Freescale Semiconductor
Application Note

Document Number: AN4176
Rev. 1, 07/2011

A Guide to Linking Multicore Application Code for StarCore MSC8156 DSPs

by *Freescale MSG Group*
Freescale Semiconductor, Inc.
Austin, TX

MSC8156 customers frequently design DSP application code with common software components that execute across all of the cores of the device. However, the application can also contain non-homogenous software: that is, code which is unique to one or more cores, but not all of them. Writing the software is generally not as difficult as *linking* the software is. The challenge lies in how to link these diverse components in a way that uses the available memory as efficiently as possible, while still providing memory protection and symbol visibility that the software design requires.

A real-time operating system (RTOS) is often added to the design, as it organizes runtime control and makes the management of shared peripherals much easier. Customers often use Freescale's SmartDSP OS or another third-party RTOS for this purpose. This can further complicate the linking process, however. This application note describes the procedure and design considerations for properly linking multicore SmartDSP-OS-based applications. The information presented here applies to the Freescale MSC8156 and its variants.

Contents

1 Hardware Considerations

When designing a multicore application for the MSC8156, several things need to be considered before starting. These include:

- Amount and type of physical memory available in the design.
- This memory's visibility to the cores.
- How to partition this memory into Memory Management Unit (MMU) descriptors.
- Setting the attributes of these MMU entries.
- Defining the sections that each MMU entry contains.

A *section* is a definition that binds logical names to the physical memory segments that the linker uses. This makes it easier to redefine the mapping of the program's elements. The use of sections is covered in Section 1.5.

1.1 Physical Memory for the StarCore MSC8156

The MSC8156 and its derivatives all have a four-level memory subsystem. It consists of two external DDR2/3 memory controllers, two levels of internal SRAM memory (M3 and M2), and two separate L1 cache memories. Figure 1 shows the memory hierarchy and latencies of each memory level for the MSC8156.

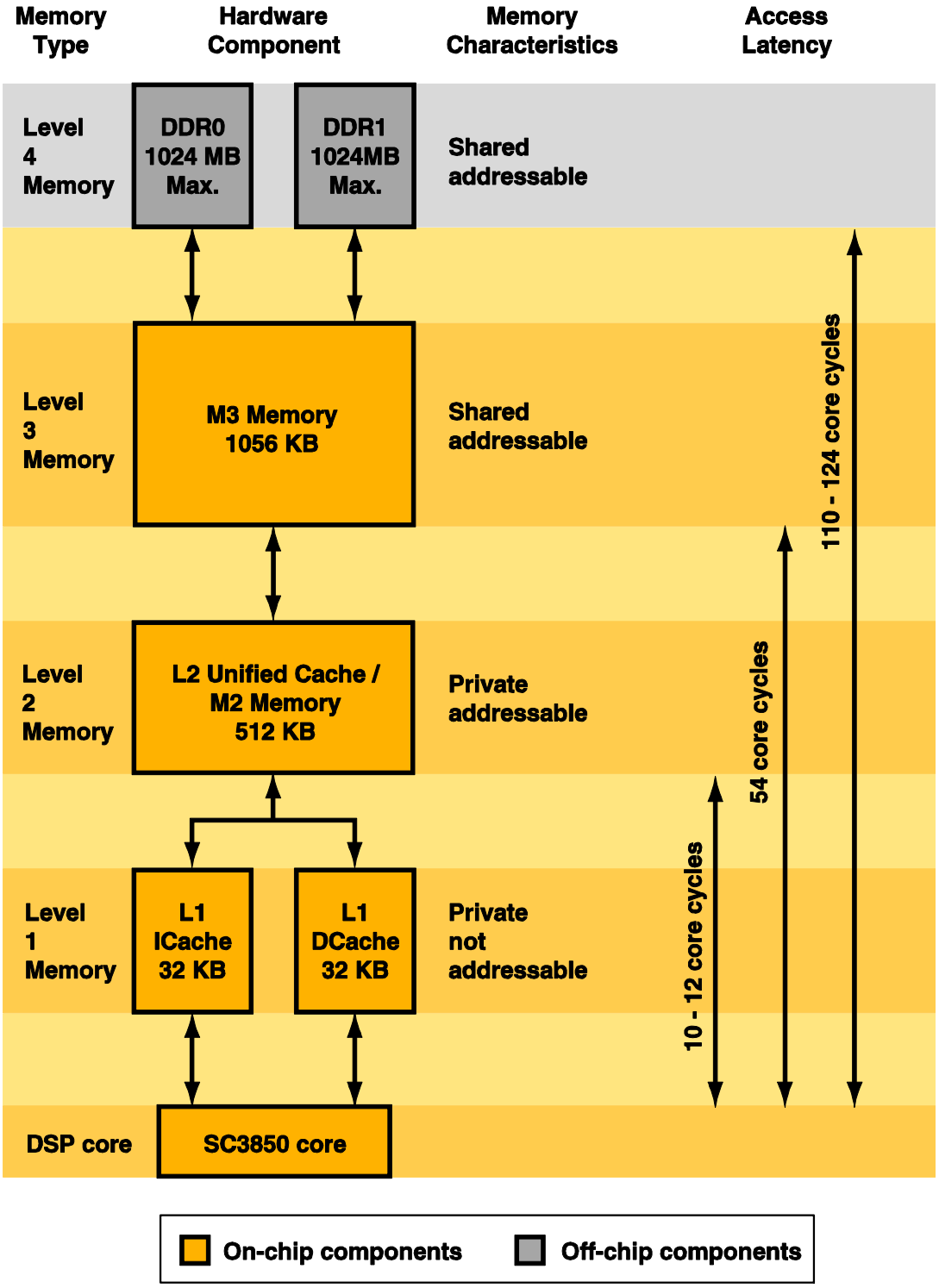

Figure 1. StarCore MC8156's Four Memory Levels

1.1.1 External DDR Memory

The MSC8156 has two DDR interfaces and controllers that manage access to the external level four memory. Each DDR controller can support up to 1 GB of DDR2 or DDR3 memory. Accesses to DDR memory can have latencies between 110 and 124 clock cycles. DDR memory can be configured to be entirely shared or entirely local memory. Or, it can be configured to be any combination of the two.

1.1.2 Internal M3 Memory

The internal level three M3 memory consists of a single block of 1056 KB of RAM. Its latency on the MSC8156 is 54 clock cycles. The M3 memory is like DDR memory in that it can be configured as shared, local, or any combination of the two. M3 memory on the MSC8156, when configured as a non-cacheable block, can also be used for software semaphores and spinlocks. This feature is used by SmartDSP OS and some runtime library code. User code can also use M3 memory this way in some instances. If this is the case, plan to set some of this memory aside for this purpose.

1.1.3 Internal M2 Memory

Each core contains a dedicated block of 512 KB of RAM. This level two M2 memory is configurable at link time. It can be split into contiguous chunks of 64 KB, with each chunk designated as either as a level 2 (L2) cache or as low-latency internal memory. When configured as M2 memory, it is always local (or private) per core. That is, M2 memory can never be used as a shared memory. When M2 memory is configured as a L2 cache, it caches both data and instructions similar to that of the level one cache. M2/L2 memory latency is 12 cycles for data accesses and 10 cycles for instruction accesses.

1.1.4 Internal level 1 ICache and DCache

Every processor core on the MSC8156 has its own level 1 (L1) instruction cache (ICache) and data cache (DCache). Each cache consists of a 32 KB block of zero wait state memory. Neither cache is directly accessible by the user.

1.2 Memory Visibility

The MSC8156 supports true shared memory. That is, memory can be shared for both data and program code. Memory can also be assigned to be local to a single core. There are limitations imposed by the physical design of the processor, however. The main consideration is each core's use of M2 memory. M2 memory is strictly visible only to the core that owns it. For example, core 0 cannot access core 3's M2 memory contents at all, and vice versa. The only method for accessing M2 memory content from one core to another is to use the system DMA controller.

M3 and DDR memories can be configured and subdivided into blocks that are either shared by all, are local per core, or have a shared attribute but an MMU entry only on a subset of cores. An access from one core to another core's local memory generates an access violation before it is allowed to complete. This is a very useful protection mechanism to have in a system. Currently, the processor cores can be grouped into a virtual subsystem through the use of shared and local memory. So, if a subsystem runs on four out the MSC8156's six cores, there is an advantage in using shared memory for the subsystem's common code and data.

1.3 Memory MMU Attributes

The MSC8156 supports memory protection, and each core has its own MMU. The MMU manages address translation, memory protection, memory sharing policy, and cache policy on a per-core basis. Each MMU has twelve program table entries and 20 data table entries. This means that each core can have up to twelve blocks of code and 20 blocks of data with distinct MMU attributes set for each.

1.4 Design Considerations

In summary, the hardware has a large impact on the application design and how the end user links the application. These include:

- Splitting the use of M2 memory between internal M2 SRAM and L2 cache.
- Placement of code and data, with respect to each memory's inherent latency.
- Placement of code and data such that they are visible only to a specific core (local), visible to a subset of cores, or visible to all of them (shared).
- Splitting of DDR and M3 memory between local and shared memory.
- Setting MMU cache policies for local and shared memories.
- Setting the address translations for local and shared memories.

1.5 Defining Memory with the Linker Command File

CodeWarrior for Starcore DSPs v10.0's linker command file first defines the memory available for the application. It then defines a list of MMU descriptors which describe the characteristics of that memory. A developer defines *sections*, which are blocks of memory that store the application's code and data. The linker command file maps these sections into the appropriate descriptors. Finally, the run-time library startup code initializes the descriptors to an MMU table using the current MMU policies and address translations. The MMU polices and address translations were defined statically during the application's linking phase.

Figure 2. The Linker Command File Section Definitions

These section definitions describe the blocks of memory that the linker works with to organize the application's code and data in the system.

Figure 2 shows the two definition lists that describe the memory space for a standard SmartDSP OS application. A portion of the memory space is shared, so the first definition specifies a shared attribute for any MMU entry whose label appears in this list. Program code and some shared data, such as flags and global variables, reside in this region. The private (local) memory is defined by the second definition list. This list sets the private attribute for those MMU entries which use the labels in this list. Private memory contains items such as heaps, stacks, and data used on a per core basis. They can also contain program code unique to a specific core, as will be shown later.

Note that linker macros are used extensively here. This is done to allow fast re-definition of the memory layout. The linker runs *N* passes on these commands, where *N* is defined in the linker command file using the command:

```
number_of_cores(N);
```

This determines how many output files are generated as the linker makes its passes through the definitions. The linker function `num_core()` determines the number of linker passes required and the number of executables files to be generated. The function `core_id()` determines the current core ID. As each link is processed, the core ID information becomes part of the macro definitions that define the start and end locations of memory.

Figure 3 and Figure 4 show a graphical representation of the standard SmartDSP OS application's use of physical memory on a MSC8156 ADS board. As expected, M2 memory is used only for local, not shared, memory.

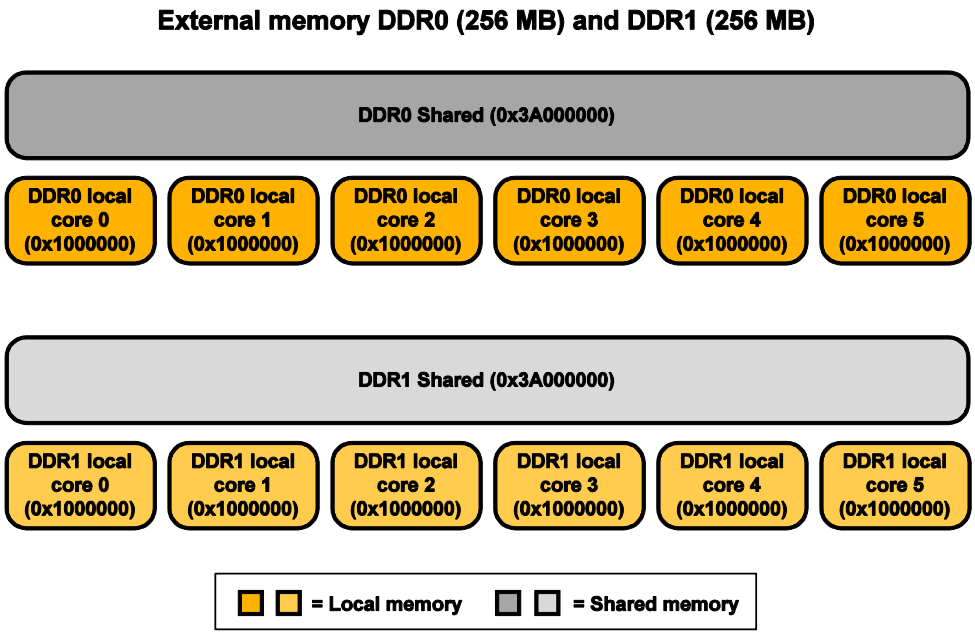

Figure 3. Off-chip Memory Map for a SmartDSP OS Application on the MSC8156 ADS

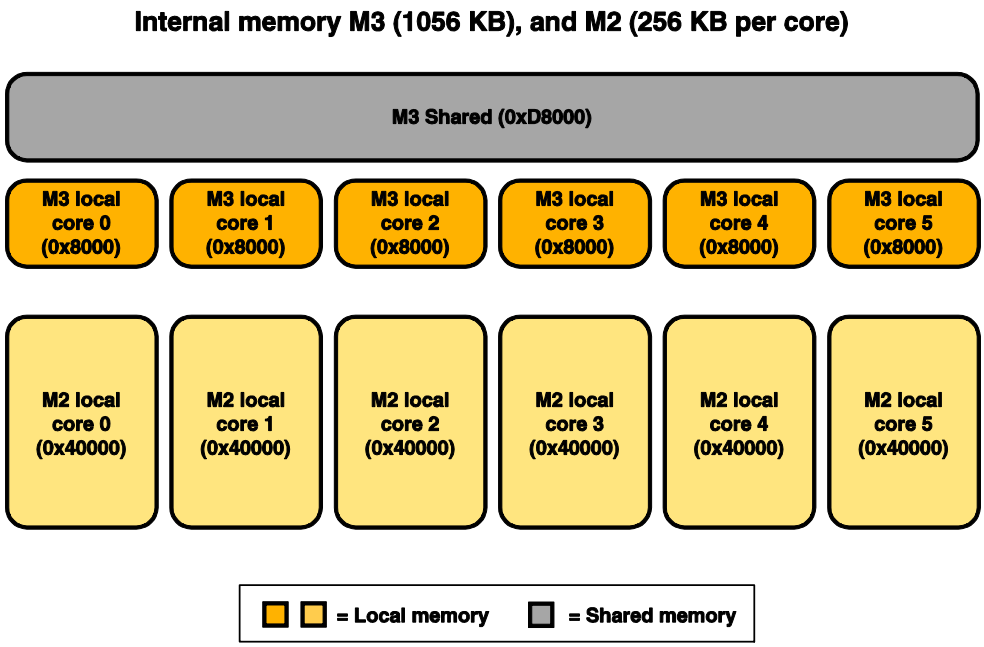

Figure 4. On-chip Memory Map for a SmartDSP OS Application on the MSC8156 ADS

2 Software Considerations

Now that the hardware's capabilities are understood, it can be seen how they impact the design of the system software. Software considerations for memory latency, cache policy, protections, and visibility all need to be taken into account. Additionally, it is important to understand the content being linked. These issues are discussed further in the sections that follow.

2.1 Memory Latency

Internal memories are at a premium, because they are fixed in size. However, they also offer two to 10 times faster access times than external memories. Careful consideration should be given towards the placement of sections in these memories, and the placement of content within those sections. When code executes in non-cacheable memory, each access to these memories incurs a cycle count equivalent to the memory's latency. Also note that first accesses to these memory regions, even when they are set as cacheable, incur the same cycle penalty. However this performance hit occurs only for the first memory access, because it also triggers a cache load. Assuming that subsequent program accesses span sequential addresses, the data can be retrieved from the lower-latency cache.

2.2 Cache Policy

The lowest latency memories on the MSC8156 system are the level 1 caches. Cache management is one of the key considerations for creating high-performance applications on the MSC8156. It is just as important to consider whether code or data will be available in cache, just as it is important to know which items need not be in cache. Additionally, the level 2 cache size is adjustable. A larger level 2 cache reduces the available M2 memory a core can use. The tradeoff is between cache performance versus low latency storage, and decisions must be made to determine the proper sizes for each memory type. Cache policy also needs to be considered for memory regions that are used for DMA source and destination locations. The system DMA and On Chip Network (OCN) DMA run underneath the MMU and the caches, so software management of coherency in these regions is necessary if the locations are set to have a cacheable policy.

2.3 Memory Protection

Memory is protected by its partitioning on the system in three ways. First, local (or private) memory cannot be accessed by cores that do not own the memory. Second, MMU entries give sections of memory either user or supervisor attributes. These attributes prevent user code or data from accessing supervisor code or data. This protects key sections of memory against unwanted accesses. Additionally, only supervisor- level code has access to the privileged mode registers and other content not made available to user-level code. For example, SmartDSP OS tasks execute with user-level permissions, and so they cannot affect supervisor mode elements. Only the SmartDSP OS kernel, drivers, and background tasks can execute in supervisor mode and access peripheral registers. Third, enabling the MMU memory protection feature protects memory against any accesses that are not specifically mapped by the MMU. In this way omitting a memory region prevents a core from accessing it. Memory can also be protected by simply leaving it out of the MMU entries for other cores. Accesses to out-of-map memory cause a MMU trap exception and therefore provide memory protection as well.

2.4 Code and Data Visibility

The biggest challenge for a multicore system which has shared and local memories is the linking and visibility of objects between cores. There are several types of application challenges involved, especially if the program design departs from the standard symmetric memory map. What has been shown so far is the standard template for symmetric code and data. It uses shared application code that links to local data. However, on each core this local data is mapped and address-translated to sit at the same logical addresses for every core. Specifically, a global variable that resides in local memory at address `0xYYYYYYYY` on core 0 can also be found at address `0xYYYYYYYY` on core 4—as well as the other four cores of the system. These six instances of the variable all look the same to the shared program code. This arrangement has a significant drawback, however. To maintain this scheme, when shared program code executes on all of the cores, the system needs an instance of every local variable. This means the memory usage is additive: the total memory used would be the sum of all of the unique variables used by all cores. However, even for those variables that it does not use, local memory space would have to be reserved on each core in order to preserve the layout and relative memory addresses of all of the variables. For completely symmetric applications, this is fine. Unfortunately, for any application that executes different code on different cores, this is an inefficient use of memory. This application note is meant to show some solutions to that problem.

This issue cannot be solved by simply defining local data variables for specific cores and not other cores. This is because the shared code is built to "see" that variable at only one location. If new variables are introduced, then the global variable is displaced to a new address, say, `0xZZZZZZZZ`. The shared program code would still access address `0xYYYYYYYY` and not at the new address `0xZZZZZZZZ` where the variable now sits. Additionally, if a variable does not exist on a specific core, when the program is linked, the shared code attempts to "see" it for all cores and this situation generates a link failure.

To solve these problems, users need to write application code such that there is a handoff point where executing shared code calls into locally executed code. To do this, simply create unique local functions with the same name, but different functionality for each core. Also be sure that these handoff functions have the exact same size on each core. This prevents anything sitting above them in memory from being shifted in address space relative to another core. Link the code sections they belong to such that they always begin at a fixed address. For information on how to do this consult the application note, AN4063, "Configuring an Asymmetric Multicore Application for StarCore DSPs".

3 Example 1: Standard SmartDSP OS Application

Most MSC8156 applications start out as a "stationery" project generated by the CodeWarrior IDE's StarCore project wizard. The stationary is a code template that implements a simple multicore SmartDSP-OS-based application. While it is more complex than a "hello world" example, this application is still about as simple a SmartDSP OS application that there could be. Because this code template is most often the starting point of the application's design (and it is a good starting point), it is important to discuss the basic demo application and its memory layout in more detail. Once it is understood how the simple application works, it can be modified to suit the requirements of a custom application.

NOTE

The SmartDSP OS stationery project is also included as basic demo in the SmartDSP OS demonstration program directory, located at `{CodeWarrior Installation}\SC\StarCore_Support\SmartDSP\demos\starcore\msc815x\basic_demo`.

3.1 Application Design

The basic SmartDSP OS project consists of an application that runs on all six cores. Each core executes identical application code on each core. This means the memory map for each core looks the same.

3.2 Application Memory Layout

The standard SmartDSP OS application template uses DDR0 and DDR1 system memory to store various shared and local objects. The contents of DDR0 memory are shown in Figure 5. As the figure indicates, external shared memory on DDR0 is used both for operating system and user program storage.

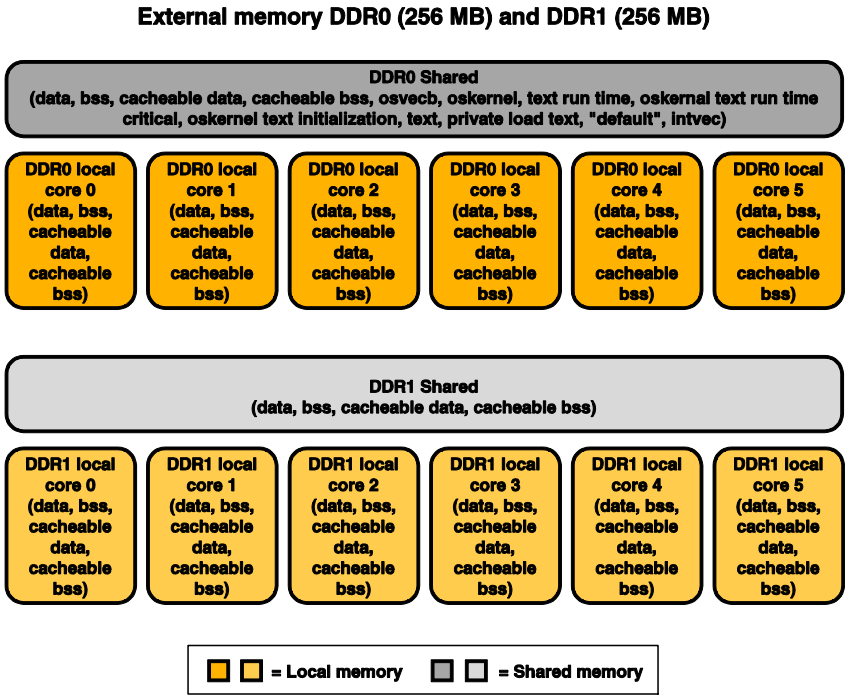

Figure 5. SmartDSP OS Application Component Placement in DDR0 Memory

Figure 6. SmartDSP OS Application Component Placement in DDR1 Memory

Figure 6 shows where the portions of SmartDSP OS and the application reside in DDR1 memory. DDR1 shared memory is used for shared `data` and `bss` (uninitialized data). Both DDR0 and DDR1 local memories store local `data` and local `bss` content for SmartDSP OS and the application.

Internal M3 memory holds shared OS `data` and shared OS `bss`, and for normal `data` and `bss` sections. Shared software mutexes, which can only be used in M3 memory, are placed here. Local M3 memory is reserved for local `data` and `bss` sections.

M2 memory contains most of the run-time critical OS `data` and `bss` sections. Additionally, user `data` is defined for this section along with initialization tables and MMU tables.

Sections in these memories are defined by default labels, or with user-defined labels called out by a `#pragma` in the source code. The method `__attribute__((section(".my_section_label")))`

allocates variables or program code into a specific section. Additional content can be allocated by use of an application file (`.appli`).

Table 1 summarizes the application's content in terms of the different sections, their purpose, and location in memory.

Table 1. Allocation of MMU Entries for SmartDSP-OS-Based Application that Executes on an MSC8156ADS

MMU Entry	Section	Memory Location and Type	Description
Data, cacheable	.oskernel_local_data	M2 local	OS-defined section
	.data	M2 local	Initialized data
	ramsp_0	M2 local	
	.oskernel_rom	M2 local	OS-defined section
	.rom	M2 local	Constants
	.exception_index	M2 local	Compiler runtime library defined for C++
	.ramsp_0	M2 local	
	.init_table	M2 local	Compiler runtime library defined
	.rom_init	M2 local	Compiler runtime library defined
	.bss	M2 local	Uninitialized data
	.rom_init_tables	M2 local	Compiler runtime library defined
	.staticinit	M2 local	Compiler runtime library defined for C++
	att_mmu	M2 local	MMU translation tables
	.oskernel_local_data bss	M2 local	OS-defined section
	.bsstab	M2 local	Compiler runtime library defined
	reserved_crt_tls	M2 local	Compiler runtime library defined
Data, cacheable	.local_data_M3	M3 local	Defined user space
	.local_data_M3_bss	M3 local	Defined user space
Data	.local_data_M3_nocacheable	M3 local	Defined user space
	.local_data_M3_bss_nocacheable	M3 local	Defined user space
Data, cacheable	.local_data_DDR0	DDR0 local	Defined user space
	.local_data_DDR0_bss	DDR0 local	Defined user space
Data	.local_data_DDR0_nocacheable	DDR0 local	Defined user space
	.local_data_DDR0_bss_nocacheable	DDR0 local	Defined user space

MMU Entry	Section	Memory Location and Type	Description
Data, cacheable	.local_data_DDR1	DDR1 local	Defined user space
	,local_data_DDR1_bss	DDR1 local	Defined user space
Data	.local_data_DDR1_nocacheable	DDR1 local	Defined user space
	.local_data_DDR1_bss_nocacheable	DDR1 local	Defined user space
Data	.os_shared_data	M3 shared	OS defined section
	.os_shared_data_bss	M3 shared	OS defined section
	reserved_crt_mutex	M3 shared	Compiler runtime library defined
Data	.shared_data_m3	M3 shared	Defined user space
	.shared_data_m3_bss	M3 shared	Defined user space
Data, cacheable	.shared_data_m3_cacheable	M3 shared	Defined user space
	.shared_data_m3_cacheable_bss	M3 shared	Defined user space
Data	.shared_data_ddr0	DDR0 shared	User defined space
	.shared_data_ddr0_bss	DDR0 shared	User defined space
Data, cacheable	.shared_data_ddr0_cacheable	DDR0 shared	User defined space
	.shared_data.ddr0_cacheable_bss	DDR0 shared	User defined space
Text, cacheable	.osvecb	DDR0 shared	OS defined section
	.oskernel_text_run_time	DDR0 shared	OS defined section
	.oskernel_text_run_time_critical	DDR0 shared	OS defined section
	.oskernel_text_initialization	DDR0 shared	OS defined section
	.text	DDR0 shared	Program code
	.private_load_text	DDR0 shared	Program code
	.default	DDR0 shared	Linker default section
	.intvec	DDR0 shared	Interrupt vectors
Data	.shared_data_ddr1	DDR1 shared	User defined space
	.shared_data_ddr1_bss	DDR1 shared	User defined space
Data	.shared_data_ddr1_cacheable	DDR1 shared	User defined space
	.shared_data_ddr1_cacheable_bss	DDR1 shared	User defined space

4 Example 2: Wireless Infrastructure Application

The next example describes a hypothetical wireless application. It discusses how the application tasks are organized into virtual subsystems, and how the linker is directed to arrange these subsystems in the system memory.

4.1 Application Design

Allocating processing power for each subsystem is a task for the system designer. The allocation of these resources depends upon the communication system's processing requirements and the interfaces it uses. For this example, the system is partitioned to have a single core running a control subsystem, a second core running the downlink subsystem, and the remaining four cores executing the subsystem code that manages the uplink.

4.2 Application Memory Layout

The wireless application uses the system's available external and internal memory as shown in Figure 7 and Figure 8.

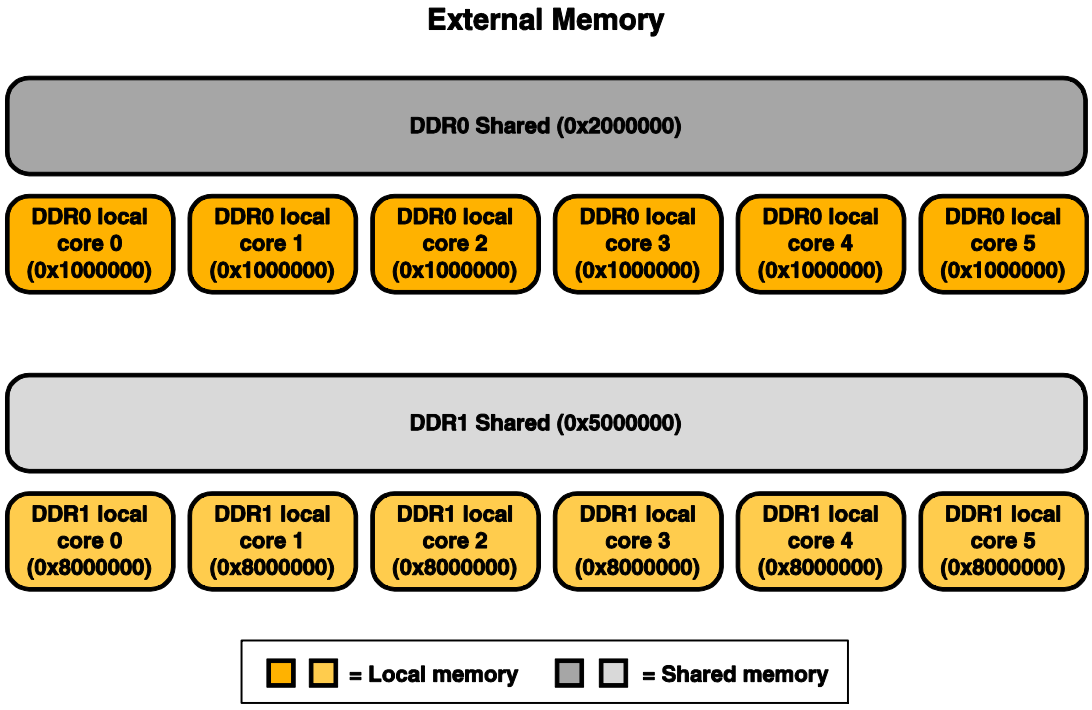

Figure 7. Map of Wireless Application Sections in External Memory

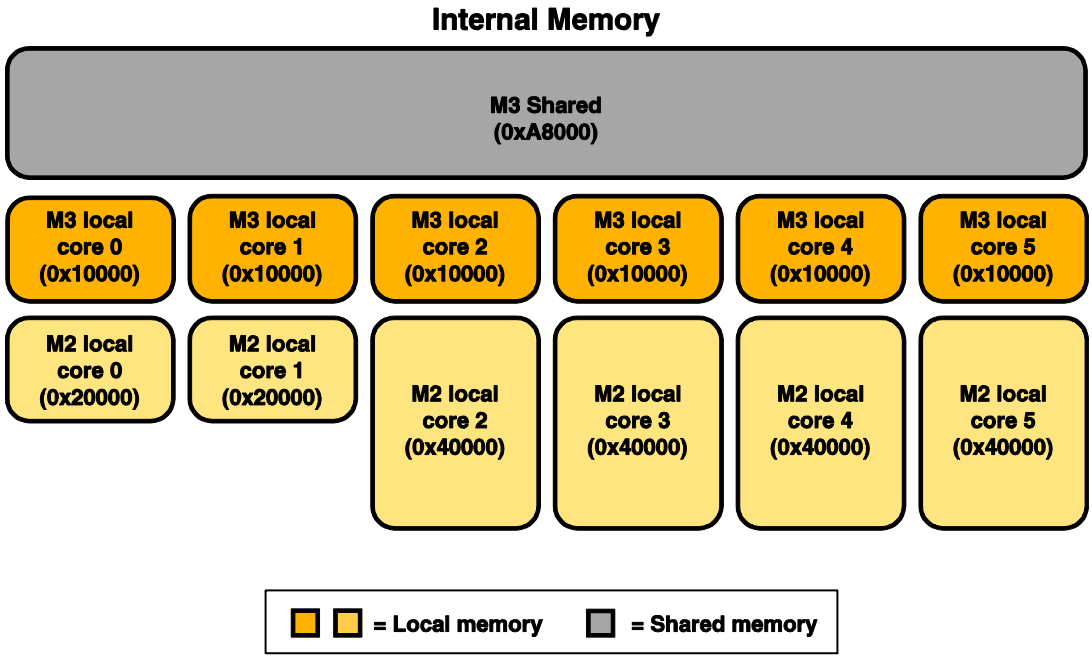

Figure 8. Map of Wireless Application Sections in Internal Memory

4.3 How to Modify the Standard SmartDSP OS Application to Implement the Design

This section describes all the modifications that are applied to the standard SmartDSP OS wizard-generated project to get to the desired application layout.

As mentioned previously, the wireless application layout is based on three subsystems:

- A subsystem running control code. This subsystem executes on core 0 and is referenced as *control subsystem* in this chapter.
- A subsystem running uplink code. This subsystem executes on core 1 and is referenced as *uplink subsystem* in this chapter.
- A subsystem running downlink code. This subsystem executes on cores 2, 3, 4, and 5 and is referenced as *downlink subsystem* in this chapter.

4.3.1 Modifications in C Source Files

4.3.1.1 Define the Sections Required to Build Each Subsystem

Table 2 shows how the sections for the control subsystem are defined.

Table 2. Memory Section Definitions for the Control Subsystem

Section name	Usage
.local_control_text	Holds the actual code for the control subsystem. This section is private to core 0, where the control code executes.
.local_control_data	Holds the initialized data for the control subsystem. This section is private to core 0, where the control code executes.
.local_control_bss	Holds the uninitialized data for the control subsystem. This section is private to core 0, where the control code executes.
.local_control_text_p	Holds the function `per_core_init`, This function is called by `app_init` and must be allocated at the same virtual address on all six cores.

Table 3 shows how the sections for the uplink subsystem are defined.

Table 3. Memory Section Definitions for the Uplink Subsystem

Section name	Usage
.local_uplink_text	Holds the actual code for the uplink subsystem. This section is private to core 1, where the uplink code executes.
.local_uplink_data	Holds the initialized data for the uplink subsystem. This section is private to core 1, where the uplink code executes.
.local_uplink_bss	Holds the uninitialized data for the uplink subsystem. This section is private to core 1, where the uplink code executes.
.local_uplink_text_p	Holds the function `per_core_init`, This function is called by `app_init` and must be allocated at the same virtual address on all six cores.

Table 4 shows the sections that comprise the downlink subsystem.

Table 4. Memory Section Definitions for the Downlink Subsystem

Section name	Usage
.local_downlink_text	Holds actual code for the downlink subsystem. This section is partially shared between core 2, core 3, core 4 and core 5, where the downlink code executes.
.local_downlink_data	Holds initialized data for the downlink subsystem. This section is symmetrical among core 2, core 3, core 4 and core 5, where the downlink code executes.
.local_downlink_bss	Holds uninitialized data for the downlink subsystem. This section is symmetrical among core 2, core 3, core 4 and core 5, where the downlink code executes.
.local_downlink_text_p	Holds the function `per_core_init`, This function is called by `app_init` and needs to be allocated at same virtual address on all six cores.

In the provided sample project, pragmas in the `.c` source files provides the linker access to the code and data allocated in the corresponding sections.

Figure 9 and Figure 10 show the placement of the application content in each of the memory regions.

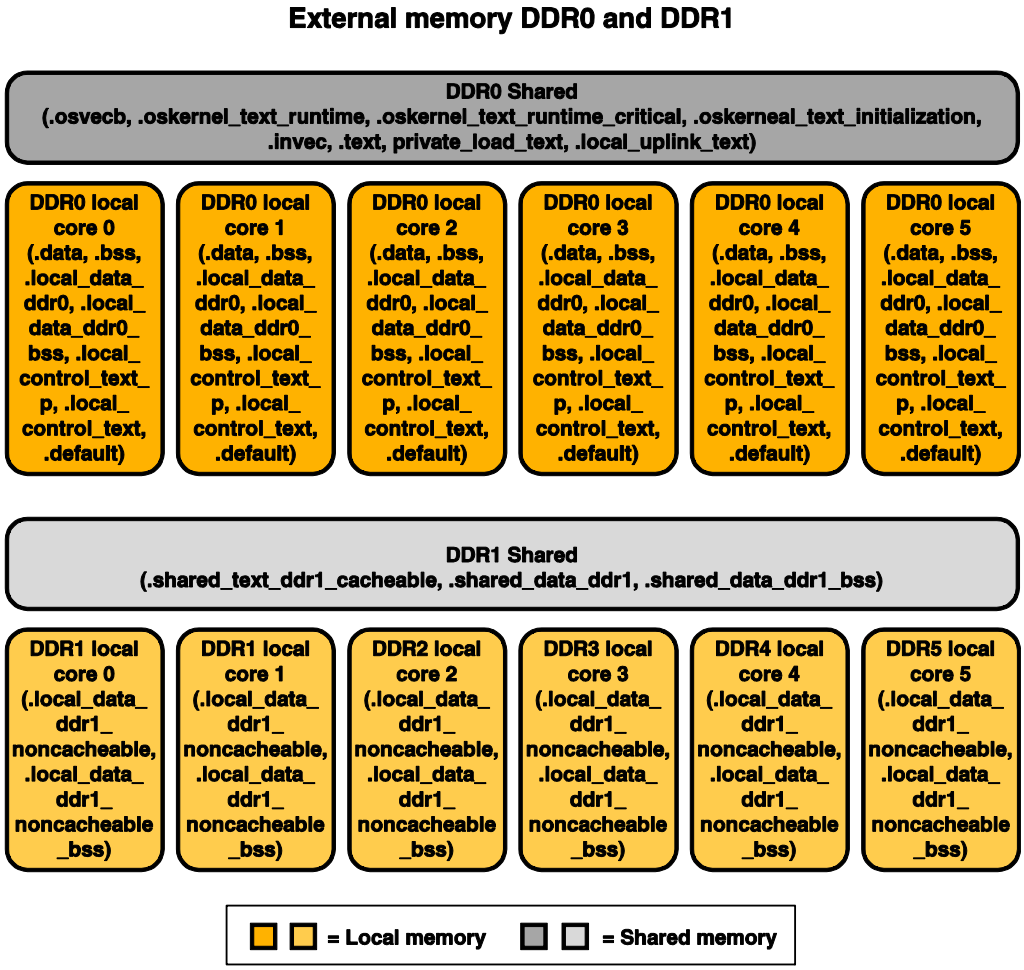

Figure 9. Allocation of the Application Sections in External Memory.

Figure 10. Allocation of Application Sections in Internal Memory.

External shared memory on DDR0 is used for all of the operating system and user program storage, while DDR1 shared memory is used for shared `data` and `bss` (uninitialized data). DDR0 and DDR1 local memory also store local `data` and local `bss` content.

Internal M3 memory is used for shared OS `data` and shared OS `bss`, and for the normal `data` and `bss` sections. It is also used for some software mutexes, which can only be used in M3. Local M3memory is reserved for `data` and `bss` sections. M2 memory contains most of the runtime critical OS `data` and `bss` sections. Finally, user data is defined for this section, along with initialization tables and the MMU tables.

4.3.1.2 Implement the Subsystem Configuration Code

The project uses a shared `main` and `appInit` function. The `appInit` function calls a subsystem-specific `per_core_init` function, which is responsible for initializing variables and creating OS objects for the various subsystems.

Function `per_core_init` is implemented respectively in the modules `control_priv.c`, `downlink_priv.c` and `uplink_priv.c`,

4.3.2 Modification in Linker Files

4.3.2.1 Define L2/M2 Mapping

Inside of file `memory_map.l3k`, the definition of the symbol `__L2_cache_size` has been made core-dependent. This allows the definition of different L2/M2 mappings on each core. This has been implemented as follows:

```
////// Local partition sizes //////////
__L2_cache_size =
         (_ID_CORE == 0) ?0x20000:
         (_ID_CORE == 1) ?0x20000:
         (_ID_CORE == 2) ? 0x40000 :
         (_ID_CORE == 3) ? 0x40000 :
         (_ID_CORE == 4) ? 0x40000 :
         (_ID_CORE == 5) ? 0x40000 :
         0x40000;
```

NOTE

Although the provided project has M2 memory partitioned as 256 KB of L2 cache and 256 KB of M2 memory, a different mapping for some of the cores can be specified. This works well provided the application uses SmartDSP OS v3.6.1 or higher. For later versions of SmartDSP OS, core 0 and core 1 get 128 KB of L2 cache, while cores 2 through 5 get 256 KB of L2 cache.

4.3.2.2 Define the Tasks for Each Core System Task.

A system task must be defined for each core to allow the allocation of sections to a specific core or set of cores. To this end, the following commands have been added to the file named `os_msc815x_Link.l3k`:

```
tasks {
  c0: task_c0, 0, 0,0;
  c1: task_c1, 0, 0,0;
  c2: task_c2, 0, 0,0;
  c3: task_c3, 0, 0,0;
  c4: task_c4, 0, 0,0;
  c5: task_c5, 0, 0,0;
}
```

4.3.2.3 Define the Memory Area for Unit Shared among All Cores

In the file `os_msc815x_link.l3k`, the memory areas used to allocate shared code and data are defined. The following changes must be applied to the default mapping created by the wizard:

- Constants are separated from data in shared DDR0 memory.
- SmartDSP-OS- shared code is separated from non-SmartDSP-OS shared code in DDR0 memory.
- A dedicated memory area for cacheable shared text is defined in DDR1 memory.

This is written as follows:

```
unit shared (*) {
  KEEP ( _interr_vector_file );
  MEMORY {
    os_shared_data_desc             ("rw"): org =_SharedM3_b;
    shared_data_m3_desc             ("rw"): AFTER( os_shared_data_desc);

    os_kernel_text_desc             ("rx"): org =_SharedDDR0_b;
    shared_text_ddr0_cacheable_desc ("rx"): AFTER( os_kernel_text_desc);
    shared_data_ddr0_cacheable_desc ("rw"): AFTER( shared_text_ddr0_cacheable_desc);
    shared_rom_ddr0_cacheable_desc  ("r") : AFTER( shared_data_ddr0_cacheable_desc) ;

    shared_text_ddr1_cacheable_desc ("rx"): org =_SharedDDR1_b;
    shared_data_ddr1_descr          ("rw"): AFTER( shared_text_ddr1_cacheable_desc);
  }
...
}
```

The section block for the new memory area appears as follows:

```
unit shared (*) {
  ...
  SECTIONS {
    descriptor_shared_rom_ddr0_cacheable {
      .rom
      .shared_data_ddr0_cacheable_rom
    } > shared_rom_ddr0_cacheable_desc;

    descriptor_os_kernel_text {
      .osvecb
      .oskernel_text_run_time
      .oskernel_text_run_time_critical
      .oskernel_text_initialization
      .intvec
    } > os_kernel_text_desc;

    descriptor_shared_text_ddr0_cacheable {
      .text
      .private_load_text
//       .default
    } > shared_text_ddr0_cacheable_desc

    descriptor_shared_text_ddr1_cacheable {
      .shared_text_ddr1_cacheable
    } > shared_text_ddr1_cacheable_desc;
  }
...}
```

The address translation block is also updated to keep track of the new memory areas:

```
address_translation (*) map11 {
    os_shared_data_desc               (SHARED_DATA_MMU_DEF): SHARED_M3;
    shared_data_m3_desc               (SHARED_DATA_MMU_DEF): SHARED_M3;

    os_kernel_text_desc               (SYSTEM_PROG_MMU_DEF): SHARED_DDR0;
    shared_text_ddr0_cacheable_desc   (SYSTEM_PROG_MMU_DEF): SHARED_DDR0;
    shared_data_ddr0_cacheable_desc   (SYSTEM_DATA_MMU_DEF): SHARED_DDR0;
    shared_rom_ddr0_cacheable_desc    (SYSTEM_ROM_MMU_DEF):  SHARED_DDR0;

    shared_text_ddr1_cacheable_desc   (SYSTEM_PROG_MMU_DEF): SHARED_DDR1;
    shared_data_ddr1_desc             (SHARED_DATA_MMU_DEF): SHARED_DDR1;
}
```

4.3.2.4 Define a Memory Area for Unit That Is Symmetric Among All Cores

In the file `local_map.l3k`, the memory areas used to allocate symmetrical data are defined. The following modifications are applied to the default wizard-generated mapping:

- Constants are separated from data in shared M2 memory.
- SmartDSP-OS-symmetrical data is separated from non-SmartDSP-OS symmetrical data in M2 memory.
- SmartDSP-OS-symmetrical constants are separated from non-SmartDSP-OS-symmetrical constants in M2 memory.

The modifications are as follows:

```
unit private (*) {
  MEMORY {
    local_os_data_desc              ("rw"): org = _VirtLocalDataM2_b;
    local_os_rom_desc               ("r"):  AFTER(local_os_data_desc) ;
    local_data_desc                 ("rw"): AFTER(local_os_rom_desc) ;
    local_rom_desc                  ("r"):  AFTER(local_data_desc) ;
    local_data_m3_desc              ("rw"): org = _VirtLocalDataM3_b;
    local_data_m3_nocacheable_desc  ("rw"): AFTER(local_data_m3_desc) ;
    local_data_ddr0_desc            ("rw"): org = _VirtLocalDataDDR0_b + reserved_size;
    local_data_ddr1_nocacheable_desc("rw"): org = _VirtLocalDataDDR1_b + _VTB_size;
  }
...
}
```

The section block for the new memory area appears as:

```
unit shared (*) {
  ...
  SECTIONS {
    descriptor_local_os_data {
      .oskernel_local_data
      LNK_SECTION(att_mmu, "rw", 0x200, 4, ".att_mmu");
      .oskernel_local_data_bss
    } > local_os_data_desc;
    descriptor_local_os_rom {
      .oskernel_rom
    } > local_os_rom_desc;
    descriptor_local_data {
```

```
        .data_m2
        .bss_m2
        .exception_index
        .init_table
        .staticinit
        .bsstab
         reserved_crt_tls
      } > local_data_desc;
      descriptor_local_rom {
        .local_rom
        .rom_init_tables
      } > local_rom_desc;
...
}
```

The address translation block is also updated to keep track of the new memory areas:

```
address_translation (*) {
  local_os_data_desc        (SYSTEM_DATA_MMU_DEF):LOCAL_M2, org = _PhysLocalDataM2_b;
  local_os_rom_desc         (SYSTEM_ROM_MMU_DEF):LOCAL_M2, AFTER(local_os_data_desc);
  local_data_desc           (SYSTEM_DATA_MMU_DEF):LOCAL_M2, AFTER(local_os_rom_desc);
  local_rom_desc            (SYSTEM_ROM_MMU_DEF):LOCAL_M2, AFTER(local_data_desc);
  local_data_m3_desc        (SYSTEM_DATA_MMU_DEF):LOCAL_M3, org = _PhysLocalDataM3_b;
  local_data_m3_nocacheable_desc(SHARED_DATA_MMU_DEF):LOCAL_M3,
AFTER(local_data_m3_desc);
  local_data_ddr0_desc      (SYSTEM_DATA_MMU_DEF):LOCAL_DDR0, org =
_PhysLocalDataDDR0_b + reserved_size;
  local_data_ddr1_nocacheable_desc (SHARED_DATA_MMU_DEF):LOCAL_DDR1, org =
_PhysLocalDataDDR1_b + _VTB_size;
#if  ((USING_USER_KA_STACK == 1) || (USING_RTLIB == 1))
  reserve (SYSTEM_DATA_MMU_DEF): _PhysLocalDataDDR0_b, _VirtLocalDataDDR0_b,
reserved_size, "rw";
#endif
}
```

4.3.2.5 Defining a Private Unit for the Control Subsystem

Now that the memory map has been defined for all system symmetrical and system shared units, it is time to focus on the actual subsystem memory mapping. This is done in `local_map_link.l3k`.

The virtual addresses of the control subsystem's private code and data are defined right after the system symmetrical data. Subsystem private data is placed in M2 memory, whereas subsystem private code is allocated in DDR0.

Section `.local_control_text_p` contains function `per_core_init` and needs to be allocated first in `local_control_text_desc`. Its virtual address must be identical on all the cores.

```
/*  Define Control core local sections */
unit private (task_c0) {
  MEMORY {
    local_control_desc          ("rw"):  AFTER(local_rom_desc);
    local_control_text_desc     ("rx"):  AFTER(local_data_ddr0_desc);
  }

  SECTIONS {
    descriptor_local_data_control {
      .local_control_data
```

```
      .local_control_bss
    } > local_control_desc;
    descriptor_local_text_control {
      .local_control_text_p
      .local_control_text
      .default
    } > local_control_text_desc;
  }
  /* remove code and data from uplink subsystem.*/
  rename "*uplink*.eln", "*", "c1`.exclude";
  /* remove code and data from downlink subsystem.*/
rename "*downlink*.eln", "*", "c1`.exclude";
}

address_translation (task_c0) {
  local_control_desc      (SYSTEM_DATA_MMU_DEF):LOCAL_M2,  AFTER(local_rom_desc);
  local_control_text_desc (SYSTEM_PROG_MMU_DEF):LOCAL_DDR0,AFTER(local_data_ddr0_desc);
}
```

4.3.2.6 Defining the Private Unit for Uplink Subsystem

The virtual addresses of the uplink subsystem's private code and data are defined right after the system symmetrical data. Subsystem private data is placed in M2 memory, whereas subsystem private code is allocated in DDR0.

Section `.local_downlink_text_p` contains function `per_core_init` and needs to be allocated first in `local_downlink_text_desc`. Its virtual address must be identical on all the cores.

```
/*  Define downlink core local sections */

unit private (task_c1) {
  MEMORY {
    local_downlink_desc            ("rw"):  AFTER(local_rom_desc);
    local_downlink_text_desc       ("rx"):  AFTER(local_data_ddr0_desc);
  }
  SECTIONS {
    descriptor_local_data_downlink {
      .local_downlink_data
      .local_downlink_bss
    } > local_downlink_desc;
    descriptor_local_text_downlink {
      .local_downlink_text_p
      .local_downlink_text
      .default
    } > local_downlink_text_desc;
  }

  /* remove code and data from uplink subsystem.*/
  rename "*uplink*.eln", "*", "c0`.exclude";
  /* remove code and data from control subsystem.*/
  rename "*control*.eln", "*", "c0`.exclude";
}

address_translation (task_c1) {
  local_downlink_desc     (SYSTEM_DATA_MMU_DEF):LOCAL_M2,AFTER(local_rom_desc);
  local_downlink_text_desc(SYSTEM_PROG_MMU_DEF):LOCAL_DDR0,AFTER(local_data_ddr0_desc);
}
```

4.3.2.7 Defining a Symmetrical Unit for the Downlink Subsystem

The virtual addresses of the downlink subsystem's private code and data are defined right after the system symmetrical data. Subsystem private data is placed in M2 memory, while subsystem private code is allocated in DDR0.

Section `.local_uplink_text_p` contains function `per_core_init` and needs to be allocated first in `local_uplink_text_desc`. Its virtual address must be identical on all the cores.

```
/*  Define uplink cores local sections */
unit private (task_c2, task_c3, task_c4, task_c5) {
  MEMORY {
    local_uplink_desc              ("rw"):  AFTER(local_rom_desc) ;
    local_uplink_text_desc_p       ("rx"):  AFTER(local_data_ddr0_desc) ;
  }
  SECTIONS {
    descriptor_local_data_uplink {
      .local_uplink_data
      .local_uplink_bss
    } > local_uplink_desc;
    descriptor_local_text_uplink_p {
      .local_uplink_text_p
      .default
    } > local_uplink_text_desc_p;
  }

  /* remove code and data from downlink subsystem.*/
  rename "*downlink*.eln", "*", "c0`.exclude";
  /* remove code and data from control subsystem.*/
  rename "*control*.eln", "*", "c0`.exclude";
}
address_translation (task_c2, task_c3, task_c4, task_c5) {
  local_uplink_desc      (SYSTEM_DATA_MMU_DEF):LOCAL_M2,   AFTER(local_rom_desc);
  local_uplink_text_desc_(SYSTEM_PROG_MMU_DEF):LOCAL_DDR0, AFTER(local_data_ddr0_desc);
}
```

4.3.2.8 Defining a Partially-Shared Unit for the Downlink Subsystem

The virtual addresses for the downlink subsystem's partially shared code are defined right after the system shared data. The subsystem shared code is allocated in DDR0.

```
unit shared (task_c2, task_c3, task_c4, task_c5) {
  MEMORY {
    local_uplink_text_desc   ("rx"):  AFTER(shared_rom_ddr0_cacheable_desc) ;
  }
  SECTIONS {
    descriptor_local_text_uplink {
      .local_uplink_text
    } > local_uplink_text_desc;
  }
}
address_translation (task_c2, task_c3, task_c4, task_c5) {
  local_uplink_text_desc (SYSTEM_PROG_MMU_DEF):SHARED_DDR0,
AFTER(shared_data_ddr0_cacheable_desc);
```

NOTE

The address translation entries in this example target cores 2 to 5 only. This provides memory protection against inadvertent access to cores 0 and 1, which do not have memory locations mapped to this area.

5 Revision History

Table 1. Revision History

Rev. Number	Date	Substantive Change
0	06/17/2011	Public release
1	07/20/2011	Revisions to Figures 1, 4, and 5.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

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

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 China Ltd.
Exchange Building 23F
No. 118 Jianguo339 Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4176
Rev:1
Date: 07/2011