Freescale Semiconductor
Application Note

Document Number: AN4220
Rev. 0, 01/2011

Asymmetric C++ Multicore Application for StarCore DSPs

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

This document is an addition to application note AN4063 “Configuring an Asymmetric Multicore Application for StarCore DSPs” and discusses the special CodeWarrior project configuration required for DSP applications that use the C++ programming language. The material focuses specifically on C++ DSP applications that utilize the Freescale SmartDSP OS (SDOS) kernel.

Readers of this document should be familiar the concepts described in the application note, AN4063. They should also be familiar with the location of the settings panels in the CodeWarrior for StarCore DSPs IDE.

The use of C++ as the programming language in an asymmetric SDOS project requires CodeWarrior for StarCore V10.1.8 or later.

Contents

© 2011 Freescale Semiconductor, Inc.

1 Configuring an SDOS Application for C++

With the release of CodeWarrior for StarCore V10.1.8, those projects created with the CodeWarrior Wizard and that use the C++ programming language in combination with SDOS are set up appropriately. However, the following adjustments must be made for the C++ code to link properly if you have initially created your project for C language.

1.1 Enable Exceptions

Exceptions need to be enabled for the compiler and the linker. This is described in the sections that follow.

1.1.1 Enable Exceptions in the Compiler

Add the option `-Cpp_exceptions on` to the **To Shell** edit box within the **StarCore C/C++ Compiler**'s **Additional Arguments** properties page (Figure 1).

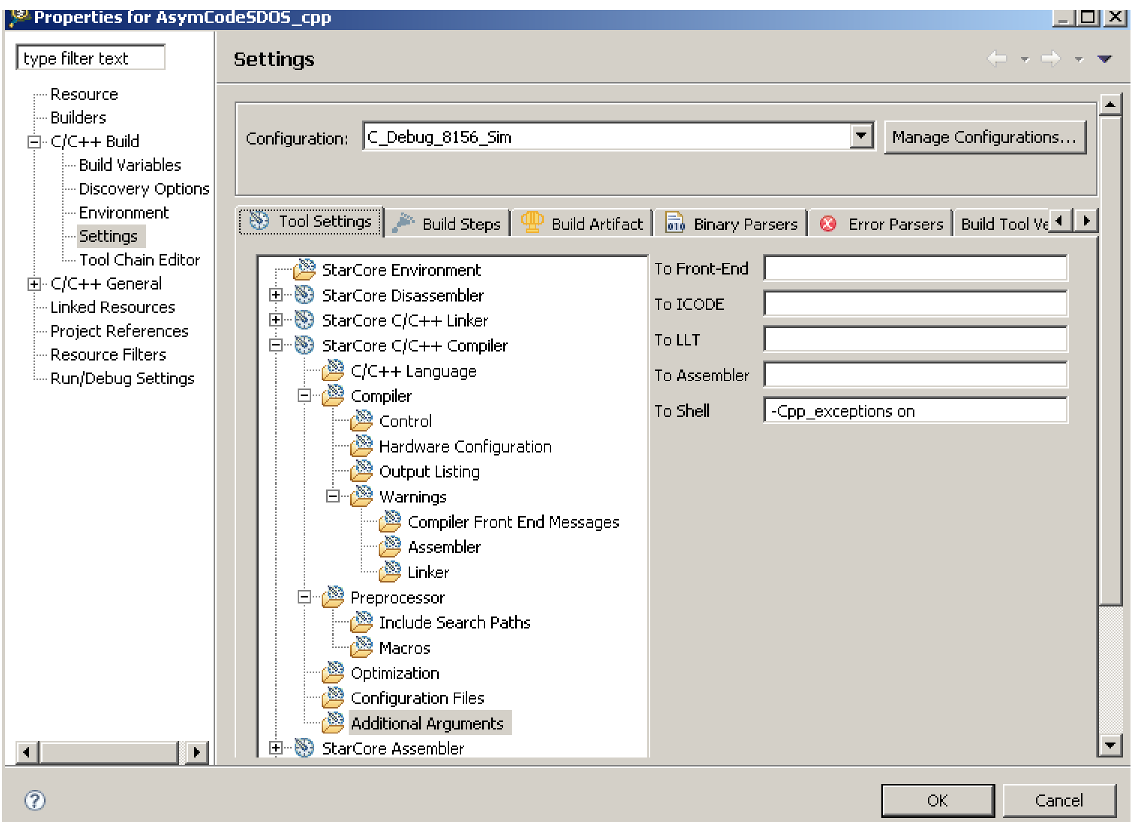

Figure 1. Setting -Cpp_exceptions Option

Once this option is enabled, the compiler recognizes C++ `try`, `catch`, and `throw` keywords. The compiler then generates additional code and data that implements the C++ exception handling.

1.1.2 Enable Exceptions in the Linker File

The symbol `ENABLE_EXCEPTION` must be set to 1. This symbol is used inside of the `.l3k` file to determine whether local symbols must be created for the exception table's start and end addresses. See section 1.3 below for more information.

The symbol can be defined directly in the `.l3k` files using following notation:

```
ENABLE_EXCEPTION=0x1;
```

Alternatively, the symbol can be defined on the linker command line by adding the following option to the **Addition Options** edit box within the **StarCore C/C++ Linker**'s **Linker Settings** properties page:

```
-DENABLE_EXCEPTION=0x1
```

1.2 Add an Exception Sections to MMU Segment

When exceptions are enabled in a C++ application, the compiler generates two sections to handle exceptions appropriately (Table 1).

Table 1. Exception Sections

Section name	Description
.exception	Holds the exception tables. This section needs to be placed in a private data descriptor.
.exception_index	Holds the exception table index. This section needs to be placed in a private data descriptor.

Both sections are placed in symmetrical memory using the file `local_map_link.l3k`:

```
unit private (*) {
    MEMORY {
      local_data_descriptor     ("rw"): org = _VirtLocalDataM2_b;
      (...)
    }
    SECTIONS {
      descriptor_local_data {
        (...)
        .exception
        .exception_index
        (...)
      } > local_data_descriptor;
      (...)
    }
}
```

1.3 Define Local Symbols Used by C++ Startup Code

1.3.1 Static Initializers

A C++ application might include definitions for some global or static class objects. These objects need to be created at startup. This is done in the function `__exec_staticinit`, which is implemented in file `staticinit__common_.c`.

This function uses the symbols `cpp_staticinit_start` and `cpp_staticinit_end` to cycle through the list of class constructors that need to be invoked. These symbols are defined in the file `local_map_link.l3k` as follows (the listing itself is just a collection of fragments extracted from the `.l3k` file):

```
unit private (*) {
  MEMORY {
    local_data_descriptor    ("rw"): org = _VirtLocalDataM2_b;
    (...)
  }
  SECTIONS {
      descriptor_local_data {
          .(...)
          .staticinit
          .(...)
      } > local_data_descriptor;
    (...)
  }
}

// Enable CPP symbols
_cpp_staticinit_start  = originof(".staticinit");
_cpp_staticinit_end    = endof(".staticinit");
```

1.3.2 Exception Table

When an exception is triggered in a C++ application, the function `FindExceptionRecord`, which is implemented in the `ExceptionHandler.cpp` module, is responsible for retrieving the exception table that corresponds to the function where the exception was thrown.

This function uses the symbols `__exception_table_start__` and `__exception_table_end__` to retrieve the appropriate exception table. These symbols are defined in the file `local_map_link.l3k` as follows (the listing is just a collection of fragments extracted from the `.l3k`file):

```
unit private (*) {
  MEMORY {
    local_data_descriptor    ("rw"): org = _VirtLocalDataM2_b;
    (...)
  }
  SECTIONS {
    descriptor_local_data {
      (...)
      .exception
      .exception_index
      (...)
    } > local_data_descriptor;
    (...)
  }
}
__exception_table_start  = (ENABLE_EXCEPTION) ? originof(".exception_index"):0;
__exception_table_end__  = (ENABLE_EXCEPTION) ? endof(".exception_index"):0;
```

1.4 Enable Usage of RTLib Heap

All instances of a class, which are created dynamically at run-time, are allocated in the Run-time library heap (RTLib Heap). So when programming in C++, the usage of the RTLib Heap needs to be enabled and a heap with the appropriate size must be defined.

The use of RTLib Heap is enabled in file `os_msc815x_link.l3k` through following command:

```
#define USING_RTLIB     1
```

The size associated with this heap can be adjusted in the file `local_map_link_l3k`. This is done through following definition:

```
__rtlibHeapSize = 0x4000;
```

In the associated example project, the RTLib Heap is allocated into private DDR0 memory.

2 Asymmetric Application Considerations

This section describes how to adjust the linker file to support C++ in an asymmetrical SDOS application. Special care must be taken as to how to manage the exception handling sections and static initialization sections.

2.1 Dealing with the Asymmetrical Memory Map

Inside of an asymmetric SDOS application implemented in C++, chances are the `.exception_index`, and `.static_init` sections have different sizes, and are allocated at different addresses on each core.

CodeWarrior for StarCore V10.1.8 and later releases do support this layout.

2.2 Exception Handling

2.2.1 Section .exception_index

The run time libraries perform a binary search on the `.exception_index` section to retrieve the exception record associated with the current function.

So there should be only one `.exception_index` section for each core image, and the section needs to be allocated in core private memory. This is described in the sections that follow.

2.2.1.1 Handling Core Private .exception_index Section

When a core specific program section is associated to a source file in an `.appli` file, the compiler creates a section ``c?`.exception_index`` (where c? stands for the core number) to store exception table index data for this module. This breaks the run-time behavior of the system. In order to ensure exceptions are processed correctly, the core specific exception table index data must be moved to the `.exception_index` section. Therefore, add the following code to the core private unit:

```
unit private (task0_c0) {
// Entries in section .exception_index need to be sorted.
//So all records need to be stored in the same section .exception_index.
```

```
    RENAME "*","c0`.exception_index",".exception_index"
    ....
}
```

The code snippet above is specific to core 0, but a similar approach can be used for code that executes on the other cores.

2.2.1.2 Allocating Section and Defining Symbols for Startup Code

The section `.exception_index` needs to be allocated in a private data MMU segment.

As the exception index table might be allocated at different address and might have different size, the symbols `__exception_table_start__` and `__exception_table_end__` need to be core specific.

This is implemented as follows in the core private unit (the listing is just a collection of fragments extracted from the `.l3k` file):

```
unit private (task0_c0) {
  RENAME "*","c0`.exception_index",".exception_index"

  memory {
    m2_private_data_0 ("rw"): org = _VirtPrivate_M2_b;
  }

  sections{
    privateData{
      . = align(4) ;
      __exception_table_start__  = .;
      ".exception_index"
       __exception_table_end__  = .;
    }
  }
}
```

At this point, remove the original definition of the symbols `__exception_table_start__` and `__exception_table_end__` in file `local_map_link.l3k`.

2.2.2 Section .exception

The `.exception_index` section contains pointers to the `.exception` sections. As the symbols defined in this section are only referenced from private constants, it is possible to keep a clean layout composed of a separate system-wide `.exception` section, a subsystem-wide section, and finally a core-specific section.

When a core-specific program section is associated to a source file in an `.appli` file, the compiler creates a section `` c?`.exception `` (where c? stands for the core number) to store exception table data for this module. In order to get a clean layout, the subsystem specific `.exception` section needs to be created. This is done using the `RENAME` command (see section 2.2.2.2 below).

2.2.2.1 System-Wide .exception Section

The system-wide `.exception` section is placed in a system symmetrical unit. The following are code fragments from the `local_map.l3k` file where this placement is done:

```
unit private (*) {
  memory {
    local_data_descriptor     ("rw"): org = _VirtLocalDataM2_b;
  }

  sections {
    descriptor_local_data {
       .exception
    } > local_data_descriptor;
  }
}
```

2.2.2.2 Subsystem-wide .exception Section

The subsystem-wide section is placed in subsystem symmetrical unit. The following are code fragments of the `system0.l3k` file show how this is done for subsystem 0:

```
unit private (task0_c0,task0_c1) {
  RENAME "*sys0_*.eln",".exception",".sys0_exception"
  memory {
    m2_SYS0_data ("rw"): AFTER(local_data_descriptor);
  }

  sections{
    sys0_data{
      ".sys0_exception"
    } > m2_SYS0_data;
  }
}
```

2.2.2.3 Core-Specific .exception Section

The core-specific section is placed in the core private unit. The following are code fragments from the `system0.l3k` file where this is done for core 0:

```
unit private (task0_c0) {
  RENAME "*","c0`.exception_index",".exception_index"
  memory {
    m2_private_data_0 ("rw"): org = _VirtPrivate_M2_b;
  }

  sections{
    privateData{
      . = align(4) ;
      __exception_table_start__   = .;
      ".exception_index"
      __exception_table_end__   = .;
      "c0`.exception"
    }> ddr0_priv_text_0;
  }
}
```

2.3 Handling Static Initializers

The run time libraries handle one single table of static initializers.

When a core-specific program section is associated to a source file in an `.appli` file, the compiler will create a section `c?`.staticinit` (where c? stands for the core number) to store static initializer data for this module.

In order for the startup code to invoke the constructor function for the global class defined in a core specific module, the `c?`.staticinit` section need to be allocated next to the `.staticinit` section. The variable `static_init_end_ptr` needs to point at the end of the core-specific constructor table.

The following are code fragments from the `system0.l3k` file where this allocation is done for core 0.

```
unit private (task0_c0) {
  RENAME "*sys0_*.eln",".staticinit",".sys0_staticinit"
  memory {
    m2_private_data_0 ("rw"): org = _VirtPrivate_M2_b;
  }

  sections{
    privateData{
      . = align(4) ;
      __cpp_staticinit_start__ = .;
      ".staticinit"
      “.sys0_staticinit”
      "c0`.staticinit"
      __cpp_staticinit_end__ = .;
    } > m2_private_data_0;
  }
  (...)
}
```

At this point make sure to remove the original definition of the symbols `_cpp_staticinit_start` and `_cpp_staticinit_end` in file `local_map_link.l3k`.

NOTE

In order to get a clean layout, a subsystem specific `.staticinit` section is created in the code snippet above (See the RENAME command). This is not mandatory; one can decide to keep subsystem-wide exception data with the system-wide ones.

3 The Example Project

3.1 Project Architecture

This section describes the configuration of an example C++ multicore DSP application. The application is a system comprised of three subsystems, as shown in Figure 2. Each of the subsystems executes its own C++ SDOS application.

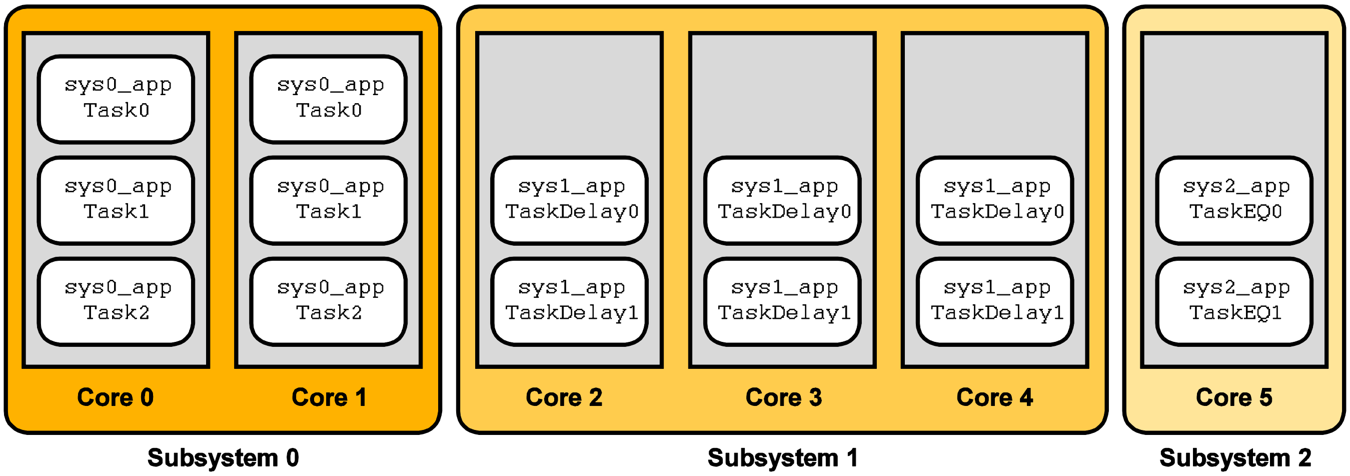

Figure 2. The Architecture of the Asymmetric DSP Application

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 SDOS 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.
  Cores running subsystem 0 define a subsystem-wide global class, `sys0_list`, which is filled with elements from the function `sys0_CreateTaskAndTimer`.
  Core 0 defines a private global class, `c0_list`, which is filled with elements from the function `c0_TaskCreate`.
- 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 ten 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.

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

Figure 3 shows the physical memory map of the example asymmetric application. The symbolic names to the right of the diagram define specific addresses.

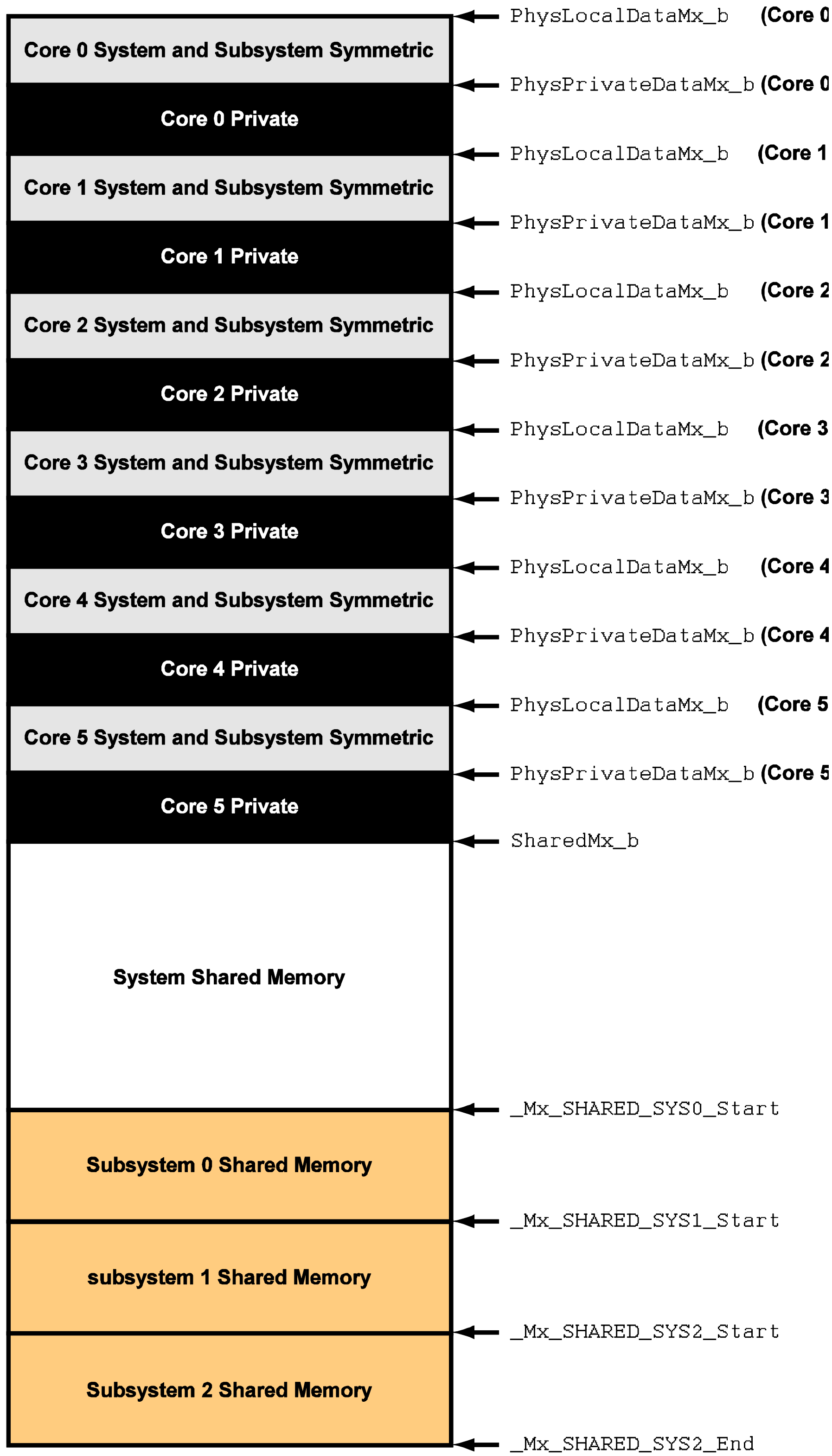

Figure 3. The Memory Map for the Example Multicore Application Described in this Article. Mx Stands for M2, M3, DDR1, and DDR2 Memories (M2 memory does not include any shared memory area)

3.3 Running the Example Program

The software archive contains an example programs that demonstrate how to implement a multicore DSP C++ application on the MSC8156. This application consists of three subsystems as described in section 3.1. To recap, subsystem zero executes on cores 0 and 1, subsystem one executes on cores 2, 3, and 4, and subsystem two runs on core 5.

The next section describes how to add and run this application with CodeWarrior for StarCore DSPs.

3.3.1 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 the CodeWarrior IDE. In the C/C++ Perspective, drag the project folder into the **CodeWarrior** view. The folder appears as a project in this view.

When importing the project in CodeWarrior V10.1.8, following message windows show up (Figure 4):

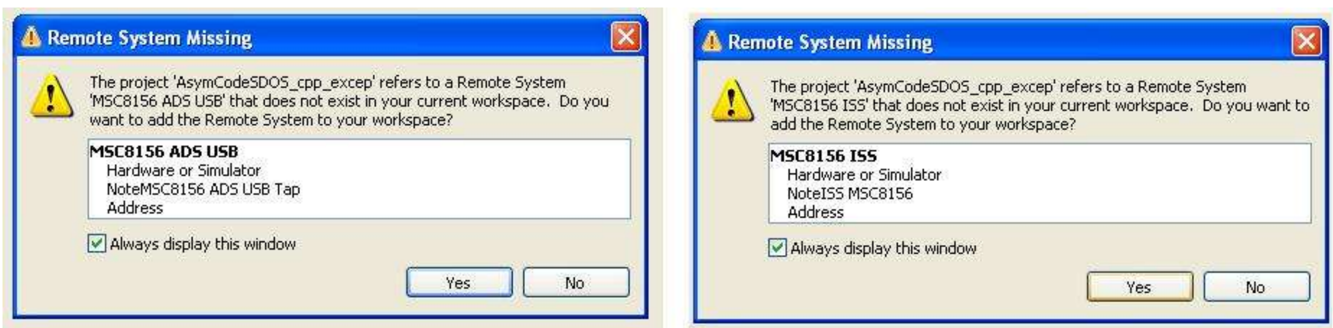

Figure 4. Remote System Missing Messages

At that point there are two choices:

a. Use the Remote connection defined in the project.
   - Click on **Yes**. The specified Remote System is added to the workspace. It is now available for any project added to the workspace or created in it.

b. A Remote System is already defined in the workspace and it is to be used with the C++ project as well.
   - Click on **No**.
   - Now it is necessary to associate the appropriate Remote System to the launch configurations. This can be done as follows:
     – Open the **Remote Systems** view. This can be done selecting the menu entry **Windows** > **Show View** > **Remote Systems**.
     – Right-click on the Remote System to associate to the ADS Launch Configuration.

– In the drop down menu select **Apply to Project** > {**ProjectName**} and select each ADS related launch configuration (Figure 5).

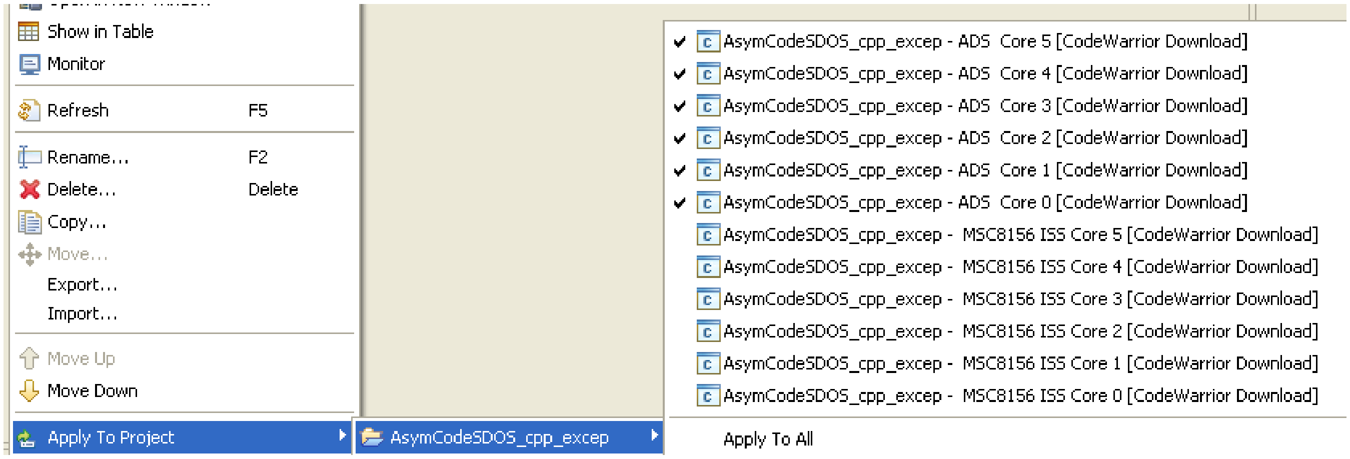

Figure 5. Apply Existing Remote System to Launch Configuration

– Apply the ISS Remote System to the ISS launch configuration in the same way.

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

The default project is implemented to generate an exception in case the RT Heap is fully used. (That is, if there is not enough heap space to allocate the C++ classes).

If the macro `_DO_TEST_EXCEPTION_` is added to the **Preprocessor** > **Macros** project Properties panel, the application throws some system-wide, subsystem-wide, and core private exceptions.

3.3.2 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 the instruction set simulator (they have the string ISS in the name) and six for an ADS hardware target (they have the string ADS in the name). Each launch configuration targets one of the six processor cores. See Figure 6. 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.

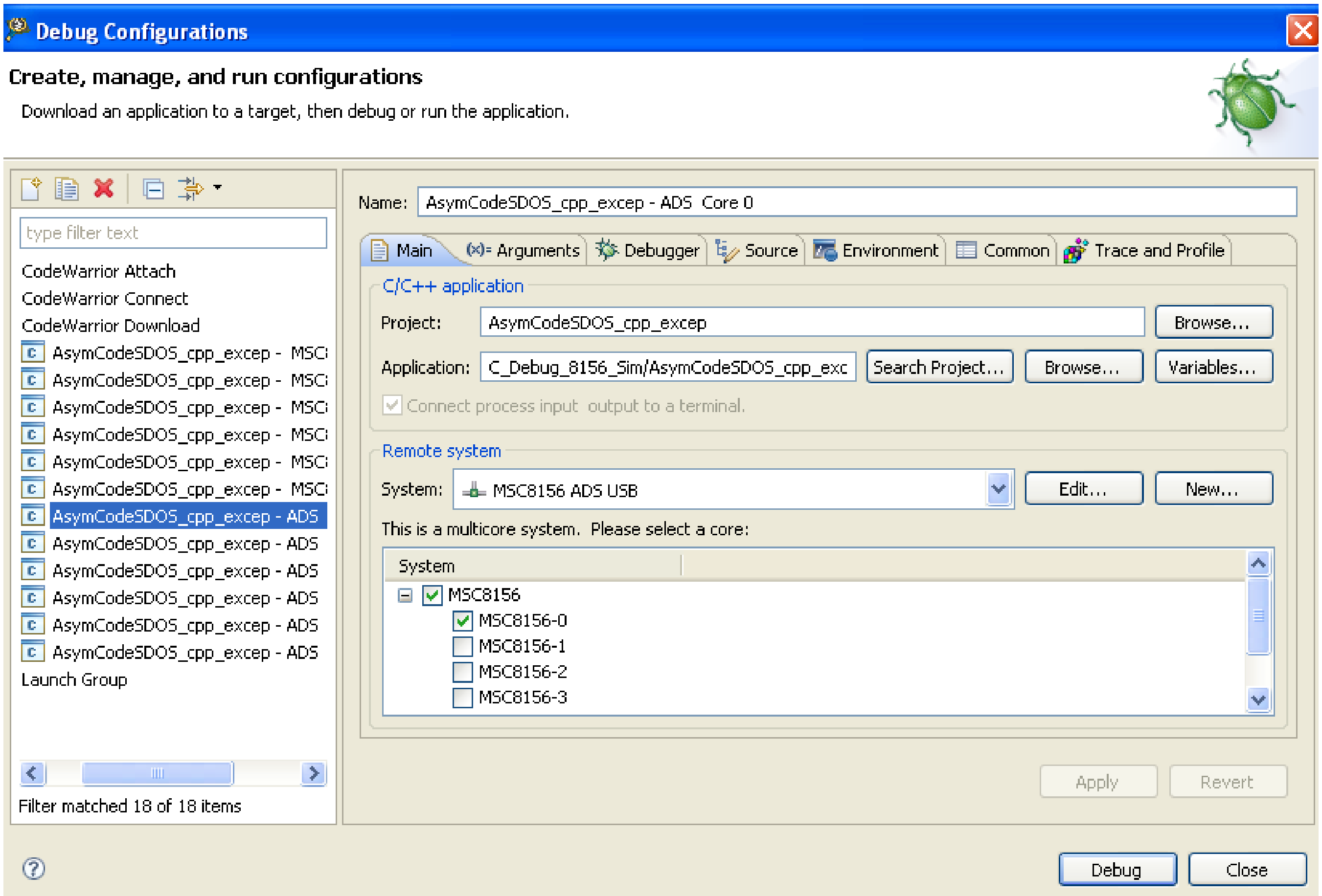

Figure 6. The Launch Configurations and Launch Groups for the C++ Project

Open each launch configuration, and use the **Debugger** tab to display the current settings. Make sure all ADS launch configuration are referring to the same Remote System.

In the same way all the ISS launch configuration must refer to the same ISS Remote System.

3.4 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 7).

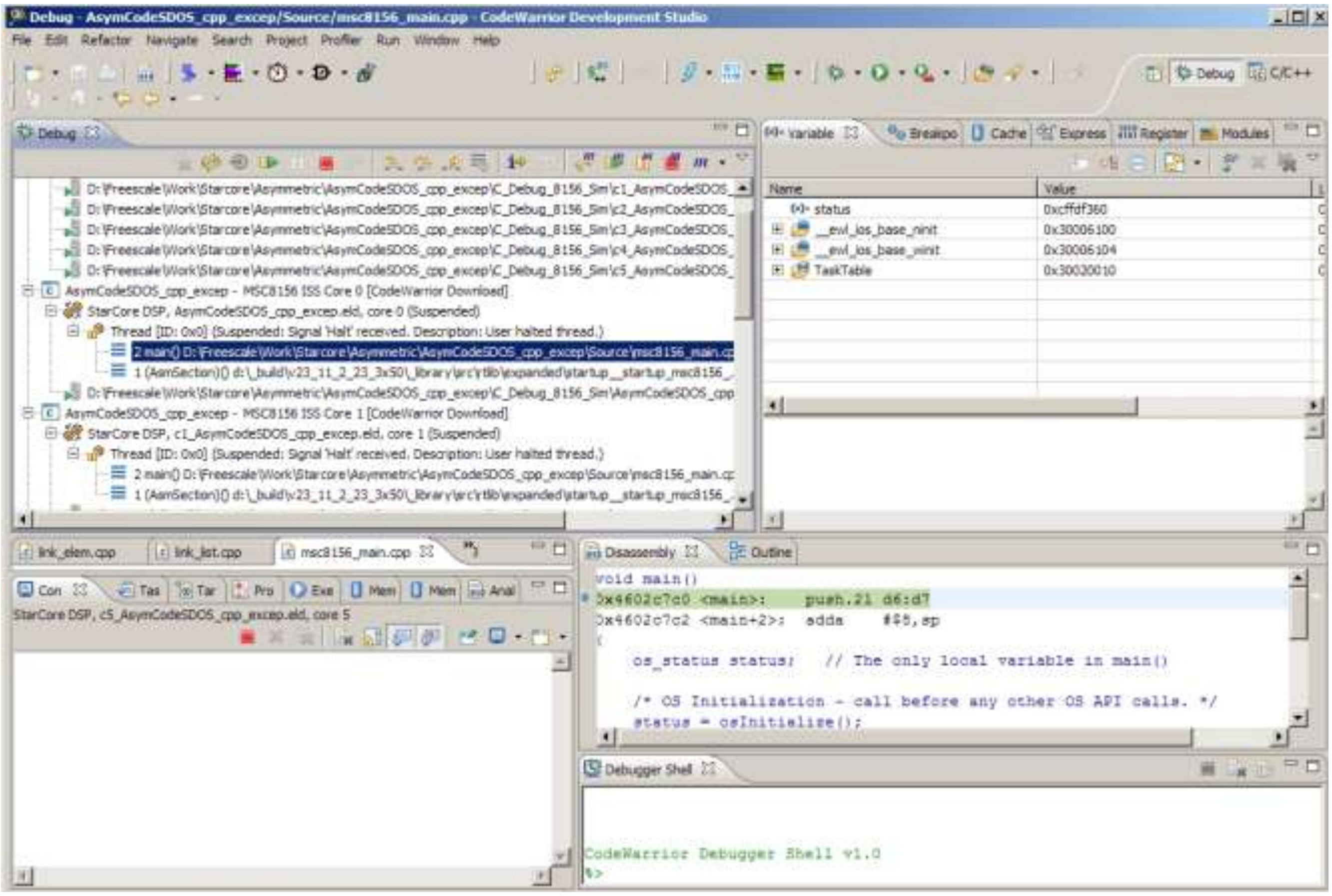

Figure 7. The Asymmetric Application's State After the Launch Group Has Started All Six Cores

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.

4 Guidelines

When changing the application memory map, make sure to follow the guidelines below.

4.1 General Purpose Guidelines

1. 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 execute.
2. To generate bootable code, the application's 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, I^2C, SPI, or any other interface.

4.2 Guidelines for SDOS Applications

1. The section that contains `_g_heap_nocache` must be allocated in the same MMU segment as the startup stack (StackStart). 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 SDOS function __target_setting must be rewritten.
2. Section `.os_shared_data` and `.os_shared_data_bss` must be allocated in M3 shared memory. These sections contain spinlocks variables used within the OS code.

   If this rule cannot be followed, multicore synchronization will not run correctly.
3. Due to the current startup code implementation, `_VBAddr` must be located at the same virtual address for all the cores running SDOS application.
   If this rule cannot be followed, revise the library module `startup__startup_msc8156_.asm`.

   If this rule cannot be followed, revise the library module startup__startup_msc8156_.asm.
4. SDOS heaps must have the same size on all cores running SDOS.

   This is an OS requirement.

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 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale, the Freescale logo, CodeWarrior, and StarCore are trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners.

© 2011 Freescale Semiconductor, Inc.

Document Number: AN4220
Rev. 0
01/2011