Freescale Semiconductor
Application Note

Document Number: AN4063
Rev. 3, 06/2011

Configuring an Asymmetric Multicore Application for StarCore DSPs

by *Devtech Customer Engineering*
Freescale Semiconductor, Inc.
Austin, TX

An asymmetric multicore DSP application is one in which the processor cores do not execute identical code. Asymmetric applications permit a complex software design to be broken down into smaller, simpler tasks. Code modules written specifically for each task implement its key functions and execute on only the required number of cores. Therefore, an asymmetric multicore design can divide and conquer the application processing requirements by distributing portions of it across the cores and so use the processor resources more effectively.

This paper describes how to configure the memory resources of several example multicore DSP applications properly so that they support asymmetric processing on Freescale StarCore DSPs. The technique described here is specific to the multicore StarCore MSC8156 and its derivatives. Example programs that demonstrate these techniques are available in a software archive and are specific to CodeWarrior for StarCore DSPs v10.1.5 and earlier.

Contents

1 Designing the Application

The most critical aspect of writing a multicore DSP application is to divide up the processor resources properly among the various task modules. In particular, care must be taken in assigning the modules to specific cores and how they use memory. For the purpose of clarity, a processor's complete set of cores and all of the software that executes on them is known as the *system*. A subset of the processor cores and the code that they execute is termed a *subsystem*. Note that the system encompasses all of the subsystems running on the processor. Subsystems exist only as a design scheme in software; there are no physical definitions of a subsystem other than the number of cores it uses.

1.1 Define the Application Memory Map

When partitioning an application into asymmetric subsystems, be aware of the application memory map. Check that each subsystem occupies the appropriate cores and accesses the correct amount of memory. Complicating this process are the different types of memory required to properly share data among other cores or tasks. Figure 1 is a conceptual diagram that depicts the relationship among the different memory types that are available to the multicore application. The system contains six cores, with one subsystem using four cores and the other subsystem using two cores. Private memory is not shown.

Each memory type has specific characteristics and purposes. The types of memory available are:

- System shared memory, which is shared among all of the processor cores
- Subsystem shared memory, which is shared among the cores within a subsystem
- Symmetric memory, which is private to each core, yet is where objects within it reside on the same virtual address for all of the cores in the system
- Subsystem symmetric memory, which is private to each core, yet is where objects reside on the same virtual address for all of the cores in the subsystem
- Private memory

NOTE

The amount of memory reserved for subsystem shared memories can vary.

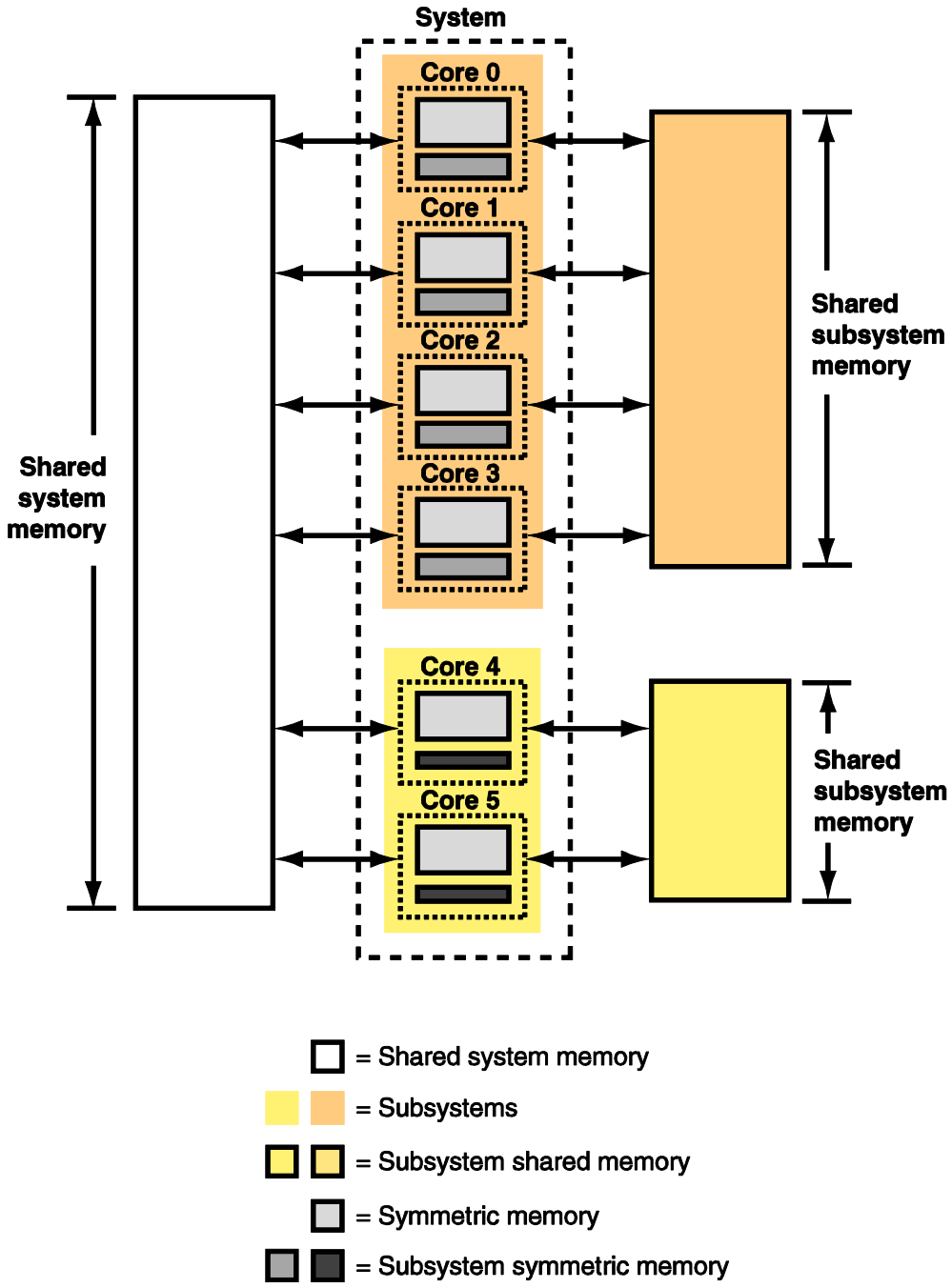

Figure 1. Conceptual Diagram of the Components of a Multicore System, Its Subsystems, and the Memory That It Uses.

It is also important to decide where these types of memory physically reside within the system. The choices are M2, M3, DDR1 or DDR2 memory. These decisions should be based on the allocation size for various objects and any latency requirements. Figure 2 shows the location of on-chip memory for the MSC8156 DSP.

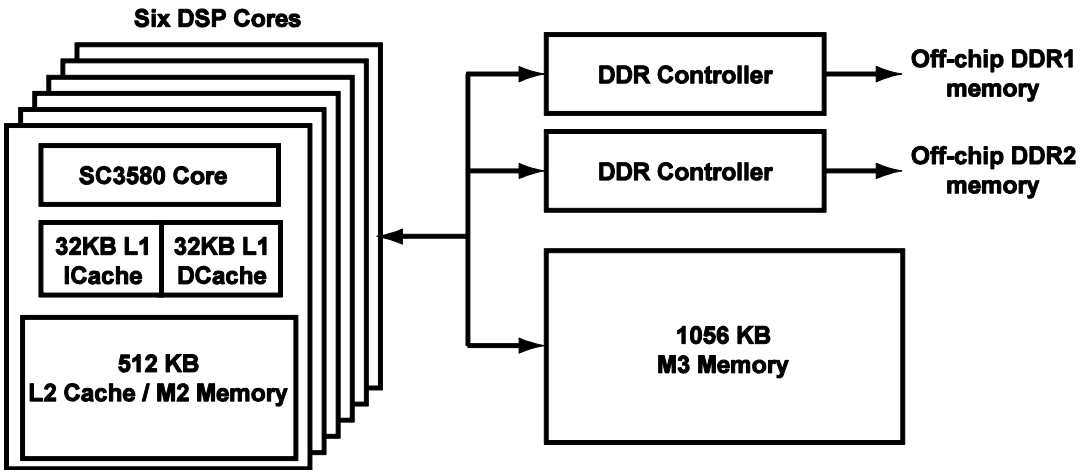

Figure 2. The Location of the Internal Memories M2 and M3 on the MSC8156 DSP. DDR1 and DDR2 Memory is Located Off-chip.

NOTE

On MSC8156 based devices, M2 memory should not be used for shared memory. Deadlocks can occur when two cores attempt to access each other's M2 memory simultaneously.

1.2 Allocating Memory

Begin the application design by carefully considering as to where its modules go into the memory map. The application resources must be partitioned by first identifying those modules that must be shared among the entire system and those modules that are used exclusively by the subsystems.

In general, code that must be shared across the entire system belongs to:

- ANSI library or runtime functions
- Startup code
- SmartDSP OS functions
- User code executed by two or more subsystems.

Therefore, locate these types of code in system shared memory.

Next, subsystem-specific code must be located into the appropriate memory types. The following general guidelines describe how to partition such application code.

1.2.1 Place Symmetric Resources in Symmetric Memory

Symmetric resources are functions or variables that have core-specific or subsystem-specific implementations, yet are referenced from shared code. For the shared code (ANSI library functions, SmartDSP OS functions, boot code and user code) to run correctly, symmetrical resources must be allocated to the same virtual address on all cores. That is, locate these resources in either system symmetric or the relevant subsystem symmetric memory.

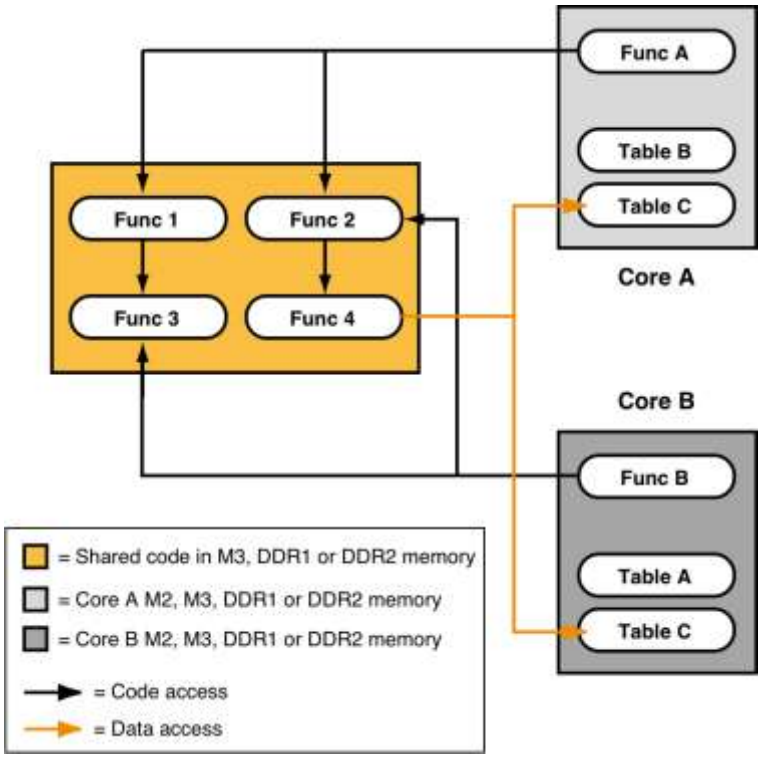

Figure 3. Accessing Symmetrical data from shared code

In the example shown in Figure 3, `Func 2` is invoked by both Core A and Core B. `Func 4` is using data from global variable `Table C`, which has a different definition (value) for each core.

To ensure `Func 2` processes the appropriate table, `Table C` must be allocated at the same virtual address on both cores and address translation must be enabled in the MMU. That is, it must be located in symmetric memory.

1.2.2 Work Within the Linker Allocation Scheme

Keep in mind that the granularity of the linker allocation scheme is based on a file. This has an impact on the application design in that certain parts of the code must reside in certain files.

First, do not mix objects (variables and functions) with different scope (system, subsystem, or core scope) in a single module. Each subsystem should have dedicated modules for its code and variables. In addition, core private functions and variables should have their own dedicated modules.

1.2.3 Build with ICODE Option `Allconst_To_Rom=TRUE`

When the application is built with this option set to `TRUE`, the compiler allocates all constants, string constants, and switch tables in the application's `rom` section instead of the `data` section.

In certain situations the compiler might generate some constants for optimization purposes. This might occur for the following data types:

- Switch tables
- String constants used in the code
- Local array or structure variables defined with initialization values

These constants are created internally by the compiler and need to have the same scope as the code that uses them. For example, if these constants are referenced by shared code, then they need to be visible to all the cores. This is done by placing them in system shared memory. If they are referenced by partially shared code, they need to be made visible to the cores running that code. Therefore, place these constants in subsystem shared memory for the appropriate subsystem.

The ICODE option `Allconst_To_Rom=TRUE` can be specified:

- In the `.appli` file, by adding following command to the current view:

```
Allconst_To_Rom=TRUE
```

- Adding following option to the compiler command line

```
-Xicode "--Allconst_To_Rom=TRUE"
```

1.3 Making Sections Subsystem Specific

The linker basically recognizes two types of sections:

- **Core-specific sections** that are only part of the image of one specific core. The name of a core-specific section is prefixed with "`c?.`" (where ? stands for the core number)
- **General-purpose sections** that are part of the image of each core. All sections whose name does *not* start with "`c?.`" are general-purpose sections.

For subsystem-based applications, it is important to instruct the linker as to which sections should not be linked into a specific subsystem image (`.eld` file).

The linker performs dead stripping when it generates the binary files for each core. If a specific function is used only on subsystem 0, removing it from the image of the cores running other subsystems prevents all objects referenced by this function to be linked to the image.

This technique can reduce the footprint of symmetrical and private memory used on each core.

As the linker does not include native support for partially shared sections, some commands are required in the `.l3k` file to remove some objects defined in general-purpose sections from a core image.

There are two ways to do this: either use the `RENAME` command or `EXCLUDE` command, both of which are described next.

1.3.1 RENAME Command

This command works by file name and relocates objects from a specified set of sections within the specified modules to another section. The syntax of the rename command is:

```
RENAME "fileName", "srcSection", "dstSection"
```

Table 1 describes the purpose of these parameters in detail.

Table 1. RENAME Command Parameters

Parameter	Description

fileName_	Name of the binary file (`.eln` or `.elb`) on which to apply the `RENAME` command. The specified file name can include the wildcard characters * or ?. The file name always starts with a * wildcard character to reflect that the binary file is not located directly in the build directory. Using a * prefix ensures that the rename can be done, no matter where the binary file is stored.
	The filename can refer either directly to an object file (`.eln`) or to an object file inside of a library.
	• Refer directly to an object file using the following notation:
	`"*objFileName"`
	For example,
	`"*startup_*.eln"`
	• Refer to an object file included in a library using the following notation:
	`"*libName(objFileName)"`.
	For example,
	`"*rtlib_*.elb(target_asm_start.eln)"`
	• Refer to all object files included in a library using the following notation:
	`"*libName(*)"`.

srcSection	Name of the section to be renamed. Wildcard characters can be used in the section name. For example, one can specify either: • The section name as it is encoded in the binary file or • Wildcard character `*` to rename all sections within the specified file.
dstSection	New name of the section as is appears in the executable file (.eld). The section name specified here can be a core-specific section or a general-purpose section. No wildcard characters are allowed here. For example • "`c0'.data`" is a core-specific section which is part of core 0 image only. • "`.data`" is a general-purpose section that is part of all core images.

The command `RENAME`, together with a core-specific section as destination section, allows excluding some sections from a subsystem.

For example:

```
unit shared (task0_c0, task0_c1) {...
  RENAME "*sys1_*.eln","*","c2`.exclude"
}
```

The command above tells the linker that when it generates the image for core 0 or 1, objects from all the sections defined in a file whose name contains the string `sys1_` are placed in a section called "`c2`.exclude`". This section is specific to core 2 (section name starts with prefix "`c2`"), so the objects will not be linked to core 0 and core 1 images. When using this command, it is good practice to use a prefix for the name of the files containing code specific to each subsystem.

Also, make sure to remove the section `.default` from the shared code section and place it in the subsystem's partially shared section. If not, the following linker message is displayed:

```
Error: In .unit "c2": The section ".default" (on module "./Source/<fileName>.eln") is
not placed into space "sp0111111" c0 (and c5, c4, c3, c2, c1) <projectName> Unknown
C/C++ Problem
```

1.3.2 EXCLUDE Command

This command excludes a specific section from the subsystem image. For example, the code snippet below excludes sections `.sys1_text_main`, `.sys1_text`, `.sys1_rom`, `.sys1_data` and `.sys1_bss` from the core 0 and 1 images:

```
unit shared (task0_c0, task0_c1) {
  exclude ".sys1_text_main"
  exclude ".sys1_text"
  exclude ".sys1_rom"
  exclude ".sys1_data"
  exclude ".sys1_bss"
}
```

When using this command, several informative messages appear:

```
 [LNK,0,6999,-1]: Information: In .unit "c0": The symbol _<symName> was forced to weak
binding for module ./Source/<fileName>.eln
```

This is normal behavior and can be ignored.

1.4 Accessing Shared Data

1.4.1 Mutual Exclusion

In any multicore application, care must be taken to prevent two or more entities (cores, tasks, etc.) from accessing and/or modifying a shared resource simultaneously. A shared resource might be a variable, a buffer, or a peripheral.

1.4.1.1 Mutual Exclusion On One Core

Mutual exclusion on one core is ensured by disabling interrupts before accessing the resource and enabling them back when the code has finished using the resource.

On SmartDSP-OS-based subsystems, this is done using OS system calls (`osHwiSwiftDisable`, `osHwiSwiftEnable`, `osHwiDisable`, `osHwiEnable`).

NOTE

When implementing a real-time application, the amount of time that interrupts are disabled should remain as small as possible. Disabling interrupts for too long might break the real time behavior of the system.

1.4.1.2 Mutual Exclusion Among Cores

Mutual exclusion among cores can be implemented using spinlocks or software semaphores.

In applications where all cores are running SmartDSP OS, spinlocks OS calls (such as `osSpinLockGet`, `osSpinLockTryGet`, `osSpinLockRelease`, and others) can implement mutual exclusion while accessing shared resources.

NOTE

On MS8156 cores, atomic operations are only possible on M3 memory. Thus all variables used as spinlocks must be allocated in M3.

1.4.2 Data Exchange Among Cores

The application note AN3855 “Multicore Support in SmartDSP OS” provides some information on the exchange of information among tasks or among cores. The section “6 Information Passing” provides more information on the matter. The application note can be downloaded from Freescale web page (www.freescale.com).

2 An Example Asymmetric Program

To move the discussion from theory to the practice, this note describes the configuration of an example multicore application. The application is a system comprised of three subsystems. Each of the subsystems executes a SmartDSP OS application. The three subsystems are implemented as follows:

- Subsystem 0—Uses processor cores 0 and 1. This subsystem creates three tasks that execute at the same priority level and a timer handler. The timer handler calls the SmartDSP OS function `osTaskYield` to force preemption of the tasks in round-robin fashion at each tick. When the third task has been awakened 30 times, the subsystem stops.
- Subsystem 1—Uses cores 2, 3, and 4. This subsystem creates two tasks that use `osTaskDelay` to wait for a specific interval and then perform some processing. The first task waits for 10 ticks and second task waits for five ticks. When second task has awakened from `osTaskDelay` 40 times, the subsystem stops.
- Subsystem 2—Uses core 5. This subsystem creates two tasks and an `EventQueue`. The first task sends data into the queue while second one reads data from this queue. When second task has read five messages from the `EventQueue`, the subsystem stops.

When each subsystem halts, it writes a status message to the console.

2.1 Naming Conventions and Memory Map

For the example application that accompanies this note, Table 2 shows the naming conventions used in the source code to identify whether the resources (either code functions or variables) are shared throughout the system, a particular subsystem, or are private to a specific core.

Table 2: Naming Conventions for the Functions and Variables

Prefix	Description
sys0_	Used on module names which contain objects used on subsystem 0. Also used for global objects that belongs to the subsystem 0 image.
sys1_	Used on module names which contain objects used on subsystem 1. Also used for global objects that belongs to the subsystem 1 image.
sys2_	Used on module names which contain objects used on subsystem 2. Also used for global objects that belongs to the subsystem 2 image.
c0_	Used on all modules that contains objects used only on core 0.
c1_	Used on all modules that contains objects used only on core 1.
c2_	Used on all modules that contains objects used only on core 2.
c3_	Used on all modules that contains objects used only on core 3.
c4_	Used on all modules that contains objects used only on core 4.
c5_	Used on all modules that contains objects used only on core 5.

NOTE

The naming conventions not only help make the code self-documenting, it also makes it easier to exclude code and data from a particular subsystem in the linker file (see section 1.3).

Figure 4 shows the physical memory map of the example asymmetric application. The symbolic names at the right define specific addresses.

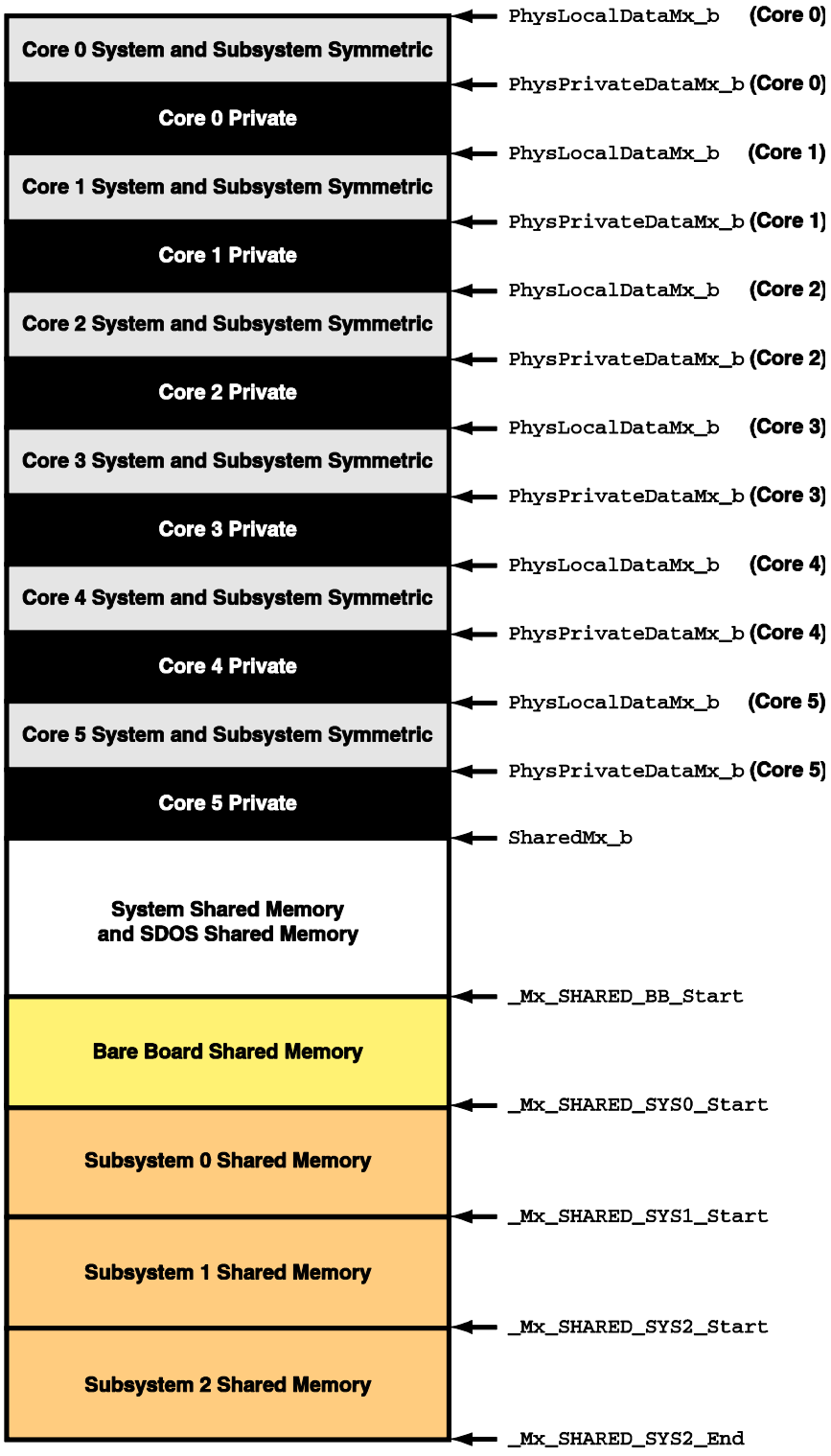

Figure 4. The Memory Map for the Example Multicore Application Described in this Article. Mx Stands for M2, M3, DDR1, and DDR2 Memories (M2 Memory Does Not Include Any Shared Memory Area).

2.2 Configure the OS Objects for Each Subsystem

Now that the structure of the asymmetric application has been determined, the best method of starting the OS objects in each subsystem must be selected. There are two possible configuration options:

- Use a private `main()` function and a subsystem-specific `appInit()` function.
- Use a shared `main()` and `appInit()` function.

The choice of which option is used is a design decision, based on programming usage or guidelines.

2.2.1 Use a Subsystem-Specific `appInit()` Function

The CodeWarrior example projects `Asym_SDOS_code` and `Asym_SDOS_code_private` demonstrate this technique. The code is structured as follows:

- There is a private `main()` function and a dedicated module for each core (`cx_main.c`, where x represents the core number).
- The `main()` function must be located at the same virtual address for each core. So it is allocated first at `_VirtPrivate_M2_b`, `VirtLocalDataM3_b`, `VirtLocalDataDDR0_b` or `VirtLocalDataDDR1_b`, depending where code is allocated. The sample projects specified above place the main functions in M2 memory. The start addresses mentioned above are all defined in file `memory_map_link.l3k`.
- Each `main()` function next calls a subsystem-specific `appInit()` function. For the example code, these functions are `sys0_appInit()`, `sys1_appInit()`, and `sys2_appInit()`, for subsystem 0, subsystem 1, and subsystem 2, respectively.
- The creation of subsystem-specific OS objects is statically coded into each `appInit()` function.

Figure 5 shows conceptually this initialization sequence.

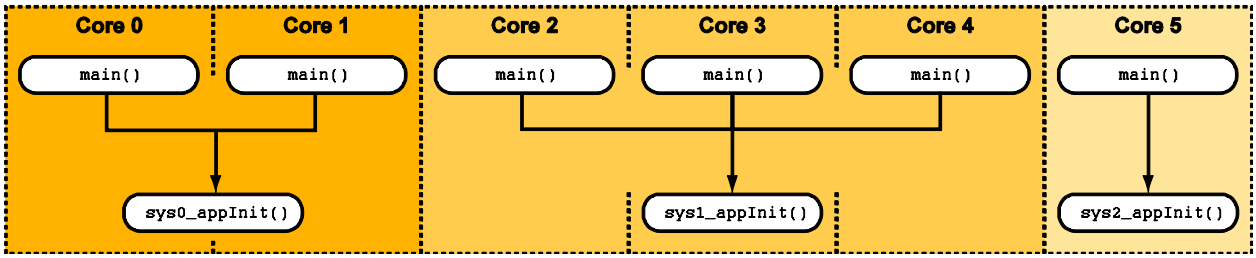

Figure 5. Conceptual Diagram of How the main() Functions Invoke the Subsystem-specific appInit() Function in the Example Asymmetric Application.

2.2.2 Use Shared `main()` and `appInit()` Functions

This initialization scheme is implemented in the CodeWarrior example project `AsymCodeSDOS_SharedMain`. The initialization code is organized as follows:

- The `main()` and `appInit()` functions are shared by all cores. They are implemented in the module `msc8156_main.c`.
- Information about the tasks that need to be created for a specific core is stored in a data structure

termed the TaskTable. This table is private for each core, and contains information relevant to the creation of the various tasks.

- The TaskTable must be located at the same virtual address on each core. So it needs to be placed first at _VirtPrivate_M2_b, VirtLocalDataM3_b, VirtLocalDataDDR0_b or VirtLocalDataDDR1_b, depending where the data is allocated
- The appInit() function uses information encoded in the TaskTable to create the required objects. For the example program, the TaskTable data is defined as follows:

```
typedef struct _TaskEntryStruct{
  /* Task Handle returned by the osTaskCreate call.. */
  os_task_handle   taskHandle;
  / This function runs when the task is activated. */
  os_task_function taskFunction;
  /*Function used to init. the task and related os object. */
  TaskCreateFunc   taskCreateFunction;
  /* Top of task's stack. */
  uint32_t         top_of_stack;
  /* The size of the above space. */
  uint32_t         stack_size;
  /* Task priority */
  os_task_priority task_priority;
  /* Task name- to identify the task in the Kernel Awareness window */
  char             *task_name;
}TaskEntryStruct;
```

- A loop creates the required OS objects by cycling through the contents of TaskTable. It parses the TaskTable's structure and for each table element retrieved, it invokes the corresponding SmartDSP OS taskCreatFunction. An element with a stackSize of zero marks the end of the table. The logic in appInit() is implemented as follows:

```
status = OS_SUCCESS;
taskCnt = 0;
while ((TaskTable[taskCnt].stack_size != 0) &&
       (status == OS_SUCCESS)) {
  status = TaskTable[taskCnt].taskCreateFunction
           (&TaskTable[taskCnt]);
  taskCnt++;
}
```

- The shared CreateTask function found in msc8156_main.c is responsible for creating most of the tasks. However, a subsystem-specific taskCreateFunction handles the creation of each core's last task, where the subsystem requires additional resources. For instance, in subsystem 0's example code, there is a sys0_CreateTaskandTimer function. It appears in the file sys0_code.c, and it creates the osTimer that subsystem 0 requires to operate properly.

3 Modifying a Wizard Created Project

When the project was created by the CodeWarrior wizard, the following steps are required to adjust the project to support an asymmetric SmartDSP OS application.

1. Define unique sections for the application (section 4.1 below).
2. Place the objects into sections (section 4.2 below).
3. Define the system task for each core (see section 5.1).
4. Define the application layout (section 5.2 to 5.4).
5. Combine sections into MMU table descriptors (See sections 5.5 to 5.7),
6. Exclude items that must be present on some cores but not others. (See sections 5.5 to 5.7).

Each of the steps above is described below.

4 Configuration Required for the Compiler

Before the code and data objects can be distributed according to the application requirements, it is necessary to define all of the memory sections. The functions, variables, and constants are then placed into the appropriate sections. One of the purposes of the application configuration file (`.appli`) is to specify these definitions and the distribution of objects within them. The sections that follow describe how this is done. An alternate method, using `pragma` and `__attribute__` modifiers, can also be used to perform this setup. See section 6.5 below for more information on this technique.

4.1 Defining Sections

To arrange the program data and code so that the linker can place these resources into the proper areas of memory, sections must be specified. 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 elements. The information that follows explains the section definitions that the linker uses to map the application elements to specific memory addresses.

4.1.1 Define the Application's Code Sections

Sections should first be defined for:

- Shared code
- Partially shared code on each subsystem
- Private code for each core

If the application is implemented with a subsystem-specific `appInit()` function (see section 2.2.1 above), there must be a dedicated private code section for every `main()` function. Since `main()` is called from startup code, it must be located on the same virtual address for all of the cores. That is, `main()` should be placed in symmetric memory.

In the `.appli` file, these definitions appear as follows:

```
section
  program = [
```

```
/* Core specific private code sections */
Entry_c0_text          : ".entry_m2_private_text" core="c0",
Entry_c1_text          : ".entry_m2_private_text" core="c1",
Entry_c2_text          : ".entry_m2_private_text" core="c2",
Entry_c3_text          : ".entry_m2_private_text" core="c3",
Entry_c4_text          : ".entry_m2_private_text" core="c4",
Entry_c5_text          : ".entry_m2_private_text" core="c5",
Text0                  : ".text" core="c0",
Text1                  : ".text" core="c1",
Text2                  : ".text" core="c2",
Text3                  : ".text" core="c3",
Text4                  : ".text" core="c4",
Text5                  : ".text" core="c5",
/* Sub-system specific shared code sections */
PgmSYS0                : ".sys0_text",
PgmSYS1                : ".sys1_text",
PgmSYS2                : ".sys2_text",
SYS0_DDR0_shared_text  : ".sys0_ddr0_cacheable_shared_text",
SYS1_DDR0_shared_text  : ".sys1_ddr0_cacheable_shared_text",
SYS2_DDR0_shared_text  : ".sys2_ddr0_cacheable_shared_text",
SYS0_DDR1_shared_text  : ".sys0_ddr1_cacheable_shared_text",
SYS1_DDR1_shared_text  : ".sys1_ddr1_cacheable_shared_text",
SYS2_DDR1_shared_text  : ".sys2_ddr1_cacheable_shared_text",
/* Shared code sections */
M3_shared_text         : ".m3_cacheable_shared_text",
DDR0_shared_text       : ".ddr0_cacheable_shared_text",
DDR1_shared_text       : ".ddr1_cacheable_shared_text",
DefaultPgm             : ".text"
```

4.1.2 Define the Application Initialized Data Sections

Next, define the initialized data sections for:

- Shared data between all cores
- Partially shared data on each subsystem
- Symmetric data for all cores
- Symmetric data for each subsystem
- Private data for each core

If the application is implemented with shared `appInit()` function (see section 2.2.2), there must be a dedicated private data section for the task table array. Since `TaskTable` is referenced from shared `appInit()` function, it must be located on the same virtual address for all of the cores. That is, `TaskTable` should be placed in symmetric memory.

These declarations appear in the `.appli` file as:

```
section
...
  data = [
    /* Shared data sections */
    /* Core-specific private data sections */
     Data0          : ".data" core="c0",
     Data1          : ".data" core="c1",
     Data2          : ".data" core="c2",
```

```
        Data3          : ".data" core="c3",
        Data4          : ".data" core="c4",
        Data5          : ".data" core="c5",
        c0_task_table  : ".task_table" core="c0",
        c1_task_table  : ".task_table" core="c1",
        c2_task_table  : ".task_table" core="c2",
        c3_task_table  : ".task_table" core="c3",
        c4_task_table  : ".task_table" core="c4",
        c5_task_table  : ".task_table" core="c5",
        SharedData     : ".sharedData", /* Shared data for all cores */
       /* Subsystem specific shared data sections */
        SDataSYS0      : ".sys0_sharedData", /*shared data sub-system 1*/
        SDataSYS1      : ".sys1_sharedData", /*shared data sub-system 2*/
        SDataSYS2      : ".sys2_SharedData", /*shared data sub-system 3*/
       /* Symmetrical data sections */
        DefaultData    : ".data"  /* symmetric data for all cores */
       /* Sub-system specific symmetrical data sections */
        DataSYS0       : ".sys0_data", /* symmetric data sub-system 0 */
        DataSYS1       : ".sys1_data", /* symmetric data sub-system 1 */
        DataSYS2       : ".sys2_data", /* symmetric data sub-system 2 */
]
```

4.1.3 Define the Application Uninitialized Data Sections

For uninitialized data, data sections must be defined for:

- Shared data among all cores
- Partially shared data among each subsystem
- Symmetric data for all cores
- Symmetric data for each subsystem
- Private data for each core

This is accomplished by placing the following declarations in the `.appli` file:

```
section
...
  bss = [
   /* Shared bss sections */
    SharedBss : ".sharedBss", /* Shared bss for all cores */
   /* Core-specific private bss sections */
    Bss0       : ".bss" core="c0",
    Bss1       : ".bss" core="c1",
    Bss2       : ".bss" core="c2",
    Bss3       : ".bss" core="c3",
    Bss4       : ".bss" core="c4",
    Bss5       : ".bss" core="c5",
   /* Subsystem-specific shared bss sections */
    SBssSYS0   : ".sys0_sharedBss", /* shared bss sub-system 0*/
    SBssSYS1   : ".sys1_sharedBss", /*shared bss sub-system 1 */
    SBssSYS2   : ".sys2_sharedBss", /*shared bss sub-system 2 */
   /* Symmetrical bss sections */
    DefaultBss : ".bss"  /* symmetric bss for all cores */
   /* Sub-system specific symmetrical bss sections */
    BssSYS0    : ".sys0_bss", /* symmetric bss sub-system 0 */
    BssSYS1    : ".sys1_bss", /* symmetric bss sub-system 1 */
```

```
    BssSYS2     : ".sys2_bss", /* symmetric bss sub-system 2 */
]
```

4.1.4 Define the Application Constants Sections

For this type of data, sections should be defined for:

- Shared constants among all cores
- Partially shared constants on each subsystem
- Private constants for each core

The following definitions in the `.appli` file specify this constant data:

```
section
...
  rom = [
      /* Core-specific private rom sections */
       Rom0       : ".rom" core="c0",
       Rom1       : ".rom" core="c1",
       Rom2       : ".rom" core="c2",
       Rom3       : ".rom" core="c3",
       Rom4       : ".rom" core="c4",
       Rom5       : ".rom" core="c5",
     /* Subsystem-specific shared rom sections */
      RomSYS0     : ".sys0_rom", /* Shared const on sub-system 0 */
      RomSYS1     : ".sys1_rom", /* Shared const on sub-system 1 */
      RomSYS2     : ".sys2_rom", /* Shared const on sub-system 2 */
     /* Shared rom sections */
      DefaultRom : ".rom"    /* const shared by all cores */
]
```

4.2 Place Functions/Variables Into the Appropriate Sections

Now that the sections have been defined, `.appli` file commands populate them with variables and constants, based on the module where these resources were defined or implemented. The procedure for doing this, starting with general cases and then narrowing to more specific situations, is described in further detail in the following sections.

4.2.1 Define the Default Allocation Scheme

First, define the default allocation scheme. To do this, place all functions in the `.text` section, all variables into `.data`, all uninitialized variables into `.bss`, and all constants into `.rom` using the following definitions in the `.appli` file:

```
program = DefaultPgm
data    = DefaultData
rom     = DefaultRom
bss     = DefaultBss
```

4.2.2 Allocate Functions/Variables for Each Subsystem

Next, allocate the code and variables for each subsystem, again using commands in the `.appli` file.

For each module that contains functions running on subsystem 0 or data used by subsystem 0 code, define a module-specific allocation scheme as follows:

```
module "sys0_code" [
  rom     = RomSYS0
  program = PgmSYS0
  bss     = BssSYS0
  data    = DataSYS0
]
```

For functions and variables used by subsystem 1, the module-specific declarations are:

```
module "sys1_code" [
   rom     = RomSYS1
   bss     = BssSYS1
   data    = DataSYS1
   program = PgmSYS1
]
```

For functions and variables used by subsystem 2, the module-specific declarations in the `.appli` file become:

```
module "sys2_code" [
   rom     = RomSYS2
   bss     = BssSYS2
   data    = DataSYS2
   program = PgmSYS2
]
```

4.2.3 Allocate Each Core Private Functions/Variables

For each module that contains core private functions or core private data, define a module-specific allocation scheme as follows. The example commands presented here are for core 0. However, similar notation can specify any core within the system:

```
module "c0_code" [
   rom     = Rom0
   bss     = Bss0
   data    = Data0
   program = Text0
]
```

If the application is implemented with subsystem-specific `appInit()` function (see section 2.2.1 above), only the module containing the implementation of the `main()` function has a special allocation scheme:

```
module "c0_main" [
    rom = Rom0
    bss     = Bss0
    data    = Data0
    program = Entry_c0_text
 ]
```

NOTE

To prevent linker problems, observe the following guidelines:

- Specify a `program` section for each module. Even if the module does not include any code, a `program` section must be present. If a `program` section in not assigned to a module, following linker messages appear:

```
Error: In .unit "c2": symbol "TextEnd_<fileName>"
undefined in ./Source/<fileName>.eln
Error: In .unit "c2": The section ".text" (on module
"./Source/<fileName>.eln") is not placed into space
"sp0111111" c0 (and c5, c4, c3, c2, c1
```

- Make sure the specified `rom`, `bss`, `data`, and `program` sections have similar scope. If the module includes core-specific code, make sure that the associated `bss`, `data`, and `rom` sections also have core scope. If the module includes subsystem-specific code, make sure that the associated `bss`, `data`, and `rom` sections have subsystem scope.

4.2.4 Placing a Resource in a Dedicated Section

There will be situations where some specific variable/constants and functions must be allocated in a section that is different from the default section associated with the module. This can be done using the `place` command in the `.appli` file. For example:

```
place (_sys0_tab)  in  SDataSYS0
```

Provided that the variable `sys0_tab` is defined in module `sys0_code.c`, the above command allocates the variable `sys0_tab` in section `.sys0_sharedData` instead of `.sys0_data`. That is, the variable `sys0_tab` appears in subsystem 0's shared memory instead of its symmetric memory.

This command works fine as long as the default section for the module and the new section planned to contain the resource have the same scope (either system, subsystem, or core private).

5 Configuration Required for the Linker

With the application's configuration described in detail to the compiler, now it is time to consider what information must be supplied to the linker so it can locate all of the application's elements into the appropriate memory areas. These descriptions are provided to the linker via linker command (`.l3k`) files.

To recap, the example application consists of three subsystems. The design calls for subsystem 0 to execute on cores 0 and 1, subsystem 1 executes on cores 2, 3, and 4, while subsystem 2 executes on core 5. For each subsystem, memory must be reserved on M2, M3, DDR1, and DDR2. The sections that follow describe how this is done.

5.1 Define the System Tasks for Each Core

Now the basic system tasks for each core must be defined to the linker. To this end, add the following commands to the file named `os_msc815x_link.l3k`:

```
tasks {
  c0: task0_c0, 0, 0,0;
  c1: task0_c1, 0, 0,0;
  c2: task0_c2, 0, 0,0;
  c3: task0_c3, 0, 0,0;
  c4: task0_c4, 0, 0,0;
  c5: task0_c5, 0, 0,0;
}
```

NOTE

When the linker file does not contain any tasks block, the default name of the system tasks created by linker per each core is `task0_cX` , is where X stands for the core number (0, 1...).

5.2 Define the Symbols that Map to M2

Note that the amount of symmetric data might vary for each subsystem, according to its needs.

The symbols that specify the start address, end address, and size for core private data must be defined. This is done in the linker file `memory_map_link.l3k`, using the following commands::

```
// Virtual local memory definitions (the same for all cores)
// This is where we load system and subsystem symmetric sections
 _VirtLocalDataM2_b = _M2Global_b;
 _VirtLocalDataM2_e = (_VirtLocalDataM2_b+ LocalDataM2_size -1);

// Virtual private memory definitions (the same for all cores)
// This is where we load core specific sections
 _VirtPrivate_M2_b = _VirtLocalDataM2_e + 1;
 _VirtPrivate_M2_e= _VirtPrivate_M2_b + _PrivateM2_size -1;
```

5.3 Define the Symbols for Memory Blocks

In the linker file `memory_map_link.l3k`, define the symbols that describe each subsystem's partially shared memory block. These symbols should define the block's start address, end address, and its size. Because the memory is shared, these blocks should reside in M3. The same approach can be used to specify blocks that occupy DDR1 and DDR2 memory.

This configuration is specified in the file `memory_map_link.l3k`:

```
_M3_SHARED_SYS0_SIZE   = 0x1000;
_M3_SHARED_SYS1_SIZE   = 0x10000;
_M3_SHARED_SYS2_SIZE   = 0x10000;

_M3_SHARED_SYS0_Start =  _M3_SHARED_end + 1;
_M3_SHARED_SYS0_End   =  _M3_SHARED_SYS0_Start +  _M3_SHARED_SYS0_SIZE -1;

_M3_SHARED_SYS1_Start =  _M3_SHARED_SYS0_End + 1;
_M3_SHARED_SYS1_End   =  _M3_SHARED_SYS1_Start +  _M3_SHARED_SYS1_SIZE -1;
```

```
_M3_SHARED_SYS2_Start =  _M3_SHARED_SYS1_End + 1;
_M3_SHARED_SYS2_End   =  _M3_SHARED_SYS2_Start +  _M3_SHARED_SYS2_SIZE -1;
```

Notice how some of these definitions correspond to the memory locations depicted in Figure 4. Once these definitions are complete, the symbol `_M3_SHARED_end` must be adjusted to avoid any overlap:

```
_M3_SHARED_end = _M3_SHARED_start+_M3_size - _M3_SHARED_SYS0_SIZE -
_M3_SHARED_SYS1_SIZE- M3_SHARED_SYS2_SIZE-(_NUMBER_OF_CORES * _PRIVATE_M3_DATA_size)-1;
```

NOTE

For this example application, the decision was made at the start as to how much shared memory must be dedicated to each subsystem. This can usually be determined roughly after application design, according to the complexity of each subsystem. Also, determining the amount of shared memory allowed for each subsystem in advance makes system integration easier when different teams are implementing each subsystem.
If a clear separation between the various subsystem memories is not required, the partially shared memory sections can be allocated along with the system shared sections. Refer to section 6.6 for more information on this technique.

5.4 Define the Physical Memory Range for Partially Shared Memory

The CodeWarrior StarCore Project wizard generates a file named `os_msc815x_link.l3k` that defines physical memory blocks available as shared memory and located in M3, DDR1, and DDR2. The definitions of these memory blocks are based upon the value in `Sharedxxx-size`. The definitions of partially shared memory blocks must be added as follows:

```
physical_memory shared  ( * ) {
  SHARED_M3  :    org = _SharedM3_b,   len = _SharedM3_size;
  SHARED_DDR0:    org = _SharedDDR0_b, len = _SharedDDR0_size;
  SHARED_DDR1:    org = _SharedDDR1_b, len = _SharedDDR1_size;
  SHARED_M3_SYS0: org = _M3_SHARED_SYS0_Start, len = _M3_SHARED_SYS0_SIZE;
  SHARED_M3_SYS1: org = _M3_SHARED_SYS1_Start, len = _M3_SHARED_SYS1_SIZE;
  SHARED_M3_SYS2: org = _M3_SHARED_SYS2_Start, len = _M3_SHARED_SYS2_SIZE;
}
```

5.5 Define the Memory Map for Partially Shared Memory

Now a memory section that is to be shared among a subset of the cores must be defined. This is the subsystem shared memory, and the subsystem partially shared code and constants are placed in it. Furthermore, this section must exclude the partially shared resources that belong to the other two subsystems. The following commands are used for these definitions:

```
unit shared (task0_c0, task0_c1) {
    memory {
        m3_shared_text_SYS0 ("rx"): org = _M3_SHARED_SYS0_Start;
        m3_shared_const_SYS0 ("rw"): AFTER(m3_shared_text_SYS0);
        }
        sections{
```

```
            shared_code_SYS0{
                 ".sys0_text"
                 .default
            } > m3_shared_text_SYS0;
            shared_rom_SYS0 {
              ".sys0_rom"
              .=align (4);
           _endOfSYS0Shared =.;
            } >m3_shared_const_SYS0;
        }
        // exclude all sections which contains sub-system 2 code and data
        // All modules containing code or data specific to sub-system 2 have
        // name starting with sys2_
  RENAME "*sys2_*.eln","*","c5`.exclude"

        // exclude all sections which contains sub-system 1 code and data
        // All modules containing code or data specific to sub-system 1 have
        // name starting with sys1_
  RENAME "*sys1_*.eln","*","c2`.exclude"
}
```

Finally, the physical-to-virtual mapping for the partially shared sections must be defined:

```
address_translation  (task0_c0, task0_c1)  {
  m3_sh_text_SYS0 (SYSTEM_PROG_MMU_DEF): SHARED_M3_SYS0, org = _M3_SHARED_SYS0_Start;
  m3_sh_const_SYS0 (SYSTEM_DATA_MMU_DEF): SHARED_M3_SYS0;
}
```

5.6 Define the Memory Map for Partially Symmetric Memory

A section of memory that must be symmetric on all cores running a particular subsystem must be defined. That is, subsystem's symmetric memory is defined in this step. The commands that follow define the symmetric memory for subsystem 0. These commands appear in the link file `system0.l3k`:

```
unit private (task0_c0, task0_c1) {
    memory {
        m2_SYS0_data ("rw"): AFTER(local_data_descriptor);
        }
         sections{
            sys0data{
              ".sys0_data"
              ".sys0_bss"
            } > m2_SYS0_data;
        }
}
address_translation  (task0_c0, task0_c1) {
    m2_SYS0_data (SYSTEM_DATA_MMU_DEF): LOCAL_M2;
}
```

Equivalent commands can be used to configure symmetric memory for the other subsystems.

5.7 Define the Memory Map for Core Private Data

Now the sections that are private to each core must be defined. Allocation sequences within the block depend whether or not the application uses a task table to initialize the OS objects. If the application is implemented with subsystem-specific `appInit()` function (see section 2.2.1 above), make sure that the `main()` function is allocated first in private memory (M2. M3, DDR1 or DDR2). In that case, the core private unit is defined as described below.

The following code is an example definition for core 0. Similar definitions are required for the other cores.

```
unit private (task0_c0) {
   memory {
      private_text_0 ("rx"): org = _VirtPrivate_M2_b;
      private_data_0 ("rw"): AFTER(m2_private_text_0);
   }
   sections{
      privateCode{
         "c0`.text_main"
         "c0`.text"
      } > private_text_0;
      privateData{
         "c0`.data"
         "c0`.bss”
         "c0`.rom"
      } > private_data_0;
   }
}

address_translation  (task0_c0) {
    private_text_0 (SYSTEM_PROG_MMU_DEF): PRIVATE_M2;
    private_data_0 (SYSTEM_DATA_MMU_DEF): PRIVATE_M2;
}
```

If the application is implemented with shared `main()` and `appInit()` functions (see section 2.2.2 above), make sure the task table is allocated first in private memory. In that case, the core private unit is defined as described below.

The following is an example for core 0. Similar definitions are required for the other cores.

```
unit private (task0_c0) {
   memory {
      private_data_0 ("rw"): org = _VirtPrivate_M2_b;
      private_text_0 ("rx"): AFTER(m2_private_data_0);
   }

   sections{
      privateData{
         "c0`.task_table"
         "c0`.data"
         "c0`.bss"
         "c0`.rom"
      } > private_data_0;
      privateCode{
         "c0`.text"
      } > private_text_0;
```

```
    }
}

address_translation  (task0_c0) {
    private_data_0 (SYSTEM_DATA_MMU_DEF): PRIVATE_M2;
    private_text_0 (SYSTEM_PROG_MMU_DEF): PRIVATE_M2;
    }
```

6 Additional Topics

This section covers items that do not quite fit into a specific category related to the compiler or linker. However, they must be dealt with to properly configure various elements of the application.

6.1 .unlikely Sections

The StarCore C/C++ compiler supports the keyword `unlikely`, which provides the compiler with branch prediction information. Inside of a `switch/case` construct, or in an `if` / `else` block, code block that are rarely executed can be marked as unlikely. When the the `unlikely` keyword is specified for a block of code in the application, the compiler moves it to the `.unlikely` section. The unlikely section can then be placed in slow access memory, leaving more fast access memory available for critical code.

The `.unlikely` section is a general purpose section (that is, it is part of each core image). If there is unlikely code present in any partially shared functions, an `.unlikely` section with subsystem scope must be specified for each subsystem where such code is defined.

For instance, suppose that there is an unlikely code bock present in subsystem 0, which is shared between core 0 and core 1. To prevent the `.unlikely` sections from modules `sys0_*.c` from being linked with the other cores, the following line must be added to the `.l3k` file:

```
unit private (task0_c0, task0_c1){
  RENAME "*sys0_*.eln", ".unlikely", ".sys0_unlikely"
...
}
```

The section `.sys0_unlikely` needs to be placed accordingly in the subsystem shared unit.

If there is unlikely code present in the program's private functions, an `.unlikely` section with core scope must be defined for each core where the unlikely code resides. For example, suppose that the core 0 private code has an unlikely code block. To specify that the `.unlikely` section from module `c0_*.c` should not be linked to the other cores, add following line to the appropriate `.l3k` file:

```
unit private (task0_c0){
  RENAME "*c0_*.eln", ".unlikely", "c0`.unlikely"
...
}
```

The section `c0'.unlikely` needs to be placed accordingly in the core private unit.

6.2 Add Tasks Common to all Subsystems

Use the following procedures to add tasks that need to execute on all cores, or a subset of them (subsystems);

- For each task, add its modules to the application where the tasks and any related global variables are defined.
- Be sure to allocate the code in the new modules to the shared code section (`.text`) and that the variables are allocated to the symmetric `.data` or `.bss` sections. No special allocation commands are required in the `.appli` file. The default allocation scheme is sufficient.
- If the application uses a subsystem-specific `appInit()` function as described in section 2.2.1, first write a function that creates the OS objects that the task(s) require. The function is called by every `appInit()` and therefore needs to be in shared memory.
- If the application uses a shared `appInit()` function as described in section 2.2.2, just add the tasks to all of the core's `TaskTable`s and associate the corresponding `taskCreateFunction` for each core.

For further information, inspect the code in the module `commontask.c` in the CodeWarrior projects `Asym_SDOS_priv_code` or `Asym_SDOS_SharedMain`.

6.3 Add Core-Specific Tasks

For those tasks that execute on one core only, proceed as follows:

- For each task, add its modules to the application where the tasks and any related global variables are defined.
- Allocate the code from these modules to the core private code section (`c0'.text`), and allocate any variables it has to the private `data` or `bss` sections (`c0'.data` or `c0'.bss` respectively). In the `.appli` file, the allocation scheme should be as follows:

```
module "c0_code" [
    program = Text0
    data    = Data0
    rom     = Rom0
    bss     = Bss0
]
```

- If the application uses a subsystem-specific `appInit()` function as described in section 2.2.1, first write a function that creates the OS objects that the task(s) require. This function should be called from the `appInit()` function for the subsystem running on that core. An empty implementation of that function must be present on the other cores that run the same subsystem code.
- If the application uses a shared `appInit()` function as described in section 2.2.2, just add the tasks the core's `TaskTable` and associate the corresponding `taskCreateFunction`.

For further information, inspect the code in the module `c0_code.c` in the CodeWarrior projects `Asym_SDOS_priv_code` or `Asym_SDOS_SharedMain`.

6.4 Handling a Different Number of .bss Sections

Occasionally when adding core-specific tasks to an asymmetric application, there is the possibility of not having the same number of `.bss` sections for all of the cores. The linker message `''Inconsistent address for _bss_count''` appears when this occurs.

The workaround for this problem is to add the following command to the `privateData` section within the private unit that is associated with the cores reporting the problem:

```
LNK_SECTION(bss, "rw", 0x10, 4, ".dummy_bbs1");
```

This issue is fixed in linker V3.0.43 (Compiler build 23.11.1.12) and higher. No `LNK_SECTION` command is required after that release.

6.5 Alternate Allocation Scheme

In section 4.2 it was explained how to place variables or functions inside of different sections using an application configuration (`.appli`) file. As an alternative, this can also be accomplished using `pragma`s and `__attribute__` modifiers.

NOTE

There is one limitation when using this method. In order to define a core private allocation scheme, or to place a variable in a specific core private section, an `.appli` file must be used. The `pragma` or `__attribute__` modifiers do *not* support this capability.

6.5.1 Defining a Default Allocation Scheme for a Module

The default allocation scheme for a module can be defined using the pragmas `pgm_seg_name`, `data_seg_name`, `bss_seg_name` and `rom_seg_name`.

For example, placing the following pragmas at the beginning of the file `sys1_code.c`:

```
#pragma pgm_seg_name  ".sys1_text"
#pragma data_seg_name ".sys1_data"
#pragma bss_seg_name  ".sys1_bss"
#pragma rom_seg_name  ".sys1_rom"
```

This is equivalent to the following notation in the `.appli` file (the following are just a collection of fragments extracted from the `.appli` file):

```
section
  program = [
    PgmSYS1      : ".sys1_text",
  ]
  data = [
    DataSYS1     : ".sys1_data",
  ]
  rom = [
    RomSYS1     : ".sys1_rom",
  ]
  bss = [
    BssSYS1     : ".sys1_bss",
```

```
  ]

module "sys1_code" [
  rom     = RomSYS1
  bss     = BssSYS1
  data    = DataSYS1
  program = PgmSYS1
]
```

Keep in mind that there can be only one pragma `pgm_seg_name`, `data_seg_name`, `bss_seg_name` and `rom_seg_name` per source file.

NOTE

Even if the module contains only data or constant definitions with subsystem scope, make sure to define a pragma `pgm_seg_name` for a section with same scope to avoid any linker issues. If a `pgm_seg_name` is not associated to a module, the following linker messages appear:

```
Error: In .unit "c2": symbol "TextEnd_<fileName>" undefined
in ./Source/<fileName>.eln
Error: In .unit "c2": The section ".text" (on module
"./Source/<fileName>.eln") is not placed into space
"sp0111111" c0 (and c5, c4, c3, c2, c1
```

6.5.2 Placing a Variable into a Dedicated Section

If a specific variable or constant must be allocated in a section that is different from the default section in the module where the object is defined, use the `__attribute__` modifier. For example, to specify that the variable `sys1_tab` should be allocated in section `.sys1_sharedData`, define it as follows:

```
int sys1_tab[SIZE] __attribute__((section(".sys1_sharedData")));
```

This is equivalent to the following command in an `.appli` file (the following code is just a collection of pieces extracted from the `.appli` file):

```
section
  data = [
    SDataSYS1 : ".sys1_sharedData",
  ]

place (_sys1_tab)  in  SDataSYS1
```

Here again this technique works as long as the default section for the module and the new section where the object is to go have the same scope (system, subsystem, or private).

6.6 Partially Shared Areas

In the supplied example application, the decision was made to define a well-separated memory area for each of the subsystem's partially shared code and data sections (Figure 6). This technique can be used when the designer has a clear picture of how to partition the memory between the different subsystems. This technique is also recommended when there are different teams working on each subsystem. A well defined memory partitioning minimizes linking issues when all the pieces are integrated together.

If the amount of memory required for each subsystem's partially shared code and data cannot be known in advance, alternatively one block of physical memory can be allocated for shared memory. The partially shared code and data can then be allocated along with the system shared code and data.

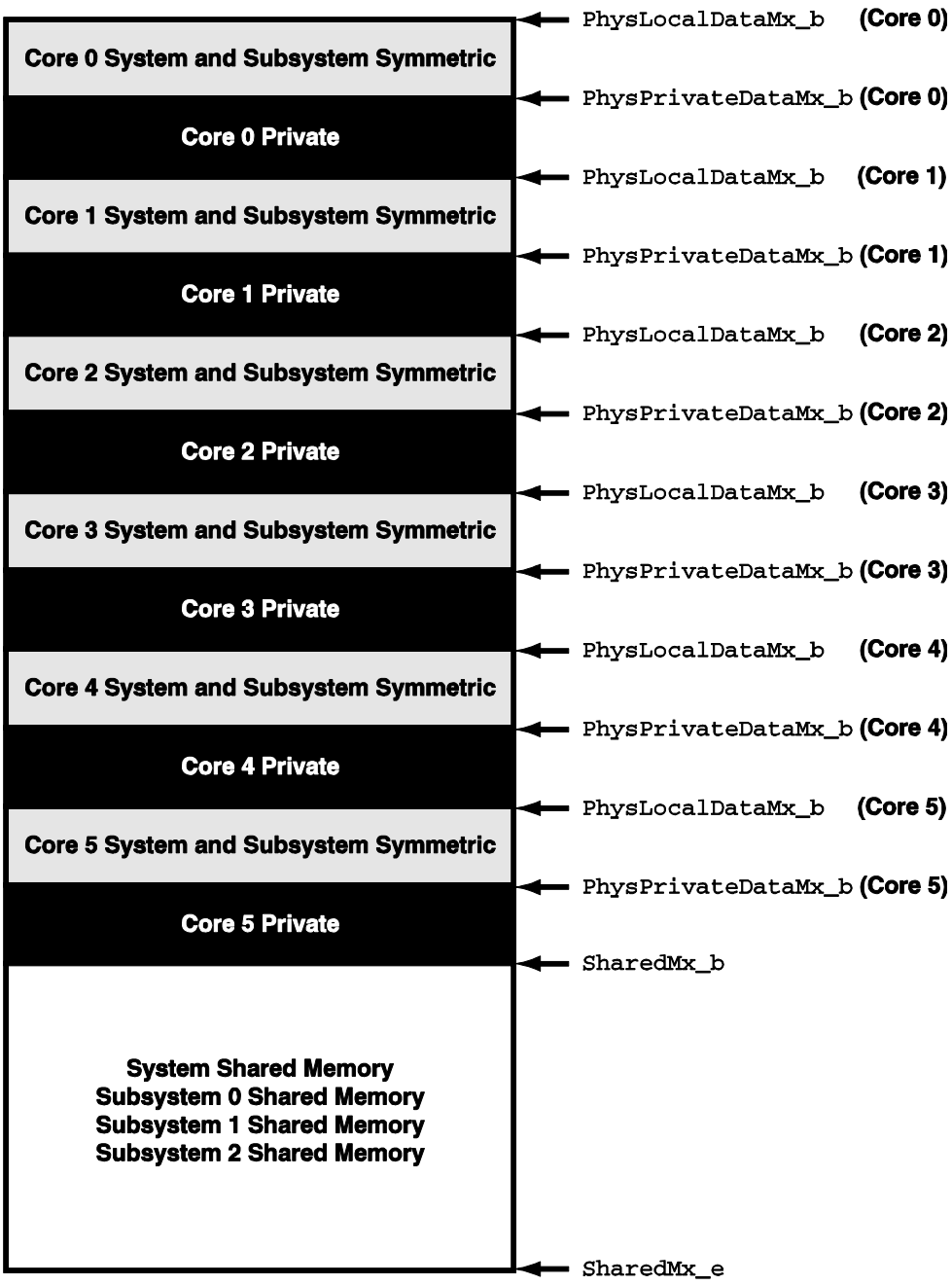

Figure 6. The Memory Map for Multi-core Application with Partially Shared Memory Allocated Together with System Shared Memory. Mx Stands for M3, DDR1 and DDR2.

To use this technique, proceed as follows:

- Keep the definition of symbols for shared memory as they were created by the wizard in

`memory_map_link.l3k` file

```
// Shared memory definitions.
// (The same for all cores - no need for shared virtual)
  SharedM3_b   = _LocalDataM3_b + (_LocalDataM3_size * num_core());
  SharedM3_size  = (_M3_e - _SharedM3_b);
  SharedM3_e   = _SharedM3_b + _SharedM3_size - 1;
```

- Keep the physical memory definition as they are created by the wizard in `os_msc815x_link.l3k`:

```
physical_memory shared  ( * ) {
  SHARED_M3   : org = _SharedM3_b,   len = _SharedM3_size;
  SHARED_DDR0 : org = _SharedDDR0_b, len = _SharedDDR0_size;
  SHARED_DDR1 : org = _SharedDDR1_b, len = _SharedDDR1_size;
}
```

- Change the unit definition in the subsystem-specific linker file. Do not specify an absolute virtual address for the memory blocks. Place them after the system shared memory blocks. The following code snippet handles the definition for subsystem 0, and a similar approach can be used for the other subsystems:

```
unit shared (task0_c0, task0_c1) {
  memory {
    m3_shared_text_SYS0 ("rx"): AFTER(shared_data_m3_descriptor;
    m3_shared_const_SYS0 ("rw"): AFTER(m3_shared_text_SYS0);
}
```

- Finally, adjust the address translation block to place the memory block in the appropriate `SHARED` physical area:

```
address_translation  (task0_c0, task0_c1)  {
    m3_shared_text_SYS0 (SYSTEM_PROG_MMU_DEF): SHARED_M3;
    m3_shared_const_SYS0 (SYSTEM_DATA_MMU_DEF): SHARED_M3;
}
```

6.7 Inter-core Communication and Synchronization

Often when an application is split into subsystems there should be a way to synchronize the different cores and exchange information between them. For instance there is a master core, which provides input data and delegates some tasks to a set of cores and waits for a result.

Inside of an asymmetric application, where all the cores are running SmartDSP OS, inter-core messaging can be used in this purpose.

The example project `Asym_SDOS_msg` demonstrates how this can be achieved.

Refer to the SDOS sample projects, `intercore_messages` and `intercore_queues`, to learn more on how to use inter-core communication within SmartDSP OS. These sample project are available in the CodeWarrior installation layout in `{Install}\SC\StarCore_Support\SmartDSP\demos\starcore\msc815x`.

6.7.1 Project Description

Core 0 is defined as being the master core for the inter-core synchronization. So a task (`c0_masterTask`), defined on core 0 only, sends inter-core messages to all cores part of subsystem 1, asking them to perform some computation.

On each core running the subsystem 1 application, a `sys1_slaveTask` is created, which basically does the following:

- Waits for message from `c0_masterTask`. The message comes with an input parameter for the computation.
- Perform the computation.
- Send message back to `c0_masterTask`, letting it know that computation is over.

On both core 0 and subsystem 1, the message handler posts a message in an event queue.

The destination task (`c0_masterTask` or `sys1_slaveTasks`) reads the data from the event queue. This is done in order to ensure the computation, which might be long, is performed in a task (in user mode) and not in a HWI (in supervisor mode).

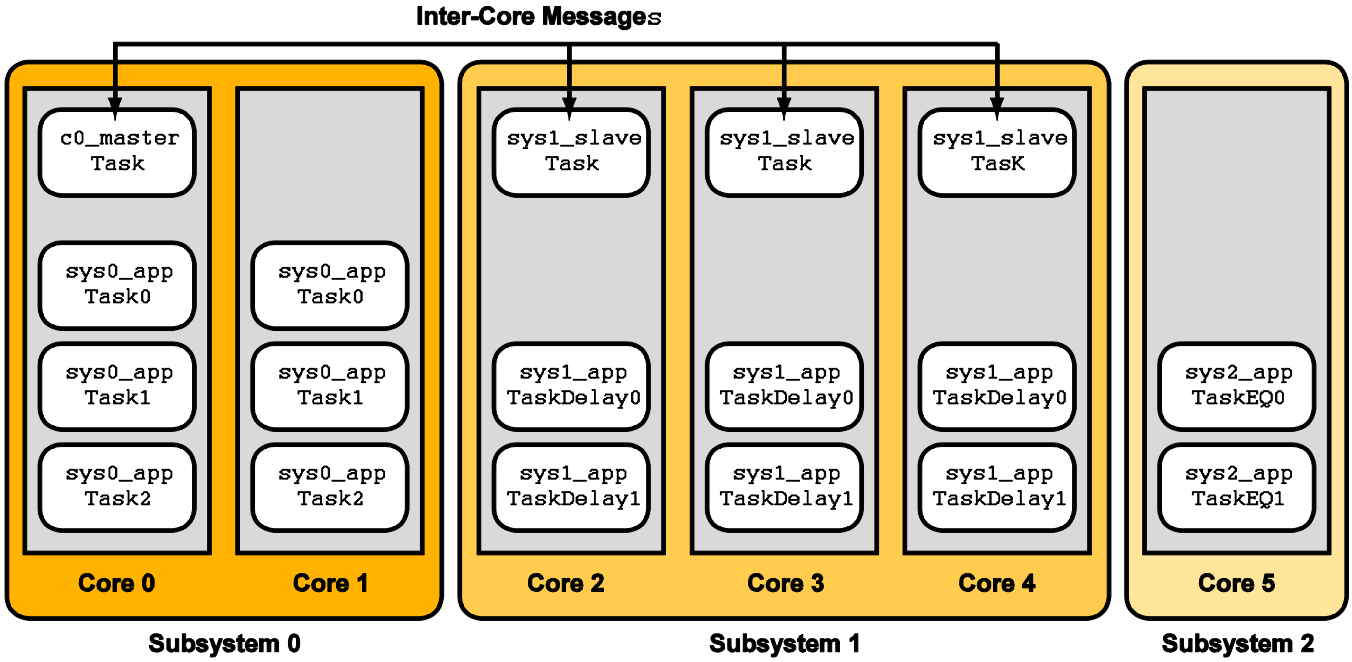

Figure 7. Available Tasks On Each Subsystem.

Figure 7 shows how the system is built up.

6.7.2 Configuration

To enable inter-core messaging, a specific configuration is needed on core 0 private code as well as on subsystem 1 partially shared code.

6.7.2.1 OS Configuration

One message only is used for the synchronization process, thus macro `OS_TOTAL_NUM_OF_INTERCORE_MESSAGES` has been set to 1 in `os_config.h`.

```
#define OS_TOTAL_NUM_OF_INTERCORE_MESSAGES  1    /* Intercore Messages Number  */
```

6.7.2.2 Application Configuration

On core 0, the following OS elements are created (see function `CoreSpecTaskCreate` in module `c0_code.c`):

- The `c0_masterTask`
- The required structure to enable inter-core messages
- The event queue used to transmit the message to the destination task

On each core running subsystem 1 code, the following OS elements need to be created (see function `sys1_appInit` in module `sys1_code.c`):

- The `sys1_slaveTask`
- The required structure to enable inter core messages
- The event queue used to transmit the message to the destination task

6.7.3 Runtime

Figure 8 below shows how inter-core messaging is used in the example project `Asym_SDOS_msg`.

Note that run time code must be written for `c0_masterTask` and `sys1_slaveTask`.

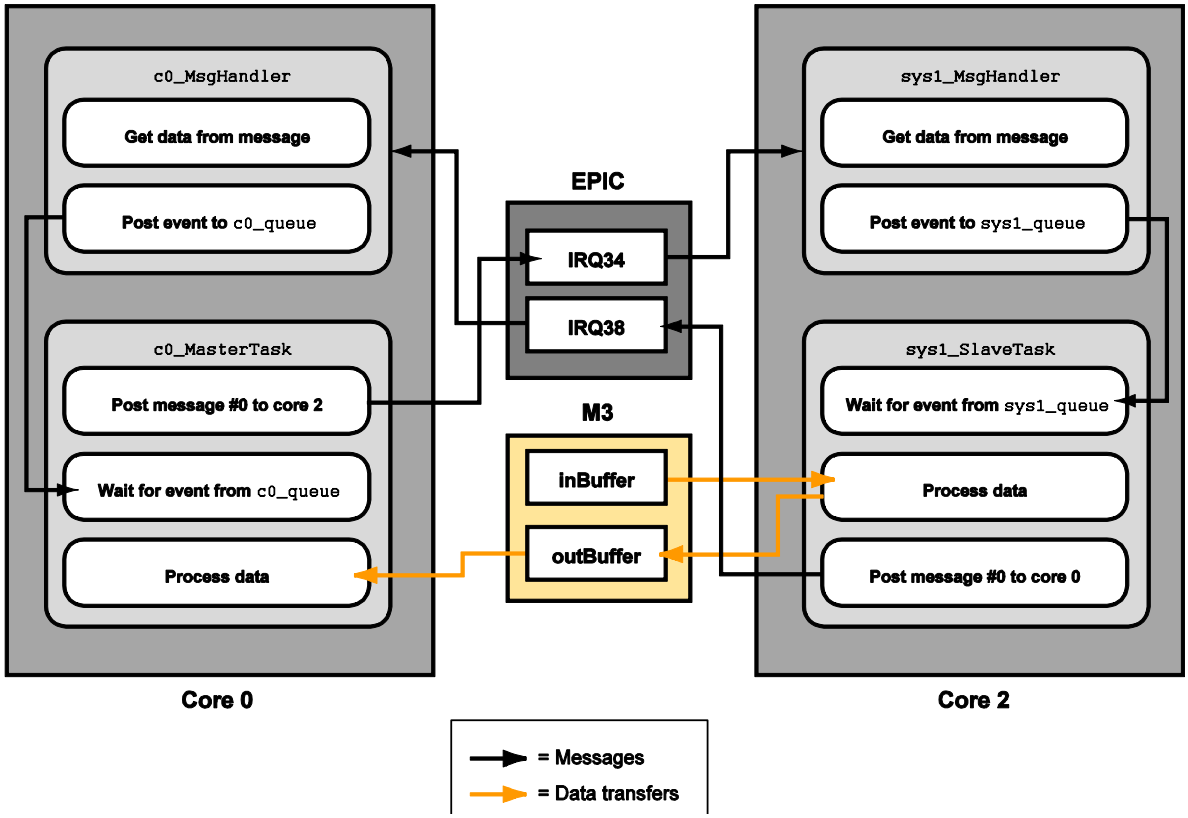

Figure 8. Inter-core Message Exchange Between Core 0 and Core 2 in `Asym_SDOS_msg`. The Same Mechanism Exists Between Core 0 and 3 and Between Core 0 and 4.

6.7.3.1 Implementation of c0_masterTask

The `c0_masterTask` first posts an inter-core message to each of the cores running subsystem 1 code (cores 2, 3 and 4). This is done through a call to `osMessagePost`.

```
/* Post a message to core 2 to start processing in sys1_slaveTask. */
  status = osMessagePost(c0_message_num, TO_CORE_2, 0x10);
  OS_ASSERT_COND(status == OS_SUCCESS);
/* Post a message to core 3 to start processing in sys1_slaveTask. */
  status = osMessagePost(c0_message_num, TO_CORE_3, 0x20);
  OS_ASSERT_COND(status == OS_SUCCESS);
/* Post a message to core 4 to start processing in sys1_slaveTask. */
  status = osMessagePost(c0_message_num, TO_CORE_4, 0x40);
  OS_ASSERT_COND(status == OS_SUCCESS);
```

It then waits for a message from each of these cores. The message signals that the job is finished. This is done through a call to `osEventQueuePend`.

```
  /* Wait for either core to finish computation */
  osEventQueuePend(c0_queue, &data, 500);
```

6.7.3.2 Implementation of sys1_slaveTask

The `sys1_slaveTask` first waits for a message from `c0_masterTask`. This is done through a call to `osEventQueuePend`.

```
  osEventQueuePend(sys1_queue, &data, 500);
```

When the event is triggered, it performs some computations. Then it sends back a message to `c0_masterTask`, letting it know the computation has finished. This is done through a call to `osMessagePost`.

As there are three cores, each sending a message to Core 0, it might be necessary to repeat the call to `osMessagePost` until core 0 is free for posting. This is the reason why the call is done as follows:

```
do {
  // In case core 0 is locked for posting, repeat the message posting.
  status = osMessagePost(sys1_message_num, TO_CORE_0, osGetCoreID());
  sys1_repeat_cnt++;
} while (status == OS_ERR_MSG_BUSY);
```

6.7.3.3 Implementation of Message Handler

The message handler is invoked by the OS scheduler when the inter-core message is received. It just reads the inter-core message and posts an event in the event queue. The message handler on core 0 (`c0_MsgHandler`) and on subsystem 1 (`sys1_MsgHandler`) are pretty similar. They are just posting to a different event queue.

The code appears as follows on subsystem 1:

```
void sys1_MsgHandler(os_hwi_arg message_id)
{
    os_status status;
    int data;
```

```
    /* Get data associated with the message. */
    data = osMessageGet(message_id);
    /* Notify sys1_slaveTask that calculation is finished on one of the
       slave cores.
     */
    status = osEventQueuePost(sys1_queue, data, NULL);
}
```

NOTE

The parameter `message_id` that is passed to the message handler transfers the source core id as well as the actual message number. `message_id` is encoded as `scrCore << 8 || msg_num`.

6.8 Using different L2/M2 Mapping on the Various Cores

As the amount of memory required in M2 memory might be different for cores running different subsystems, it might be necessary to define a different L2/M2 mapping, depending on the subsystem.

This can be done easily in `memory_map.l3k` as follows:

```
////// Local partition sizes //////////
// on subsystem 0, we are using 0x4000 bytes of M2 as L2 cache.
// on subsystem 1 and 2, we are using 0x2000 bytes of M2 as L2 cache.
_ _M2_Setting = (core_id() == 0) ? 0x1f :
                (core_id() == 1) ? 0x1f :
                 0x3f;

// M2 size
 _M2_size =
         (_M2_Setting == 0x01) ? 0x10000 :
         (_M2_Setting == 0x03) ? 0x20000 :
         (_M2_Setting == 0x07) ? 0x30000 :
         (_M2_Setting == 0x0f) ? 0x40000 :
         (_M2_Setting == 0x1f) ? 0x50000 :
         (_M2_Setting == 0x3f) ? 0x60000 :
         (_M2_Setting == 0x7f) ? 0x70000 :
         (_M2_Setting == 0xff) ? 0x80000 :
         0x0.
//512KB minus the area dedicated to the L2 cache
 __L2_cache_size = (_M2_L2_Size - _M2_size);
```

NOTE

Using different L2/M2 mapping between cores running SmartDSP OS applications is possible when using SmartDSP OS V3.6.1 or higher.

NOTE 2

In subsystems running SmartDSP OS, the entire M2 memory cannot be used as L2 cache. A portion of M2 memory is required to store the heap, stack and `.att_mmu` sections.

7 Running the Example Programs

The software archive `AN4063WSW.zip` contains several example programs that demonstrate how to implement multicore DSP applications on the MSC8156. All four of these applications follow the design described in section 2. Each example application consists of three subsystems, where subsystem zero executes on core 0 and 1, subsystem one executes on cores 2, 3, and 4, and subsystem two runs on core 5. All of the example applications used SmartDSP OS.

The purposes of the four variations of the DSP applications are:

- AsymSDOS_code—Demonstrates the use of a private `main()` and a private `appInit()` function. The application does not execute common tasks or core-specific tasks.
- AsymSDOS_priv_code—Demonstrates the use of a private `main()` and a private `appInit()` function. A task, `CommonTask`, is defined that executes on all of the subsystems. In addition, it defines a private task the runs only on core 0.
- AsymCodeSDOS_SharedMain—This example application is similar to AsymSDOS_priv_code except that it uses a shared `main()` function to invoke private `appInit()` functions.
- AsymSDOS_Msg—This example application is similar to AsymSDOS_priv_code, except that it uses inter-core messaging to synchronize subsystem 0 and subsystem 2.

The next section describes how to add and run these applications in CodeWarrior for StarCore.

7.1 Add the Project and Build It

First, extract the desired example application from the archive to obtain a folder that contains the project files. Launch CodeWarrior for StarCore v10.1.3 or later. In the C/C++ Perspective, drag the project folder into the CodeWarrior view. The folder appears as a project in this view.

Choose **Project > Clean** and then **Project > Build Project** to build the project.

7.2 Check the Launch Configurations

To access the launch configurations, choose **Run > Debug Configurations**. This displays the **Debug Configuration** dialog. Since this is a multicore project, there are multiple launch configurations. The example project has twelve launch configurations: six for an ADS hardware target (they have the string ADS in the name) and six for the instruction set simulator (they have the string ISS in the name). Each launch configuration targets one of the six processor cores. See Figure 9. There are also two launch groups, one for the hardware target, and one for the simulator. The launch groups are used to start the application on all six cores.

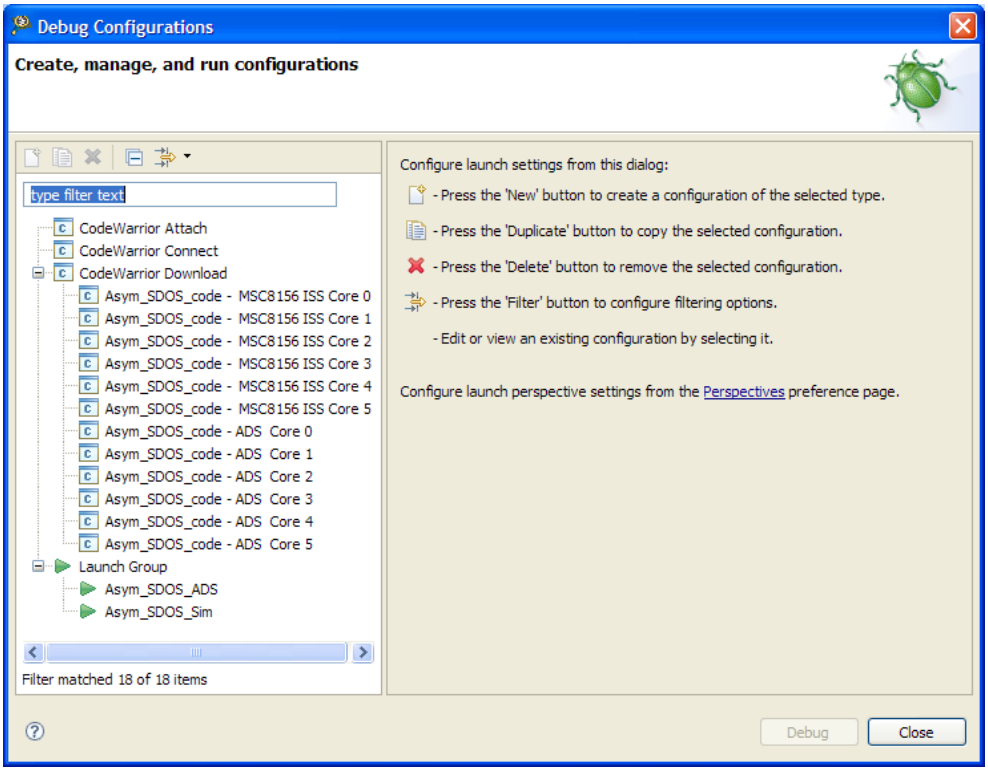

Figure 9. The Launch Configurations and Groups for the Asymmetric Project.

Open each launch configuration, and then use the **Debugger** tab to display the current settings. In particular, click on the **Connection** tab in this view to inspect the connection settings. They should match the hardware setup being used to run the project. If they do not, the connection settings must be modified in all six of the launch configurations.

NOTE

The screenshots and location of the connection settings described here are for CodeWarrior for StarCore DSPs v10.1.5 or earlier. For the location of the connections settings in CodeWarrior v10.1.8 or later, consult the *Targeting StarCore DSPs* manual.

7.3 Launch the Application

To start the asymmetric application, click on the appropriate launch group, then **Debug**. The Debug Perspective appears, and all six launch configurations are started in succession. When the launch process completes, the code on all six cores is suspended at its `main()` function (Figure 10).

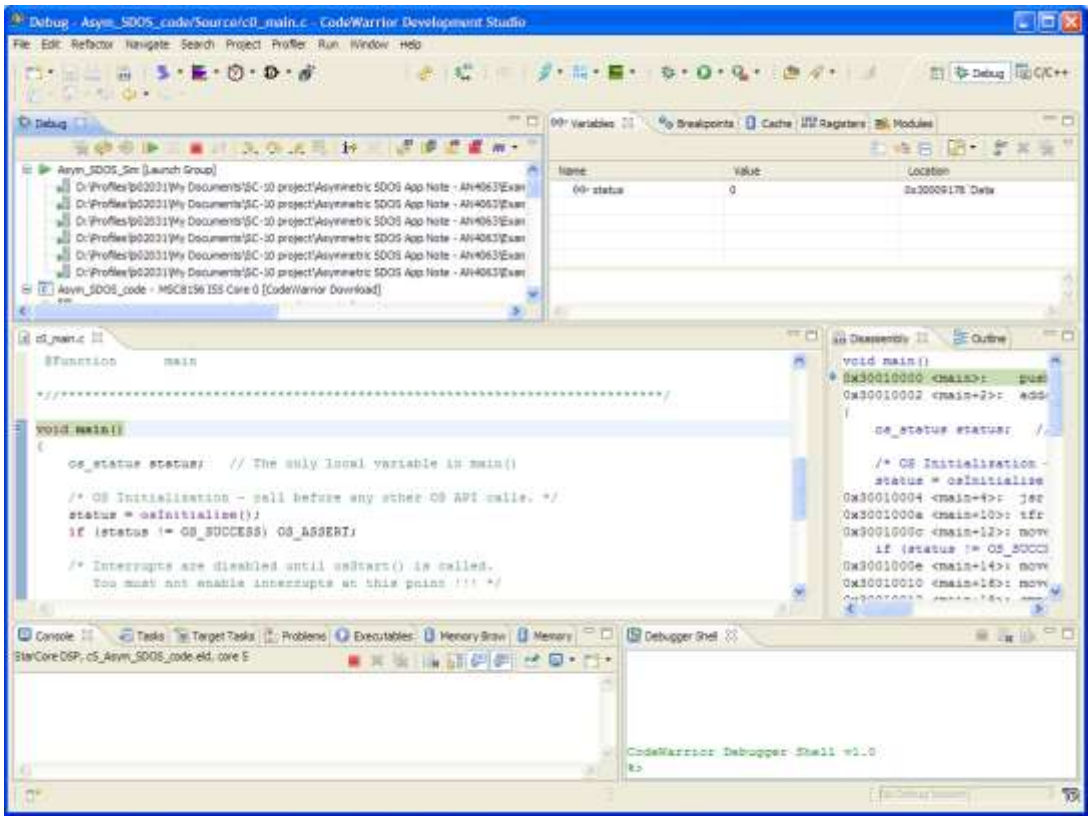

Figure 10. The Asymmetric Application's State after the Launch Group Has Started All Six Cores.

Click on **Multicore Resume** to start all of the cores at once. As each subsystem completes, it writes a `System x Test: Passed` message to the console. Clicking on each core thread in the **Debug** view displays the console associated with the subsystem that uses that core.

The other two example programs can be run by following the steps described above

8 Guidelines

Whenever the application memory map must be changed, make sure to follow guidelines below:

8.1 General Purpose Guidelines

1. The section `.att_mmu` and startup stack (label `_StackStart`) must be located in the same MMU segment.
   This is a runtime library requirement and applications that do not follow that scheme will not pass startup code. If this rule cannot be followed, the function `___target_asm_start` must be rewritten.
   a) For cores running bare board applications, this means sections `.att_mmu` and `.stack` must be allocated in the same MMU segment.
   b) For cores running SmartDSP OS applications, this means sections `.att_mmu` and `.oskernel_local_data_bss` must be allocated in the same MMU segment.
2. Application entry code and startup code must be allocated in a memory area with 1:1 mapping between virtual and physical address.
   This is a hardware requirement and applications that do not follow that scheme will not start.
3. Due to a current library implementation, the run time library heap needs to be allocated at the same virtual address on all cores.
   If this rule is not followed, the source code of `alloc.c` library module needs to be modified.
4. To generate bootable code, the application entry point should be located at the same physical address on all cores.
   This is a hardware requirement and applications that do not follow this scheme will not work when attempting to boot the application over Ethernet, I2C, SPI, or any other interface.

8.2 Guidelines for SmartDSP OS Applications

1. The entire M2 memory cannot be used as L2 cache.
   If this rule cannot be followed, the SmartDSP OS function `___target_asm_start` must be rewritten.
2. Sections `.att_mmu` and `.oskernel_local_data_bss` must be allocated in M2 memory.
   If this rule cannot be followed, the SmartDSP OS function `___target_asm_start` must be rewritten.
3. The section that contains `_g_heap_nocache` must be allocated in the same MMU segment as `.att_mmu` and the startup stack. That means the section `.oskernel_local_data` must also be allocated in same MMU segment as `.att_mmu` and `.oskernel_local_bss`.
   If this rule cannot be followed, the SmartDSP OS function `__target_setting` must be rewritten.
4. Section `.os_shared_data` must be allocated in M3 shared memory. This section contains spinlocks variables used within the OS code.
5. Due to the current startup code implementation, `_StackStart` and `_VBAddr` must be located at the same virtual address for all the cores running SmartDSP OS application.
   If this rule cannot be followed, revise the library module `startup__startup_msc8156_.asm`.

6. For the provided `.l3k` file, the size of KA buffer and VTB must be identical on all 6 cores. If this rule cannot be followed, the file `local_map_link.l3k` must be modified to move the KA buffer and VTB respectively to the end of DDR0 and DDR1 local memory.

.

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

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Error! No text of specified style in document.
Rev. 3
22 June 2011