections are placed on first fit basis algorithm. This is a default option.
`-disable-exception`	Removes exception support for the application: • Excludes `.exception` and `.exception_index` sections from linking process. • Defines symbols `ENABLE_EXCEPTION`, `__unexpected`, `__throw`, `__rethrow`, and `__end_catch` with value 0. (The default LCF file uses `ENABLE_EXCEPTION`; the other symbols inhibit exception support from the runtime library.) (Opposite option is `-enable exception`, which is the default.)
`-disable-mmu-support`	Inhibits automatic overlay support for sections that the `.att_mmu` and `.concatenate` directives mention. (Opposite option is `-enable-mmu-support`, which is the default.)
`-disable-placing-after-same-segment-on-first-fit-basis`	Tells the linker to set the placing to start at the address specified by the .firstfit directive on a first fit basis.
`-disable-remove-dead-symbols`	Tells the linker to keep unreferenced symbols in the ELD file, preserving backwards compatibility. (Opposite option is `-enable-remove-dead-symbols`, which is the default.)
`-disable-rewrite-padding-between-objects`	Tells the linker to accept the assembler's padding between objects.
`-disable-sort-exception-index`	Tells the linker to *not* sort the exception table index. (Opposite option is `-enable-sort-exception-index`, which is the default.)
`-disable-seq-link`	Turns off sequential linking mode. This is the default behavior.
`-disable-stack-effect`	Tells the linker to *not* compute the stack effect (Default option. Opposite option is `-enable-stack-effect`.)

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-disable-warn-stack-effect`	Tells the linker to *not* generate a warning if stack-effect estimation must consider recursively called functions, even though such functions impair the accuracy of the estimate. (Opposite option is `-enable-warn-stack-effect`.)
`-disable-warn-placing-section-on-first-fit-basis`	Tells the linker to inhibit warning message generation for each section when it is placed using the first fit basis algorithm. This is the default.
`-disable-xd`	Tells linker to preserve backward compatibility by keeping compiler/assembler symbols for debugging sections. (Default option. Opposite option is `-xd`.)
`-disable-warn-section-has-not-been-found`	Tells the linker to inhibit warning message generation for each section not found. This is the default option.
`-display-info<level>-in-map`	Specifies information that a map file displays: level = 0: all global symbols level = 1: all global symbols, local objects, and local functions level = 2: all global symbols, local objects, local functions, and local no-type symbols
`-D`*`symbol`*`=`*`value`*	Defines a global symbol, overriding any `.provide` symbol of the same name that may exist. The `value` argument specification may be in octal (leading `0`), decimal, or hexadecimal (leading `0x` or `0X`).
`-enable-check-ELF`	Tells the linker to check the ELF format for input files. (Default option. Opposite option is `-disable-check-ELF`.)
`-enable-compress`	Enables compression support for program overlay sections. (Without this option, overlay sections lack this support, because the `SEC_WRITE` property is not set.)
`-enable-display-padding-info`	Displays information for padding in map file. (Opposite option is `-disable-display-padding-info`, which is the default.)
`-enable-emit-att_mmu`	Tells the linker to create an `.att_mmu` section in response to any reference to the _address_translation_table_mmu[]. (Default option. The `.att_mmu` section specifies the sections involved in the memory address translation table. Any undefined reference to the __address_translation_table_mmu symbol blocks dead stripping. Opposite option is `-disable-emit-att_mmu`.)

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-enable-emit-bsstab`	Tells the linker to create a `.bsstab` section in response to any reference to the _bss_table[]. (Default option. The startup file uses the _bss_table[] to initialize .bss sections with zero at runtime. Any undefined reference to the __bss_table symbol blocks dead stripping. Opposite option is `-disable-emit-bsstab`.)
`-enable-emit-note`	Tells the linker to create a `.note` section. (Default option. Opposite option is `-disable-emit-note`.)
`-enable-emit-ovltab`	Tells the linker to create an `.ovltab` section in response to any reference to the _overlay_table[]. (Default option. Any undefined reference to the __overlay_table symbol blocks dead stripping. Opposite option is `-disable-emit-ovltab`.)
`-enable-emit-shared-segment2cores`	Tells the linker to emit the PT_LOAD type of segment for all cores. This is the usual way to emit the loadable information.
`-enable-emit-shared-segment2cores-as-dynamic`	Tells the linker to emit segments as follows: - For the core that exports the shared space the linker emits the PT_LOAD type of segment. This is the usual way to emit the loadable information. - For the cores that import the shared space the linker emits the PT_DYNAMIC type of segment. The StarCore loader tool doesn't download this type of segment. This option is enabled by default when the MSC8144 architecture is selected.
`-enable-emit-shared-segment2cores-as-null`	Tells the linker to emit segments as follows: - For the core that exports the shared space the linker emits the PT_LOAD type of segment. This is the usual way to emit the loadable information. - For the cores that import the shared space the linker emits the PT_NULL type of segment. The StarCore loader tool doesn't download this type of segment. This option is enabled by default when the MSC8144 architecture is selected.
`-enable-emit-signature-info`	This option enables the storing of information in the `other/ovl_other` field from MMU table/overlay table if the signature for section is not set.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-enable-error-progbits-after-nobits`	Generates a warning message and continue the linking process, if SHT_PROGBITS section (a non-BSS section) is placed after SHT_NOBITS section (a BSS section). This is the default behavior. To disable, use `-disable-error-progbits-after-nobits`.
`-enable-exception`	Activates exception support, including ascending sorting of the exception table index by function address. (Default option. Opposite option is `-disable-exception`.)
`-enable-mmu-support`	Enables overlay support for all sections that the `.att_mmu` and `.concatenate` directives mention. (Default option. Opposite option is `-disable-mmu-support`.)
`-enable-placing-after-same-segment-on-first-fit-basis`	Tells the linker to set the placing to start after the same segment on a first fit basis, unless a similar segment is found, in which case the placing starts at the address specified by the .firstfit directive.
`-enable-remove-dead-symbols`	Tells the linker to remove unreferenced symbols from the ELD file. (Default option. Opposite option is `-disable-remove-dead-symbols`.)
`-enable-sort-exception-index`	Tells the linker to sort the exception index table ascendingly, by function address. (Default option. Opposite option is `-disable-sort-exception-index`.)
`-enable-seq-link`	Reduces the linker memory consumption by using core-by-core internal processing. This option can be used for a subset of multicore applications that follow these rules: • only core0 exports the shared sections, while all other cores import those shared sections • for each core, the linker control file has identical occurrences for the following directives: .xref .xref_module .exclude .rename
`-enable-stack-effect`	Tells the linker to estimate the stack effect. (Opposite option is `-disable-stack-effect`, which is the default.)
`-enable-undef`	Permits undefined symbols in a self-contained library. Valid only with the option `-self-contained-library`.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`enable-warn-no-stack-effect`	Generates a warning (with the list of files) for functions without `pragma stack_effect` information, when `-enable-stack-effect` option is enabled. This is the default option. To disable, use `disable-warn-no-stack-effect`.
`-enable-warn-placing-section-on-first-fit-basis`	Tells the linker to generate a warning message for each section when it is placed using the first fit basis algorithm.
`-enable-warn-stack-effect`	Tells the linker to generate a warning if stack-effect estimation must consider recursively called functions; such functions impair the accuracy of the estimate. (Default option. Opposite option is `-disable-warn-stack-effect`.)
`-env <path>`	This command line option is used to define the path of the compiler. The given path must be the root of the compiler to be used. This option overrides the `SC100_HOME` environment variable.
`-exclude` *`section{, section}`*	Tells linker to *not* link the specified sections.
`-exec_padding` *string*	Specifies padding characters for program sections of type SHT_PROGBITS. Without this option, padding value is {x90, 0xC0} (big-endian mode) or {0xC0, 0x90} (little-endian mode). (See padding note after this table.)
`-exec_padding16bits` *value*	For program sections, changes the default padding to the specified 16-bit value. Example: `-exec_padding16bits 0x9e7c`. (See padding note after this table.)
`-exec_padding32bits` *value*	For program sections, changes the default padding to the specified 32-bit value. Example: `-exec_padding32bits 0x9f799e7c`. (See padding note after this table.)
-force-self-contained-library	Tells the linker to ignore any `.library_concatenate_sections` directive present in the LCF and creates a normal self-contained library.
`-fsub`	Try folding substring.
`-inhibit-o2-place name_section[,name_section...]`	Inhibits the size optimization for the specified sections. (Opposite option is `-02-place`.)

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-init-table`	Tells the linker to create a `.rom_init_tables` section, in response to any reference to the _rom_init_tables[]. (Default option. With the compiler `-mrom` command-line option, the startup file uses _rom_init_tables[] to copy initial ROM variable values to RAM, Any undefined reference to the __rom_init_tables symbol blocks dead stripping,)
`-init-table-load`	Tells linker to use load address (physical address) when it creates .rom_init_tables section. (Changes default behavior for msc8144 architectures.)
`-init-table-run`	Tells linker to use run address (virtual address) when it creates .rom_init_tables section. (Changes default behavior for all architectures but msc8144.)
`-j`	Displays symbol index for symtab section.
`-k` *`depth`*	Specifies maximum depth for the relocation stack.
`--keep_symbols`	Removes all symbols, and keeps only those symbols that match a specified criteria. For example, the following command removes all symbols except for the global symbols that match the specified pattern: `sc100-ld -s --keep_symbols "pattern1, pattern2, ..." ...` You use `--keep_symbols` option with `-s` option (strip all symbol information). The pattern can be a symbol name or may contain ? or * wildcards as well.
`-l`	Displays original name of static symbols.
`-l`*`archive`* `...`	Links against the specified `archive`, in standard search path `$SC100_HOME/lib`.
`-L`*`searchdir`* `...`	Searches the specified directory for the archive that the `-l`*`archive`* option specified. This `-L`*`searchdir`* option may appear multiple times in the command line, but all must precede `-l`*`archive`*.
`-m`	Displays unmangled names for C++ symbols that have mangled names.
`-M`	Outputs a memory map to the standard output.
`-Map` *`mapfile`*	Outputs a memory map to the specified file.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-mrom_compress`	Tells linker to compress code stored in ROM (that is, code generated via the `scc -mrom` option).
`-n`	Inhibits dead-code and dead-data stripping.
`-nc`	Inhibits dead-code stripping.
`-nd`	Inhibits dead-data stripping.
`-nf`	Disable any folding optimization.
`-nfc`	Disable folding code, still folding data.
`-nfd`	Disable folding data, still folding code.
`-non-ovl`	Restricts overlay support to application sections that `.overlay` and `.union` directives mention.
`-no_exec_padding` *value*	Specifies padding value for data sections of type SHT_PROGBITS. Without this option, padding value is zero. (See padding note after this table.)
`-no_exec_padding16bits` *value*	For data sections, changes the default padding to the specified 16-bit value. Example: `-no_exec_padding16bits 0x9f79`. (See padding note after this table.)
`-no_exec_padding32bits` *value*	For data sections, changes the default padding to the specified 32-bit value. Example: `-no_exec_padding32bits 0x9f799e7c`. (See padding note after this table.)
`-no-reserve-ovl-run`	Does not reserve memory space for overlay-section run addresses.
`-N`	Displays never-called functions and never-used data.
`-Nc`	Displays never-called functions.
`-Nd`	Displays never-used data.
`-o` *`outfile`*	Assigns the specified file name to the executable object file. The default file name is `a.eld`.
`-old-init-table`	Tells the linker to *not* create a `.rom_init_tables` section, to use instead old-style ROM initialization. (For backward compatibility only, with older runtime libraries or startup files.)
`-o0-place`	Uses an unoptimized algorithm to place sections in memory space. This default option helps preserve backward compatibility.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-o1-place`	Uses an optimized algorithm to place sections in memory space.
`-o2-place`	Enables the size optimization for intra-section space. (Opposite option is `-o0-place`.)
`-remove-time-stamp-from-prefix`	This option removes the time stamp information from the symbol prefix in the self-contained library, but you must use this option only with the `-self-contained-library` option.
`-remove-veneer`	Removes unnecessary veneer functions that StarCore Compiler may generate. This option will not take effect when used with `-set-cache1` or `-o2-place`. By default, `-remove-veneer` option is disabled.
`-reread-lib`	Forces the linker to reread all libraries until it cannot resolve any more references.
`-rewrite-padding-between-objects`	Forces the linker to rewrite padding between objects. (Default option.)
`-s`	Strips all symbol information from the executable object file.
`-sa`	Specifies aggressive stripping for both dead-code and dead-data.
`-sac`	Specifies aggressive dead-code stripping.
`-sad`	Specifies aggressive dead-data stripping.
`-saf`	Ignores address taken information, assumes no address taken.
`-section-alignment` *factor*	Specifies factor for section alignment: all sections in memory begin at addresses that are multiples of this number. Default factor value is `0x1`.
`-self-contained-library`	Creates a self-contained library: one that does not have any external references.
`-set-cache1`	Enables cache optimization in the linker.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-set-mmu-info<level>`	This option sets the level of information that can be removed when MMU support is used. <level>: 0 - Forces the linker to emit signature for the sections that are involved in MMU support. This is default option. 1 - Forces the linker not to emit signature for the sections that are involved in MMU support. 2 - Forces the linker to eliminate padding required by MMU 3 - Forces the linker to keep the SHT_NOBITS type alive for BSS section by using the SHT_MIX_OVERLAY type if the BSS sections are used only in MMU support.

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description

`-set-mmu-info4`	This option forces the linker to re-use the physical space required to align the size of section to a power of two for descriptors generated by `.att_mmu` directive. The `reserved_mmu_padding` is a new parameter in the `.att_mmu` directive. If this parameter is specified for a section, The space required to align the size of a section to power of two is reserved in the physical space. We can call MMU padding the space required to align the size of section to a power of two. If this parameter is not specified, the MMU padding can be used by another section (the MMU padding is not reserved in the physical memory space). If the parameter is specified, it cannot be overwritten by the -`set-mmu-info4` option from linker.
	
	The syntax is:
	
	`.att_mmu "name", \`
	`{task_id: absolute_value,}\`
	`[{task_id: absolute_value,} ...] \`
	`start_address, end_address\`
	`[{, start_address, end_address} ...]\`
	`, _RESERVED_, \`
	`size: absolute_value, \`
	`region_type: "data"|"program", \`
	`attribute: absolute_value, \`
	`base_address: absolute_value, \`
	`physical_address: absolute_value \`
	`| , "section_name" \`
	`{, reserved_mmu_padding } \`
	`{, attribute: absolute_value} \`
	`{, single_mapped: absolute_value}\`
	`{, base_address: absolute_value} \`
	`{, physical_address |`
	`after_physical_address:absolute_value}\`
	`[, _RESERVED_, \`
	`size: absolute_value, \`
	`region_type: "data"|"program", \`
	`attribute: absolute_value, \`
	`base_address: absolute_value, \`
	`physical_address: absolute_value \`
	`| , "section_name",\`
	`{, reserved_mmu_padding }, \`
	`{, attribute: absolute_value } \`
	`{, single_mapped: absolute_value}\`
	`{, base_address: absolute_value} \`
	`{, physical_address |`
	`after_physical_address:absolute_value},...]`

Table 2.1 StarCore Linker Command-Line Options (*continued*)

Option	Description
`-start-reread-lib<libraries> -end-reread-lib`	These two options tell the linker to solve interdependencies among the specified libraries by rereading them until it cannot resolve any more references. Only library names may be between these two options.
`-stop-link-after-first-error`	Specifies that linking stop as soon as the linker finds the first error; reports warnings and error messages at end of the link phase.
`-S`	Strips debug information from the executable object file.
`-U`*`symbol`*	Creates an unresolved reference to the specified `symbol`.
`-unmangle`	In error messages and map files, displays unmangled names for C++ symbols that have the format `mangle_name{unmangle_name}`.
`-v`	Verbose mode: reports progress through each linking stage.
`-V`	Displays the current version of the linker and exits.
`-w`	Suppresses warnings.
`-W<level>`	Sets the warning level: Specify `level` value 0 (the default) to have the linker report most warnings and remarks. Specify `level` value 1 to have the linker report all warnings and remarks.
`-x`	Removes all local symbols from the symbol table.
`-xd`	Tells linker to remove compiler/assembler symbols for debugging sections. Does not affect current debugging capabilities, but can interfere with backward capability. (Opposite option is `-disable-xd`, which is the default.)
`@<file>`	Tells the linker to accept options and arguments in the specified file, just as if they were options of the command line itself.

You may define regularly used linker options in the `.sc100` argument file. The linker processes these options definitions immediately after program defaults. The linker ignores all options not designated for `sc100-ld` or `all`.

NOTE The linker pads program (executable) sections or data (non-executable) sections of type SHT_PROGBITS according to the command-line options you specify. Options –`exec_padding`, `-exec_padding16bits`, and `-exec_padding32` bits pertain to program sections. (For program sections, flag SHF_EXECINSTR is set.) Options `-no_exec_padding`,

`-no_exec_padding16bits`, and `-no_exec_padding32` bits pertain to data sections. (For data sections, flag SHF_EXECINSTR is clear.)

2.2.1 Specifying a Linker Command File

If you do not specify a command file, the linker uses the default file `crtscsmm.cmd`, located in `$SC100_HOME/etc`. To override this default behavior, use any of these options:

`-c commandfile`

Specifies the linker command file that the linker will use; the argument may include an optional pathname.

`-C`

Suppresses the search for a linker command file; this lets you link without a command file. This option may be appropriate for hand-coded programs in which you have allocated all memory and explicitly initialized the status register.

This example command tells the linker to use command file `crtsctmm.cmd`, in directory `$SC100_HOME/etc`;

```
sc100-ld -c $SC100_HOME/etc/crtsctmm.cmd foo.eln
```

2.2.1.1 Specifying Arguments in a File

An argument file is an ASCII text file of options and arguments, just as they would appear on the command line. An argument file is convenient for arguments that you use frequently.

An argument file can contain multiple lines. The end-of-line character counts as a space. Use the \ character to continue a line. Start comments with the # character.

Use the @ character in the command line to specify an argument file, as in this example:

```
sc100-ld @arg_file
```

When the linker executes this command, it opens file `arg_file`, processing all the arguments the file contains.

2.2.1.2 Enabling Shared Symbol References

Ordinarily it is not appropriate to have references from symbols between shared-to-private or shared-to-shared spaces. The linker even verifies references from symbols, in this regard.

Rarely, however, such references are appropriate. If this is the case with your application, you may suppress automatic linker verification of shared symbol references. To do so, include this option in the command line:

```
-enable-shared2private
```

2.2.1.3 BSS Section Zeroing

The linker automatically creates a `.bsstab` section, which contains these symbols:

- `.__bss_table` — an array of type Elf32_bsstab that contains an entry for each `.bss` section.
- `.__bss_count` — the number of entries in `.__bss_table`; an unsigned, 32-bit integer.

The startup file uses these symbols to fill the .bss sections with zeros.

The structured type Elf32_bsstab is:

```
typedef struct {
Elf32_Addr  start_address;   /*start address of .bss section*/
ELF32_Word  length;          /*size of .bss section in bytes*/
}Elf32_bsstab;
```

File `bsstab.h` defines the data structures and variables needed for the `.bsstab` section.

NOTE Do not let the linker place the `argv` and `argxc` symbols in a `.bss` section. If necessary to prevent this placement, use the `-disable-emit-bsstab` option to stop creation of the `.bsstab` section.

2.2.1.4 Reread Libraries

To force the linker to reread all libraries until it cannot resolve any more references, use this option on the command line:

```
-reread-lib
```

2.2.1.5 Omit Signatures

A default linker action is appending a signature to each overlay section. To disable this behavior, use this option on the command line:

```
-disable-emit-signature2overlay
```

2.2.1.6 Self-Contained Libraries

A library that, in theory, should not have any external references is *self-contained*. The advantages of such libraries are:

1. Appropriately removing dead code and dead data.
2. Hiding inside libraries all resolved symbols that are not entry points or public symbols.

To build a self-contained library, use the flag `-self-contained-library` in the command line, as in this example

```
sc100-ld -self-contained-library -c linker-command-file.lcf
module1.eln library1.elb ... -o self_contained_library.elb
```

Five new directives help you create and manage a self-contained library:

`.library_entry_points “entry_point_symbols”, ...`

> Defines the entry point symbols; mandatory for creating a self-contained library. (The compiler prefixes all C function names with an underscore.) The linker does not strip these entry point symbols (functions), even if they are not referenced.

`.library_public_symbols “public_symbols”, ...`

> Defines public symbols; required for creating any self-contained library that contains public symbols. (The compiler prefixes all C function names with an underscore.) The linker does not strip these public symbols (variables), even if they are not referenced.

`.library_undefined_symbols “symbols”, ...`

> Provides for unresolved global symbols. If the linker cannot resolve global symbols, it checks this directive. Only if the symbol is not in this directive does the linker issue an error message.

`.library_prefix “prefix_name”`

> Adds the specified prefix to all symbols that are not entry points, are not public symbols, or that are not resolved. Each self-contained library should have a unique prefix.

`.library_concatenate_sections "name", "section_name_pattern" [, "section_name_pattern" [, ...]]`

> Combines the sections that parameter `section_name_pattern` specifies and merges them into a new section. The parameter `name` specifies the name of the new section.

The related option `-enable-undef` permits undefined symbols in a self-contained library, but you must use this option only in a -self-contained library command.

Additional points:

- In verbose mode, the linker displays information about undefined symbols.
- The linker can accept multiple LCF files in command-line options, but the order of the LCF files is important.
- You can use a self-contained library as you would any other library.

The compiler includes two functionalities for self-contained libraries:

1. Using `-selflib` inside scc, provided that you specify all directives needed to create a self-contained library. For this option, the scc omits the startup file and adds the runtime library at the end

of the list, with object files you specify. (The linker links the startup file with the application at the last stage, when it builds the executable file.)

The only undefined symbols (external references) are those that the `.library_undefined_symbols` directive mentions. Usually, you must mention global symbols from the startup file in this directive. Examples are `.library_undefined_symbols`, `"___break"`, `"___syscall"`, `"___crt0_end"`, `"___stack_safety"`, and `"___mem_limit"`.

This functionality is useful for creating independence from the runtime library. It is possible to combine the `-selflib` option with -Xlnk `-enable-undef`, in which case you need not specify all undefined symbols in a `.library_undefined_symbols` directive.

2. Using `-complib` as an `scc` command line option, provided that you use it only for global optimizations. The compiler uses an automatically generated linker control file to pass the information to the linker.

NOTE If you create a library that uses global optimizations, you must also include in the application file the list of entry points and variables that must be visible to the outside world.

2.2.2 Adding Directories to the Linker Search Path

Use the -L option to add directories to the linker's standard search paths:

`-L`*`searchdir`*

Searches the specified directory for the archive that the `-l`*`archive`* option specifies. This `-L`*`searchdir`* option may appear multiple times in the command line, but all must precede `-l`*`archive`*.

The linker first searches the standard search path, `$SC100_HOME/lib`. If it does not find the archive, it subsequently searches the directories the `-L` option specifies, in their command-line order.

This example command links archive `libalpha.elb` with the file `corr.eln`. The linker searches the `$SC100_HOME/lib` directory, then `/starcore/sclib`, then `/scprogs/mylib`.

```
sc100-ld -L/starcore/sclib -L/scprogs/mylib -llibalpha corr.eln
```

2.2.3 Defining Symbols

To define symbols from the linker command line, use these options:

`-Dsymbol=value`

Defines a global symbol, which overrides any `.provide` symbol of the same name that may exist. You may specify the `value` argument in octal (leading `0`), decimal, or hexadecimal (leading `0x` or `0X`).

`-Usymbol`

Creates an unresolved reference to the specified `symbol`. This option forces linking of a module that contains a definition for `symbol`, if that module is in a library, so would not otherwise be linked.

2.2.4 Estimating Stack Effect

The more possible entry points there are in an application, the larger the stack that the application needs. The default entry points are the function `main()`, plus all function pointers (functions that data sections reference). You can add additional entry points by using the `-U` option on the command line, or by using the `.xref` or `.library_entry_points` directives in the linker command file.

Stack effect is a term for the necessary stack size. To have the linker estimate the stack effect, and print the estimate in the map file, use this option:

```
-enable-stack-effect
```

To stop such estimation (restoring default behavior), use the opposite option:

```
-disable-stack-effect
```

If your application allows recursive calls to functions, the linker cannot know in advance how many such calls there will be during execution. Consequently, the estimated stack effect for such an application cannot be as accurate as the estimate for a non-recursive applications. The linker's default behavior is warning you any time that recursive calls could impair the accuracy of a stack-effect estimate. To deliberately specify this default behavior, use this option:

```
-enable-warn-stack-effect
```

To stop such warnings, use the opposite option:

```
-disable-warn stack-effect
```

2.2.5 Renaming the Object File

The default executable object file is `a.eld`. To assign a different name to this file, use the `-o` option:

`-o outfile`

Assigns the specified file name, exactly as entered, to the executable object file. Overwrites any file with the same name.

This example command links object file `corr.eln`, creating executable object file `corr.eld`:

```
sc100-ld -o corr.eld corr.eln
```

2.2.6 Modifying the Contents of the Object File

To modify the contents of the object file the linker creates, use any of these options:

`-S`

Strips debug information from the executable object file.

`-s`

Strips all symbol information from the executable object file. (This keeps sections `.symtab`, `.strtab`, and `.mw_info` from erasing debugger information from the executable object file.)

`-x`

Removes all local symbols from the symbol table.

2.2.7 Dead-Code, Dead-Data Stripping

The linker automatically removes never-used data and the code of never-called functions. To restrict this default behavior, use any of these options:

`-n`

Inhibits dead-code and dead-data stripping.

`-N`

Displays never-used data and never-called functions.

`-sa`

Specifies aggressive stripping for both dead code and dead data.

2.2.7.1 Dead-Code Stripping

The linker strips dead code only if there is a defined end-of-function symbol `F<function_name>` for each function symbol `function_name`. A more consistent way to specify the size of a function is to use the assembler SIZE directive.

While generating the executable files, Linker performs dead-code stripping only if the entry function `main()` (or `_main` in the ASM source) is defined. For self-contained libraries, Linker performs dead-code stripping using the entry function information from `.library_entry_points` directive in the LCF.

There are limitations to dead-code stripping, such as its basis: relocation information. In the example below, the linker performs an incorrect operation, removing the code of function `_Func1`. This is because an absolute value references the function, so the assembler does not generate relocation information.

```
       ....
       jsr $1000
       ....
       org$1000
_func1 TYPE FUNC
```

```
       <function code>
F_func1_end
```

To prevent incorrect code removal, application code must reference all functions by symbol, not by absolute (constant) values.

If two functions overlap and only one is called, the linker does not strip either function. It does not even strip the un-overlapped part of the uncalled function. And if the linker cannot determine a function's size (that is, the linker cannot find the ending symbol), he linker does not perform dead-code stripping at all.

The command-line options for dead-code stripping are:

`-nc`

Inhibits dead-code stripping.

`-Nc`

Displays never-called functions.

`-sac`

Specifies aggressive dead-code stripping: rounds the function size up to a multiple of the section alignment. If there is no new symbol definition in this newly allocated space, the linker strips function code, including this padding space. But if there is a new symbol definition in this newly allocated space, the linker performs its default dead-code stripping.

2.2.7.2 Dead-Data Stripping

Correct linker operation does not depend on default variables; you may specify global symbols that the linker will not strip, even if they are not referenced. To do so, use the directive `.library_public_symbols`, from the linker control file.

And to prevent the stripping of useful data, you can use two new, vendor-specific symbol types, *VARIABLE* and *INITIALIZER*. These are sub-types of the 'OBJECT" ELF type, in the ELF file.

VARIABLE symbols are for regular variables. A symbol of this type defines a regular C variable. It always should have an associated size (the memory that the variable occupies).; the default size is 0. The ELF type value for VARIABLE is STT_OBJECT; the sub-type (as written in the `.mw_info` section) is VARIABLE.

VARIABLE rules are:

1. VARIABLE symbols do not have a mandatory location — the linker is free to move them anywhere in the same section, so long as it maintains the same alignment. Objects that must have a fixed location should not have the type VARIABLE.
2. If there is no explicit reference to the symbol that defines a VARIABLE, then the VARIABLE is not used, so should be stripped.
3. Overlapping VARIABLES mean an exception to rule 2. A reference to any VARIABLE counts as a reference to all overlapping VARIABLES, preventing stripping.

4. An object of any type, even NO_TYPE, can reference a VARIABLE symbol. Any external reference to a symbol or group of symbols prevents stripping.
5. The alignment of a symbol always is considered to be a power of 2. Never assume that a VARIABLE symbol could be aligned on 3, 19, 35, or any such value.
6. The only object that may overlap a VARIABLE symbol is another VARIABLE symbol.
7. If an object of non-VARIABLE type starts within an object of type VARIABLE:
    a. The linker is free to strip the VARIABLE object, even if there is a reference to the other symbol; the two symbols would have the same address.
    b. The linker is free to move the VARIABLE object. This action moves the other symbol as well, preserving the relative distance from the start of the VARIABLE symbol.

INITIALIZER symbols are for `.init_table`-like sections. Some applications require that data be kept in a non-initialized (`.bss`) section, initialized from a ROM section. In order for the startup code to do the initialization, you must define another (`init_table`) section that specifies how to copy the initial variable values from *rom* to *bss*. As startup code always references all these variables, stripping them normally is not possible — even if the application does not actually reference them.

But INITIALIZER symbols make such stripping possible. The ELF type value for INITIALIZER is STT_OBJECT; the sub-type (as written in the `.mw_info` section) is INITIALIZER. An INITIALIZER has the same properties and requirements as a VARIABLE symbol, except that:

1. An INITIALIZER may not overlap any other symbol
2. If an INITIALIZER contains any reference to a VARIABLE, the VARIABLE implicitly contains a reference to the INITIALIZER.

INITIALIZERS and VARIABLES form groups of symbols that make circular references to each other:

- If the application does not reference the VARIABLE, the linker can strip the entire group (VARIABLE from `bss`, VARIABLE from `rom`, and INITIALIZER from `init_table`).
- Any application reference to anything in the group (typically the VARIABLE from `bss`) prevents stripping any member of the group.

The command-line options for dead-data stripping are:

`-nd`

Inhibits dead-data stripping.

`-Nd`

Displays never-used data.

`-sad`

Specifies aggressive dead-data stripping, beyond the scope of linker default dead-data stripping.

2.2.8 Code and Data Folding

Code and data folding is a symbol-level size optimization in the linker. If two read-only symbols have same contents or one of them is a sub-string of another, these two symbols could be allocated in memory overlapped.

If the application has the following example statements, linker would allocate array b in array a, from index 6 of array a:

```
static const char a[] ="linkertesting";
static const char b[] ="testing";
```

If application is sensible with address of symbols, like comparing addresses of two symbols, this optimization is not safe. Compiler will analyze code sources and try to find if some address is taken and let linker know that it is not safe to do the optimization.

Compiler does not catch all the scenarios, like library compiled by old compiler or some assembly code. In this case, user should provide information if some address is taken in this kind of code. If no address exists, user could use `-saf` option in the linker to do aggressive folding optimization and get the most size reduction.

By default, linker would turn on this optimization, check information of the address taken and not fold sub-string symbols.

The command-line options for code and data folding are:

`-nf`

Disables any folding optimization.

`-nfc`

Disable folding code, still folding data.

`-nfd`

Disable folding data, still folding code.

`-fsub`

Try folding substring.

`-saf`

Ignores address taken information, assumes no address taken.

2.2.9 Cache Optimization

The cache optimization in the linker lets you reduce the paging traffic by placing the routines close to their callers in the virtual memory. In addition, the optimization also reduces the collisions that related and frequently used routines may have in the I/D cache.

The linker propagates the information about frequency of object call by using call graph of the respective functions and data. Also, it places the objects in the section in decreasing order of their call frequency (which is propagated in the call graph).

The cache optimization also helps to avoid the conflict that may appear when number of called objects, having same line index in cache, is greater than available way slots.

NOTE You can obtain line index of the cache from virtual address of the object.

The linker places the objects in distinct line index from the cache. The PLRU algorithm is used to select an appropriate way slot.

NOTE Use command line option `-set-cache1` to enable the cache optimization. In addition, use `.cache_setting` directive to specify cache settings, such as instruction cache structure. You may also use `.frequency` directive to specify function call frequency and cycle count. See Table 3.4 for more details on these directives.

2.2.10 Generating a Linker Map File

To have the linker generate a linker map file, use one of these options:

`-M`

Outputs a memory map to the standard output. This map lists section allocations in virtual memory, and contains tables that show the absolute locations of all local and global symbols.

`-Map mapfile`

Outputs a memory map to a new file; the name of this new file is the exact `mapfile` value.

This example command links file `corr.eln`, generating executable file `corr.eld` and map file `corr.map`.

```
sc100-ld -o corr.eln -Map corr.map corr.eln
```

The map file lists all sections the linker encounters in the linker input stream. For each section, the map file includes:

- Total size of the section
- Address of the section after relocation
- Details of all section fragments: fragment size, address after relocation, input file or library module, and any symbols the fragment references.

To identify sections, section fragments, and symbols in a map file, note their positions in the Symbol field:

- A section always begins in column 1. The label `Section:` precedes the section name.

- A section fragment is on the first indent of the Symbol field. The label `Section:` precedes the fragment name. After the fragment name comes the name, in parentheses, of the input file or library module in which the linker found the fragment.

  Each fragment has the same name as the section to which it belongs.
- A symbol is on the second indent of the Symbol field; it applies to the closest previous fragment listed.

Figure 2.1 shows a partial map file that lists five sections: .intvec, .data, .bss, .default, and .text. Notice that the .text section has an address of $10000 after relocation, and has the size of 360 bytes. This size represents the total of all .text-section fragments assimilated from separate input files.

Table 2.2 explains the four fragments that make up the .text section.

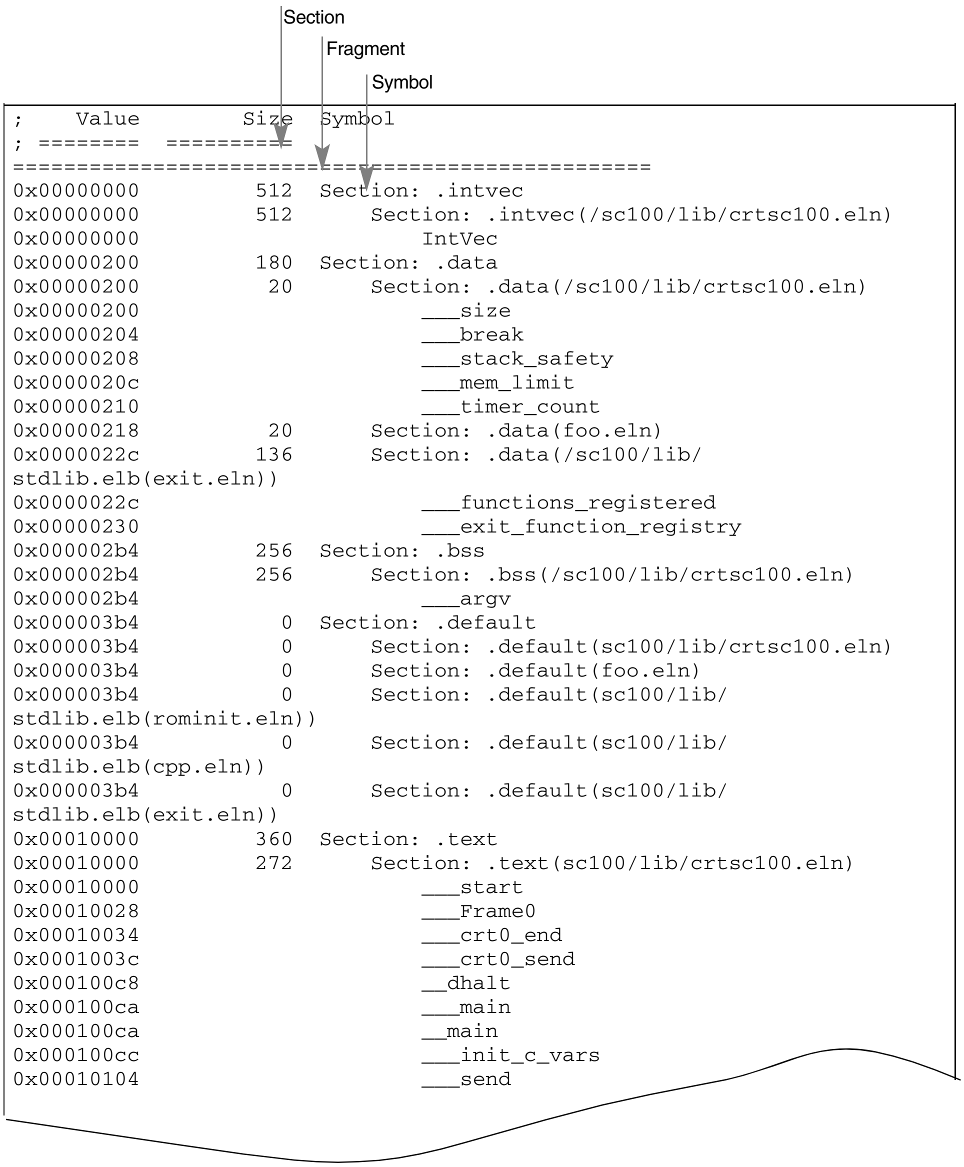
Table 2.2 Composition of .text Section

Fragment	Size	Originating Input File	Symbols Referenced Within Fragment

1	272	`/sc100/lib/crtsc100.eln`	`___start`
			`___Frame0`
			`___crt0_end`
			...

2	4	`foo.eln`	`_main`
3	4	`cpp.eln` module of the library, `/sc100/lib/stdlib.elb`	`__do_cppini` `__do_cppfin`
4	80	`exit.eln` module of the library, `/sc100/lib/stdlib.elb`	`_exit` `_exit_end`

Figure 2.1 Example Linker Map File


Section
Fragment
Symbol

```
;    Value         Size   Symbol
; ========   ==========
==================================================
0x00000000          512   Section: .intvec
0x00000000          512       Section: .intvec(/sc100/lib/crtsc100.eln)
0x00000000                        IntVec
0x00000200          180   Section: .data
0x00000200           20       Section: .data(/sc100/lib/crtsc100.eln)
0x00000200                        ___size
0x00000204                        ___break
0x00000208                        ___stack_safety
0x0000020c                        ___mem_limit
0x00000210                        ___timer_count
0x00000218           20       Section: .data(foo.eln)
0x0000022c          136       Section: .data(/sc100/lib/
stdlib.elb(exit.eln))
0x0000022c                        ___functions_registered
0x00000230                        ___exit_function_registry
0x000002b4          256   Section: .bss
0x000002b4          256       Section: .bss(/sc100/lib/crtsc100.eln)
0x000002b4                        ___argv
0x000003b4            0   Section: .default
0x000003b4            0       Section: .default(sc100/lib/crtsc100.eln)
0x000003b4            0       Section: .default(foo.eln)
0x000003b4            0       Section: .default(sc100/lib/
stdlib.elb(rominit.eln))
0x000003b4            0       Section: .default(sc100/lib/
stdlib.elb(cpp.eln))
0x000003b4            0       Section: .default(sc100/lib/
stdlib.elb(exit.eln))
0x00010000          360   Section: .text
0x00010000          272       Section: .text(sc100/lib/crtsc100.eln)
0x00010000                        ___start
0x00010028                        ___Frame0
0x00010034                        ___crt0_end
0x0001003c                        ___crt0_send
0x000100c8                        __dhalt
0x000100ca                        ___main
0x000100ca                        __main
0x000100cc                        ___init_c_vars
0x00010104                        ___send
```

2.2.11 Controlling Linker Messages

To control the information content of linker messages, use any of these options:

`-V`

Displays the current version of the linker and exits.

`-v`

Specifies verbose mode: the linker reports progress through each of these stages:

- `|Loading FILE|`

  The linker is reading the named file, merging its section contents with existing sections of the same name, and adding its global symbol values to the internal symbol table.
- `|Placing SECTION|`

  The linker is assigning an address to the named section.
- `|Adjusting symbol values|`

  The linker is adjusting the values of all symbols to reflect section addresses.
- `|Relocating SECTION(FILE)|`

  The linker is resolving external references in the named section.
- `|Writing executable file.|`

  The linker is writing the executable file.

`-w`

Suppresses warnings.

This example command links archive `lib1401.elb`, file `crtnosat1401.eln`, and file `main.eln` into an executable object file. This command also tells the linker to reports progress. Listing 2.1. is sample verbose-mode output.

```
sc100-ld -v -llib1401 crtnosat1401.eln main.eln
```

Listing 2.1 Sample Output of Linker Verbose Mode

```
Loading /opt/pulsar/opt/SC100/1.99/lib/crtnosat1401.eln
Loading main.eln
Loading /opt/pulsar/opt/SC100/1.99/lib/lib1401.elb(rominit.eln)
Loading /opt/pulsar/opt/SC100/1.99/lib/lib1401.elb(cpp.eln)
Loading /opt/pulsar/opt/SC100/1.99/lib/lib1401.elb(exit.eln)
Placing .intvec
Placing .data
Placing .bss
Placing .default
Placing .text
Placing .init_table
```

```
Adjusting symbol values
Relocating .text(/opt/pulsar/opt/SC100/1.99/lib/crtnosat140l.eln)
Relocating .data(/opt/pulsar/opt/SC100/1.99/lib/crtnosat140l.eln)
Relocating .intvec(/opt/pulsar/opt/SC100/1.99/lib/crtnosat140l.eln)
Relocating .init_table(/opt/pulsar/opt/SC100/1.99/lib/
crtnosat140l.eln)
Relocating .SC100.delay_slots(/opt/pulsar/opt/SC100/1.99/lib/
crtnosat140l.eln)
Relocating .text(/opt/pulsar/opt/SC100/1.99/lib/lib140l.elb(exit.eln))
Writing executable file
```

2.2.12 Defining Unlikely Block of Code as Private Block of Code in a Multi-core Application

When you specify the `unlikely` keyword for a block of code in your application, the compiler moves that block of code to the `.unlikely` section in the LCF.

In a multi-core application, however, specifying the `unlikely` keyword for private block of code on multiple cores leads to linking errors because, by default, `unlikely` blocks of code are not considered as private blocks of code.

In order to avoid such linking errors, follow these steps to explicitly define `unlikely` blocks of code as private blocks of code in a multi-core application:

1. Rename the relevant `.unlikely` section to make it specific to each of the cores (if the multi-core application model is true private), or specific to each of the subset of cores that share the code (e.g. if the function code is common only to core 0 and core 1).

   For example, for `C0` private functions defined in the module `C0M2function.c`, use the following linker directive:

   ```
   .rename "*C0M2function.eln", ".unlikely", ".unlikely_C0M2function"
   ```

2. Place the renamed section in the private memory areas specific to the hosting core.
3. If a subset of cores share the code, export the renamed section to the other cores in the sharing subset, and have each core in the sharing subset import the renamed section.

   For example:

   ```
   .export ".unlikely_C0M2function" ; on the hosting core
   .import ".unlikely_C0M2function" ;on each core in the sharing subset
   ```

   In addition, exclude the renamed section from the cores in the non-sharing subset.

   For example:

   ```
   .exclude ".unlikely_C0M2function"
   ```

3

Linker Command File

The linker lets you specify your output code by using directives in a linker command file (LCF). This chapter explains the linker command file and its directives.

- 3.1 Pre-Built Linker Command Files
- 3.2 Linker Command File Syntax
- 3.3 Sections
- 3.4 Linker Directives

3.1 Pre-Built Linker Command Files

The factory provides two linker command files, one for each memory model that the StarCore architecture supports. These files are:

- `crtscsmm.cmd`, for small memory mode (the default)
- `crtscbmm.cmd`, for big memory mode

These files are located in `{CodeWarrior}\StarCore_Support\Compiler\etc`.

3.2 Linker Command File Syntax

This section explains the syntax of the linker command file.

- 3.2.1 Command File Structure
- 3.2.2 Expressions and Symbols
- 3.2.3 Operators
- 3.2.4 Comments

3.2.1 Command File Structure

A linker command file is a text file; its filename ends with extension `.lcf` or `.cmd`. This file contains the linker directives that define your program's memory layout, segment allocations, and entry point.

To create a linker command file for a simple project:

1. Identify the important memory addresses of the processor. Use the `.provide` directive to define symbols for them.
2. Define the memory ranges — use the `.memory` and `.reserve` directives.
3. Combine similar sections — use the `.segment` directive. Map sections into memory — use the `.org` directive.
4. Define the entry point.

Listing 3.1 shows a sample linker command file.

Listing 3.1 Example Linker Command File

```
.provide _ROMStart, 0x7f000
.provide _SR_Setting, 0xe4000c
.provide _sdram0, 0x20000000
.provide _StackStart, 0x20c00000
.provide _TopOfStack, 0x20fffff0
.provide __BottomOfHeap, _StackStrt
.provide __TopOfHeap, _TopOfStack
.provide _sdram1, 0x20ffffff
.memory 0x0 , 0xfffff , "rwx"
.memory 0x20000000 , 0x20ffffff , "rwx"
.reserve _StackStart ,_TopOfStack
.org 0x0
.segment .intvec , ".intvec"
.org _sdram0
.segment .rotable , ".init_table"
.segment .roinit , ".rom_init"
.segment .data , ".data", ".ramsp_0", ".bss"
.segment .text , ".text", ".default"
.org _sdram0 + 0x400000
.segment .text , ".text"
.entry 0x0
```

Note these lines of the example LCF:

```
.provide _ROMStart, 0x7f000

.provide _SR_Setting, 0xe4000c

.provide _sdram0, 0x20000000

.provide _StackStart, 0x20c00000

.provide _TopOfStack, 0x20fffff0

.provide __BottomOfHeap, _StackStrt

.provide __TopOfHeap, _TopOfStack

.provide _sdram1, 0x20ffffff
```

The `.provide` directives define symbols for memory locations important to the program. This lets you refer to these locations by symbol.

```
.memory 0x0 , 0xfffff , "rwx"
.memory 0x20000000 , 0x20ffffff , "rwx"
.reserve _StackStart ,_TopOfStack
```

The `.memory` directive defines the memory regions for program and data storage. The `.reserve` directive blocks off a portion of a memory region, preventing the linker from putting anything there. This `.reserve` directive reserves space for the stack and heap.

```
.org 0x0
 .segment .intvec , ".intvec"
```

The `.org` directive specifies the address where the linker places the subsequent `.segment` or segments.

```
.org _sdram0
 .segment .rotable , ".init_table"
 .segment .roinit , ".rom_init"
 .segment .data , ".data", ".ramsp_0", ".bss"
.segment .text , ".text", ".default"
.org _sdram0 + 0x400000
 .segment .text , ".text"
```

The `.segment` directive combines sections. The linker locates these segments at the address the preceding `.org` directive specified.

```
.entry 0x0
```

The `.entry` directive specifies the entry point of the program.

3.2.1.1 Dynamic Stack/Heap Configuration

In the Listing 3.1 example, one `.provide` directive defined symbols `_StackStart` and `__BottomOfHeap` with the same value. Another `.provide` directive defined symbols `_TopOfStack` and `__TopOfHeap` value. This is *dynamic* configuration — the default — in which the stack and the heap use the same memory space.

Earlier versions of the linker offered only this dynamic configuration. The stack and heap had to use the same memory space, because there were no bottom and start symbols for the heap.

3.2.1.2 Static Stack/Heap Configuration

The new symbols `__BottomOfHeap` and `__TopOfHeap` make possible a *static* configuration: one in which the heap and the stack use separate areas of memory. Simply give these symbols different address values from their stack counterparts (`_StackStart` and `_TopOfStack`).

NOTE Old applications and LCFs lack definitions for the new symbols `__BottomOfHeap` and `__TopOfHeap`, so you cannot link them with a new runtime library. To make such linking possible, you must add definitions of these new symbols, setting `__BottomOfHeap` to the `_StackStart` value, and setting `__TopOfHeap` to the `_TopOfStack` value. These new definitions preserve the *dynamic* configuration of the old applications and LCFs.

3.2.2 Expressions and Symbols

An expression is a combination of symbols, constants, operators, and parentheses that represent a value to be used as a linker-directive operand. Expressions may contain user-defined labels and their associated integer values, or any combination of integers and linker functions.

Symbols that you define in the linker command file may begin with the underscore character, but this character is not mandatory. To refer to a symbol from C source code, use the underscore prefix.

To create global symbols and assign addresses to them, use the `.provide` and `.set` directives. For this example, the linker would evaluate the `.provide`, `.segment`, and `.set` directives in their order here:

```
.provide _textSegmentStart, 0x100000
.org _textSegmentStart
.segment text, ".text"
.set _textSegmentEnd, _textSegmentStart + @segsize(text)
```

Table 3.1 lists the functions you can use within expressions. You must enclose section names in quotes, as section names may include non-identifier characters.

Table 3.1 Linker Expression Functions

`@iif(condition, true_result, false_result)`	Condition evaluator: gives true result if condition is true, false result if condition is false
`@mmu_align(expression)`	Calculates and returns a number that meets all these conditions: • can be represented as power of two (2^x) • greater than or equal to the specified expression • greater than minimum descriptor size as required by MMU. For example, for 8144, the minimum descriptor size is 0x100.
`@pc()`	Position counter referring to the physical memory
`@secaddr("section-name")`	Starting address of section
`@secalign("section-name")`	Alignment of section
`@secend("section-name")`	First address after this section — same as `@secaddr("section-name") +@secsize("section-name)`
`@secsize("section-name")`	Size of section
`@segaddr(segment-name)`	Starting address of segment
`@segalign(segment-name)`	Alignment of segment
`@segsize(segment-name)`	Size of segment
`@vsecaddr("section-name")`	Starting virtual address of section
`@vsecend("section-name")`	First virtual address after this section

NOTE The linker allows multiple definitions of absolute symbols (assembly EQUs and SETs) if all symbols have the same value. The linker supports MULTIDEF symbol binding of the assembler. That is, the linker accepts the first definition of such a symbol, as if the symbol were global. Any other symbol redefinition is a fatal error.

3.2.3 Operators

You may use standard C arithmetic, unary, shift, relational, bitwise, and logical operations when you define or use symbols in the linker command file.

All operators are left-associative. The linker evaluates expressions according to this operator precedence order:

1. Parenthesis
2. Unary plus, unary minus, one’s complement, logical negation
3. Multiplication, division, modulo
4. Addition, subtraction
5. Shift
6. Relational operators: less, less or equal, greater, greater or equal
7. Relational operators: equal, not equal
8. Bitwise AND, OR, EOR
9. Logical AND, OR

Table 3.2 lists the arithmetic operators.

Table 3.2 Arithmetic Operators

Type	Operator	Description
Unary	highest (1)	`- ~ !`
	2	`* / %`
Relational	3	`+ -`
	4	`>> <<`
	5	`asdf`
	6	`&`
	7	|
	8	`&&`
	9	||

3.2.4 Comments

To add comments to your linker command file, use the semi-colon character (;). The LCF parser ignores comments.

```
; This is a comment
```

3.3 Sections

A section is a relocatable block of code or data that is encapsulated by the SECTION and ENDSEC assembler directives and has an associated section name and type. Although you can create any name for a section, some section names are reserved by the debugger and the SmartDSP Operating System. The application must not use these reserved names (refer to the assembler user's guide and the corresponding SmartDSP OS documentation).

In addition, the assembler recognizes conventional ELF sections such as `.text`, `.data`, `.rodata`, and `.BSS`.

A space can be shared by multiple cores, if the following conditions are true:

- the core, which defines the space shares it by using the .export directive
- all other cores import the shared space by using the .import directive

The core, which imports a shared space, cannot place any private code or private data in the shared region of physical memory. The linker automatically reserves this region of physical memory. In addition, the linker keeps the debug information consistent across all the cores.

The following rules apply when you access the symbols defined in a shared or private space:

- symbol defined in a private space can be accessed from:
  - other private spaces of the same unit
  - a shared space, only if the accessed symbol is defined at the same virtual address in all the cores. In this case, the descriptors of all the cores must have the same starting virtual address (the `base_address` field from `.att_mmu` directive).
- symbol defined in a shared space S can be accessed from:
  - any private space of the same unit
  - any private space of another unit, which imports space S
  - any shared space, whose import list is included in the import list of space S

The following linker directives are involved in providing multi-core support to the linker:

- .export
- .import
- .space
- .unit

Table 3.4 lists more information on these directives.

For each section that is generated by the linker, the linker expands the section size to the biggest size occupied by this section, in order to share the same size for all cores. It is very useful to have the same size for the object that is placed in private memory.

There are two types of sections:

1. Core specific section

The section that is prefixed by the name of core is visible only in this core (For example, `c0`.data`, `c0`.private_data`, `c0`.text`, `c0`.private_text` sections are visible only for the "c0" core).

2. Non-core specific section

The section that is not prefixed by the name of the core (For example, `.data`, `.data_private`, `.text`, `.private_text` sections) is visible for all cores. These sections can be placed in the private or shared space.

Following sections are generated by the CodeWarrior linker and compiler:

Table 3.3 Sections in LCF

Section Name	Usage
.att_mmu	Data section that is used in startup file/ runtime library and system operation to set the MMU registers. File `att_mmu.h` defines the data structures and variables needed for the `.att_mmu` section. Placed in a private data descriptor.
.bss	Un-initialized data section that is placed in a private data descriptor.
.bsstab	Read-only data section that is used in the startup file to fill the `.bss` sections with zeros. File `bsstab.h` defines the data structures and variables needed for the `.bsstab` section. Placed in a private data descriptor.

Table 3.3 Sections in LCF

Section Name	Usage
`.compress_table`	Contains these symbols to decompress the compressed section: • `__compress_table`: an array of type `.compress_table` that contains an entry for every compressed section. The table is sorted by `load_address` field • `__compress_count`: the number of entries in `__compress_table`; an unsigned, 32-bit integer
.data	Data section that is placed in a private data descriptor.
.default	Program section that is created by assembler for code that is not put between section `<name>` and endsec directives. In Single Instruction Multi Data (SIMD) application model, needs to be placed in a shared program descriptor. In Multi Instruction Multi Data (MIMD) application model, needs to be placed in a private program descriptor.
.exception	Read-data section that is used in the startup file/runtime library to catch the C++ Exception (Exception table). Placed in a private data descriptor.
.exception_index	Read-data section that is used in the startup file/runtime library to catch the C++ Exception (Exception table index). Placed in a private data descriptor.
.init_table	Read-only data section that is used to initialize the global variable ROM to RAM (`-mrom` option from scc). File `init_table.h` defines the data structures and variables needed for the `.init_table` section. Placed in a private data descriptor.

Table 3.3 Sections in LCF

Section Name	Usage
.intvec	Program section that is used to define the interrupt vector code. Recommended to be placed in a shared program descriptor, if placed in a private program descriptor, need to set the VBA register again, as the support from runtime library assumes that the virtual and physical address for VBA share the same value.
os_*	These sections are the system operation sections. These sections can be for code, data, read-only data or bss.
.ovltab	Data section that is used by overlay manager. File `overlay.h` defines the data structures and variables needed for the `.ovltab` section. Placed in a private data descriptor.
reserved_crt_tls	Data section that is used in reentrant runtime library. The context local data variable is defined in this section. Placed in a private data descriptor.
reserved_crt_mutex	Data section that is used in reentrant runtime library. The MUTEX variables are defined in this section. These variables are used by the critical region. This section needs to be mentioned in a non-cacheable descriptor from MMU among all cores. Placed in a shared data descriptor.
.rom	Un-initialized data section that is placed in a private data descriptor.
.rom_init_tables	Read-only data section that is used to initialize the global variable from ROM to RAM. File `init_table.h` defines the data structures and variables needed for the `.rom_init_tables` section. Placed in a private data descriptor.

Table 3.3 Sections in LCF

Section Name	Usage
.staticinit	Read-only data section that is used in the startup file/runtime library to initialize the C++ static objects. Placed in a private data descriptor.
.text	Program section that is placed in a shared program descriptor.
task1_*	These sections are the user sections. These sections can be for code, data, read-only data or bss.
.zdata	Data section that is fitted in the fist first 64k of memory.Placed in a private data descriptor.

3.4 Linker Directives

Table 3.4 lists linker directives — the commands that control linker operation. Subsequent text explains each directive in more detail.

Table 3.4 Linker Directives

.align	Sets the alignment requirement of a segment (alignment must be a power of 2)
.assert	Declares the specified expression as true. The linker reports an error, if the expression evaluates to zero (evaluates to false)
.att_mmu	Creates section .att_mmu if address translation table involves overlay sections
.att_mmu_settings	Defines MMU configuration parameters.
.bss	Creates a BSS segment in the executable file
.cache_setting	Specifies the cache optimization settings
.concatenate	Concatenates a list of overlay sections.
.define_compress	Enables compression for overlay sections.
.define_overlay	Enables overlay support for sections compiled without overlay support.

Table 3.4 Linker Directives (*continued*)

.define_region_to_map_virtual_addressing	Defines region-to-map virtual addressing.
.define_single_mapped_virtual_addressing	Defines single-mapped virtual addressing for non-overlay (non-translation) sections.
.entry	Defines the program's entry point
.exclude	Tells the linker to *not* link specified sections
.export	Specifies the spaces shared by the current core with other cores
.firstfit	Forces the linker to place segments on a first-fit basis
.frequency	Specifies the caching frequency of an object within a function.
.group	Specifies the run-time overlay section ordering
.import	Specifies the spaces defined in other cores that are shared by the current core
.include	Appends the content of specified LCF at the current location.
.inhibit_compress	Prevents compression for overlay sections.
.inhibit_folding_symbols	Enables to turn off folding optimization if it is found not appropriate for certain symbols.
.inhibit_folding_modules	Enables to turn off folding optimization if it is found not appropriate for certain modules.
.init_table_section	Tells the linker to treat sections that match the specified pattern as init_table sections.
.library_concatenate_sections	Combines the sections in a self-contained library into a new section.
.memory	Specifies a region in memory available for linking
.non_ovl	Specifies sections that will not have overlay support
.org	Specifies a starting address to begin linking segments
.overlay	Specifies which overlays will share a run address
.place_symbols	Places the specified symbols in a target section
.provide	Inserts a global symbol into the symbol table of the executable file

Table 3.4 Linker Directives (*continued*)

.puncture	Makes one section of all those that a `.segment` directive mentions
.rename	Renames ELF sections before processing
.reserve	Specifies a region in memory that is not available for linking
.segment	Specifies which sections to link at the current location counter
.set	Inserts a global symbol into the symbol table of the executable file
.space	Specifies a memory device in a multiple core device
.union	Specifies which pure data overlays share a run address
.unit	Specifies the core name in a multiple core device
.virtual_memory	Defines a memory region available for dynamic sements
.xref	Notifies the Linker that the specified symbol has a reference, even if the Linker does not see the reference during processing
.xref_module	Inhibits dead code or dead data stripping from specified module.

.align

Aligns the program counter to the specified byte boundary. The linker aligns a segment definition that follows an align statement accordingly.

```
.align n
```

Parameter

`n`

Byte boundary. Must be a power of 2.

Remarks

If the sections within a segment also have their own alignment, the linker may align the segment to a slightly higher address than that specified by the `.align` directive.

Example

```
.org _CodeStart
.align 16
```

```
     .segment .libtext, ".libtext"
```

.assert

Declares the specified expression as true. The linker reports an error, if the expression evaluates to zero (evaluates to false).

```
.assert expression
```

Parameter

`expression`

Expression to be evaluated.

Remarks

This directive is useful for restricting or checking the maximum size of a segment. For example, the `.assert` directive in the following example specifies that the INTVEC segment (the interrupt vector table) must be exactly 512 bytes. If it is not, the linker reports an error.

Example

```
     .org 0
     .segment INTVEC, ".intvec"
     .assert @segsize(INTVEC) == 512
```

.att_mmu

Creates an address translation table (ATT) section, of type SHT_MW_ATT_MMU (SHT_LOPROC + 4), from existing sections or overlays.

```
.att_mmu "name",\
     {task_id: absolute_value,}\
     [{task_id: absolute_value,}...]\
     start_address, end_address, \
     [{, start_address, end_address,}...]\
     ,_RESERVED_,\
          size: absolute_value,\
          region_type: "data"|"program",\
          attribute: absolute_value,\
          base_address: absolute_value,\
          physical_address: absolute_value\
     |"section_name",\
          {, reserved_mmu_padding}\
          {, attribute: absolute_value}\
          {, single_mapped: absolute_value}\
          {, base_address: absolute_value}\
          {, physical_address|after_physical_address: absolute_value}\
     [, _RESERVED_,\
          size: absolute_value,\
          region_type: "data"|"program",\
          attribute: absolute_value},\
          base_address: absolute_value,\
          physical_address: absolute_value\
     |"section_name",\
          {, reserved_mmu_padding}
          {, attribute: absolute_value}\
          {, single_mapped: absolute_value}\
          {, base_address: absolute_value}\
          {, physical_address|after_physical_address:
     absolute_value},...]
```

Parameters

`name`

Name for new section entry in the memory attribute translation table (MATT); double quotes must enclose the name.

`task_id`

Optional task-identifier value for a section declared inside this directive. Use commas to separate multiple `task_id` values. If you omit this value, the system assigns sections to the system task that directive `.att_mmu_settings` defines.

`start_address`

Starting address in the virtual address space (virtual memory) for the new ATT section. With the `end_address` value, specifies the section range: any power of 2, from 256 bytes to 4 gigabytes. Use commas to separate multiple `start_address/end_address` values.

`end_address`

Ending address in the virtual address space (virtual memory) for the new ATT section. With the `start_address` value, specifies the section range: any power of 2, from 256 bytes to 4 gigabytes. Use commas to separate multiple `start_address/end_address` values.

`_RESERVED_`

Keyword for reserving a memory region in both virtual and physical space. With its subordinate parameter values, forms an entry in the ATT. Particularly useful for programming peripherals.

`size`

Number of bytes in the reserved memory region.

`region_type`

Memory type of the reserved region: “data” or “program”.

`attribute`

If subordinate to `_RESERVED_`, a mandatory absolute value that specifies extended section attributes.
If subordinate to `section_name`, an optional absolute value that fills the same role for the named section.

(The RTOS or debugger, not the linker, interprets `attribute` values.)

`single_mapped`

If this attribute is set, the linker generates a "`section_name`" segment that is placed starting at `<absolute_value>` value on a first fit approach in virtual and physical space, but the section preceding it will use the same range of addresses for virtual and physical space.

NOTE This field cannot appear together with the `base_address`, `physical_address` and `after_physical_address` for a section.

`base_address`

If subordinate to `_RESERVED_`, a mandatory absolute value that positions the section relative to the preceding section in virtual address space that the translation tables define. For the first section, must have value 0. If not 0, must be a multiple of the section size.

If subordinate to `section_name`, an optional absolute value that fills the same role for the named section.

`physical_address`

If subordinate to `_RESERVED_`, a mandatory absolute value that positions the section relative to the preceding section in physical address space. For the first section, must have value 0. If not 0, must be a multiple of the section size.

If subordinate to `section_name`, an optional absolute value that fills the same role for the named section.

`section_name`

Name of a section for which the system creates a descriptor in the memory attribute translation table MATT). The system automatically increases the size of this section until it conforms to the minimum that directive `.att_mmu_settings` defines.

`after_physical_address`

Optional address value. Tells the linker to place the named section at or after this address, wherever it first fits.

Remarks

With other directives `.att_mmu_settings`, `.define_region_to_map_virtual_addressing`, and `.define_single_mapped_virtual_addressing`, provides control of the memory management unit (MMU).

This directive checks restrictions that the memory attribute translation table (MATT) defines in the MMU:

- Confirms the memory-region size that directive `.att_mmu_settings` specifies.
- Confirms that base_address alignment is 0 or a multiple of the region size.
- Sets section attributes.
- Sets task identifiers.

The task sections of `.att_mmu` directives define objects; for each such object, the linker generates map-file entries for the virtual address, physical address, size, and symbol name. If several tasks share a section, the linker makes sure that no other section overlaps this section in the virtual space — unless the `.att_mmu_settings` directive permits overlapping.

If the linker finds any sections not included in `.att_mmu` directives, it generates a warning.

If an `.att_mmu` directive specifies a section larger than the `max_descr_size` value (of the `.att_mmu_settings` directive), the linker generates an error message. If the `.att_mmu` directive specifies a section smaller than the `min_descr_size` value, the linker pads the section up to that minimum.

The linker places sections in virtual memory according to their `base_address` values. For sections without `base_address` values, the linker uses an optimization algorithm to control placement in virtual memory.

The linker places sections in physical memory according to their `physical_address` and `after_physical_address` values. For sections without these parameter values, the linker follows the guidance of the `.org` directive.

Examples

The Overlays chapter includes two extended examples of the `.att_mmu` directive.

.att_mmu_settings

Defines memory management unit (MMU) configuration parameters, such as region size, maximum counter values, and prevention of descriptor overlap in virtual memory space.

```
.att_mmu_settings min_descr_size : absolute_value,
     max_descr_size : absolute_value,\
     {system_task : absolute_value,}\
     {max_data_descr_count : absolute_value,}\
     {max_program_descr_count : absolute_value,}\
     {can_not_overlap : absolute_value
      [{,can_not_overlap : absolute_value},...],}\
     {force_overlap : absolute_value}
```

Parameters

`min_descr_size`

Minimum size of a descriptor in the memory attribute translation table (MATT), but the size of the descriptor must be aligned to the power of two. Default value: 256 bytes.

`max_descr_size`

Maximum size of a descriptor in the memory attribute translation table (MATT), but the size of the descriptor must be aligned to the power of two. Default value: 4 gigabytes.

`system_task`

Optional value that identifies the system task. RTOS code and data can make use of this value. Default value: 0.

`max_data_descr_count`

Optional value defining the number of data descriptors in the MATT. If the number of system_task data descriptors plus the number of data descriptors for any other task exceeds this value, the linker generates an error message. Default value : 0, which tells the linker to not perform this validation.

`max_program_descr_count`

Optional value defining the number of program descriptors in the MATT. If the number of system_task data descriptors plus the number of program descriptors for any other task exceeds this value, the linker generates an error message. Default value : 0, which tells the linker to not perform this validation.

`can_not_overlap`

Optional value defining the bit-set a descriptor attribute must contain — to *prevent* descriptor-range overlap in virtual memory. Separate multiple `can_not_overlap` values with commas.

`force_overlap`

Optional value defining the bit-set a descriptor attribute must contain — to *permit* descriptor-range overlap in virtual memory.

Remarks

With other directives `.att_mmu`, `.define_region_to_map_virtual_addressing`, and `.define_single_mapped_virtual_addressing`, provides control of the memory management unit (MMU).

The RTOS uses the `force_overlap` information to determine whether a descriptor has low or high priority in the MATT. (You can define the MATT priority scheme. If only two descriptors overlap in virtual memory, the linker can determine which has the highest priority.

Use the `can_not_overlay` and `force_overlay` values to define attributes of directives `.att_mmu` and `.define_single_mapped_virtual _addressing`. This stores this important information as bit-sets, eliminating the need to hard-code the information.

.bss

Creates an un-initialized BSS section in the executable file.

```
.bss "section_name", "flags", length, alignment
```

Parameters

`section_name`

Name of the section.

`flags`

A string containing any combination of `r`, `w`, or `x` (read, write, or execute, respectively).

`length`

Length of the section.

`alignment`

Alignment of the section, which must be a power of 2.

Remarks

If a `.segment` directive does not explicitly specify the BSS section, the linker automatically links the BSS section, on a first-fit basis after processing the linker command file.

Example

```
   .bss ".bss_example", "rw", 0x100, 0x2
```

.cache_setting

Specifies the cache optimization settings.

```
.cache_setting \
      type: <name_type>, \
      way: <number_of_way>, \
      line: <number_of_line> , \
      size_of_line: <size_value>, \
      line_index_mask: <line_index_mask_value>
```

Parameters

`name_type`

Specifies either Data or Instruction cache setting. Possible values are:

- “`L1Data`” – Data cache, level one
- ”`L1Instruction`” – Instruction cache, level one

`number_of_way`

Number of way slots for the specified cache type.

`number_of_line`

Number of line of the specified way.

`size_value`

Size value for the specified cache line.

`line_index_mask_value`

You can calculate the line index by performing AND bit operation between this value and the address of the object. The linker uses the line index when number of called objects is greater than available way slots.

Remarks

It is mandatory for the linker to perform this optimization.

Example 1: L1 Data Cache Description Information for MSC8144 Platform

```
.cache_setting \
type: "L1Data", \
way: 8, \
line: 16 , \
size_of_line: 256, \
line_index_mask: 0xF00
```

Example 2: L1 Instruction Cache Description Information for MSC8144 Platform

```
.cache_setting \
type: "L1Instruction", \
way: 8, \
line: 8 , \
size_of_line: 256, \
     line_index_mask: 0x700
```

.concatenate

Concatenates a list of overlay sections, in the specified order.

```
.concatenate "name", "section_pattern" \
     {,unmatch_pgm ("exception_sec_pattern")} \
     {,unmatch_data ("exception_sec_pattern")} \
     {,unmatch_bss ("exception_sec_pattern")} \
     {,unmatch_rom ("exception_sec_pattern")}\
     [{," section_pattern"...}, \
       {,unmatch_pgm ("exception_sec_pattern")} \
       {,unmatch_data ("exception_sec_pattern")} \
       {,unmatch_bss ("exception_sec_pattern")} \
       {,unmatch_rom ("exception_sec_pattern")}...]
```

Parameters

`name`

Name of the section that contains concatenated sections.

`"section_pattern"`

Pattern that identifies a section; may include wildcards (*, ?, and []) to specify an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

`unmatch_pgm ("exception_sec_pattern")`

Includes all program sections (that are not defined in any other directive) in the concatenated section.

`unmatch_data ("exception_sec_pattern")`

Includes all data sections (that are not defined in any other directive) in the concatenated section.

`unmatch_bss ("exception_sec_pattern")`

Includes all un-initialized data sections (that are not defined in any other directive) in the concatenated section.

`unmatch_rom ("exception_sec_pattern")`

Includes all read only data sections (that are not defined in any other directive) in the concatenated section.

Remarks

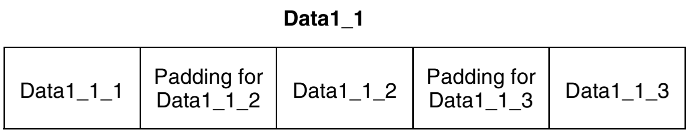
Pertains only to overlay sections. Overlay sections in the list must share such properties as flags and section type. The linker forms each section of the list by concatenating overlay-section fragments, according to the specified linking order.

Example 1

```
.concatenate "Data1_1", "Data1_1_1", "Data1_1_2", "Data1_1_3"
```

Figure 3.1 depicts the result of this command: section list Data1_1.

Figure 3.1 Section List Data1_1



Example 2

Consider Listing 3.2.

Listing 3.2 .concatenate Directive Example

```
.concatenate "descriptor__m2__cacheable__sys__shared__text", \
".m2__cacheable__sys__shared__text", \
".text", ".default"

.concatenate "descriptor__m3__cacheable__sys__shared__text", \
".m3__cacheable__sys__shared__text"

.concatenate "descriptor__ddr__cacheable__sys__shared__text", \
".ddr__cacheable__sys__shared__text",unmatch_pgm()

.att_mmu "Shared_mmu_m2", \
_M2_SHARED_start, _M2_SHARED_end, \
"descriptor__m2__cacheable__sys__shared__text", \
attribute: SYSTEM_PROG_MMU_DEF, \
after_physical_address: _M2_SHARED_start

.att_mmu "Shared_mmu_m3", \
_M3_SHARED_start, _M3_SHARED_end, \
"descriptor__m3__cacheable__sys__shared__text", \
attribute: SYSTEM_PROG_MMU_DEF, \
after_physical_address: _M3_SHARED_start
```

```
.att_mmu "Shared_mmu_ddr", \
_DDR_SHARED_start, _DDR_SHARED_end, \
"descriptor__ddr__cacheable__sys__shared__text", \
attribute: SYSTEM_PROG_MMU_DEF, \
after_physical_address: _DDR_SHARED_start
```

The Listing 3.2 produces the following memory map in the ELD output file. Note that even though the `.unlikely` section is not explicitly included in the MMU directive, it is concatenated in the `descriptor__ddr__cacheable__sys__shared__text` descriptor.

```
;0x40000000  0x40000000 54   Section:
descriptor__ddr__cacheable__sys__shared__text
;0x40000000  0x40000000 12   Section:
.ddr__cacheable__sys__shared__text(msc8144_main.eln)
;0x40000000  0x40000000 12   _main
;0x4000000c  0x4000000c      F_main_end
;0x40000010  0x40000010 38   Section: .unlikely(msc8144_main.eln)
;0x40000010  0x40000010 38   msc8144_mainmsc8144_mainPFO (local)
```

.define_compress

Enables compression for specified overlay sections.

`.define_compress "section"[,"section"...]`

Parameter

`section`

Name of section.

Remarks

To prevent compression for overlay sections, use the opposite directive: `.inhibit_compress`.

.define_overlay

Enables overlay support for sections compiled without overlay support.

```
.define_overlay "section"[,"section"...]
```

Parameter

`section`

Name of section.

Remarks

Converts the specified sections from type SHT_PROGBITS or SHT_NOBITS to type SHT_STARCORE_OVERLAY. This enables overlay support, even for sections compiled without overlay support.

NOTE Overlay support via this linker directive may not be the complete equivalent of compiling sections with overlay support. If a debugger must have old-style overlay information, only the compiler can provide that information.

.define_region_to_map_virtual_addressing

Defines region-to-map virtual addressing, useful for programming peripherals when memory protection is on.

```
.define_region_to_map_virtual_addressing type_region,\
     physical_start_address, virtual_start_address, size\
     {,attribute : absolute_value}\
     {,task_id : absolute_value}\
     [{,task_id : absolute_value}...]
```

Parameters

`type_region`

Memory type of the reserved region: “data” or “program”.

`physical_start_address`

Absolute value that specifies the start of the physical address.

`virtual_start_address`

Absolute value that specifies the start of the virtual address.

`size`

Absolute value that specifies the size of the region.

`attribute`

Optional absolute value that specifies extended region attributes. The RTOS and debugger can interpret this value.

`task_id`

Optional task-identifier value for a section of the region to which this directive pertains. Use commas to separate multiple `task_id` values. If you omit this value, the system assigns sections to the system task that directive `.att_mmu_settings` defines.

Remarks

With other directives `.att_mmu`, `.att_mmu_settings`, and `.define_single_mapped_virtual_addressing`, provides control of the memory management unit (MMU).

This directive inserts an entry in the `.att_mmu` section for each task.

.define_single_mapped_virtual_addressing

Defines single-mapped virtual addressing for non-overlay (non-translation) sections.

```
.define_single_mapped_virtual_addressing\
     "section_name"\
             {,attribute: absolute_value}\
             {,task_id: absolute_value}\
             [{,task_id: absolute_value}...]\
     [,"section_name"\
             {,attribute: absolute_value}\
             {,task_id: absolute_value}\
             [{,task_id: absolute_value}...]...]
```

Parameters

`section_name`

Name of section.

`attribute`

Optional absolute value that specifies extended region attributes. The RTOS and debugger can interpret this value.

`task_id`

Optional task-identifier value for the section to which this directive pertains. Use commas to separate multiple `task_id` values. If you omit this value, the system assigns sections to the system task that directive `.att_mmu_settings` defines.

Remarks

With other directives `.att_mmu`, `.att_mmu_settings`, and `.define_region_to_map_virtual_addressing`, provides control of the memory management unit (MMU).

Use commas to separate multiple `section_name` clauses.

All sections this directive lists must have the same size and alignment requirements as the translation sections. The system stores information about virtual-addressed sections in the `.att_mmu` section.

.entry

Assigns the address at which to begin executing the program.

```
.entry expression
```

Parameter

`expression`

The beginning address.

Example

```
.entry 0x8000
```

.exclude

Tells the linker to *not* link the specified sections.

```
.exclude "section"[,"section"...]
```

Parameter

`section`

Name of section.

Example

```
.exclude .sec1, .sec2
```

.export

In a multicore environment, defines the memory devices the current core shares. (Other cores should use the .import directive to access the segments placed in the shared spaces.)

```
.export "space_pattern"[,"space_pattern"...]
```

Parameter

`space_pattern`

Memory-device identifier.

.firstfit

Modifies the behavior of the linker: places segments on a first-fit basis.

```
.firstfit [absolute_address]
```

Parameters

`absolute_address`

The absolute_value, if specified, is an absolute start address of placing. If this parameter is omitted, the start address of placing is zero.

Remarks

Linker default behavior is to place segments consecutively, starting at the address that the `.org` directive has specified. The .firstfit directive modifies this behavior by switching to a first-fit placement mode. In this mode, the linker uses the lowest available memory region for a segment, and segments need not be consecutive.

Example

In this example, segment A is at address 0x1000, followed by segment B, then by segment C, with no gaps between them except possibly for alignment padding. However, segments D, E, and F are each at the lowest possible address. These segments might not be consecutive, and no memory order can be inferred.

```
.org 0x1000
.segment A, ".section_a"
.segment B, ".section_b"
```

```
    .segment C, ".section_c"
    .firstfit
    .segment D, ".section_d"
    .segment E, ".section_e"
    .segment F, ".section_f"
```

.frequency

Specifies the caching frequency of an object within a function.

```
.frequency \
    function: <function_name> [<self>], \
    object: <object_name>, < frequency> {[, object, < frequency>] …} \
    {[, function: <function_name>, \
    object: <object_name>, < frequency> {[, object, < frequency>] …} …
    }
```

Parameters

`function_name`

Name of the function.

`self`

Function's cycle count, excluding the sub-function.

`object_name`

Name of the object that is called from within the specified function.

`frequency`

Number of calls for the object in the specified function.

Remarks

If the frequency information is missing in the LCF, the linker uses the static information.

Example

```
    .frequency \
    function: "___doprnt" 791,\
      object: "_fwrite", 3,\
      object: "_isdigit", 1,\
      object: "_memcpy", 1,\
    function: "_fwrite" 209,\
      object: "_atexit", 1,\
      object: "_memcpy", 1,\
      object: "___write", 1,\
    function: "___target_c_start" 14890
```

You can get this information by profiling the application. Use this method:

- Generate `prof_func.rep` file by running this simulator command: `runsim -p prof ...`
- Run the following Perl script, available in `StarCore_Support\compiler\bin` directory, and create a text file containing the profiling information:

  ```
  cache_optimization.pl prof_func.rep > prof.txt
  ```

- For each core, include the `prof.txt` file in the LCF. For example, add this line in `local_data.txt`:

  ```
  .include "prof.txt"
  ```

.group

Defines a logical name for a group of sections. Also defines a partial ordering of the sections, by runtime address, that match `section_group_pattern`.

```
.group "group_name"{{, load_Address},
     segment_type},"section_group_pattern"{+offset}
     [,"section_group_pattern"{+offset}...]
```

Parameters

`load_address`

Address where the linker loads the group of sections — as a segment. If the directive includes a load_address value, it also must include a `segment_type` value.

`segment_type`

Type specifier: 1 = load segment; 2 = dynamic segment.

`section_group_pattern`

Name of a section, group of sections, or overlay. This name may include wildcards (*, ?, and []) to specify an arbitrary character sequence.

`offset`

Hexadecimal value for offset placement into the group. This offset value must dovetail with section alignment.

Remarks

Expressions that include expression functions, either directly or by substitution, may not be arguments for this directive.

Example 1

You can use this directive with the .overlay directive to create a hierarchy of overlay sections.

```
.overlay "Overlay1", "rwx", "Pgm7", "Pgm8", "Pgm9"
.group "Group1", "Pgm1", "Pgm2"
.group "Group2", "Pgm3", "Pgm4", "Pgm5"
.group "Group3", Overlay1", "Pgm6"
.overlay ".MyOverlay", "rwx", "Group1", "Group2", "Group3"
.org 0x10000
.segment OVER, ".MyOverlay"
```

Example 1 produces this memory mapping at run time:

```
0x10000:    [Pgm1]  [PGM3]  [Pgm7/Pgm/8/Pgm9]
   .        [Pgm2]  [Pgm4]  [Pgm6]
   .                [Pgm5]
   .
0x7ffff:
```

Example 2

```
.group "G1", "sec_ovl1"+0x8,"sec_ovl2","sec_ovl3"+0x100
.group "G2", "sec_ovl4"+0x8,"sec_ovl2","sec_ovl5"
.overlay "OVL","wxr", "G1","G2"

.org 0x4000
.segment RUN_OVL, "OVL"
```

Example 2 produces this memory mapping at run time:

```
0x4000:     {unused}     [sec_ovl4]
0x4008:     [sec_ovl1]   {unused}
            [sec_ovl2]   [sec_ovl2]
            {unused}     [sec_ovl5]
0x4100:     [sec_ovl3]
    .
    .
    .
0x7ffff:
```

.group_firstfit_start

Same as the .group directive, but for use with the firstfit algorithm.

```
.group_firstfit_start [absolute_address]
```

Parameters

absolute_address

The absolute_value, if specified, is an absolute start address of placing. If this parameter is omitted, the start address of placing is zero.

.import

In a multiple core environment, specifies the external memory devices that are visible to the current core. (Any symbols defined in the imported spaces are visible to the current core, even if they are not part of the object file for this core.)

```
.import "space_pattern"[,"space_pattern"...]
```

Parameter

`space_pattern`

Memory-device identifier.

.include

Appends the content of specified LCF at the current location.

```
.include “lcf_name”
```

Parameters

`lcf_name`

Name of the LCF. Specify absolute path to include LCF from a different directory.

Remarks

Use this directive to segregate a large LCF into smaller parts, or to modify a LCF to link against different architectures and memory mappings.

Example

```
.include "file1.lcf" //if file1.lcf is local to the current
directory
```

```
.include "x:/dir1/dir2/file2.lcf" //if file2.lcf exists in some
other directory
```

.inhibit_compress

Prevents compression for specified overlay sections.

```
.inhibit_compress "section"[,"section"...]
```

Parameter

`section`

Name of section.

Remarks

To enable compression for overlay sections, use the opposite directive: `.define_compress`.

.inhibit_folding_symbols

Inhibit folding the specified symbols.

```
.inhibit_folding_symbols symbol [, symbol...]
```

Parameter

`symbol`

Any valid symbol.

Remarks

Code/data folding optimization is removing duplicated code (code that has the same contents) and duplicated data (constant data that has the same contents).This directive lets you turn off this folding optimization if it is not appropriate for certain symbols.

.inhibit_folding_modules

Inhibit folding duplicated code or data from the specified module.

```
.inhibit_folding_modules "module_pattern" [,"module_pattern"...]
```

Parameter

`module_pattern`

Name of an ELF module, or a pattern that specifies a module name. Double quotes must enclose the pattern; the pattern may include wild-card characters *, ?, or [.

Remarks

Code/data folding optimization is removing duplicated code (code that has the same contents) and duplicated data (constant data that has the same contents). This directive lets you turn off this folding optimization if it is not appropriate for certain modules.

Wild-card characters match any module-name character, including a slash or leading period. If you specify the name of a library, you must start `module_pattern` with a * character, which tells the linker to ignore the library path.

.init_table_section

Tells the linker to consider sections that match the specified identifier pattern to be init_table sections. (For backward compatibility, the linker uses patterns “`.init_table`” and “`*'.init_table`” to recognize default init table sections.)

```
.init_section_table "section_pattern"[,"section_pattern"...]
```

Parameter

`section_pattern`

Section identifier.

.library_concatenate_sections

Combines the sections in a self-contained library into a new section.

```
.library_concatenate_sections "name", "section_name_pattern" [,
     "section_name_pattern" ...]
```

Parameters

`"name"`

Name of the new section where specified sections are put together. Double quotes must enclose the name.

`"section_name_pattern"`

Pattern that identifies a section; may include wildcards (*, ?, and []) to specify an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

Remarks

When this directive is present in the LCF, the linker will create a re-locatable ELF file by incremental linking. You may suppress this behavior by using this command line directive: `–force-self-contained-library`.

NOTE The `.library_concatenate_sections` directive is valid only when creating a self-contained library. See 2.2.1.6 Self-Contained Libraries for more information.

Example

Consider the ELF input files that Listing 3.3 shows.

Listing 3.3 ELF Input Files

```
Module1.eln
 section .sec1:
  _f11
  …
  F_f11_end
 endsec

 section .rela.sec1
  ….
 endsec

Module2.eln
 section .sec1:
  _f21
  …
  F_f21_end
 endsec

 section .sec2:
  _f22
  …
  F_f22_end
 endsec

 section .rela.sec2
  ….
 endsec
```

The following command in the LCF merges the section patterns `.sec1` and `.sec2` into a new section `.sec_conc`:

```
.library_concatenate_sections ".sec_conc", ".sec1", ".sec2"
```

Listing 3.4 shows the final ELF output file.

Listing 3.4 ELF Output File

```
Module_output.eln
 section .sec_conc:
  _f11
  …
  F_f11_end
  _f21
  …
  F_f21_end
  _f22
  …
  F_f22_end
 endsec

 section .rela.sec_conc
  …
  …
 endsec
```

.memory

Defines a region in memory that is available for linking.

```
.memory lo_addr, hi_addr(, "flags")
```

Parameters

`lo_addr`

32-bit expression that sets the region's low address.

`hi_addr`

32-bit expression that sets the region's high address.

`flags`

Optional string: any combination of `r`, `w`, or `x` (read, write, or execute, respectively).

Remarks

Expressions that include expression functions, either directly or by substitution, may not be used as arguments for this directive.

Example

```
.memory 0, 0xFFFFF
```

.non_ovl

Specifies sections that will not have overlay supports.

```
.non_ovl "name_section"[,"name_section"...]
```

Parameter

`name_section`

Name of an overlay section (one compiled with overlay support) or a section that the .overlay directive helps define.

Remarks

For all sections this directive mentions, the linker changes types, for example SHT_STARCORE_OVERLAY to SHT_PROGBIT and SHT_STARCORE_UNION to SHT_NOBITS.

.org

Specifies a starting address for linking segments. (Use this directive in combination with the `.segment` directive.)

```
.org address
```

Parameter

`address`

Beginning address for consecutive placement of all `.segment` directives occurring after this directive but before the next `.org` directive. Section alignment might result in a slightly higher starting address being used.

Specifies only the physical address.

Example

In this example, the `.org` directive instructs the linker to begin linking the `.text` segment at the value represented by the `CodeStart` variable.

```
.org CodeStart
.segment .text, ".text"
```

.overlay

Specifies which overlays share a run address. The linker combines into a new .bss section all sections that match a `section_pattern` argument. The order of these section in the .bss section is the same as their order in the `.overlay` directive.

```
.overlay "section_name," "flags", "section_pattern"{*\-}
     [,"section_pattern"{*\-} ...]
```

Parameters

`"section_name"`

Name for the new BSS section. Double quotes must enclose the name.

`"flags"`

`p` - progbits, `r` - read, `w` - write, `x` - execute; double quotes must surround this parameter value.

`"section_pattern"`

Pattern that identifies a section; may include wildcards (*, ?, and []) to specify an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

Remarks

You may designate *one* of the overlay sections as a default, making the section type .progbits instead of .bss. To do so, append the * or - character:

- The * character tells the linker to load the default section at both its load address and its run address.

- The - character tells the linker to load the default section at its run address, but not at its load address.

This directive creates an overlay section large enough to contain any of the specified sections. This directive determines where the overlay will be loaded. If the overlay section is not named in a `.segment` directive, the linker links the section on a first-fit basis after processing the linker command file.

Use the `p` flag to specify type SHT_PROGBITS for the section, instead of the default type SHT_NO_BITS.

Example

This example shows usage of the `.overlay` and `.segment` directives. The results: at a given time you may run only one of the sections `.ovl_alpha1`, `.ovl_alpha2`, or `.ovl_alpha3`; at a given time, you may run either `.ovl_beta1` or `.ovl_beta2`, but not both.

```
.overlay ".ovlalpha", "rwx",
".ovl_alpha1",".ovl_alpha2",".ovl_alpha3"
.overlay ".ovlbeta", "rwx", ".ovl_beta1",".ovl_beta2"

.org 0x200
.segment TEXT, ".text"
.segment OVLALPHA, ".ovlalpha"
.segment OVLBETA, ".ovlbeta"

.org 0x10000
.segment RODATA, ".rodata"
.segment OVERLAYS, ".ovl_*"
```

.place_symbols

Places the specified symbols in a target section.

`.place_symbols "file_pattern", "symbol_pattern", "section_name"`

Parameters

`“file_pattern”`

Pattern that identifies an ELF file; may include wildcards (*, ?, and []) for specifying an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns. You must use an asterisk (*) to start the name of a library. This tells the linker to ignore the library’s path.

`"symbol_pattern"`

Pattern that identifies an ELF symbol; may include wildcards (*, ?, and [) for specifying an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

`"section_name"`

Name of the target section. If the section you specify already exists in the linker, the symbols are moved directly to that section. If the section does not exist, the linker creates a new section using the attributes from the symbol's original section, and then moves the symbols.

Remarks

You can also set a target section for only one symbol. Use one of the following options to enable or disable one symbol per section:

- -Xllt --one_symb_per_sect0: disabled
- -Xllt --one_symb_per_sect1: enabled
- -Xllt --one_symb_per_sect2: enabled only for functions
- -Xllt --one_symb_per_sect3: enabled only for variables

If the ELF file is not one symbol per section, the linker renames the original section of the symbol (where the symbol is defined) to the specified target section name.

Example

This example shows how the library members are expressed in the file pattern search as `archive(member)`. In this example, all library symbols whose names begin with “_foo” are moved to the .libtext section.

```
.place_symbols "*.elb(*)", "_foo*", ".libtext"
```

.provide

Creates a global, absolute symbol in the symbol table of the executable file.

```
.provide symbol_name, expression
```

Parameters

`symbol_name`

Name of the symbol to be inserted.

`expression`

Value of the symbol.

Example

```
.provide StackStart, 0x20000
.provide ROMStart, 0x7FFF0
.provide SR_Setting, 0xE40008
```

.puncture

Combines into one section all the sections that the following `.segment` directive mentions, in the order of the input files.

```
.puncture “section_name”
```

Parameter

`section_name`

Name of the section to be created.

Example

Create one section, `data_text`, that contains the concatenation of input-file sections `.data` and `.text`:

```
.puncture “data_text”
.org 0x30000
.segment seg_data_text, “.data”, “.text”
```

.rename

Renames ELF sections before processing

```
.rename "file_pattern", "section_pattern", "new_name"
```

Parameters

`"file_pattern"`

String to match the name of an ELF file.

`"section_pattern"`

String to match the name of an ELF section.

`"new_name"`

String: the new section name.

Remarks

If the file and section names of an incoming ELF file match the `file_pattern` and `section_pattern` arguments, the system makes the `new_name` value the new name of the section.

You may use wildcards (*, ?, and []) in the `file_pattern` and `section_pattern` arguments to specify an arbitrary character sequence. Note that the * and ? wildcards match any character, including a slash or leading dot.

You *must* use an asterisk (*) to start the name of a library. This tells the linker to ignore the library's path.

Example

This example shows how library members are expressed in the file pattern search as `archive(member)`. In this example, all library .text sections are renamed to `.libtext`.

```
.rename "*.elb(*)", ".text", ".libtext"
```

.reserve

Defines a region in memory that is not available for linking. If any LCF directive tries to write to this reserved region, the linker issues a warning.

```
.reserve lo_addr, hi_addr
```

Parameters

`lo_addr`

32-bit expression that sets the region's low address.

`hi_addr`

32-bit expression that sets the region's high address.

Remarks

Expressions that include expression functions, either directly or by substitution, may not be used as an argument for this directive.

Example

```
.reserve StackStart, TopOfMemory
```

.segment

Combines all sections and symbols that match the specified patterns into a new segment, at the current location counter.

```
.segment seg_name {, segment_type}, "section_pattern" |
     ("file_pattern", "symbol_pattern") [,"section_pattern" |
     ("file_pattern", "symbol_pattern")...]
```

Parameters

`seg_name`

Name for the new segment.

`segment_type`

Optional p_type field number that specifies the segment structure: 1 for loadable segment, 2 for dynamic segment, and so on.

`section_pattern`

Pattern that identifies a section; may include wildcards (*, ?, and [ ) for specifying an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

`file_pattern`

Pattern that identifies an ELF file; may include wildcards (*, ?, and []) for specifying an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns. You must use an asterisk (*) to start the name of a library. This tells the linker to ignore the library's path.

`symbol_pattern`

Pattern that identifies an ELF symbol; may include wildcards (*, ?, and []) for specifying an arbitrary character sequence. Double quotes must enclose each pattern; commas must separate multiple patterns.

Remarks

You can also set a target section for only one symbol. Use one of the following options to enable or disable one symbol per section:

- -Xllt --one_symb_per_sect0: disabled
- -Xllt --one_symb_per_sect1: enabled
- -Xllt --one_symb_per_sect2: enabled only for functions
- -Xllt --one_symb_per_sect3: enabled only for variables

If the ELF file is not one symbol per section, the linker moves the original section of the symbol (where the symbol is defined) to the specified location in the segment.

The `("file_pattern", "symbol_pattern")` in the directive syntax identifies the symbols.

Section and symbol order in the new segment reflect these rules:

- For sections and symbols within an executable, the order of section and symbol patterns in the .segment directive. The symbol pattern has a higher priority than section pattern in case of a conflict.
- For like-named sections or symbols from different files, the file order on the command line.

Consecutive .segment directives in the command file produce contiguous segments in the object file, with possible alignment padding inserted.

The linker uses the same segment allocation algorithm, regardless of the segment type. The default type is loadable (type 1); the loader loads each type 1 segments into its load address. The loader does not load dynamic (type 2) segments; you must use a DMA or other external device to load a type 2 segment.

To specify where to begin placing segments in memory, use the .org directive with the .segment directive.

Example

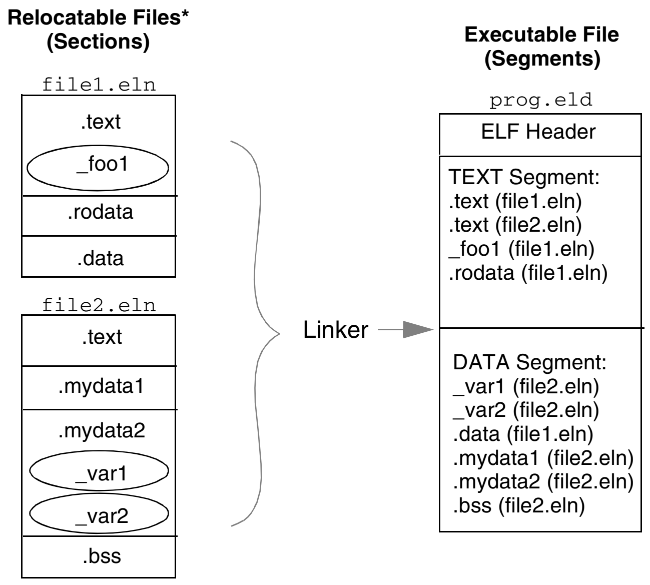
```
.org 0
.segment TEXT, ".text", (“file1.eln”, “_foo1”), ".rodata"
.segment DATA, (“file2.eln”, “_var*”), ".data", ".mydata*", ".bss"
```

These directives:

- Construct a segment named TEXT, which contains all sections named .text, symbol named _foo1 in the file1.eln, followed by all sections named .rodata.
- Construct a segment named DATA, which contains all symbols whose names begin with “_var” in the file2.eln, then all sections named .data, then all sections whose names begin with “.mydata”, then all sections named .bss.
- Place both segments consecutively in memory, beginning at address 0.

Figure 3.2 shows the placement of these sections and symbols. (For the purposes of this example, assume that `file1.eln` was linked before `file2.eln`.)

Figure 3.2 Placement of Sections and Symbols Using .segment Directive



.set

The .set directive creates a 32-bit global symbol in the symbol table of the executable file and assigns a value to it. The linker generates an error if the symbol already exists.

```
.set symbol_name value
```

Parameters

`symbol_name`

Name of the symbol to be inserted.

`value`

Value of the symbol.

Example

```
    .set TextSize, @secsize("text")
```

.space

Defines a space for a group of segments. This space corresponds to a physical memory device in a multiple-core environment.

```
.space space_name, start_address, end_address, {"default",}
     "segment_pattern"[,"segment_pattern"...]
```

Parameters

`space_name`

Name of the space.

`start_address`

Starting address of the space.

`end_address`

Ending address of the space.

`"default"`

Optional flag. If set, specifies that all unmatched sections (those that do not match a segment definition) be placed into the defined space.

`"segment_pattern"`

String that matches the name of one or more segments.

Remarks

Use this directive to group all segments that are to be placed in a given memory device. Spaces may not overlap, as they are uniquely identified by name and range address.

Expressions that include expression functions, either directly or by substitution, may not be used as an argument for this directive.

.union

Determines which pure data overlay shares a run address.

```
.union <section_name>, "flags", "section_pattern" [, "section
     pattern"...]
```

Parameters

`section_name`

Name of a BSS section.

`"flags"`

r - read, w - write, x - execute.

`"section_pattern"`

String that matches a section name.

Remarks

The linker combines all the sections that match the `section_pattern` argument into a BSS section with the specified flags and section name. The sections are combined in the order specified by the pattern argument.

The section resulting from the union is large enough to contain any of the sections that match the pattern argument. The overlay is linked normally using the `.segment` directive, which determines the run address of the union section. If the overlay section is not named in a `.segment` directive, the linker links the section on a first-fit basis after processing the linker command file.

.unit

Splits a linker command file into multiple parts, so that the linker can create multiple object files from a single linker command file. This is useful for programming in a multiple-core environment, as a one project can produce a separate object file for each core.

```
.unit unit_name
```

Parameters

`unit_name`

Name of the output object file

Remarks

All directives between two .unit directives are considered to be for one object file. The `unit_name` is appended to the object file name. The object file contains:

- all sections that are mapped to the unit segments
- all sections whose names begin with unit_name that are not specified in the linker command file (for example: abs sections)

The scope of a segment name is the unit in which it is used.

Example

```
.unit c0
.org _CodeStart_L10
.segment .text, "c0'.text"
.org _DataStart_L10
.segment .data, "c0'.data"

.unit c1
.org _CodeStart_L11
.segment .text, "c1'.text"
.org _DataSTart_L11
.segment .data, "c1'.data"
```

From this example, the linker generates two object files: `c0_a.eld` and `c1_a.eld`.

If you call the linker with two input files containing these sections:

```
input1.eln
  c0'.text, c1'.data, c0'.org.text, .bss
input2.eln
  c1'.text, c1'.data, c1'.data
```

Then the output files will contain these sections:

```
c0_a.eld
  c0'.text, c0'.org.text, .bss
c1_a.eld
  c1'.data, c1'.data, c1'.data, .bss
```

.virtual_memory

Defines a memory region available for dynamic segments (segments that will not be loaded).

`.virtual_memory lo_addr, hi_addr(, “flags”)`

Parameters

`lo_addr`

32-bit expression that sets the region’s low address.

`hi_addr`

32-bit expression that sets the region’s high address.

`"`*`flags`*`"`

r - read, w - write, x - execute.

Remarks

Parameter values must not be expressions that include expression functions, either directly or by substitution.

Example

```
.virtual_memory 0xFF00000, 0xFFFFFFF, “rwx”
```

.xref

Notifies the Linker that the specified symbol has a reference, even if the Linker does not see the reference during processing.

```
.xref symbol
```

Parameter

`symbol`

Any valid symbol.

Remarks

An undefined reference to a symbol prevents the linker from dead stripping. This is useful for forcing the linking of a module containing the specified `symbol`, even if there are no actual references to the symbol in the other modules.

The .xref directive is comparable to the `-U` linker command-line option. You can create an undefined reference to a symbol either by using the .xref directive in the linker command file, or by using the `-U`*`symbol`* option on the command line.

.xref_module

Inhibits stripping of dead code or dead data from the specified module.

```
.xref_module “module_pattern”
```

Parameter

`module_pattern`

Name of an ELF module, or a pattern that specifies a module name. Double quotes must enclose the pattern; the pattern may include wild-card characters `*`, `?`, or `[`.

Remarks

A common optimization is stripping dead code (code that never could be executed) and dead data (data that never could be used). But this directive lets you turn off this stripping optimization if it is not appropriate for certain modules.

Wild-card characters match any module-name character, including a slash or leading period. If you specify the name of a library, you must start module_pattern with a * character, which tells the linker to ignore the library path.

Examples

The directive

```
.xref_module "*gamma.eln"
```

prevents the dead code/data stripping mechanism from removing any objects from module `gamma.eln`.

The directive

```
.xref_module "*lib1.elb(delta.eln)"
```

inhibits dead code/data stripping for module `delta.eln`. of library `lib1.elb`.

An undefined reference to a symbol prevents the linker from dead stripping. This is useful for forcing the linking of a module containing the specified `symbol`, even if there are no actual references to the symbol in the other modules.

The .xref directive is comparable to the `-U` linker command-line option. You can create an undefined reference to a symbol either by using the .xref directive in the linker command file, or by using the `-U`*`symbol`* option on the command line.

4

Overlays

This chapter explains how to create and use overlays in StarCore projects.

- 4.1 Using Overlays
- 4.2 Overlay Manager
- 4.3 Overlay Header Table
- 4.4 Address Translation Table Examples

4.1 Using Overlays

To use overlays in your program, you must:

1. Define the overlays sections in your source code with the overlay pragma
2. Define the overlay space in the linker command file.
3. Add an overlay manager to your source code to copy the overlay sections from the load address to the run address

4.2 Overlay Manager

An overlay manager copies the overlay sections from their load addresses to the run addresses. You must implement your own overlay manager; the runtime does not contain one.

To determine the load and run addresses of each overlay, the overlay manager must read and interpret the overlay header table created by the linker. Listing 4.1 is a simple implementation of an overlay manager.

Listing 4.1 A Simple Overlay Manager

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct ovltab {
  void *ovl_run;
  void *ovl_load;
  unsigned long int ovl_size;
  unsigned long int ovl_checksum;
  unsigned long int ovl_flags;
  unsigned long int ovl_other;
  unsigned short int ovl_shndx;
  unsigned short int ovl_parent;
  unsigned short int ovl_sibling;
  unsigned short int ovl_child;
};

extern struct ovltab _overlay_table[];
extern unsigned long int _overlay_count;

static int Loaded_Segment  = -1;

void *_overlay_manager(void *load_addr)

{
  int i;
#ifdef TRACE_OVL
  printf("Searching load addr: %X\n", load_addr);
#endif
  for (i = 0; i<_overlay_count; i++)
    if ((unsigned long) _overlay_table[i].ovl_load <= (unsigned long)
load_addr &&
     ((unsigned long) _overlay_table[i].ovl_load +
_overlay_table[i].ovl_size) > (unsigned long) load_addr) {
#ifdef TRACE_OVL
      printf("Match at index %d, Run: %X, Size: %d\n", i,
_overlay_table[i].ovl_run, _overlay_table[i].ovl_size);
#endif
      if (Loaded_Segment == -1 || Loaded_Segment != i) {
#ifdef TRACE_OVL
        printf("Loading segment from load address ....\n");
#endif
        Loaded_Segment = i;
        return memcpy(_overlay_table[i].ovl_run,
_overlay_table[i].ovl_load, _overlay_table[i].ovl_size);
      } else {
#ifdef TRACE_OVL
        printf("Segment Already loaded ....\n");
#endif
        /* Already loaded, simply return the run address */
        return (_overlay_table[i].ovl_run);
      }
    }

  printf("Overlay manager: Failed to locate this load address: %X\n",
load_addr);
#ifdef TRACE_OVL
  printf("Overlay structure is:\n");
  for (i = 0; i<_overlay_count; i++) {
    printf("%5d -> Run: %8X, Load: %8X, Size: %8d\n",
          i, _overlay_table[i].ovl_run, _overlay_table[i].ovl_load,
_overlay_table[i].ovl_size);
  }
#endif
  return NULL;
}
```

4.3 Overlay Header Table

The linker creates a section, .ovltab, wherever overlay sections are involved. This section contains two symbols:

- `__overlay_table`

  This is an array of type Elf32_Ovl that contains an entry for each overlay section.
- `__overlay_count`

  This is an unsigned 32-bit integer that represents the number of entries in __overlay_table

The structured type, `Elf32_Ovl`, is:

```
typedef struct{
  Elf32_Addr ovl_run;      /*overlay run address*/
  Elf32_Addr ovl_load;     /*overlay load address*/
  Elf32_Word ovl_size;     /*size of overlay section in bytes*/
  Elf32_Word ovl_checksum; /*checksum of the overlay data*/
  Elf32_Word ovl_flags;   /*overlay flags used by overlay manager*/
  Elf32_Word ovl_other;    /*other information*/
  Elf32_Half ovl_shndx;    /*overlay section index*/
  Elf32_Half ovl_parent;   /*parent overlay*/
  Elf32_Half ovl_sibling;  /*next sibling overlay*/
  Elf32_Half ovl_child;    /*first child overlay*/
} Elf32_Ovl;
```

The field `ovl_other` is a bitset that may contain these flags:

- `OVL_OTHER_NONE 0` — ordinary text section.
- `OVL_OTHER_WRITE 1` — ordinary data section.
- `OVL_OTHER_DEF_LOADED 2` — section the linker loads at its run address.

Define these ovl_other values in the `overlay.h` file.

Only the linker should write field `ovl_other`. The overlay manager may use this field to copy back only the data sections, or to know which sections the linker will load by default at their run addresses.

4.4 Address Translation Table Examples

The previous chapter explained the `.att_mmu` directive, which creates an address translation table (ATT) section. As this command can work with overlays, its examples are in this overlay chapter.

NOTE Appendix B consists of two longer, complex examples: one for a multi-core environment, and another that makes use of the `.att_mmu_setting` directive.

4.4.1 Example 1: Non-Overlay

Suppose that an application includes the sections that Table 4.1 lists. You want the linker to set the base address for each section, and to place these sections into memory efficiently.

Table 4.1 Non-Overlay Example: Initial Sections

Section	Bytes (Decimal)	Bytes (Hexadecimal)	Alignment
`Data1_1`	230	E6	1
`Bss1_1`	3914	F4A	4
`Rom1_1`	30	1E	4
`Pgm1_1`	3260	CBC	16
`Data1_2`	391	187	1
`Bss1_2`	17010	4272	4
`Rom1_2`	18	12	4
`Pgm1_2`	3064	8F8	16
`Data2_1`	432	1B0	1
`Bss2_1`	510	1FE	4
`Rom2_1`	18	12	4
`Pgm2_1`	2536	98E	16
`Data2_2`	28	187	1
`Bss2_2`	10	A	4

Table 4.1 Non-Overlay Example: Initial Sections

Section	Bytes (Decimal)	Bytes (Hexadecimal)	Alignment
`Rom2_2`	18	12	4
`Pgm2_2`	5200	1450	16

Use the `.att_mmu` directive to create a memory address translation table (MATT) section:

```
.att_mmu "MATT", 0x0, 0xffffff, \
    "Data1_1", "Bss1_1", "Rom1_1", Pgm1_1", \
    "Data1_2", "Bss1_2", "Rom1_2", Pgm1_2", \
    "Data2_1", "Bss2_1", "Rom2_1", Pgm2_1", \
    "Data2_2", "Bss2_2", "Rom2_2", Pgm2_2"
```

The linker:

- Specifies a memory range that will accommodate each section: a power of 2, 256 bytes to 4 gigabytes. The new size of this range also serves as the alignment for each section.
- Organizes memory ranges from largest to smallest, assigning virtual starting addresses so as to fill a contiguous range of virtual memory.
- Maps each section to an appropriate range of physical memory.

Table 4.2 shows the new sizes (decimal and hexadecimal) for each section, and lists the new addresses for each section. Figure 4.1 depicts the layout of virtual memory, and Figure 4.2 depicts the mapping to physical memory.

Table 4.2 Non-Overlay Example: Final Sections

Section	Initial Size	New Size (decimal)	New Size (hex)	Virtual Address	Physical Address
`Data1_1`	230	256	100	0xE600	0x1900
`Bss1_1`	3914	4096	1000	0xD000	0x14000
`Rom1_1`	30	256	100	0xE900	0x6B00
`Pgm1_1`	3260	4096	1000	0xA000	0x12000
`Data1_2`	391	512	200	0xE400	0x1200
`Bss1_2`	17010	32768	8000	0x0000	0x8000
`Rom1_2`	18	256	100	0xE700	0x6A00

Table 4.2 Non-Overlay Example: Final Sections

Section	Initial Size	New Size (decimal)	New Size (hex)	Virtual Address	Physical Address
`Pgm1_2`	3064	4096	1000	0xC000	0x13000
`Data2_1`	432	512	200	0xE000	0x1400
`Bss2_1`	510	512	200	0xE200	0x1000
`Rom2_1`	18	256	100	0xEA00	0x1B00
`Pgm2_1`	2536	4096	1000	0xB000	0x7000
`Data2_2`	28	256	100	0xEB00	0x6800
`Bss2_2`	10	256	100	0xEC00	0x1A00
`Rom2_2`	18	256	100	0xE800	0x6900
`Pgm2_2`	5200	8192	2000	0x8000	0x10000

Figure 4.1 Non-Overlay Example: Virtual Memory Layout

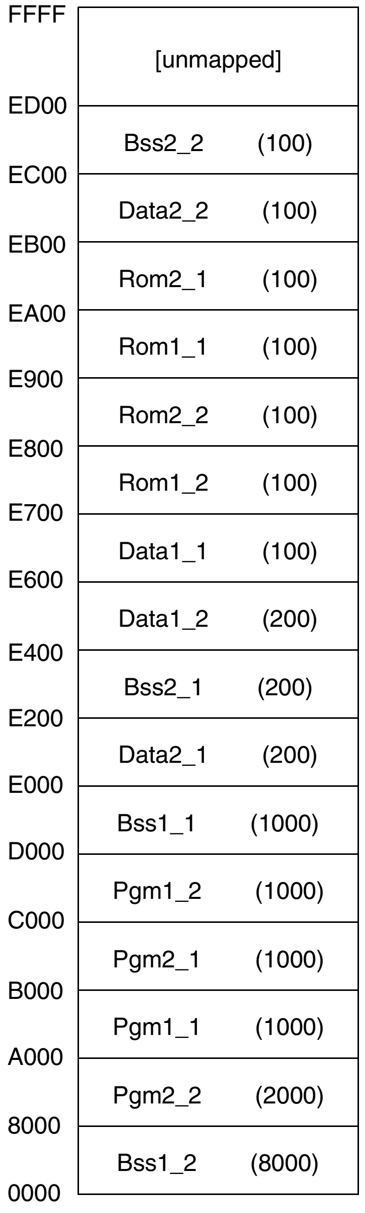
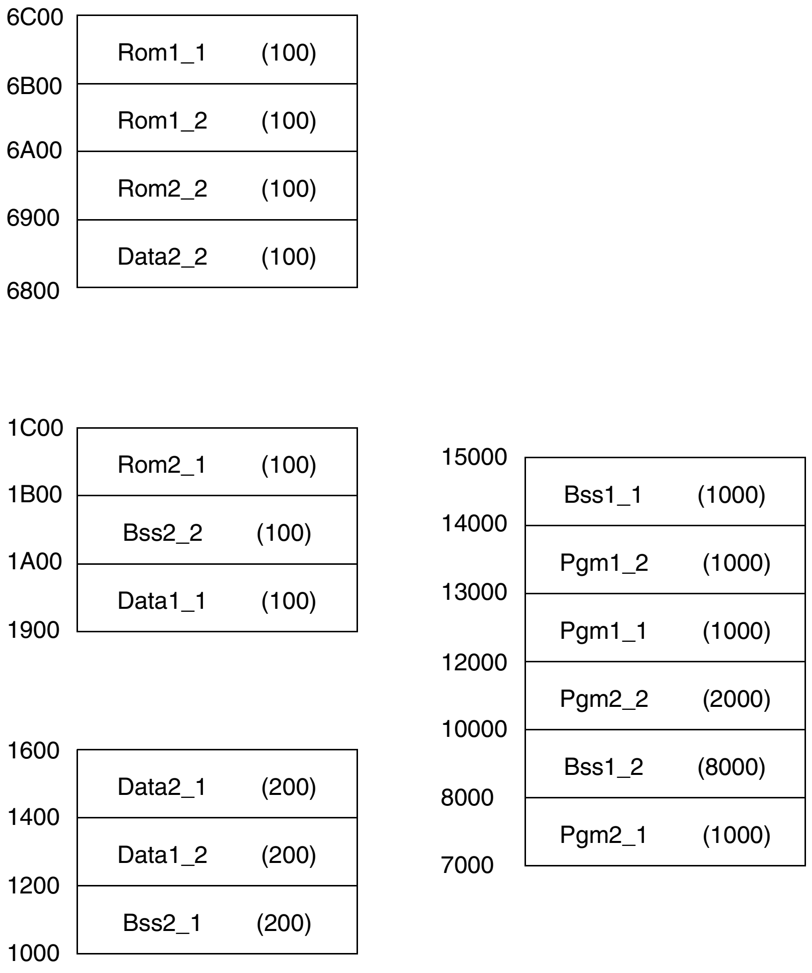


Figure 4.2 Non-Overlay Example: Physical Memory Layout



4.4.2 .att_mmu section

Wherever overlay sections are involved in the address translation table, the linker creates an `.att_mmu` section, which contains these symbols:

1. `__address_translation_table_mmu`

   This is an array of type `ELF32_ATT_MMU`. Listing 4.2 explains the structure of this type. (Sorting in this array is in ascending order of the `physical_address` field.)

2. `__address_translation_table_mmu_count`

   This is an unsigned, 32-bit integer.

3. `__task_table`

   This is an array of type `Elf32_TASK_DESCRIPTOR`. Listing 4.3 explains the structure of this type. (Sorting in this array is in ascending order of the `task_id` field, then the `desc_index` field.)

4. `__task_table_count`

   This is an unsigned 32-bit integer.

Listing 4.2 ELF32_ATT_MMU Structure

```
typedef struct att_mmu{
  Elf32_Addr base_address;       //virtual address/
  Elf32_Addr physical_address;   //physical address
  Elf32_Word physical_size;      //size (bytes) of overlay section
  Elf32_Word attribute;          //bit-set for section attributes
                                 //(like cacheable...), which RTOS,
                                 //debugger, and so forth can use.
  Elf32_Half physical_shndx;     //overlay section index
  Elf32_Half other;              //other information
} Elf32_ATT_MMU;
```

The `other` field is a bit set that may contain these flags:

```
OVL_OTHER_NONE       0 — ordinary section
OVL_OTHER_WRITE      1 — ordinary data section
OVL_OTHER_EXEC        8 — ordinary text section
```

`is in ascending order of the physical_address field.`

Listing 4.3 ELF32_TASK_DESCRIPTOR Structure

```
typedef struct task_map{
  Elf32_Half task_id;            //task identifier
  Elf32_Half desc_index;         //descriptor index in
                                 // __address_translation_table_mmu
                                 //table
} Elf32_TASK_DESCRIPTOR;
```

File `att_mmu.h` defines the data structures and variables needed to work with the `.att_mmu` section.

4.5 Advanced Examples

The final examples of this chapter help clarify how the address translation table feature can be of help to you.

NOTE Examples 2, 3, and 4 each pertain to a task — that consists of `.text`, `.data`, `.rom`, and `.bss` sections — and which is to execute in virtual address space.

4.5.1 Example 2: Irrelevant Section Placement

Requirements:

1. Virtual address space is from `0x0` to `0xffff`.
2. Within this space, it does not matter where the linker places each section.

The appropriate linker directive is:

```
.att_mmu"task",0x0,0xffff,".text",".data",".rom",".bss"
```

As this command does not specify any base addresses, the linker places all sections in virtual memory on a first-fit basis.

4.5.2 Example 3: Specific Base Address

Requirements:

1. Virtual address space is from `0x0` to `0xffff`.
2. The `.text` section must start at address `0x100` in virtual memory.

The appropriate linker directive is:

```
.att_mmu"task",0x0,0xffff,".text",base_address:0x100,".data",".rom",".
bss"
```

This command specifies a base address for the `.text` section, so the linker's first action is placing this section — that is, fixing its position in virtual memory at address `0x100`. Next, the linker places the `.data`, `.rom`, and `.bss` sections on a first-fit basis.

4.5.3 Example 4: Two Ranges, Specific Base Addresses

Requirements:

1. Virtual address space is from `0x0` to `0xffff`, and from `0xf0000` to `0xf1000`.
2. The `.text` section must start at address `0x200` in virtual memory — and at address `0x10000` in physical memory.

3. The `.data` section must start at address `0xf0000` in virtual memory — and its extended section attribute must have the value `0x4`.

The appropriate linker directives are:

```
.att_mmu"task",0x0,0xffff,0xf0000,0xf1000,\
        ".text",base_address:0x200,\
        ".data",base_address:0xf000,attribute:0x4,\
        ".rom",".bss"
.org 0x10000
.segment text, ".text"
```

In response, the linker performs these actions:

1. Per the `.att_mmu` directive, places the `.text` section in virtual memory at address `0x200`.
2. Per the same directive, places the `.data` section in virtual memory at address `0xf0000`, assigning the `0x4` attribute value.
3. Per the same directive, places the `.rom`, and `.bss` sections in virtual memory on a first-fit basis.
4. Per the `.org` and `.segment` directives, places the `.text` section in physical memory. (The `.att_mmu` directive does not control section placement in physical memory. Just as with placing an ordinary section in physical memory, placing the `.text` section there requires the `.org` and `.segment` directives.)

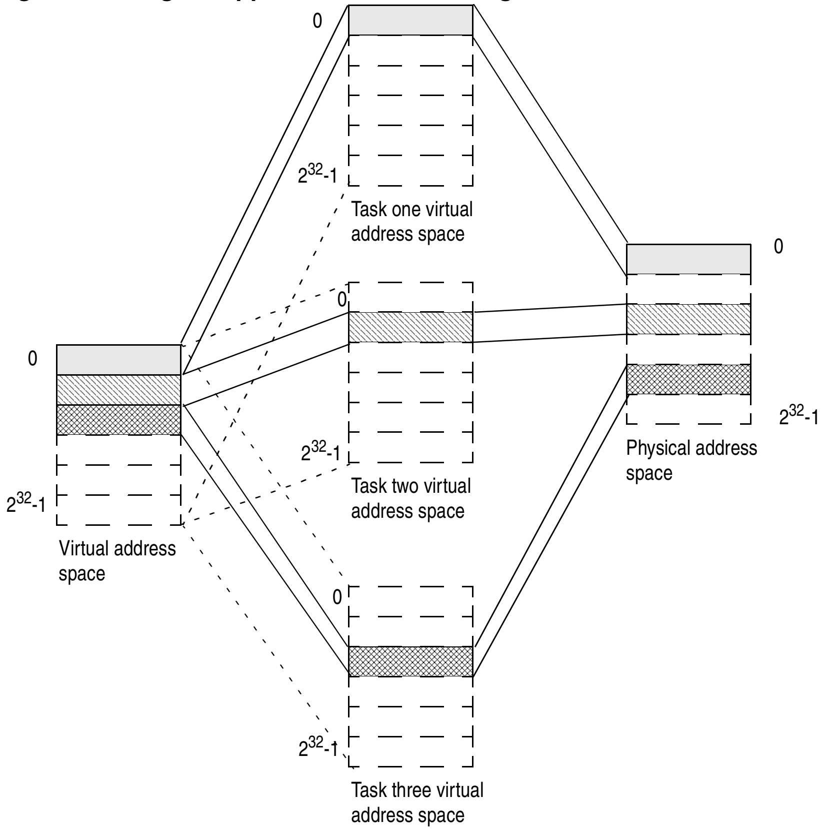
4.5.4 Examples 5: Disjunct Virtual Spaces

Examples 5a and 5b (as well as examples 6a and 6b) pertain to three tasks, each consisting of four sections:

- Task one — `".text1"`, `".data1"`, `".rom1"`, and `".bss1"`
- Task two — `".text2"`, `".data2"`, `".rom2"`, and `".bss2"`
- Task three — `".text3"`, `".data3"`, `".rom3"`, and `".bss3"`

For examples 5a and 5b, execution of the three tasks takes place in disjunct virtual address space. Within the system, use of each virtual address is unique. This is *single-mapped virtual addressing*, which means that different tasks do not use the same virtual addresses at the same time. Figure 4.3 represents this arrangement.

Figure 4.3 Single-Mapped Virtual Addressing



4.5.4.1 Example 5a: All Sections in Virtual Memory

Requirement: Place all sections of all tasks in virtual memory.

The appropriate linker directive is:

```
.att_mmu"tasks",0x0,0xfffff,\
        ".text1",.data1",".rom1",".bss1"\
        ".text2",.data2",".rom2",".bss2"\
        ".text3",.data3",".rom3",".bss3"
```

4.5.4.2 Example 5b: Task-Defined Spaces

Requirement: Place the sections corresponding to each task in the virtual address spaces that each task defines.

The appropriate linker directives are:

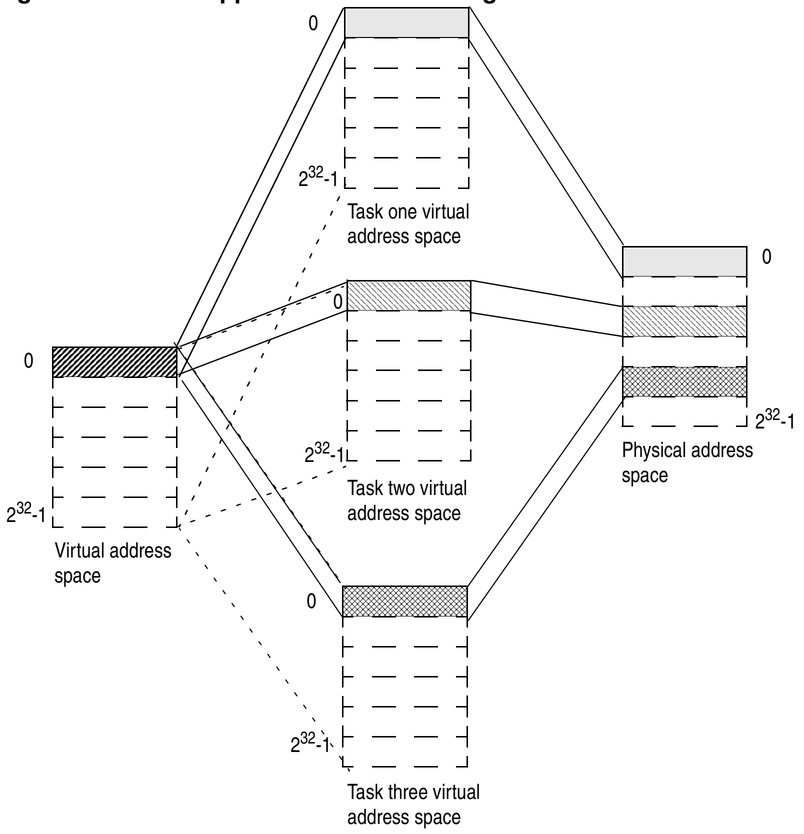
```
.att_mmu"task1",0x0,0xfffff,\
        ".text1",.data1",".rom1",".bss1"\
.att_mmu"task2",@secaddr("task1)+@secsize("task1"),0xfffff,\
        ".text2",.data2",".rom2",".bss2"\
.att_mmu"task3",@secaddr("task2)+@secsize("task2"),0xfffff,\
        ".text3",.data3",".rom3",".bss3"
```

By using the linker function, you can compute the size and starting address in the virtual memory of the sections that `.att_mmu` defines. This function is useful for preserving virtual memory space.

4.5.5 Examples 6: Shared Virtual Spaces

In examples 6a through 6c, execution of the three tasks takes place in shared virtual address space. Each task generates virtual addresses; address translation maps these virtual addresses to different physical memory. This is *multi-mapped virtual addressing*, which Figure 4.4 represents.

Figure 4.4 Multi-Mapped Virtual Addressing



4.5.5.1 Example 6a: Definitions for All Tasks

Requirement: Define each individual task.

The appropriate linker directives are:

```
.att_mmu“task1”,0x0,0xfffff,”.text1”,”.data1”,”.rom1”,\
       ”.bss1”
.att_mmu“task2”,0x0,0xfffff,”.text2”,”.data2”,”.rom2”,\
       ”.bss2”
.att_mmu“task3”,0x0,0xfffff,”.text3”,”.data3”,”.rom3”,\
       ”.bss3”
```

4.5.5.2 Example 6b: Directives for Data and Program

Requirements:

1. Use only .att_mmu directives.
2. Define each individual task.

The solution is to use two .att_mmu directives for each task — one for virtual data and the other for virtual program. The appropriate linker directives are:

```
.att_mmu “task1_data”, task_id:0x1,0x0,0xfffff,”data1”,
   ”.rom1”,”.bss1”
.att.mmu ”task1_program”,task_id:0x1,0x0,0xfffff,”.text1”

.att_mmu “task2_data”, task_id:0x2,0x0,0xfffff,”data2”,
   ”.rom2”,”.bss2”
.att.mmu ”task2_program”,task_id:0x2,0x0,0xfffff,”.text2”

.att_mmu “task3_data”, task_id:0x3,0x0,0xfffff,”data3”,
   ”.rom3”,”.bss3”
.att.mmu ”task3_program”,task_id:0x3,0x0,0xfffff,”.text3”
```

4.5.5.3 Example 6c: .concatenate Directive

Requirements:

1. Define each individual task, as for Example 6b.
2. Reduce the size and number of MMU descriptors.

As with the previous example, two `.att_mmu` directives are necessary to define each task. But satisfying the second requirement requires the `.concatenate` directive. The appropriate linker directives are:

```
.concatenate”data1”,”.data1”,”.rom1”,”.bss1”
.concatenate”data2”,”.data2”,”.rom2”,”.bss2”
.concatenate”data3”,”.data3”,”.rom3”,”.bss3”
.att_mmu”task1_data”,task_id:0x1,0x0,0xfffff,”data1”
.att_mmu”task2_data”,task_id:0x1,0x0,0xfffff,”data2”
.att_mmu”task3_data”,task_id:0x1,0x0,0xfffff,”data3”
.att_mmu”task1_program”,task_id:0x1,0x0,0xfffff,”text1”
.att_mmu”task2_program”,task_id:0x1,0x0,0xfffff,”text2”
.att_mmu”task3_program”,task_id:0x1,0x0,0xfffff,”text3”
```

NOTE Appendix B consists of two longer, complex examples: one for a multi-core environment, and another that makes use of the `.att_mmu_setting` directive.

A

Linker Messages

This appendix lists linker warnings and error messages. The linker always routes such messages to the standard output.

- Table A.1 lists linker warnings.
- Table A.2 lists linker error messages.
- Table A.3 lists linker error codes and their explanations.
- Table A.4 lists various directives and their associated priority in the LCF.

Table A.1 Linker Warnings

Warning	Explanation

`Warning: can't close directory <DIRECTORY>.`
`Warning: can't open directory <DIRECTORY>.`
`Warning: can't remove directory <DIRECTORY>.`
`Warning: can't remove temporary file <FILE>.`
	The linker created its output file, but could not remove the temporary directory or file that it also created. Any of these warnings should include a brief explanation for the failure.

`Warning: Code/data stripping cannot be performed.`
 	
In verbose mode, these additional messages are possible:
`Information: Duplicate function symbol <function_name>.`
`Information: Function <function_name> not found.`
`Information: In module <module_name>: end of function symbol <symbol_name> not defined.`
	• Not all function symbols `function_name` had the corresponding end-of-function symbol definitions `F<function-name>_end`.
	• For applications, the entry function `_main` is not defined. For self-contained libraries, no entry functions are defined using `.library_entry_points` directive.
	• Not all function symbols had valid sizes.

Table A.1 Linker Warnings (*continued*)

Warning	Explanation
`Warning: Ignore overlay support for section <section_name>.`	Command-line option `-non-ovl` specified sections that the .overlay or .union directive named. (Otherwise, the linker would change section types: SHT_STARCORE_OVERLAY to SHT_PROGBIT, SHT_STARCORE_UNION to SHT_NOBITS.) Another way to avoid this warning is to use the .non_ovl directive in the linker control file.
`Warning: In module <module_name>: size of symbol <symbol_name> has a wrong value <value>.`	STT_FUNC or STT_OBJECT symbol failed the size validity check, so dead-code/dead-data stripping is not possible. The sum of the symbol value plus symbol size must not exceed the size of the section that contains the symbol definition.
`Warning: integer overflow.`	A numeric quantity in the linker control file is too big for internal representation, so has been clamped.

`Warning: no linker command file; using internal defaults.`	No -c option specified the command file, nor the linker find one in file `$SC100_HOME/etc/crtscsmm.cmd`. The linker used default options, producing behavior as for a command file containing only:
	`.memory 0, 0xffffffff`
	`.entry 0`

`Warning: non-standard escape sequence \<CHAR>.`	A quoted string in the linker command file contained a backslash character not recognized as a valid escape sequence; the linker ignored the backslash. (Valid escape sequences are `\"`, `\'`, `\?`, `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`, as well as octal and hexadecimal character literals.)

`Warning: The <section_name> absolute overlay section has a different alignment <value>; the current alignment is <value>.`
`Warning: The <section_name> absolute overlay section has a different size <value>; the current size is <value>.`
	Linker accepts unmatched absolute overlay sections (sections whose names do not match any .overlay directives). The linker generates an absolute bss section for all overlay sections that have the same run address. The section size is the maximum over the overlay section sizes. The linker generates a warning if the absolute overlay sections have different sizes or alignments.

Table A.2 Linker Error Messages

Error Message	Explanation
`Error: <FILE>: unknown SC100 core type.` `Error: <FILE>: unknown <CORE> core revision.`	An input file was marked as being generated for a core type (or core revision) that the linker did not recognize. The linker may be out of date with respect to the assembler, but either message could indicate an assembler bug or a vendor-specific extension.
`Error: <SECTION>: absolute section may not be combined.`	The input files contained two or more absolute sections that had the same name. Absolute sections should have unique names; this is why the StarCore assembler incorporates addresses into section names.
`Error: <SYMBOL> is unresolved, referenced from <FILE> ...`	All input files were read, but a required symbol remained undefined. This message lists all files that reference this symbol.
`Error: absolute section <SECTION> cannot be linked at address <ADDRESS>. The address range [<LOW>,<HIGH>] is unavailable.`	Linking was not possible for an absolute section. Either: (1) Another absolute section already occupied all or part of its address range, (2) The range was not a valid memory region -- reserved with the .reserve directive or never allocated with the .memory directive.
`Error: can't create a temporary directory (<DIRECTORY> is full).`	The temporary directory is full. If you cannot correct this condition, set the TMPDIR environment variable to point to a different temporary directory.
`Error: can't create temporary file <FILE>.`	The linker could not create a temporary file; the message includes the reason for this failure.
`Error: can't have .segment before first .org or .firstfit.`	A .segment directive preceded the first .org or .firstfit directive.
`Error: can't fit section <SECTION> at address <ADDRESS>.`	The section did not fit at the memory location that the linker control file specified.
`Error: Can't link group section <section_name> explicitly.`	Attempt to use a .group section for the .segment directive.
`Error: can't link non-allocatable section <SECTION>.`	You requested linking for a debugging section, or some other section not intended to be linked.
`Error: can't link section <section_name> (<size_of_section> bytes, align <value>).`	Insufficient memory space for this section.

Table A.2 Linker Error Messages (*continued*)

Error Message	Explanation
`Error: can't link section <section_name> (<size_of_section> bytes, align <value>) at <offset_value>.`	The section did not fit into the memory location that the linker control file specified.
`Error: cannot find <LIBRARY> in any of <DIRECTORY> ...`	The -l option specified a library that the linker could not find in any search path that the -L option specified.
`Error: Computation of overlay section cannot be done because there are sections <ection_1_name>, <section_2_name> that have different run addresses.`	1. At least one section specified in the .group directives was to be place at two run addresses. 2. Possible loops.
`Error: directory <DIRECTORY> does not exist or is not writable.`	The linker could not access the system's temporary directory. If you cannot correct this condition, set the TMPDIR environment directory to point to a different temporary directory.
`Error: Do not reserve running space for overlay. It is illegal to use a default configuration for overlay.`	No space was reserved for the run address, so it was not possible to load a configuration in that space. Remove an asterisk or minus sign, so that you no longer have a default configuration.
`Error: export <export_name> already exists in command file.`	More than one .export directive used the same export name.
`Error: expression is not absolute.`	A linker directive expected a parameter that evaluated to an absolute quantity, but the parameter was undefined or referred to a relocatable address.
`Error: group <group_name> already exists in command file.`	More than one .group directive used the same group name.

Table A.2 Linker Error Messages (*continued*)

Error Message	Explanation
`Error: illegal external references to symbol <symbol_name> in <module_name> definition in <space_name> of unit <unit_name>. reference from <space_name> of unit <unit_name>.`	1. A symbol was defined in private space; a reference to the symbol was not in another private place of the same unit. 2. A symbol was defined in private space; a reference to the symbol was from shared space, but shared spaces were not defined at the same address in all cores. 3. A symbol was defined in shared space S; a reference to the symbol was not in any private space of the unit, nor was it in any private space of another unit that imported S. 4. A symbol was defined in shared space S; a reference to the symbol was not from shared space whose import list was included in the S import list.
`Error: illegal mapping [<LOW,<HIGH>] (low > high).`	A .memory or .reserve directive specified a memory region that did not make sense.
`Error: import <import_name> already exists in command file.`	More than one .import directive used the same import name.
`Error: invalid alignment (must be power of 2).`	The .align directive received an inappropriate alignment value.

`Error: invalid token at <CHARACTER>`
`Error: expected <TEXT> at <CHARACTER>.`
	A parse error occurred during the reading of the linker control file. The input was malformed (for example, a number contained a non-digit) or the expected type of text was not found (for example, a string appeared where a directive was expected).

`Error: It is illegal to have the section <section_name> in default configuration for overlay, because it is not a group section or an overlay section.`	Attempt to specify a non-group or non-overlay section as default configuration for an .overlay directive.
`Error: It is illegal to have two default configurations into an overlay directive.`	Attempt to have the linker load two sections at the same run address. Remove an asterisk or a minus sign.
`Error: mapping overlaps existing mapping [<LOW>,<HIGH>].`	A .memory directive specified a memory region that a previous .memory directive already specified (partially or completely).

Table A.2 Linker Error Messages (*continued*)

Error Message	Explanation
`Error: mapping overlaps existing segment <SEG<ENT> at <LOW ADDRESS>...<HIGH ADDRESS>.`	An .org directive specified an address that overlaps the start address of another .org directive.
`Error: module <FILE> contains no symbol table.`	An input file contained relocations but no symbol table. This prevents linking any section that contains external references or section-relative internal references.
`Error: multiple absolute sections at address <ADDRESS>.`	The input files contained two or more absolute sections that had the same starting address.
`Error: multiple entry points specified.`	The .entry directive appeared more than once in the linker control file.
`Error: .org values must increase monotonically.`	The linker control file contained a .org directive with a starting address less than that of the previous .org directive. The linker requires section specification from the lowest address to the highest address.
`Error: out of memory.`	The linker ran out of memory. Some operating systems let you increase available memory by closing open applications or specifying a resource limit.
`Error: overlay <overlay_name> already exists in command file.`	More than one .overlay directive used the same overlay name.
`Error: part or all of address range is already unmapped.`	A .reserve directive specified a memory region (1) already partially or completely specified by a previous .reserve directive or (2) never allocated via a .memory directive.
`Error: redefinition of symbol <SYMBOL> in <FILE>.`	An input file defined a symbol already defined by either the linker control file or another input file.
`Error: Relocation Error.`	A relocation could not be performed; the message includes additional details. A likely reason for this message: the input file contained a data or code reference to a symbol value, but the symbol was unaligned or outside the machine instruction’s allowable representation. For example, you may have compiled or assembled code assuming that all data would be in the first 64K of memory, but the linker control file specified data placement outside that range.

Table A.2 Linker Error Messages (*continued*)

Error Message	Explanation
`Error: <section_name> has already been overlaid.`	More than one .overlay directive mentioned the section.
`Error: <section_name> is not an overlay/union section.`	An .overlay/.union directive specified a non-overlay section.
`Error: <section_name> section is not placed into space.`	The section was not placed in memory space. Check all rename directives for inappropriate renaming of this section.
`Error: section <SECTION> can not be linked.`	The section would not fit into any available memory region.
`Error: section <SECTION> in <FILE> has flags <FLAGS> incompatible with existing flags <FLAGS>.`	The input files contained two or more sections with the same name but different flag sets. For example, a .data section could be writable in one file but not in another. This is why the StarCore assembler assigns standard flags to the conventional section names (.text, .rodata, .data, and .bss).
`Error: section <SECTION> is already linked.`	The linker control file specified linking a section more than once: the section's name matches more than one pattern of a .segment directive.
`Error: section <section_name> already exists in overlay <overlay_name>.`	An overlay directive specified the same section twice.
`Error: Section <section_name> in <module_name> has flags <value> (<list_of_flag_name>). The section punctured must have flags <value> (<list_of_flag_name>).`	At least one section following a punctured section did not have flags specified in the flag directive.
`Error: Section <section_name> in <module_name> has loading space (<space_name> in unit <core_name>) incompatible with running space (<space_name> in unit <space_name>), both spaces must have the same unit.`	Loading and running spaces were not in the same unit, keeping necessary information from the overlay manager.
`Error: Section <section_name> was not mentioned in any .overlay/ .union directive.`	An overlay/union section must be present in only one overlay/union directive.

Table A.2 Linker Error Messages (*continued*)

Error Message	Explanation
`Error: space <space_name> already exists in command file.`	More than one .space directive used the same space name.
`Error: The directive .library_entry_points was omitted in the linker command file.`	Attempt to create a self-contained library without this mandatory directive.
Error: The <name> instruction is forbidden at offset <value> in (section index:<value> and module name: <name>).	The instruction is not compatible with the specified target architecture.
`Error: The symbol <symbol_name> has different addresses on all cores. Value <value> from unit <unit_name> in <module_name>.`	A symbol defined in a private space can be referred to from shared space on defined at the same address in all cores.
`Error: union <union_name> already exists in command file.`	More than one .union directive used the same union name.
`Error: unit <unit_name> already exists in command file.`	More than one .unit directive used the same unit name.
`Error: This is a symbol <symbol_name> in a section <section_name> that was cloned for overlay purposes. Cannot find the section <section_name>.`	Linker could not set the index of the section for this symbol.
`Error: unknown option <OPTION>.`	A command-line option was not recognized.
`Error: unrecognized directive <DIRECTIVE>.`	The linker could not recognize a directive in the linker control file.
`Error: You cannot place section <section_name> at offset <value> in group <group_name>. Section <section_name> could not be aligned to <value>. Try again using offset <value>.`	Linker could not use the specified offset. Try the suggested offset.
`Error: You could try to place segment <segment_name> at <address>.`	Linker could not use the specified address. Try the suggested address.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Error(E1001): LCF syntax: unknown directive 'bad_directive'. segment8.cmd(1): .bad_directive`	Unsupported directive. The linker interprets any text beginning with a dot as a directive.

`Error(E1002): unit c0: LCF syntax: missing token ',' after 'shared_m2'. unit c0: os_msc814x_link.lcf(36): .space shared_m2`
 	
 `Error(E1002): unit c0: LCF syntax: missing token ',' before '".shared_data_m2"'. unit c0: os_msc814x_link.lcf(37): .space shared_m2, 0xc000, 0xcfff ".shared_data_m2"`
	Incomplete directive definition.

`Error(E1003): LCF syntax: unknown permission flag 'e', expect [rwx]. bss2.cmd(21): "rew"`	Incorrect permission flag for .bss section. The LCF only accepts ‘r’, ‘w’, ‘x’ or a combination of these three characters as .bss section flag.
`Error(E1004): LCF syntax: invalid or missing directive name, expect string or identifier after '.group'. group2.cmd(53): .group 1234, "Pgm4"`	Incorrect directive name. Directive name should be a string or an identifier.
`Error(E1005): LCF syntax: invalid or empty expression, expect segment_type or section_pattern. segment3.cmd(24): .segment DATA1,`	Incomplete directive definition.
`Error(E1006): LCF syntax: unexpected identifier 'DATA' at 'DATA) + segsize(DATA)'. segment2.cmd(23): DATA) + @segsize(DATA)`	Incorrect directive definition.

Table A.3 Linker Error Codes

Error Code/Message	Explanation

`Error(E1007): LCF syntax: unexpected expression after token '\'.`
`align1.cmd(11): \ "rwx"`
	Incorrect syntax after the line continuation character '\'. Do not specify any other character after the line continuation character. Move the text after the continuation character to a new line.

`E1008`	RESERVED.

`Error(E1009): LCF syntax: invalid or empty expression, expect constant expression after '.align'.`
`align3.cmd(26): .align sad`
	Incorrect expression in the directive definition. Specify a constant value expression.

`Error(E1010): LCF syntax: invalid or empty expression, expect constant expression for length field in .bss directive.`
`bss4.cmd(21): .bss "BSS","rxw",sada`
	Incorrect directive definition. Some non-trivial directives may have more than one field in the syntax structure, which must be written in the specified order.

`Error(E1011): LCF configuration: redefinition of directive .group with same name 'G1'.`
`group3.cmd(52): .group "G1","Pgm1","Pgm2"`
`group3.cmd(53): .group "G1"`
	Identical directive name in the LCF. Specify a unique name for the directive definition.

`Error(E1012): LCF configuration: redefinition of directive .rename with same pattern '*.elb(*) .text .libtext'.`
`rename8.cmd(1): .rename "*.elb(*)", ".text", ".libtext"`
`rename8.cmd(2): .rename "*.elb(*)", ".text", ".libtext"`
	Same pattern definition for more than one directive of the same type. Specify unique pattern definition for each directive.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Error(E1013): LCF configuration: redefinition of directive .entry. entry5.cmd(20): .entry @segaddr(INTVEC)`	Two definitions for the .entry directive. Specify a single entry point for an application.
`Error(E1014): LCF configuration: duplicate pattern '_Usymbol' in/for directive .xref. xref3.cmd(31): .xref _Usymbol xref3.cmd(34): .xref _Usymbol`	Duplicate `in/for` pattern expression for a directive.
`Error(E1015): LCF configuration: invalid expression for alignment field in directive .align, expect power of 2 alignment. align4.cmd(26): .align 19`	Incorrect align expression. Specify an alignment with a power of 2.
`Error(E1016): LCF configuration: undefined section '.libtext' used in expression @secalign(...). rename6.cmd(26): @secalign(".libtext")`	Unable to find the definition of section ".libtext" in the LCF.
`Error(E1017): LCF configuration: undefined segment 'DATA9' used in expression @segsize(...). segment2.cmd(23): _DataStart + @segsize(DATA9) + @segalign(DATA)`	Unable to find the definition of segment "DATA9" in the LCF. Compared with @secxxxx(...) expression, the argument for @segxxxx(...) does not require quotes, while the argument for secxxxx(...) requires quotes. For example: @secalign("sec1"), @segalign(seg1).
`Error(E1018): LCF configuration: too late for directive .xref, because reading input files is already done. xref2.cmd(31): .xref _Usymbol fdw eqf`	Incorrect priority in the LCF. Linker processes the directives with a low priority number before processing the directives with a higher priority number. This error occurs when the linker encounters any inconsistencies in the directive priorities.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Fatal(F1019): System error: fail to run directive .align. align4.cmd(26): .align 19`	Incorrect directive definition. This error generally occurs because of an earlier reported error from the same memory location.
`E1020`	RESERVED.
`Error(E1021): LCF configuration: directive .align cannot precede the first .org directive align1.cmd(18): .align 16`	Directive .align cannot occur before the first .org directive. Check the first occurrence of .align directive in the LCF and make sure that some .org directive precedes it.
`E1022`	RESERVED.
`Error(E1023): LCF configuration: section '.unlikely' is not placed explicitly in linker control file.`	Error placing a section on first fit basis. Use one of the following method to explicitly place a section in the LCF: • place the section in a .segment and then place this segment by the .org directive • place the section in a .concatenate directive and then place this concatenate section in .att_mmu directive with argument “physical _address” set. Alternatively, place the section in .att_mmu directive directly. • place the section in a .group directive with argument “load_address” set. This error occurs only when `-enable-error-placing-section-on-first-fit-basis` is used on command line.
`Fatal(F1024): LCF configuration: missing overlay definition for section 'Pgm3', expect it's defined by .overlay or .att_mmu directive.`	Missing .overlay directive in the LCF. The linker assumes that when .att_mmu directive is present in the LCF and the hardware architecture is equipped with the MMU, all sections receive the overlay attribute during link time. Make sure that the section specified in the error message is defined in an .overlay or .att_mmu directive.
`E1025`	RESERVED.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`E1026`	RESERVED.
`E1027`	RESERVED.
`E1028`	RESERVED.
`Fatal(F1029): LCF configuration: cannot create segment for group 'G2', same name segment exists. group1.cmd(62): .segment G2,".data",".ramsp_0",".default",".bss" group1.cmd(52): .group "G2", _Load_G2,Load_Segment,"Pgm3", "Pgm2","Pgm4"`	Duplicate segment name. The linker creates a segment with the name specified for a .group directive. This error message occurs when the linker finds a segment already defined with .group directive name.
`Error(E1030): LCF configuration: incorrect memory range(0x0000ffff, 0x0000fffe), low_addr >= high_addr. reserve4.cmd(17): .reserve _CodeStart-1, _CodeStart-2,`	Incorrect memory range specified in the directive definition.
`Fatal(F1031): LCF configuration: memory range(0x00010000, 0x00ffffff) overlapped with another memory range. memory11.cmd(15): .memory 0x10000, 0xfffff, "r" memory11.cmd(16): .memory 0x10000, 0xffffff, "rxw"`	Overlapped .memory directives. The .memory directive is used to configure available physical memory block in the system.
`Fatal(F1032): LCF configuration: cannot reserve memory(0x0000fffe, 0x0000ffff), memory range is not covered by any .memory directive. align1.cmd(13): .reserve _CodeStart-2, _CodeStart-1`	Missing .memory directive definition. Check .memory directive definitions in the LCF and make sure that the requested memory region is included.

Table A.3 Linker Error Codes

Error Code/Message	Explanation

`Error(E1033): unit c0: LCF configuration: cannot reserve space shared_m2(0xd0020000, 0xd0900000), memory range is used by other segment/ reserve(s) or not covered by any .memory directive. Space memory map for unit "c0":`
`|0x40000000..0x60000000| free`
`|0xc0000000..0xc0080000| free`
`|0xd0000000..0xd001ffff| free`
`|0xd0020000..0xd0900000| shared_m3`
`|0xd0900001..0xd0a00000| free`
	Conflict in specified memory range. This error indicates that linker cannot allocate memory region for the specified space, because:
	• the memory region is used by another segment or space
	• the memory region is reserved by .reserve directive or reserve clause in .att_mmu directive
	• the memory region is not covered by any .memory directive

`E1034`	RESERVED.
`Error(E1035): LCF configuration: cannot allocate memory(0x00010000, 0x00010349) for segment TEXT(842 bytes, align 16, perms r-x), incompatible permission with .memory directive. memory10.cmd(15): .memory 0x10000, 0xfffff, "r"`	Incorrect access attributes. Verify .memory and .segment directive definitions in the LCF and make sure that the segment is placed in a compatible memory region. In addition: • all the sections placed in a segment must have same access attribute • segment must be placed in a memory region with compatible access attributes
`Error(E1036): LCF configuration: cannot allocate memory(0x00000000, 0x00000fff) segment .intvec(4096 bytes, align 2, perms r-x), too large to fit into remaining memory.`	Unable to allocate the requested memory for the specified segment. Expand the .memory directive so that it covers the memory region specified in the error message.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Error(E1037): LCF configuration: cannot allocate memory(0x00014e20, 0x0001501f) for segment SEG(512 bytes, align 16, perms rwx), the memory range is overlapped with other segment/reserve(s).. Memory map: r-x |0x00000000..0x00000fff| segment INTVEC (4096 bytes) rwx |0x00001000..0x0000112b| segment DATA (300 bytes) rwx |0x0000112c..0x0000fffd| free (61138 bytes) |0x0000fffe..0x0000ffff| reserved (2 bytes) r-x |0x00010000..0x00010349| segment TEXT (842 bytes) rwx |0x0001034a..0x0001034f| free (6 bytes) r-x |0x00010350..0x00015595| segment LIBTEXT (21062 bytes) rwx |0x00015596..0x00027fff| free (76394 bytes) |0x00028000..0x0007eff0| reserved (356337 bytes) rwx |0x0007eff1..0x00ffffff| free (16257039 bytes) You could try to place segment SEG at 0x000155A0.`	Overlapped memory region. The specified memory region is used by another segment or reserve.

Table A.3 Linker Error Codes

Error Code/Message	Explanation

`Error(E1038): LCF configuration: cannot allocate memory(0x00014e20, 0x0001501f) for segment SEG(512 bytes, align 16, perms rwx), the memory range is not defined by any .memory directive.`
`Memory map:`
`r-x |0x00000000..0x00000fff| segment INTVEC (4096 bytes)`
`rwx |0x00001000..0x0000112b| segment DATA (300 bytes)`
`rwx |0x0000112c..0x0000fffd| free (61138 bytes)`
`|0x0000fffe..0x0000ffff| reserved (2 bytes)`
`r-x |0x00010000..0x00010349| segment TEXT (842 bytes)`
`rwx |0x0001034a..0x0001034f| free (6 bytes)`
`r-x |0x00010350..0x00015595| segment LIBTEXT (21062 bytes)`
`rwx |0x00015596..0x00027fff| free (76394 bytes)`
`|0x00028000..0x0007eff0| reserved (356337 bytes)`
`rwx |0x0007eff1..0x00ffffff| free (16257039 bytes)`
`You could try to place segment SEG at 0x000155A0.`
	Unable to allocate the requested memory because no .memory directive defines the specified memory region.

Table A.3 Linker Error Codes

Error Code/Message	Explanation

`Error(E1039): LCF configuration: cannot find available free memory for segment .vfrw_buffers_bss(16777216 bytes, align 16777216, perms rw-).`
`Memory map:`
`|0x00000000..0x3fffffff| reserved (1073741824 bytes)`
`rwx |0x40000000..0x40000020| free (33 bytes)`
`|0x40000021..0xbfffffff| reserved (2147483615 bytes)`
`rwx |0xc0000000..0xc0080000| free (524289 bytes)`
`|0xc0080001..0xcfffffff| reserved (267911167 bytes)`
`rwx |0xd0000000..0xd0a00000| free (10485761 bytes).`
	Unable to find a free memory block to place the specified segment using first fit strategy. Add a new .memory directive or adjust an existing directive to accommodate the specified segment.

`Error(E1040): unit c0: LCF configuration: cannot fit segment os_shared_data_seg(0xc0040000, 0xc00400ff) into space shared_ddr(0x40004000, 0x60000000).`
`unit c0: os_msc814x_link.lcf(84): .segment .os_shared_data_seg, ".os_shared_data" ;Non cacheable`
	Unable to place the segment in specified space memory region because the segment is already placed using fixed or first-fit method in another memory region which is not covered by the space memory region.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Error(E1041): LCF configuration: inconsistent flag for section components in output section .text. d:\cwsc\StarCore_Support\compiler\lib\sc140\rtlib_le.elb(exit.eln)(.data): flag= SEC_ALLOC SEC_WRITE SEC_BITS d:\cwsc\StarCore_Support\compiler\lib\sc140\startup_le.eln (.text): flag= SEC_ALLOC SEC_EXEC SEC_BITS`	Inconsistent attributes composed into an output section. The linker expects all the section components of an output section to have consistent attributes, such as loading_space, running_space, flags, etc. This error may also occur because of wrong .rename definitions: `.rename "*.elb(*)", ".data", ".text"`
`E1042`	RESERVED.
`E1043`	RESERVED.
`E1044`	RESERVED.
`Fatal(F1045): LCF configuration: cannot link section '.data', section already linked to segment 'DATA' segment7.cmd(24): .segment DATA1,".data",".ramsp_0",".bss",".default"`	Duplicate section placement. Verify the section placement in the LCF.
`Fatal(F1046): LCF configuration: multiple link section '.text' by directives: segment5.cmd(32): .concatenate cc1, ".text" segment5.cmd(24): .segment TEXT, ".text"`	Duplicate section placement. When you place a section using .concatenate directive, you cannot place the same section using .segment directive again.
`E1047`	RESERVED.

Table A.3 Linker Error Codes

Error Code/Message	Explanation
`Fatal(F1048): LCF configuration: cannot link section 'G2' explicitly, because it's a group section. group1.cmd(61): .segment .data,".data",".ramsp_0",".de fault",".bss", "G2"`	Incorrect section specification. The group section can only be linked with .group and .overlay directive. In addition, once linked with .group and .overlay directive, the group section cannot be used in .segment or .concatenate section again.
`E1049`	RESERVED.
`Fatal(F1050): unit c0: LCF configuration: cannot include concatenated section '.os_kernel_text' into another .concatenate directive. unit c0: os_msc814x_link.lcf(43): .concatenate ".os_kernel_text", ".osvecb", ".text", ".oskernel_text_run_time", ".oskernel_text_run_time_crit ical", ".private_load_text",".defaul t", ".intvec" unit c0: os_msc814x_link.lcf(44): .concatenate "CC", ".data", ".os_kernel_text"`	Incorrect directive specification. The concatenated sections cannot be included in another .concatenate directive.
`Fatal(F1051): unit c1: LCF configuration: undefined unit 'bad_unit' in linker command file. unit c1: os_msc814x_link.lcf(119): .import "bad_unit`shared_m3"`	Undefined unit used as section or space name prefix in the LCF.
`Error(E1052): Symbol resolution: undefined symbol '_Usymbol' used in .xref directive. xref3.cmd(31): .xref _Usymbol`	Undefined symbol used in the .xref directive.

Table A.3 Linker Error Codes

Error Code/Message	Explanation

`Error(E1053): unit c0: Symbol resolution: found inconsistent address for symbol '_g_os_pram_memory_size' which is defined in section '.os_shared_data'.`
`Value 0xC0040010 from unit c0 in src/os_msc814x_debug_noassert.elb (os_shared_data.eln)`
`Value 0x40000010 from unit c3 in src/os_msc814x_debug_noassert.elb (os_shared_data.eln)`
`Value 0x40000010 from unit c2 in src/os_msc814x_debug_noassert.elb (os_shared_data.eln)`
`Value 0x40000010 from unit c1 in src/os_msc814x_debug_noassert.elb (os_shared_data.eln)`
	Inconsistent addresses for the symbols with same name on different cores. Do one of the following:
	• move the section containing the symbol definition into a shared space.
	• if the section containing the symbol definition is required in a private space, make sure that this section is placed on the same virtual address on all the cores.

`E1054`	RESERVED.
`Error(E1055): Symbol resolution: unresolved reference to symbol 'dsadas'.`	Undefined symbol ‘dsadas’ encountered at the linking time. Use the compiler option ‘-v’ to show full line option, all the input modules and the libraries for the linker.
`E1056`	RESERVED.

`Fatal(F1057): Symbol resolution: redefinition of symbol '_main'.`
`set8.cmd(32): .set _main, @secalign(".libtext") hello.eln(.text)`
	Duplicate definitions for a symbol name. Use one of the following methods to define a symbol:
	• define the symbol in a source file and assemble it to a section in the module file.
	• define the symbol using the linker command line option: -Dsymbol=value
	• define the symbol using .set or .provide directive in the LCF

Table A.4 Linker Directives Priority Table

Directive Name	Priority
`.unit`	0
`.include`	0
`.assert`	0
`.entry`	0
`.force`	0
`.memory`	0
`.virtual_memory`	0
`.non_ovl`	0
`.provide`	0
`.rename`	0
`.library_entry_points`	0
`.library_undefined_symbols`	0
`.library_prefix`	0
`.library_public_symbols`	0
`.inhibit_folding_symbols`	0
`.inhibit_folding_modules`	0
`.reserve`	0
`.set`	0
`.xref_module`	0
`.xref`	0
`.concatenate`	0
`.define_overlay`	0
`.att_mmu_settings`	0
`.define_single_mapped_virtual_addressing`	0

Table A.4 Linker Directives Priority Table

Directive Name	Priority
`.inhibit_compression`	0
`.define_compression`	0
`.library_concatenate_section`	0
`.place_symbols`	0
`.space`	1
`.export`	2
`.import`	3
`.group`	10
`.align`	10
`.bss`	10
`.firstfit`	10
`.org`	10
`.overlay`	10
`.union`	10
`.puncture`	10
`.segment`	10
`.att_mmu`	10
`.exclude`	10
`.init_table_section`	10
`.define_region_to_map_virtual_addressing`	10
`.group_firstfit_start`	10
`.group_firstfit_end`	10

B

Complex Examples

This appendix provides two complex examples of multi-mapped virtual addressing:

- B.1 Multi-Core Environment Example
- B.2 .att_mmu_settings Example

B.1 Multi-Core Environment Example

One of the last examples in Chapter 4 was using the `.concatenate` directive to reduce the size and number of MMU descriptors. The appropriate linker directives are:

```
.concatenate"data1",".data1",".rom1",".bss1"
.concatenate"data2",".data2",".rom2",".bss2"
.concatenate"data3",".data3",".rom3",".bss3"
.att_mmu"task1_data",task_id:0x1,0x0,0xfffff,"data1"
.att_mmu"task2_data",task_id:0x1,0x0,0xfffff,"data2"
.att_mmu"task3_data",task_id:0x1,0x0,0xfffff,"data3"
.att_mmu"task1_program",task_id:0x1,0x0,0xfffff,"text1"
.att_mmu"task2_program",task_id:0x1,0x0,0xfffff,"text2"
.att_mmu"task3_program",task_id:0x1,0x0,0xfffff,"text3"
```

A translation of this previous example is appropriate for a SIMD model in a multi-core environment. We split shared memory into two logical parts:

- logical private memory — for data
- logical shared memory — for the program

Follow these configuration steps:

1. Enter the directives to define Core 0:

```
.unit"c0",""
.provide _Core_ID,0
```

2. Enter the directives that establish the number of cores:

```
.provide _Number_of_cores,2
.provide _Private_size,0x10000
.provide _Shared_size,0x20000
.provide _Mem_start,0x0
.provide _Mem_end,_Private_size*_Number_of_cores
   +_Shared_size
```

3. Enter the directives that define the logical private memory:

```
.provide _Private_start,_Mem_start+_Core_ID*_Private_size
.provide _Private_end,_Mem_start+_Core_ID*_Private_size
   +Private_size-1
.memory _Private_start,_Private_end,"rwx"
```

4. Enter the directives that define the logical shared memory:

```
.provide _Shared_start,_Private_size*_Number_of_cores
.memory _Shared_start,_Mem_end,"rwx"
```

5. Enter the directives that define *shared* space:

```
.space "shared",_Private_size*_Number_of_cores,_Mem_end,
   ".seg_task_pgm
.export "shared"
```

6. Group sections `.dataX`, `.romX`, and `.bssX` into section `dataX`, in order to decrease the memory size and number of MMU descriptors. Enter these directives:

```
.concatenate"data1",".data1",".rom1",".bss1"
.concatenate"data2",".data2",".rom2",".bss2"
.concatenate"data3",".data3",".rom3",".bss3"
```

7. Define virtual data memory by putting sections `data1`, `data2`, and `data3` into segment `.seg_task_data`. This segment is private for each core, so each core needs a definition of virtual/physical memory. Enter these directives:

```
.att_mmu"task1_data",task_id:0x1,0x0,0xfffff,"data1"
.att_mmu"task2_data",task_id:0x1,0x0,0xfffff,"data2"
.att_mmu"task3_data",task_id:0x1,0x0,0xfffff,"data3"
```

8. Define virtual program memory by putting sections `.text1`, `.text2`, and `.text3` into the segment `.seg_task_pgm` that this core exports. The core that imports *shared* space from Core 0 does not need to redefine virtual/physical memory for the program, because the `.import` directive automatically propagates this information. Enter these directives:

```
.att_mmu"task1_program",task_id:0x1,0x0,0xfffff,"text1"
.att_mmu"task2_program",task_id:0x1,0x0,0xfffff,"text2"
.att_mmu"task3_program",task_id:0x1,0x0,0xfffff,"text3"
```

9. Enter the directives that configure physical memory and put data into logical private memory:

```
.org _Private_start
.segment".seg_task_data","data1","data2","data3"
```

10. Enter the directives to put the program into logical shared memory:

```
.org _Shared_start
.segment".seg_task_pgm","text1","text2","text3"
```

11. Enter the directives to define Core 1:

```
.unit”c1”
.provide _Core_ID,1
```

12. Enter the directives that establish the number of cores:

```
.provide _Number_of_cores,2
.provide _Private_size,0x10000
.provide _Shared_size,0x20000
.provide _Mem_start,0x0
.provide _Mem_end,_Private_size*_Number_of_cores
   +_Shared_size
```

13. Enter the directives that define the logical private memory:

```
.provide _Private_start,_Mem_start+_Core_ID*_Private_size
.provide _Private_end,_Mem_start+_Core_ID*_Private_size
   +Private_size-1
.memory _Private_start,_Private_end,”rwx”
```

14. Enter the directives that define the logical shared memory:

```
.provide _Shared_start,_Private_size*_Number_of_cores
.memory _Shared_start,_Mem_end,”rwx”
```

15. Enter the `.import` directive, which automatically propagates *shared*-space information from Core 0:

```
.import “c0‘shared”
```

16. Group sections `.dataX`, `.romX`, and `.bssX` into section `dataX`, in order to decrease the memory size and number of MMU descriptors. Enter these directives:

```
.concatenate”data1”,”.data1”,”.rom1”,”.bss1”
.concatenate”data2”,”.data2”,”.rom2”,”.bss2”
.concatenate”data3”,”.data3”,”.rom3”,”.bss3”
```

17. Define virtual data memory by putting sections `data1`, `data2`, and `data3` into segment `.seg_task_data`. This segment is private for each core, so each core needs a definition of virtual/ physical memory. Enter these directives:

```
.att_mmu”task1_data”,task_id:0x1,0x0,0xfffff,”data1”
.att_mmu”task2_data”,task_id:0x1,0x0,0xfffff,”data2”
.att_mmu”task3_data”,task_id:0x1,0x0,0xfffff,”data3”
```

18. Enter the directives that configure physical memory and put data into logical private memory:

```
.org _Private_start
.segment”.seg_task_data”,”data1”,”data2”,”data3”
```

This completes the example.

B.2 .att_mmu_settings Example

This example involves five tasks to be executed in virtual memory space:

- Task 1 — a system task, consisting of sections `.text`, `.data`, `.share_data`, `.rom`, and `.bss`.
- Task 2 — a user task, consisting of sections `.sw1_text`, `.sw1_data`, `.sw1_rom`, `.sw1_bss`, `.sw12_text`, `.sw12_data`, `.sw14_text`, and `.sw14_data`.
- Task 3 — a user task, consisting of sections `.sw2_text`, `.sw2_data`, `.sw2_rom`, `.sw2_bss`, `.sw12_text`, and `.sw12_data`.
- Task 4 — a user task, consisting of sections `.sw3_text`, `.sw3_data`, `.sw3_rom`, and `.sw3_bss`.
- Task 5 — a user task, consisting of sections `.sw4_text`, `.sw4_data`, `.sw4_rom`, `.sw4_bss`, `.sw14_text`, and `.sw14_data`.

Notice that sections `.sw12_text` and `.sw12_data` are common to Tasks 2 and 3. Similarly, sections `.sw14_text` and `.sw14_data` are common to Tasks 2 and 5. This means that these sections *must not* overlap in virtual memory unless code enables a priority mechanism.

The directives of Listing B.1, below, provide the appropriate configuration, which involves these considerations:

1. The minimum size of the descriptor section is 256 bytes. If the original section size is smaller, the linker expands the size to 256 bytes.
2. The maximum size of the descriptor section is 0x10000 bytes. If section size is greater, the linker generates an error message.)
3. All descriptors that have attributes `MMU_PROG_DEF_SYSTEM` or `MMU_DATA_DEF_SYSTEM` cannot overlap in virtual memory unless the priority mechanism is enabled.
   a. Task 4 enables the priority mechanism, because `.data_sw3` includes attribute `MMU_DATA_DEF_SYSTEM` and `.sw3_text` includes attribute `MMU_PROG_DEF_SYSTEM`. This means that:
      - `.data_sw3` can overlap the `.sw12_data` and `.sw14_data` spaces (`.sw12_data` and `.sw14_data` do not overlap in virtual memory).
      - `.sw3_text` can overlap the `.sw12_text` and `.sw14_text` spaces (`.sw12_text` and `.sw14_text` do not overlap in virtual memory).

   b. Task 5 does not enable the priority mechanism, so:
      - `.data_sw4` *cannot* overlap the `.sw12_data` space.
      - `.sw4_text` *cannot* overlap the `.sw12_text` space

Listing B.1 .att_mmu_setting Configuration Example

```
.provide MMU_PROG_GLOBAL_PROGRAM_1,     0x00100000
.provide MMU_PROG_GLOBAL_PROGRAM_0,     0x0008000
```

```
.provide MMU_PROG_PREFETCH_ENABLE,      0x00040000
.provide MMU_PROG_BURST_SIZE_8,         0x00030000
.provide MMU_PROG_BURST_SIZE_4,         0x00020000
.provide MMU_PROG_BURST_SIZE_2,         0x00010000
.provide MMU_PROG_BURST_SIZE_1,         0x00000000

.provide MMU_PROG_DEF_SHARED,           0x00000010
.provide MMU_PROG_DEF_CACHEABLE,        0x00000008
.provide MMU_PROG_DEF_XPERM_USER,       0x00000004
.provide MMU_PROG_DEF_XPERM_SUPER,      0x00000008
.provide MMU_PROG_DEF_SYSTEM,           MMU_PROG_DEF_SHARED
.provide _MMU_PROG_DEF_SYSTEM,          MMU_PROG_DEF_SHARED

.provide MMU_DATA_NONCACHEABLE_WRITE_THROUGH_STALL,   0x00C00000
.provide MMU_DATA_NONCACHEABLE_WRITE_THROUGH,         0x00800000
.provide MMU_DATA_CACHEABLE_WRITE_BACK,               0x00400000
.provide MMU_DATA_CACHEABLE_WRITE_THROUGH,            0x00000000
.provide MMU_DATA_GLOBAL_DATA_2,                      0x00200000
.provide MMU_DATA_GLOBAL_DATA_1,                      0x00100000
.provide MMU_DATA_GLOBAL_DATA_0,                      0x00080000
.provide MMU_DATA_PREFETCH_ENABLE,                    0x00040000
.provide MMU_DATA_BURST_SIZE_8,                       0x00030000
.provide MMU_DATA_BURST_SIZE_4,                       0x00020000
.provide MMU_DATA_BURST_SIZE_2,                       0x00010000
.provide MMU_DATA_BURST_SIZE_1,                       0x00000000

.provide MMU_DATA_DEF_MIXED_ENDIAN_MEMORY_REGION,     0x00000040
.provide MMU_DATA_DEF_SHARED,                         0x00000020
.provide MMU_DATA_DEF_WPERM_USER,                     0x00000010
.provide MMU_DATA_DEF_RPERM_USER,                     0x00000008
.provide MMU_DATA_DEF_WPERM_SUPER,                    0x00000002
.provide MMU_DATA_DEF_RPERM_SUPER,                    0x00000004

.provide MMU_DATA_DEF_SYSTEM,                    MMU_DATA_DEF_SHARED
.provide _MMU_DATA_DEF_SYSTEM,                   MMU_DATA_DEF_SHARED

.provide USER_DATA_MMU_DEF,    MMU_DATA_CACHEABLE_WRITE_THROUGH|\
                                  MMU_DATA_PREFETCH_ENABLE|\
                                  MMU_DATA_DEF_WPERM_USER|\
                                  MMU_DATA_DEF_RPERM_USER|\
                                  MMU_DATA_DEF_WPERM_SUPER|\
                                  MMU_DATA_DEF_RPERM_SUPER|\
                                   MMU_DATA_BURST_SIZE_4

.provide SHARED_DATA_MMU_DEF,   MMU_DATA_NONCACHEABLE_WRITE_THROUGH|\
                                  MMU_DATA_PREFETCH_ENABLE|\
                                  MMU_DATA_DEF_SHARED|\
                                  MMU_DATA_DEF_WPERM_USER|\
                                  MMU_DATA_DEF_RPERM_USER|\
                                  MMU_DATA_DEF_WPERM_SUPER|\
                                  MMU_DATA_DEF_RPERM_SUPER|\
                                   MMU_DATA_BURST_SIZE_4

.provide SYSTEM_DATA_MMU_DEF, MMU_DATA_CACHEABLE_WRITE_THROUGH|\
                                  MMU_DATA_PREFETCH_ENABLE|\
                                  MMU_DATA_DEF_SHARED|\
                                  MMU_DATA_DEF_WPERM_USER|\
                                  MMU_DATA_DEF_RPERM_USER|\
                                  MMU_DATA_DEF_WPERM_SUPER|\
                                  MMU_DATA_DEF_RPERM_SUPER|\
                                   MMU_DATA_BURST_SIZE_4

.provide USER_PROG_MMU_DEF,     MMU_PROG_DEF_CACHEABLE|\
                                  MMU_PROG_PREFETCH_ENABLE|\
                                  MMU_PROG_DEF_XPERM_USER|\
                                  MMU_PROG_DEF_XPERM_SUPER|\
                                   MMU_PROG_BURST_SIZE_4

.provide SHARED_PROG_MMU_DEF,   MMU_PROG_DEF_CACHEABLE|\
                                  MMU_PROG_PREFETCH_ENABLE|\
                                  MMU_PROG_DEF_SHARED|\
                                  MMU_PROG_DEF_XPERM_USER|\
                                  MMU_PROG_DEF_XPERM_SUPER|\
                                   MMU_PROG_BURST_SIZE_4
.provide SYSTEM_PROG_MMU_DEF,    SHARED_PROG_MMU_DEF
.provide MMU_HIGH_PRIORITY,      0x10000000

.set PRIVATE_Mx_start, 0xC0000000
.set SHARED_Mx_start,  0xC0040000
.set VIRTUAL_SYSTEM__DATA_start, 0x0
.set VIRTUAL_SYSTEM__DATA_size,  0x8000
.set VIRTUAL_SYSTEM__DATA_end, VIRTUAL_SYSTEM__DATA_start
    + VIRTUAL__SYSTEM_DATA_size -1
.set VIRTUAL_SHARED_DATA_start, 0x40000
.set VIRTUAL_SHARED_DATA_size,  0x10000
.set VIRTUAL_SHARED_DATA_end, VIRTUAL_SHARED_DATA_start
    + VIRTUAL_SHARED_DATA_size -1

.set PERIPHERAL_start, 0xFFF10000
.set PERIPHERAL_size, 0x000F0000
.set PERIPHERAL_end, PERIPHERAL_start + PERIPHERAL_size -1

.set VIRTUAL_SHARED_PROGRAM_start, 0x40000
.set VIRTUAL_SHARED_PROGRAM_size,  0x10000
.set VIRTUAL_SHARED_PROGRAM_end, VIRTUAL_SHARED_PROGRAM_start
    + VIRTUAL_SHARED_PROGRAM_size -1
```

```
.set VIRTUAL_DATA_USER_start, VIRTUAL_SYSTEM_DATA_end
.set VIRTUAL_DATA_USER_size,  0x8000
.set VIRTUAL_DATA_USER_end, VIRTUAL_DATA_USER_start
    + VIRTUAL_DATA_USER_size -1

.set VIRTUAL_PROGRAM_start, 0x0
.set VIRTUAL_PROGRAM_size,  0x10000
.set VIRTUAL_PROGRAM_end, VIRTUAL_PROGRAM_start
    + VIRTUAL_PROGRAM_size -1

.set _task_two_id, 0x2
.set _task_three_id, 0x3
.set _task_four_id, 0x4
.set _task_five_id, 0x5

.att_mmu_settings min_descr_size: 256, max_descr_size: 0x10000, \
      system_task: 0, \
      max_data_descr_count: 20, \
      max_program_descr_count: 12, \
      can_not_overlap: MMU_DATA_DEF_SYSTEM, \
      can_not_overlap: MMU_PROG_DEF_SYSTEM, \
      force_overlap: MMU_HIGH_PRIORITY


; System Task
.concatenate "data_local",".data",".bss"
.att_mmu "data_task_one_mmu", \
      VIRTUAL_SYSTEM__DATA_start, VIRTUAL_SYSTEM_DATA_end \
      "data_local", \
            attribute: SYSTEM_DATA_MMU_DEF, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu "shared_data_task_one_mmu", \
      VIRTUAL_SHARED_DATA_start, VIRTUAL_SHARED_DATA_end \
      ".rom", \
            attribute: SYSTEM_DATA_MMU_DEF, \
            after_physical_address: SHARED_Mx_start
      ".share_data", \
            attribute: MMU_SHARE_DATA, \
            after_physical_address: SHARED_Mx_start

.att_mmu "peripheral_task_one_mmu", \
      PERIPHERAL_start, PERIPHERAL_end \
     _RESERVED_, \
            size: PERIPHERAL_size, \
            region_type: "data", \
            attribute: SYSTEM_DATA_MMU_DEF, \
            base_address: PERIPHERAL_start, \
            physical_address: PERIPHERAL_start

.att_mmu “program_task_one_mmu”, \
      VIRTUAL_SHARED_PROGRAM_start, VIRTUAL_SHARED_PROGRAM_end \
      “.text”, \
            attribute: MMU_SYSTEM_PROGRAM, \
            base_address: SHARED_Mx_start, \
            physical_address: SHARED_Mx_start


; Task two - user task
.concatenate “data_sw2”,”..sw2_data”,”.sw2_rom”,.sw2_bss”
.att_mmu “data_task_two_mmu”, \
      task_id: _task_two_id,
      VIRTUAL_DATA_USER_start, VIRTUAL_DATA_USER_end \
      “.data_sw2”, \
            attribute: USER_DATA_MMU_DEF, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu “program_task_two_mmu”, \
      task_id: _task_two_id, \
      VIRTUAL_PROGRAM_start, VIRTUAL_PROGRAM_end \
      “.sw2_text”, \
            attribute: USER_PROG_MMU_DEF, \
            after_physical_address: SHARED_Mx_start


; Task three - user task
.concatenate “data_sw3”,”..sw3_data”,”.sw3_rom”,.sw3_bss”
.att_mmu “data_task_three_mmu”, \
      task_id: _task_three_id,
      VIRTUAL_DATA_USER_start, VIRTUAL_DATA_USER_end \
      “.data_sw3”, \
            attribute: USER_DATA_MMU_DEF, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu “data_task_two_three_mmu”, \
      task_id: _task_three_id, task_id: _task_two_id, \
      VIRTUAL_DATA_USER_start, VIRTUAL_DATA_USER_end, \
      “.sw23_data”, \
            attribute: USER_DATA_MMU_DEF| MMU_DATA_DEF_SYSTEM, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu “program_task_three_mmu”, \
      task_id: _task_three_id, \
      VIRTUAL_PROGRAM_start, VIRTUAL_PROGRAM_end \
      “.sw3_text”, \
            attribute: USER_PROG_MMU_DEF, \
            after_physical_address: SHARED_Mx_start

.att_mmu “program_task_two_three_mmu”,\
      task_id: _task_three_id, task_id: _task_two_id, \
      VIRTUAL_PROGRAM_start, VIRTUAL_PROGRAM_end,\
      “.sw23_text”, \
            attribute: USER_PROG_MMU_DEF| MMU_PROG_DEF_SYSTEM, \
            after_physical_address: SHARED_Mx_start


; Task four - user task
.concatenate “data_sw4”,”..sw4_data”,”.sw4_rom”,.sw4_bss”
.att_mmu “data_task_four_mmu”, \
      task_id: _task_four_id,
      VIRTUAL_DATA_USER_start, VIRTUAL_DATA_USER_end \
      “.data_sw4”, \
            attribute: USER_DATA_MMU_DEF| MMU_HIGH_PRIORITY, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu “program_task_four_mmu”, \
      task_id: _task_four_id, \
      VIRTUAL_PROGRAM_start, VIRTUAL_PROGRAM_end \
      “.sw4_text”, \
            attribute: USER_PROG_MMU_DEF| MMU_HIGH_PRIORITY, \
            after_physical_address: SHARED_Mx_start


; Task five - user task
.concatenate “data_sw5”,”..sw5_data”,”.sw5_rom”,.sw5_bss”
.att_mmu “data_task_five_mmu”, \
      task_id: _task_five_id,
      VIRTUAL_DATA_USER_start, VIRTUAL_DATA_USER_end \
      “.data_sw5”, \
            attribute: USER_DATA_MMU_DEF| MMU_HIGH_PRIORITY, \
            after_physical_address: PRIVATE_Mx_start

.att_mmu “program_task_five_mmu”, \
      task_id: _task_five_id, \
      VIRTUAL_PROGRAM_start, VIRTUAL_PROGRAM_end \
      “.sw5_text”, \
            attribute: USER_PROG_MMU_DEF| MMU_HIGH_PRIORITY, \
            after_physical_address: SHARED_Mx_start
```

Index