AN3788
4/2010

Overlay Support in StarCore Compiler

This document is intended to present the StarCore Compiler support for overlays. It also includes some examples of tool configurations.

Note: This document refers to sc100-ld StarCore linker. The configuration required for sc3000-ld StarCore linker is not explained in this document.

Note: This document refers to non-MMU based overlays.

CONTENTS

freescale™ semiconductor

1 Reference Documents

- Assembler Manual, available as a PDF file in CodeWarrior
- SC100 Linker User Guide, available as PDF file in CodeWarrior
- Configuration_mgt.txt, available as release note in CodeWarrior

2 Overview

The StarCore Compiler provides support for both code/data overlay and pure data overlay. Code/data overlay is meant to represent a mechanism that adds support for a number of sections to be loaded in memory at different addresses, but run from the same address. Pure data overlay differs from code/data overlay, the pure data section content is undefined at startup (pure data overlay sections can hold only uninitialized data).

In order to use the overlay support from the compiler one should be familiar with application files and linker command files. If assembler support is needed, the sectype directive should also be used.

It is users responsibility to write an overlay manager which copies an overlay section from its load to its run address. Such an overlay manager (both in C and assembly) is presented in the Examples chapter.

3 Overlay Sections

An overlay section is similar to a “progbits” section except that it has two starting addresses instead of one:

- Load address
- Run address

The run address is the address at which the section will begin when its code is being executed. References to symbols in an overlay section refer to the run address.

The load address is the address at which the section is linked. It is the responsibility of an overlay manager to copy the section from its load address to its run address. All references to symbols in an overlay section refer to the run address. To refer to a global symbol's load address, prefix the name with "LoadAddr_".

For each overlay section the linker creates a “LoadAddr_<sec_name>” global symbol which may be used by the overlay manager to copy the section from its load to its run address.

4 Tool Support

4.1 Tool Command Line Options

SCC accepts the following command line options:

- for the linker command file: -mem <linker_command_file>
- for the application file: -ma <application_file>

The linker accepts –c <linker_command_file>.

CodeWarrior IDE Options

In the CodeWarrior IDE the application file may be specified in Project Settings-> StarCore Compiler-> PassThrough-> Use Application Configuration File and the linker command file in Project Settings-> Linker-> Enterprise Linker-> Additional by adding -mem <linker_command_file>.

4.3 Application Configuration Files

The application configuration file contains information about the interaction between the application software and the hardware. This file indicates to the compiler how to compile specific software units in order to ensure efficient sharing of hardware resources in particular memory space. This information can be modified to suit the requirements of your application. An application configuration file can be specified using `-ma` command line option.

4.3.1 Syntax

The application configuration file syntax is:

```
view <View_name>
        <View_Body>
end view
use view <view_name>
```

where:

```
<View_Body>:
        <Section_definitions>
        <Section_settings>
        <Variable settings>
        <Control_option_settings>
        <Calling_convention_setting>
        <List Of Modules> ; A module name is the file without the extension

<Module>:
        module <module_name> [
                <Variable settings>
                <Control_option_settings>
                <Calling_convention_setting>
                <List of functions>
        ]
<Function>:
        function <function_name>
                <Control_option_settings>
                <Calling_convention_setting>
        ]
```

Section definitions:

It is a way to bind physical segments (used by the linker) to logical names in order to be able to easily redefine a mapping. This information does apply to the whole view.

```
<Section_definitions>:
        section
                <Section_list>
        end section
<Section>:
```

```
        <Section_type> = [ <Binding_List> ]
<Section_type>:
        data
        program
        bss
        rom
        init
<Binding>:
        <Logical_Name> : <Physical_Name> <Optionnal_Qualifier>
<Qualifier>:
        overlay
```

Example:

```
        ....
section
        data = [
                data1: "My_Data_Seg1",
                data2: "My_Data_Seg2" overlay,
                data3: "My_Data_Seg3" overlay
                ]
        program = [Pgm1: "My_Pgm_Seg1"]
end section
        ....
```

Section settings:

Once the section is defined it is possible to specify a setting in each context. If nothing is specified then the settings are inherited from the hierarchy. For example, if a setting is defined in a module, all the functions in this module will inherit from this setting unless locally overridden.

```
<Section_settings>:
        <Section_type> = <Logical_Name>

<Section_type>:
data
program
bss
rom
init

<Logical_Name> must be defined in Section_Definitions.
```

Example:

```
module "My_Module"
        ...
        data = data2            /* Use data2 as a data space for the whole module */
        program = Pgm1          /* Use Pgm1 as a program space for the whole module */
        ...
        function _foo
                ...
                data = data3 /* Use data3 as a data space for function _foo, overrides data = data2 */
                ...
        ]
]
```

Variable settings:

way to specify additional information on user variables:

- Alignment:

  Force it: the variable is simply forced to the provided alignment value
  Inform: Let the compiler know something about a pointer alignment

- Segment information:

  Overrides the data segment in scope for a list of variable.

```
<Variable_settings>:
        <Alignment_List>
        <Placement_List>

<Alignment>:
        align <Var_Name> <Value>
        |
        align * <Var_Name> <Value>

<Placement>:
        place ( <Var_Name_List> ) in <Logical_Name>
```

Example:

```
        view My_View
        section
                data = [data1: "My_Data_Seg1", data2: "My_Data_Seg2" overlay, data3: "My_Data_Seg3"
overlay]
                program = [Pgm1: "My_Pgm_Seg1"]
                end section

                place ( _A, _B, _C) in data2
                        /*Globals _A, _B and _C will be allocated in segment data2 (e.g My_Data_Seg2)*/
                place ( _X, _Y) in data3
                        /* Globals _X, _Y will be allocated in segment data3 (e.g My_Data_Seg3) */

                module "file1" [
                        data = data2
                        opt_level = speed
                        program = Pgm1
                        place (_AB, _CD) in data1
                        /* Global or static _AB, _CD will be allocated in segment data1 (e.g
My_Data_Seg1) */
        function _foo [
                        align _X 2 /* Local _X will be forced to 2 byte boundary */

                        align *_Y 4 /* Let the compiler know that Y is pointing to a 4 byte aligned
location */

                ]
        ]
        end view
```

Note: The rest of the application configuration file settings are irrelevant to overlay support and will not be detailed.

Overlay Manager Support

The StarCore compiler provides two header files which may be used for overlay support — prototype.h and overlay.h. The prototype.h header provides access to the Ovl_Load_Address function which adds support for the load address of the overlay sections.

The following piece of code provide access to the load address of an overlay section:

```
extern void *Pgm1;
{
_overlay_manager(Ovl_Load_Address(&Pgm1));
}
```

The overlay.h header defines the ovltab structure.

4.5 Assembler Support

A regular section is defined with the SECTION directive and possibly modified by the SECFLAGS and SECTYPE directives. An overlay section is defined by using the OVERLAY operand of the SECTYPE directive as shown in the below example:

```
section .ovl_foo local
secflags nowrite,alloc,execinstr
sectype overlay
```

An overlay section has two starting addresses — a load address and a run address:

- The run address is the address at which the section begins when its code is executed.
- The load address is the address at which the section is linked.

All references to symbols in an overlay section refer to the run address. To refer to a global symbol's load address, prefix the name with LoadAddr_, as shown in the below example. Local symbols cannot be referenced this way.

```
        section .text local
        global _main
_main:
        push r0
        move.l #LoadAddr__foo,r0
        jsr __overlay_manager
        . . .
        pop r0
        jsr _foo
        rts
        section .ovl_text local
        secflags alloc,execinstr,nowrite
        sectype overlay
        global _foo
_foo:
        . . .
        rts
```

This code first loads the overlay section ".ovl_text" to its run address by calling the overlay manager. The overlay manager needs to know the load address of the section since that is unique (many overlays could run at the same address). After the overlay manager finishes, it should be safe to call code in the overlay section.

Note: The linker defines the "LoadAddr_<section_name>" symbols at the start of the overlay sections with the section load address as their values.

sembler also provides support for pure data overlays using the sectype union directive. The resulting section content is undefined, therefore it can hold only uninitialized data.

4.6 Linker Command Files

4.6.1 .overlay directive

Overlay sections must be linked normally (using .segment directives) and also grouped with the .overlay directive. The .segment directive determines where an overlay will be loaded. The new .overlay directive determines which overlays will share a run address. The syntax is:

```
.overlay "SECTION-NAME", "FLAGS", "PATTERN" [, "PATTERN" ...]
```

The directive specifies that a section with specified FLAGS (r, w, and x — read, write and execute respectively) should be created large enough to contain any of the overlay sections listed in the PATTERNs. PATTERNs may include wildcards (*, ?, and [ ) for specifying an arbitrary character sequence. Each pattern must be enclosed in double quotes (").

Example:

```
.overlay ".ovlfoo", "rwx", ".ovl_foo1", ".ovl_foo2", ".ovl_foo3"
.overlay ".ovlbar", "rwx", ".ovl_bar1", ".ovl_bar2"
.org 0x200
.segment TEXT, ".text"
.segment OVLFOO, ".ovlfoo"
.segment OVLBAR, ".ovlbar"
.org 0x10000
.segment RODATA, ".rodata"
.segment OVERLAYS, ".ovl_*"
```

This example implies that only one of .ovl_foo1, .ovl_foo2, or .ovl_foo3 is runnable at any given time; and only one of .ovl_bar1 or .ovl_bar2 is runnable at any given time.

An asterisk may be appended to one of the overlay sections specified in the .overlay directive. In that case this section will be loaded by the linker not only at its load address but also at its run address.

If a default section is specified in the .overlay directive the resulting section will have progbits type, otherwise it becomes bss.

Example:

```
.overlay ".My_Overlay", "rx", ".My_Pgm1", ".My_Pgm2"*, ".My_Pgm3"
```

4.6.2 .group directive

```
.group "grp_name"{,{load_address,} segment_type}, "section_grp_pattern"
[,"section_grp_pattern"...]
```

The .group directive defines a logical name for a group of sections, and a partial order of the sections that match the section_group_pattern, by their runtime addresses. The section_group_pattern is a pattern of sections which represents groups of sections, and/or overlay sections.

This directive used in conjunction with the .overlay directive creates a hierarchy of overlay sections.

egment type field is specified, the linker generates a segment with the type segment_type containing all the group sections.

The most usual segment types are:

1. Load segment
2. Dynamic segment

When the segment type is specified a load address may also be mentioned. In this case the linker assures that the segment is placed at the desired address else it will be placed according to the first fit basis.

The loader doesn't load dynamic segments, as it does for load segments, instead another mechanism should be used (like a DMA channel). Note that the .group directive is the only way to create dynamic overlay segments. See also the .virtual_memory directive.

Example:

```
.overlay "Overlay_1", "rwx", "Pgm_7", "Pgm_8", "Pgm_9"
.group "Group1", "Pgm_1", "Pgm_2"
.group "Group2", "Pgm_3", "Pgm_4", "Pgm_5"
.group "Group3", "Overlay_1", "Pgm_6"
.overlay ".My_Overlay", "rwx", "Group_1", "Group_2", "Group_3"
.org 0x10000
.segment OVER, ".My_Overlay"
```

Memory map:

```
0x10000: [Pgm_7 Pgm_8 Pgm_9] [Pgm_1] [Pgm_3]
. . .    [Pgm_6]             [Pgm_2] [Pgm_4]
                                     [Pgm_5]
```

The same section may appear in more than one .group directives:

```
.group "Group1", "Pgm_1", "Pgm_2"
.group "Group2", "Pgm_3", "Pgm_4", "Pgm_2"
.group "Group3", "Pgm_1", "Pgm_4"

.overlay ".My_Overlay", "rwx", "Group1", "Group2", "Group3"

.org 0x10000
.segment OVER, ".My_Overlay"
```

Memory map:

```
0x10000: [Pgm_1] [Pgm_3] [Pgm_1]
         Unused  [Pgm_4] [Pgm_4]
         [Pgm_2] [Pgm_2]
```

4.6.3 .virtual_memory

```
.virtual_memory lo_addr, hi_addr [, "flags"]
```

The .virtual_memory directive defines a region in memory that is available for dynamic sections (sections that will not be loaded). The lo_addr and hi_addr arguments are 32-bit expressions that set the region's low and high addresses respectively. The optional flags argument is a string containing any combination of r, w, or x (read, write and execute respectively).

irective used in conjunction with the .group directive is the only method of creating dynamic segments.

4.6.4 .union

The .union directive determines which pure data overlay sections will share a run address.

```
.union <section-name>, "flags", "section-pattern" [, "section-pattern" …]
```

The linker combines all sections matching the section_pattern argument(s) into a BSS section named section_name with the specified flags (r, w, or x). Sections are added in the order specified by the pattern arguments. Patterns may include wildcards (*, ?, and [ ) for specifying an arbitrary character sequence. Each pattern must be enclosed in double quotes ("). Multiple patterns must be separated by commas. All the sections in section patterns must be defined as bss.

The linker assures that the resulting union section is large enough to hold any of the sections listed in the specified patterns.

The resulting section is linked normally using the .segment directive which determines the run address of the union section. If the union section is not mentioned in a .segment directive, the section will be linked on a first-fit basis after the linker command file is processed.

This directive may be combined with .group directive.

Note that .union and .overlay directives cannot be interchanged because:

- The content of the section resulting from .union directive is undefined, therefore it can hold only uninitialized data.
- The result of the .union directive is a section large enough to accommodate all the sections specified in section-patterns. Since these sections have SHT_STARCORE_UNION type, only one section is emitted in the linked file (the result of the .union directive) and its run address is equal to its load address. The SHT_STARCORE_OVERLAY sections used in .overlay directive are emitted in the linked file and also the result of the .overlay directive is placed in the .eld file.
- .union does not accept a default configuration.

4.7 Ovltab Section

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

- __overlay_table is an array of type Elf32_Ovl that contains an entry for each overlay section.
- __overlay_count is an unsigned 32-bit integer that represents the number of entries in __overlay_table.

```
typedef struct{
Elf32_Addr ovl_run;          /*overlay run address*/
Elf32_Addr ovl_load;         /*overlay load address*/
Elf32_Word ovl_size;         /*size in bytes of the overlay section*/
Elf32_Word ovl_checksum;     /*checksum of the overlay data*/
Elf32_Word ovl_flags;        /*overlay flags, used by the 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;
```

- Elf32_Addr is a 32-bit unsigned value
- Elf32_Word is a 32-bit integer value
- Elf32_Half is a 16-bit integer value

The ovl_run and ovl_load fields contain the run and load addresses of the overlay.

The ovl_size field contains the size of the overlay data.

The ovl_checksum field may contain a checksum of the overlay data. The StarCore linker always sets this field to zero.

The ovl_flags field is for use by the overlay manager. A typical use would be to indicate whether the overlay section is currently loaded.

The ovl_other field is a bitset which may contain the following flags:

- OVL_OTHER_NONE 0 — ordinary text section
- OVL_OTHER_WRITE 1 — ordinary data section
- OVL_OTHER_DEF_LOADED 2 — section loaded by the linker at the its run address

The above-mentioned values are defined in the overlay.h file.

The field should be written only by the linker and may be used by the overlay manager to copy back only the data sections and to know which sections are loaded by default by the linker at their run address.

The ovl_shndx field is the section number of the corresponding overlay.

Although the ovl_parent, ovl_sibling, and ovl_child fields can be used to represent a dependency tree, since the StarCore linker command file permits an overlay section to appear in as many group directives as necessary, these fields are not used (they are always set to zero).

4.8 Debug Support

The compiler generates for each overlay section ovlsec separate debug sections:

.debug_lineovlsec, .debug_infoovlsec, …

The linker appends a signature to each overlay section to let the debugger automatically identify the overlay section that is currently loaded into the run space.

The signature consists of two fields:

- uint16_t ovl_shndx — overlay section index
- uint32_t ovl_size — overlay section size including the signature

The linker also generates an overlay table for each debug section.

.debug_line_ovltab, .debug_info_ovltab, …, which contains the following fields for each overlay debug section:

- uint16_t ovl_shndx — index in ovltab of the corresponding overlay section
- uint16_t debug_shndx — index of the debug information (.debug_infoovlsec) corresponding to the overlay section

- uint32_t ovl_run — offset from the beginning of the debug section (.debug_infoovlsec) where the debug information for the overlay section is located
- uint32_t ovl_size — the size of the debug information corresponding to the overlay section

In order to generate consistent debug information a module containing an overlay/union section must not include other text sections. This remark applies only to modules containing debug information.

4.9 Examples

A simple C overlay manager in presented below:

```
void *_overlay_manager(void *load_addr) {
unsigned long int k;
void *res;
for (k=0; k<_overlay_count; k++) {
if (_overlay_table[k].ovl_load == load_addr) {
res = memcpy(_overlay_table[k].ovl_run, _overlay_table[k].ovl_load,
_overlay_table[k].ovl_size);
return res;
}
}
printf("ERROR: Unable to find an overlay section with load address =
%08lX\n",load_addr);
return NULL;
}
```

If an assembly version is needed a simple implementation is presented below:

```
;input: r0 - the load address of the overlay section to be loaded
;output: r0 - the run address if an overlay section was found or 0
; otherwise
global __overlay_manager
align 16
__overlay_manager type func OPT_SPEED
[
move.l <__overlay_count,d0 ;no of overlay sections
suba r1,r1 ;software loop counter
]
[
tsteq d0 ;test if there are overlay sections
push r6 ;ABI
push r7 ;ABI
]
[
adda #<8,sp ;prepare for memcpy()
bt <L2 ;no overlay sections
]
move.l #__overlay_table+4,r3 ;pointer to overlay_table - load_addr
move.l #__overlay_table,r2 ;pointer to overlay_table - run_addr
falign
L1
[
move.l (r3),r4 ;the load address of the overlay section
adda #>32,r3,r3
;pointer to the next entry in overlay_table - load_addr
```

```
]
nop ;AGU stall
cmpeqa r4,r0 ;check if load_addr of the current
entry in overlay_table == input
bf <L3 ;this isn't the input section
;we've found the section
[
move.l (r2),r0 ;take the run_addr of the current section
adda #<8,r2 ;pointer to ovl_size in overlay_table - for memcpy()
]
;prepare call to memcpy()
[
move.l (r2),d1
tfra r4,r1
]
jsrd _memcpy ;perform memcpy()
move.l d1,(sp-4) ;third parameter for memcpy()
jmp L6 ;go to the end
L3
[
adda #<1,r1 ;increment the loop counter
adda #>32,r2,r2
;pointer to the next entry in overlay_table - run_addr
]
move.l r1,d2
;check if we finished all the entries in overlay_table
cmphi d2,d0
bt <L1 ;software loop
L2
;an overlay section with the specified load address wasn't found
suba r0,r0 ;return NULL
L6
;we've found an overlay section with the specified load address
suba #<8,sp ;restore the stack
[
pop r6 ;ABI
pop r7 ;ABI
]
rts
F__overlay_manager_end
endsec
```

In order to call the overlay manager one should use the following piece of code:

```
extern void *Pgm1, *Pgm2,*Data1,*Data2; /*name of overlay sections*/
void main()
{
_overlay_manager(Ovl_Load_Address(&Pgm1));
_overlay_manager(Ovl_Load_Address(&Data1));
}
```

The assembly-written overlay manager may be called using:

```
extern void *Pgm1, *Pgm2; /*name of overlay sections*/
extern void *_overlay_manager(void *);
void main()
{
```

```
void *res;
//called from C
res = _overlay_manager(Ovl_Load_Address(&Pgm1));
if (!res)
return;
//called from asm
asm(" move.l #LoadAddr__Pgm2,r0");
asm(" jsr __overlay_manager");
}
```

A simple application file is presented below:

```
configuration
view My_View
section
program = [pgm1: "Pgm1" overlay,
pgm2: "Pgm2" overlay,
pgm3: "Pgm3" overlay,
pgm4: "Pgm4" overlay,
pgm5: "Pgm5" overlay,
]
data = [
data1: "Data1" overlay,
data2: "Data2" overlay,
data3: "Data3" overlay,
data4: "Data4" overlay,
data5: "Data5" overlay,
data_non_ovl: “Data_non_ovl”
]
end section
module "f1" [
program = pgm1
data = data1
place (“variables_not_overlaid”) in data_non_ovl
]
module "f2" [
program = pgm2
data = data2
place (“variables_not_overlaid”) in data_non_ovl
]
module "f3" [
program = pgm3
data = data3
place (“variables_not_overlaid”) in data_non_ovl
]
module "f4" [
program = pgm4
data = data4
place (“variables_not_overlaid”) in data_non_ovl
]
module "f5" [
program = pgm5
data = data5
place (“variables_not_overlaid”) in data_non_ovl
]
end view
```

```
use view My_View
end configuration
```

Using this application file creates overlay sections for the whole code in f1, f2, f3, f4 and f5.c modules and overlay sections for data in the same modules. The variables specified in `variables_not_overlaid` will be placed in sections, an ordinary .data section (not an overlay section).

A simple linker command file which overlays two text sections (Pgm1 and Pgm2) and two data sections (Data1 and Data2) should contain:

```
.provide Run_OVER, 0x4000
.memory 0, 0x7ffff, "rwx"
;;overlay definition
.overlay ".My_Program_Overlay", "rwx","Pgm1","Pgm2"
.overlay ".My_Data_Overlay", "rw","Data1","Data2"
;; overlay run address
.org Run_OVER
.segment .ovl_program ,".My_Program_Overlay"
.segment .ovl_data ,".My_Data_Overlay"
```

If groups are needed one may use this:

```
.provide Run_OVER, 0x4000
.memory 0, 0x7ffff, "rwx"
.group ".My_Prog1", "Pgm1", "Pgm3"
;Pgm1 and Pgm3 may exist at the same time
.group ".My_Prog2", "Pgm2", "Pgm4"
;Pgm2 and Pgm4 may exist at the same time
.group ".My_Data1", "Data1", "Data3"
;Data1 and Data3 may exist at the same time
.group ".My_Data2", "Data2", "Data4"
;Data2 and Data4 may exist at the same time
;;overlay definition
.overlay ".My_Program_Overlay", "rwx",".My_Prog1",".My_Prog2"
;.My_Prog1 and .My_Prog2 are run at the same address
.overlay ".My_Data_Overlay", "rw",".My_Data1",".My_Data2"
;.My_Data1 and .My_Data2 are run at the same address
;; overlay run address
.org Run_OVER
.segment .ovl_program ,".My_Program_Overlay"
.segment .ovl_data ,".My_Data_Overlay"
```

The map file will look like:

```
0x4000 [Pgm1] [Pgm2]
. . .  [Pgm3] [Pgm4]
0x4100 [Data1] [Data2]
. . .  [Data3] [Data4]
```

If an overlay section is needed in more than one group:

```
.provide Load_Pgm2G, 0x2000
.group "Pgm1G", "Data1", "Pgm1", "Data5","Pgm5"
.group "Pgm2G", Load_Pgm2G, Load_Segment, "Data2", "Pgm2"
.group "Pgm3G", "Data3", "Pgm3"
.group "Pgm4G", "Data4", "Pgm4"
.group "G1", "Pgm1G", "Pgm2G"
.group "G2", "Pgm3G", "Pgm4G", "Pgm2G"
```

```
.group "G3", "Pgm3G", "Pgm2G"
.overlay ".My_Overlay", "rwx","G1","G2,"G3"
.org 0x1000
.segment .ovl, ".My_Overlay"
```

The map file (assuming all section are 50 bytes long):

```
0x1000          [Data1]         [Data3]         [Data3]
                [Pgm1]          [Pgm3]          [Pgm3]
                [Data5]         [Data4]         [empty]
                [Pgm5]          [Pgm4]          [empty]
                [Data2]         [Data2]         [Data2]
                [Pgm2]          [Pgm2]          [Pgm2]
```

In the following example the Initial_ovl section will be also loaded by the linker at its run address:

```
.overlay ".My_Overlay", "rwx","Ovl1","Ovl2","Init_ovl"* ,"Ovl3"
```

5 Object File Interface

ELF section type SHT_STARCORE_OVERLAY (0x70000000) equivalent to a SHT_PROGBITS section except that at link time its run address and load address are different. The sh_addr field contains the run address and the sh_info field contains the link address. The sh_link field points to the section number of the file's overlay table. In a relocatable object file, all of these fields are zero.

The linker is responsible for creating a segment large enough to hold the largest overlay. This created segment will be of type PT_LOAD with p_filesz==0.

ELF section type SHT_STARCORE_OVLTAB (0x70000001) is a loadable, writeable section (SHF_ALLOC | SHF_WRITE) which contains information about the overlays in an executable. This structure is used by a run-time overlay manager. The name of the section is .ovltab, it contains the _overlay_table array of overlay structures followed by _overlay_count, an unsigned 32-bit integer containing the number of array entries. The structure of the _overlay_table array is presented in Ovltab Section.

ELF section type SHT_STARCORE_UNION (0x70000002) equivalent to a SHT_NOBITS section, generated only by the assembler.

6 Linker Error Messages

The linker may give the following error messages applicable to overlays:

- Could you try to place segment <segment_name> at <address>.

  The linker cannot place the segment at the user specified address. Try to use the suggested address.
- Section <section_name> was not mentioned in any .overlay/.union directive.

  An overlay/union section must be present in exactly one overlay/union directive.
- Computation of overlay section cannot be done because […].

  At least a section specified in the .group directives is to be placed at two run addresses. Possible loops.
- <section_name> has already been overlaid.

  The specified section was mentioned in more than one .overlay directive.

- Can’t link group section <section_name> explicitly.

  A .group section can’t be used for the .segment directive.

- <section_name> is not an overlay/union section.

  A non-overlay section was specified in a .overlay/.union directive.

- It is illegal to have two default configurations into overlay directive.

  Two sections are to be loaded by the linker at the same run address. Remove an asterisk.

- Section <section_name> already exists in overlay <overlay_name>.

  The same section was specified twice in the same overlay directive.

- Overlay <overlay_name> already exists in command file.

  The same overlay name was used in more than one .overlay directive.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order No.: AN3788
4/2010