TEXAS INSTRUMENTS

Keystone Multicore Workshop

Lab Manual

Contents

Lab 1 – SRIO Loopback Direct IO

Purpose

The purpose of this lab is to demonstrate how to build and run a very basic Code Composer Studio v5 project on the C6678 EVM using the Direct IO Loopback example delivered with the MCSDK.

Project Files

The exact location of the project files will depend on where the MCSDK was installed and which version you are using. They can be found here:

<MCSDK_DIR>\pdk_C6678_<your pdk version>\packages\ti\drv\exampleProjects\SRIO_LoopbackDioIsrexampleproject

EVM Configuration

Set the EVM the 'no boot' mode as shown below.

Boot Mode	**DIP SW3 (Pin1, 2, 3, 4)**	**DIP SW4 (Pin1, 2, 3, 4)**	**DIP SW5 (Pin1, 2, 3, 4)**	**DIP SW6 (Pin1, 2, 3, 4)**
No boot	(off, on, on, on)	(on, on, on, on)	(on, on, on, on)	(on, on, on, on)

Additional EVM switch settings are available at the following link:
http://processors.wiki.ti.com/index.php/TMDXEVM6678L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings

Task 1: Import the Example Project

1. Open CCS.
2. Set the Perspective to CCS Edit.
3. Import the project.
    - Project | Import Existing CCS/CCE Eclipse Project
    - Select search directory to

    <MCSDK_DIR>\pdk_C6678_<your pdk version>\packages\ti\drv\exampleProjects

    - From the list of Discovered projects, choose SRIO_LoopbackDioIsrexampleproject and then click Finish.
4. SRIO_LoopbackDioIsrexampleproject should now appear in your Project Explorer.

Task 2: Set/Verify the Project Properties

5. Select the SRIO_LoopbackDioIsrexampleproject.
6. Right click and select Properties.
7. Select General and choose the Main Tab.
8. Set the following Device Properties.
    - Device Family = C6000
    - Variant = Generic C66x Device
9. Under Build/C6000 Compiler, select Basic Options and set the following compiler debug properties:
    - Target processor version = 6600
    - Debugging model = Full symbolic debug
    - Optimization level = 0
    - Optimize for code size = 0
10. Click OK.

Task 3: Build the Project

11. Select SRIO_LoopbackDioIsrexampleproject.
12. Build the project.
    - Project | Build Project

    OR
    - Right Click and select Build Project
13. Verify that the build was successful.

Was the file SRIO_LoopbackDioIsrexampleproject.out generated? _____

> **Hint:** From the CCS Edit perspective, check the Binaries or Debug directory. From the CCS Debug perspective, check the Console.

Task 4: Connect to the Target EVM

1. Set the Perspective to CCS Debug.
2. Create a new User-Defined Target:
    - View | Target Configurations
    - Select User Defined
    - Click the New Target button or Right-click and select New Target Configuration
3. Define the C6678L/LE EVM as a new target:
    - File name = EVM6678L or EVM6678LE
    - Location = <local>\ti\CCSTargetConfigurations
    - Select the emulator type in the connection drop-down menu
        i. If you are using the on-board XDS100, you should choose TI XDS100v1
        ii. If you are using a Mezzanine Emulator, you should choose Blackhawk XDS560v2 Mezzanine Card
    - Specify the Board or Device by checking the appropriate box (TMS320C6678)
    - Click the "Advanced" tab at the bottom of the screen and add the appropriate GEL file for Core 0 by selecting C66x_0 and choosing the initialization script.
        i. The GEL file is located here:

        <CCS_INSTALL_DIR>\ccsv5\ccs_base\emulation\boards\evm6678l\gel\evmc6678.gel
    - Click Finish
4. Make sure the EVM is powered ON and connect your PC/laptop to the emulator port on the EVM using the provided USB cable.
5. Launch the target configuration (e.g., EVM6678LE.ccxml).
    - Select the target.
    - Right click and select Launch Selected Configuration.
6. Select Core 0, right click, and select Connect Target.

Task 5: Load and Run the Program

7. Select Core 0 and load the .out file created earlier in the lab.
    - Run | Load | Load Program
    - Click Browse Project
    - Select SRIO_LoopbackDioIsrexampleproject.out and Click OK.
    - Click OK to load the application to the target (Core 0)
8. Run the application.

Did the application execute successfully? _____

Hint:	Check the console.

Lab 2 – Hyperlink

Purpose

The purpose of this lab is to demonstrate how to build and run a HyperLink loopback application on the C6678 EVM using the example code as delivered with MCSDK. In addition, you will make modifications to the application parameters to vary the transfer rate. Optionally, you will run the application on two boards and modify the transfer rate to determine the maximum throughput allowed with this example configuration.

Project Files

The exact location of the project files will depend on where the MCSDK was installed and which version you are using. They can be found here:

<MCSDK_DIR>\pdk_C6678_<your pdk version>\packages\ti\drv\exampleProjects\hyplnk_exampleproject

Task 1: Import the Example Project

1. Open CCS.
2. Set the Perspective to CCS Edit.
3. Import the project.
    - Project | Import Existing CCS/CCE Eclipse Project
    - Select search_directory: \pdk_C6678_<your_pdk_version>\packages\ti\drv\exampleProjects
    - From the list of Discovered projects, choose hyplnk_exampleproject and then click Finish.
4. hyplnk_exampleproject should now appear in your Project Explorer.

How many lanes are configured? _____

What is the baud rate? (Note: 01p250 means 1.25GBaud) _____

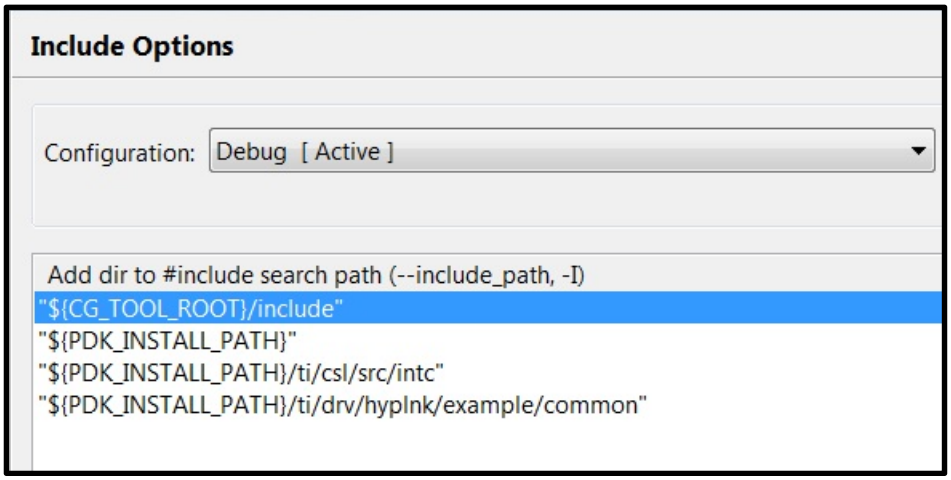
> **Hint:** Refer to hyplnkLLDCfg.h

5. Look for the following line in hyplnkLLDCfg.h and ensure that it is uncommented.

```
#define hyplnk_EXAMPLE_LOOPBACK
```

Task 2: Set the Project Properties

6. Select the hyplnk_exampleproject.
7. Right click and select Properties.
8. Select General and choose the Main Tab.
9. Set the following Device Properties.
    - Device Family = C6000
    - Variant = Generic C66x Device
10. Under Build/C6000 Compiler, select Basic Options and set the following compiler debug properties:
    - Target processor verison = 6600
    - Debugging model = Full symbolic debug
    - Optimization level = 0
    - Optimize for code size = 0
11. Click OK.
12. Under Build/C6000 Compiler, select Include Options and verify the following paths:



Task 3: Build the Project

13. Select hyplnk_exampleproject.
14. Build the project.
    - Project | Build Project

    OR
    - Right Click and select Build Project
15. Verify that the build was successful.

Was the file hyplnk_exampleproject.out generated? _____

> **Hint:** From the CCS Edit perspective, check the Binaries or Debug directory. From the CCS Debug perspective, check the Console.

Task 4: Connect to the EVM

16. Set the Perspective to CCS Debug.
17. Create a new User-Defined Target:
    - View | Target Configurations
    - Select User Defined
    - Click the New Target button or Right-click and select New Target Configuration
18. Define the C6678L/LE EVM as a new target:
    - File name = EVM6678L or EVM6678LE
    - Location = <local>\ti\CCSTargetConfigurations
    - Click Finish
19. Make sure the EVM is powered ON and connect your PC/laptop to the emulator port on the EVM using the provided USB cable.
20. Launch the target configuration (e.g., EVM6678LE.ccxml).
    - Select the target.
    - Right click and select Launch Selected Configuration.
21. Select Core 0, right click, and select Connect Target.

Task 5: Load and Run the Program

22. Load the .out file created earlier in the lab.
    - Run | Load | Load Program
    - Click Browse Project
    - Select hyplnk_exampleproject.out and Click OK.
    - Click OK to load the application to the target.
23. Run the application.

Did the application execute successfully? _____

> **Hint:** Check the console.

Task 6 (Optional): Board-to-board Hyperlink Example

Modify the example to run the HyperLink application on two EVMs.

Hardware requirements:

- Two C66x EVMs
- One HyperLink cable
- Connector Board

24. Modify the example code for hyplnk_exampleproject
    - Open hyplnkLLDCfg.h
    - Search for “#define hyplnk_EXAMPLE_LOOPBACK”
    - Comment out this command.
    - Change the Baud Rate back to 6.25 Gbaud
25. Build the code, load to both targets, and run only on Core 0.

Did the application execute successfully? _____

> **Hint:** Check the console.

26. Modify the example code for hyplnk_exampleproject
    - Open hyplnkLLDCfg.h
    - Change the Baud Rate to a higher rate.
27. Build the code, load to both targets, and run only on Core 0.

Did the application execute successfully? _____

What is the highest transfer rate that can be achieved using this example? _____

Lab 3 – SRIO Type 11

Purpose

The purpose of this lab is to demonstrate how to use Type 11 SRIO in an application.

Project Files

The following files are used in this lab

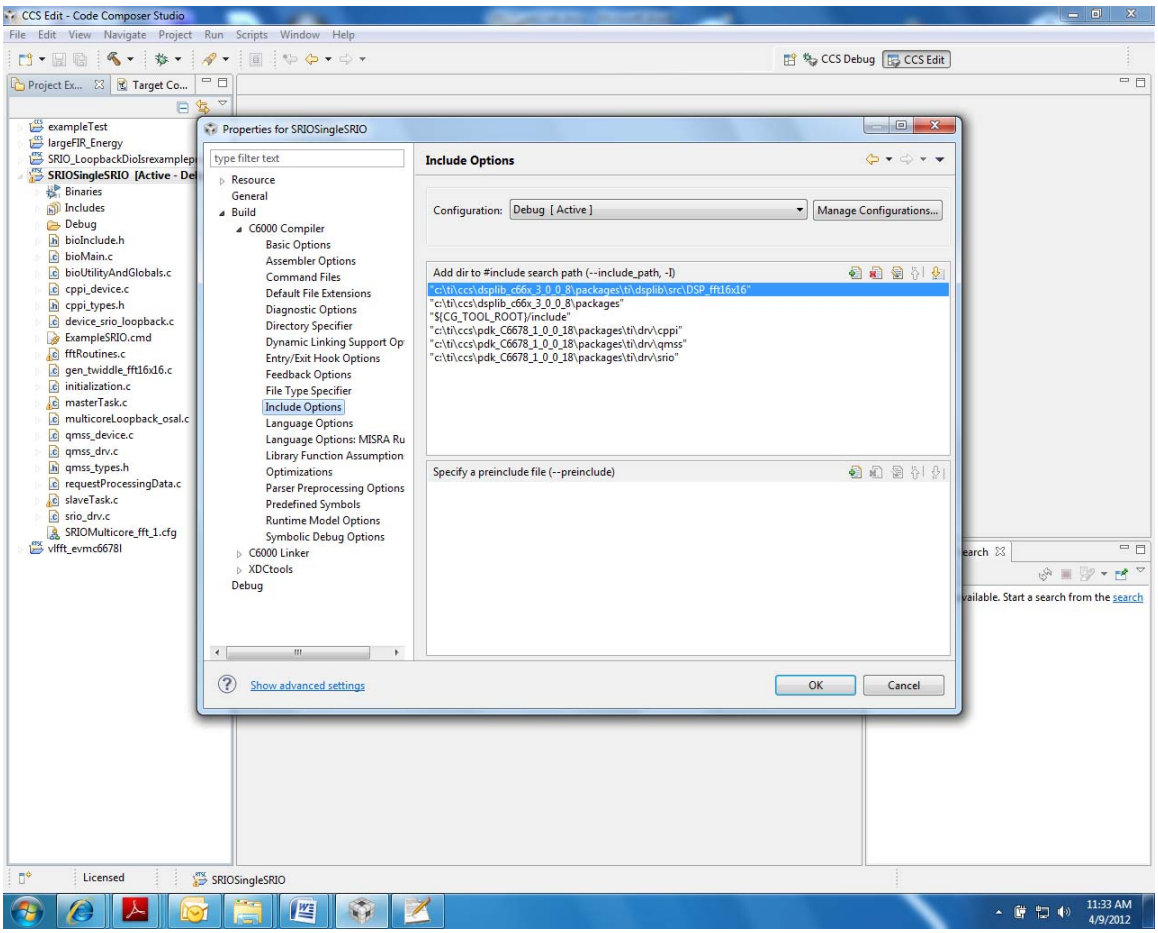
- bioInclude.h
- bioMain.c
- bioUtilityAndGlobals.c
- cppi_device.c
- cppi_types.h
- device_srio_loopback.c
- ExampleSRIO.cmd
- fftRoutines.c
- gen_twiddle_fft16x16.c
- initialization.c
- masterTask.c
- multicoreLoopback_osal.c
- qmss_device.c
- qmss_drv.c
- qmss_types.h
- requestProcessingData.c
- slaveTask.c
- srio_drv.c
- SRIOMulticore_fft_1.cfg

Task 1: Load the Project

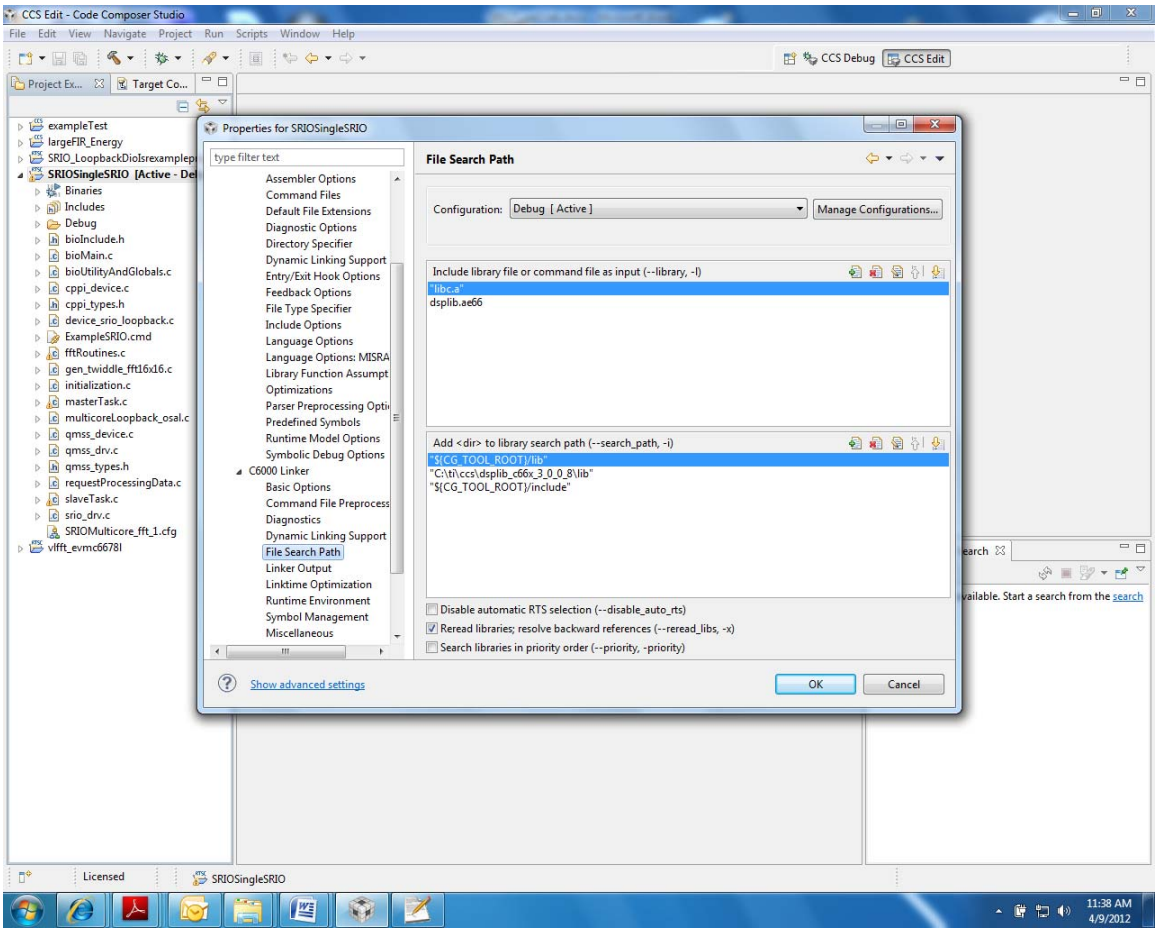
1. Copy the project folder to your local development environment. The instructor will pass around a USB drive or point you to a location where the folder can be downloaded.

2. Start CCS and import the project.

3. Update the include path in the Project Properties. Refer to the path as defined below. You must modify all paths to either an absolute address where you put your tools, or refer to a relative address. You can use {PDK_INSTALL_PATH} or similar if it is defined.



4. Do the same for the Linker File Search Path, as shown below.

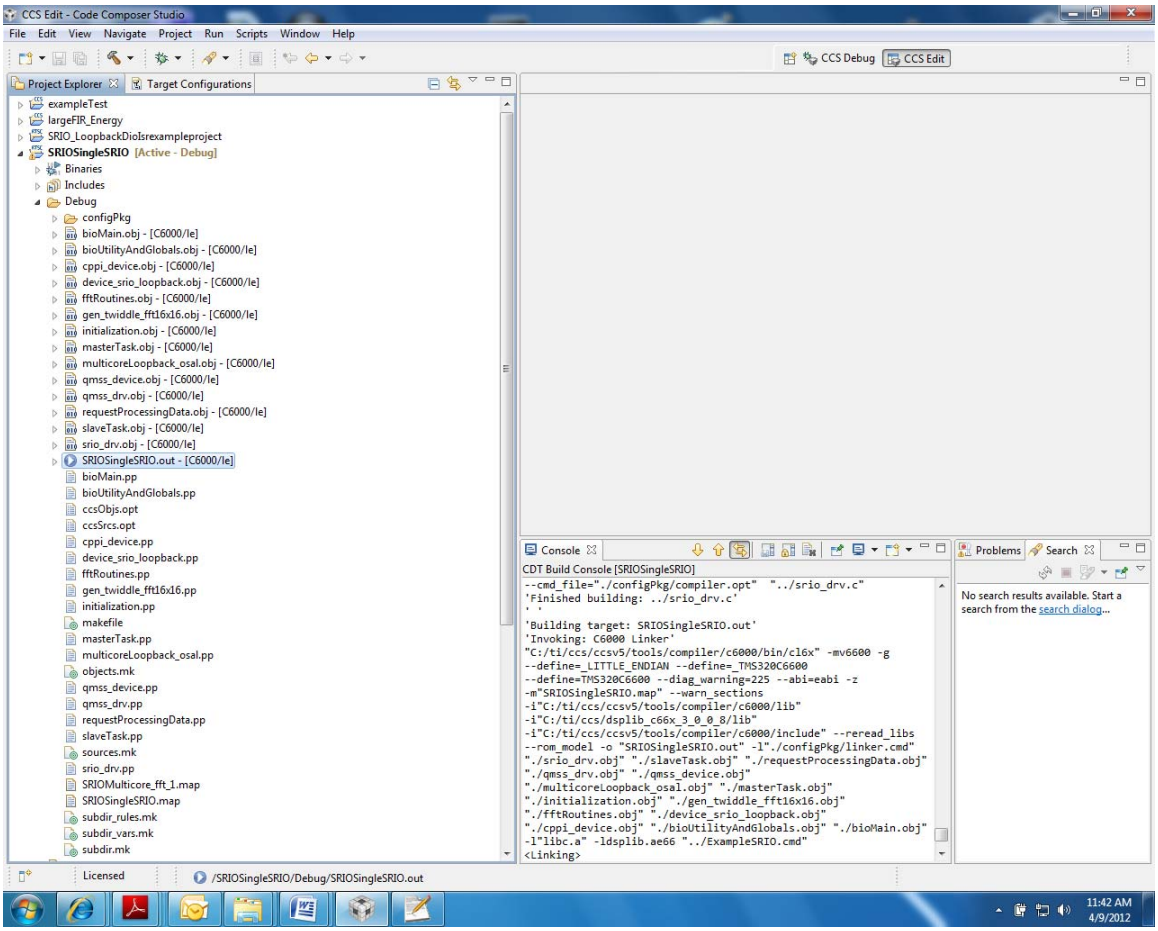


Task 2: Build the Application

1. Clean the build.

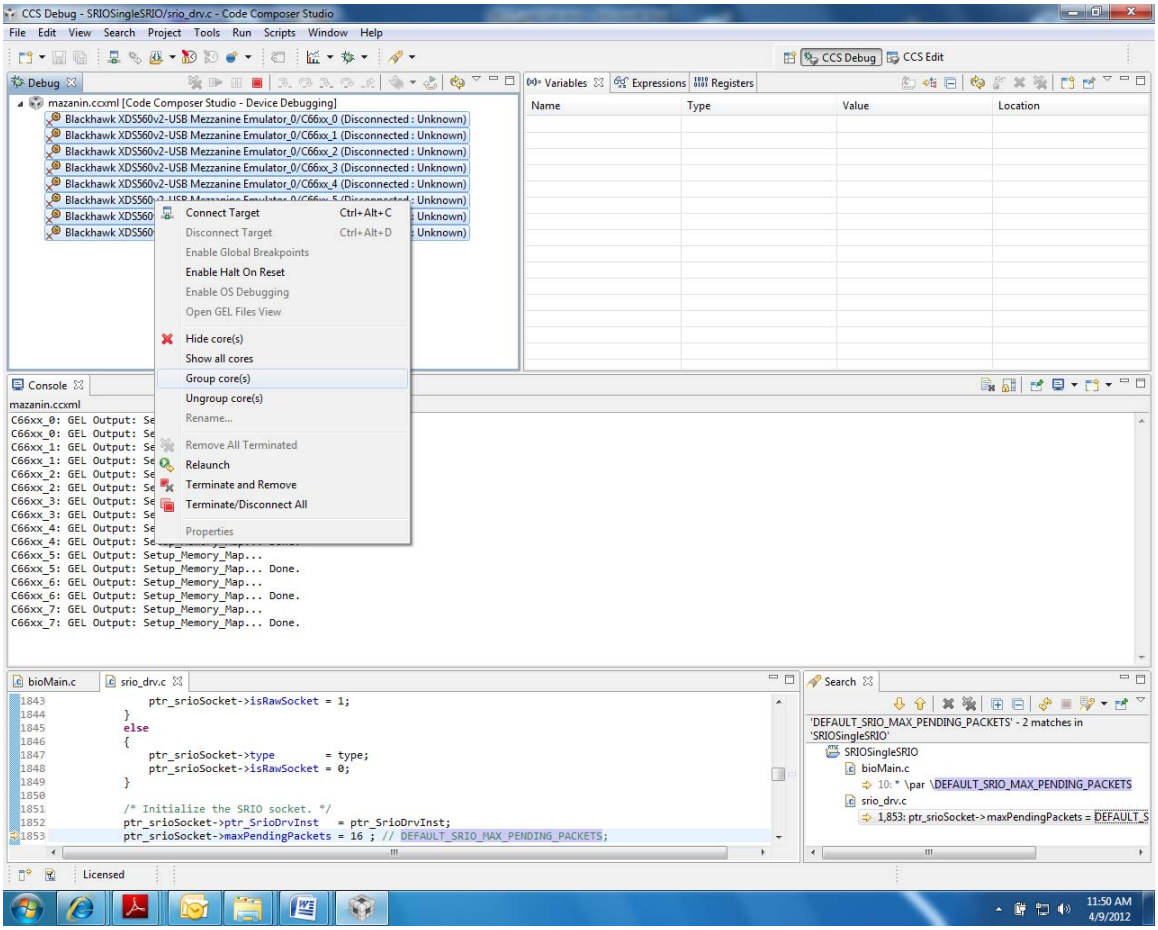
2. Build the project.

3. Verify that the executable (.out) was built by looking at the debug directory (assuming the build configuration is debug configuration).

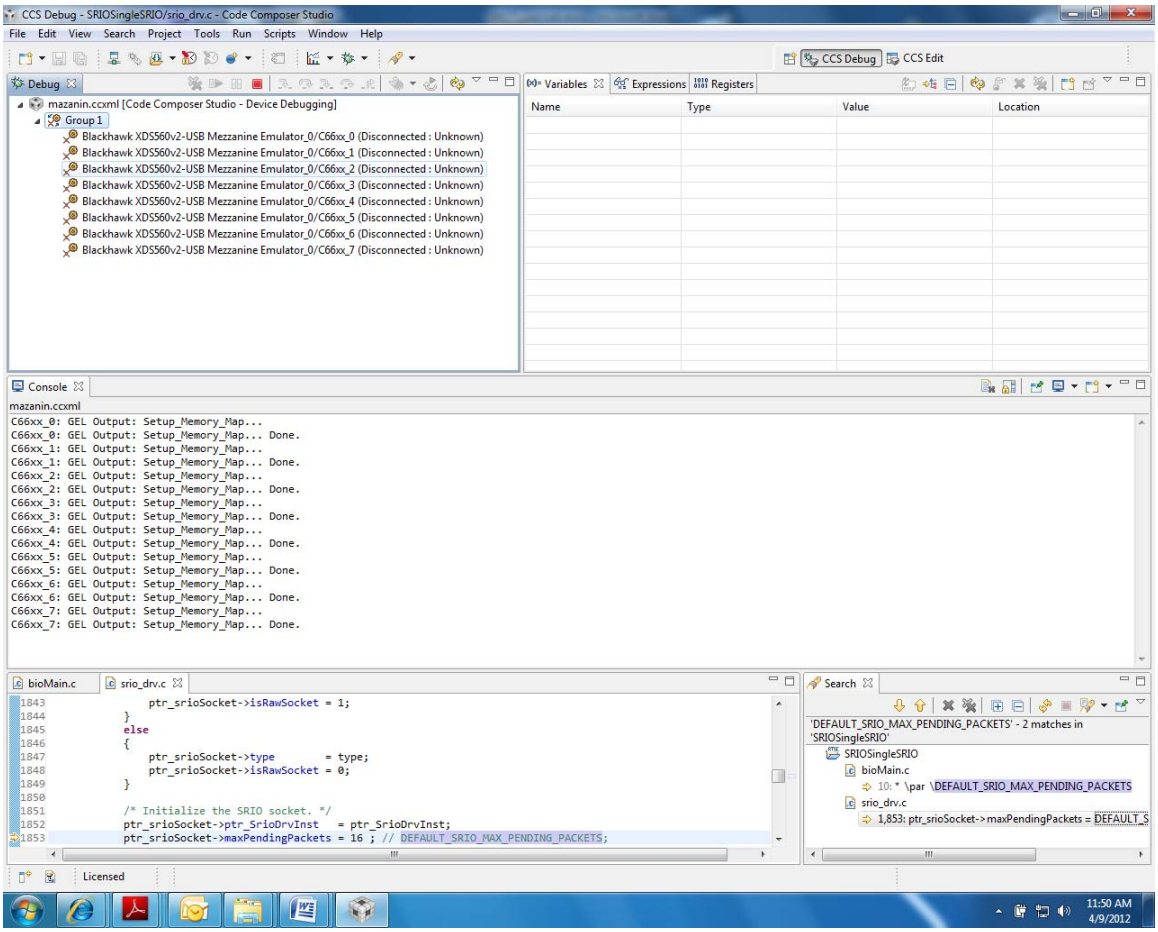


Task 3: Launch the Debugger

1. Power on the EVM, connect the USB cable to the emulator, wait for the EVM to finish boot (the red light is ON).
2. Launch your debugger.
3. Group all cores into one group as follows:



After grouping, Group 1 is defined and displayed as shown below:

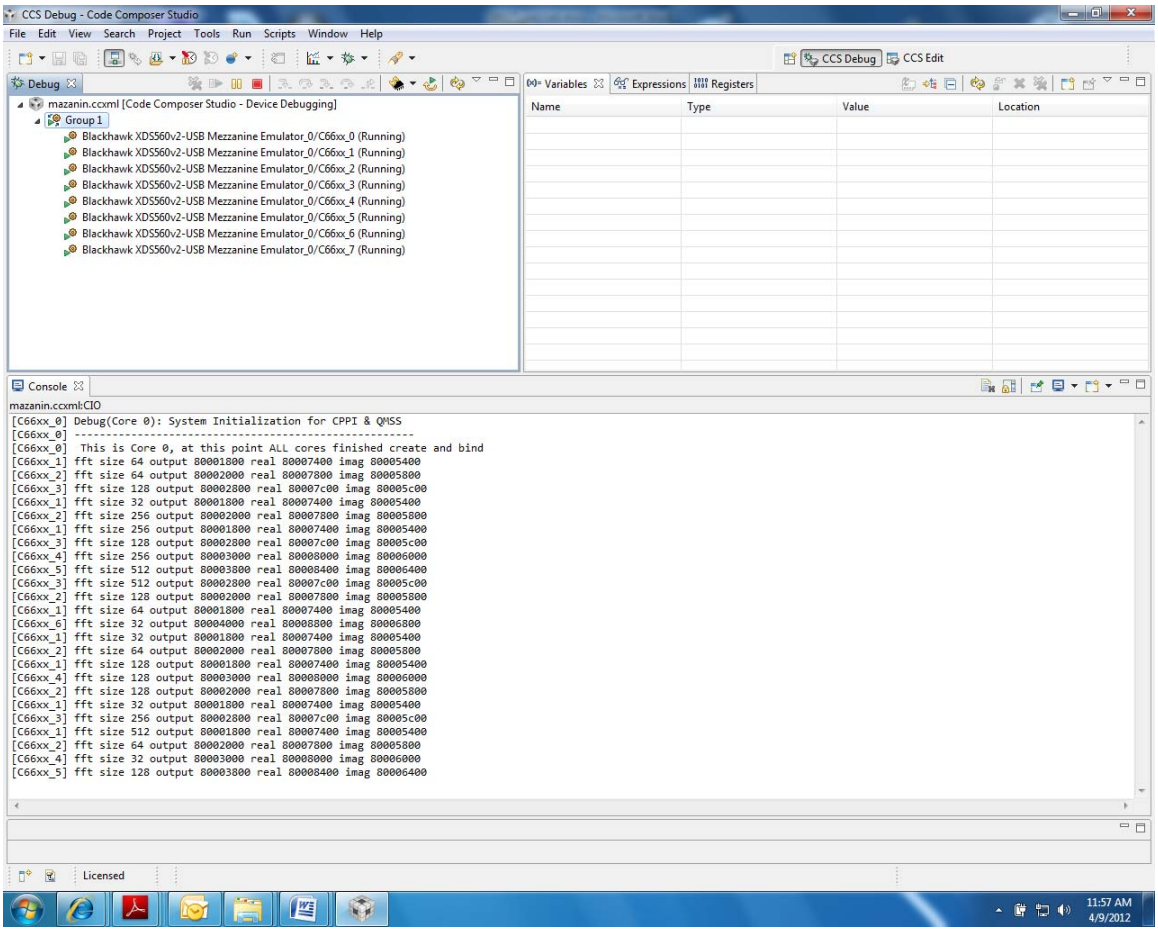


Task 4: Load and Run

1. Select Group1, and connect all cores in the group by one of the three ways:
    - From the RUN menu, select Connect Target.
    - Right click on the group name and choose Connect Target.
    - Click the Connect Target icon.
2. Load the code to all cores in the group:
    - From the RUN menu, select Load.

    OR
    - Click the Load icon.
3. Run the code in one of the following ways:
    - Press F8.
    - From the RUN menu, select Resume.
    - Click on the Resume icon (green arrow).

The output results appear as follows:



4. Observe the results, then suspend the run:

   - From the RUN menu, choose Suspend

   OR

   - Click on the Suspend icon (the yellow “pause” lines)

Task 5 (Optional): Debug the Project

When you have finished the previous task, you can optionally download the same project with some embedded bugs and then debug the code. Your instructor will point you to the location to download the buggy project.

5. Load the project to your development environment:
6. Start CCS and import the project.

Challenge Question: Can you debug the code and make it build and run properly?

Hint: There are two bugs. One bug is in the build process. Something is missing from the .cfg file.
The second bug is in the run. The code was originally developed for 6670 and then converted to 6678.

Lab 4 – Optimization Exercise

Purpose

The goal of this lab is to demonstrate some basic optimization techniques.

This lab executes on an EVM board, or can be used with the simulator in conjunction with the estimated cycle count.

Project Files

The following files are used in this lab:

1. firMain.c
2. naturalCFilter.c
3. intrinsicCFilter.c
4. utilities.c
5. test.h
6. linker.cmd

Task 1: Build and Run the Project

1. Open CCS
2. Create new project
3. Create new project file location
4. Choose “Empty Project” under “Project Templates and Examples”
5. Delete default main.c. The “main” function will now be in the file firMain.c
6. Examine the code in firMain.c to understand the functions that are being called. The generateData function generates the data sets to be operated on. naturalCFilters and intrinsicCfilters execute filters on the generated data. The former is implemented completely in C, the latter also takes advantage of compiler intrinisics.
7. Copy project files to new folder. (Your instructor will tell you where to get the files from)
8. Look at the source code
9. Open Properties and set the debug model to full and optimization levels to minimum.
10. Create an Environment Variable called PDK_ROOT that points to the root directory of the PDK. (e.g. c:\ti\ pdk_C6678_<your pdk version>\). Do this under the Project Properties->Build->Environment.
11. Start a debug project. Compile and link. Generate an out file
12. If the target is not defined yet, define a target. Assign gel file to core 0.
13. Launch the target debugger (emulation), connect and load the program into core 0
14. From the run menu, enable the clock (Run->Clock->Enable)

15. Run the code and record the cycles time for natural C function and for intrinsic function

Task 2: Compiler Optimization

16. Change the project build option. Suppress all debug features and enable the highest time optimization.

17. Re-build and re-run.

18. Record the optimized project cycles time for natural C function and for intrinsic function.

How much improvement is noted for the natural C code? _____

How much improvement is noted for the intrinsic code? _____

What issues exist within the code, if any? _____

> **Hint:** Do intrinsic functions better utilize the processor?

Task 3: Enable Software Pipelining

19. Keep the assembly file. In the project properties build compiler tab go to assembly option and check the appropriate tab

20. Rebuild the code. Find the assembly file. Hint, if the build configuration is debug, the assembly files are in the debug directory.

21. Open the interisicCFilter.asm file.

Was the compiler able to schedule the software pipeline? _____

What are the general reasons that the compiler might not schedule the software pipeline?

_____

_____

> **Hint:** Think about cases that can cause randomness in the execution timing.

22. Open the interisicCFilter.asm file

23. Analyze and answer the following question.

What reason can you see that the compiler might not be able to schedule the software pipeline?

_____

_____

> **Hint:** Think about inline function

24. Substitute the intrinsic function instead of the regular function in all the loops

25. Re-build and re-run. Look at the intrinsicCFilter.asm. Did the compiler schedule software pipeline?

26. Record the optimized project cycles time for natural C function and for intrinsic function with software pipeline

Task 4: Align the Data

27. In the intrinsicCFilter.c code, the data is read from the memory.

Challenge Question? What is the alignment of the input data? What is the alignment of the filter coefficients (in the stack)?

> **Hint:** Find Pragma that align the data. What other ways there to align the data on 64 bit boundary?

28. Change the code to tell the compiler that the data is loaded from aligned memory

29. Re-build and re-run

30. Record the optimized project cycles time for natural C function and for intrinsic function with software pipeline and aligned load.

Task 5: Cache Considerations

31. In test.h, change the number of elements to 4K, 8K and 16K

32. Record the cycle counts for each case

Challenge Question? Why is the non-linear jump in the performances

> **Hint:** Think about cache trashing

33. Change the code to take full advantage to the cache

> **Hint:** Break the data into chunks, and call each routine multiple times. Make sure to keep the sum between calls

34. Re-build and re-run

35. Record the final optimization cycle count.

36. Do you have any ideas how to further reduce execution time?

Lab 5 – Interprocessor Communication (IPC)

5A – Shared Memory Transport

Purpose

The goal of this lab is to become familiar with how to use the Interprocessor Communication Module (IPC) to communicate between applications running on different cores. We will build a project that will use the MessageQ module of IPC to pass messages between arbitrary cores on the 6678. Initially, we will be using shared memory in order to pass data between the cores. Later, we will re-configure the project to use Multicore Navigator.

Project Details

The project will generate a single .out file that will run on all cores. Core 0 is designated the "Master" core, meaning that it will be the one that is responsible for initialization tasks. A token message will be passed between randomly chosen cores 100 times. The current count of the number of passes is part of the token message. When a core receives the token message, it will send an acknowledge back to which ever core the token came from. The core that receives the 100th token message pass will also be responsible for freeing the memory used by the token message and then sending a "Done" message to all of the cores. Upon receipt of the "Done" message, each core will do its cleanup and then exit.

The application is implemented with a single task, and a polling implementation is used. Once initialized, each task will poll it's MessageQ to see if there is a message waiting. If there is, the message is read, and then action is taken based on the message type.

Task 1: Import and Examine the Skeleton Project

1. Import the InterprocessorCommunication project into Code Composer Studio v5.
2. Examine the InterprocessorCommunication.cfg project.
    1. Right Click on the file in the Project Explorer
    2. Choose Open With -> XDCScript Editor
3. Note the following lines in the .cfg that are necessary for the IPC example.
    1. The following lines include the modules that are necessary for using IPC in this manner. The MultiProc module handles the management of the various processor IDs. The IPC module is used to initialize the subsystems of IPC. The HeapBufMP module manages allocation of memory buffers from the shared heap. And the MessageQ module supports the sending and receiving of variable length messages.

```
var MessageQ    = xdc.useModule('ti.sdo.ipc.MessageQ');
var Ipc         = xdc.useModule('ti.sdo.ipc.Ipc');
var HeapBufMP   = xdc.useModule('ti.sdo.ipc.heaps.HeapBufMP');
var MultiProc   = xdc.useModule('ti.sdo.utils.MultiProc');
```

    2. The following line defines which processors will be used. In this case, we will be using all of them.

```
Ipc.procSync = Ipc.ProcSync_ALL;
```

3. These lines define the shared memory location.

```
var SHAREDMEM           = 0x0C000000;
var SHAREDMEMSIZE       = 0x00200000;
```

4. These include the SharedRegion module, which manages the shared memory allocation across processors and defines the specific location of the shared memory.

```
var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
SharedRegion.setEntryMeta(0,
    { base: SHAREDMEM,
      len:  SHAREDMEMSIZE,
      ownerProcId: 0,
      isValid: true,
      name: "DDR2 RAM",
    })
```

4. Open and examine InterprocessorCommunication.h. Note specifically the enumeration for the different message types, and the myMsg structure. The exact myMsg structure definition is arbitrary, except for the requirement that a MessageQ_MsgHeader element is the first item. In this case, the message consists only of a Message header, a message type, and an int that holds the count of the number of times that the token has been passed.
5. Open and Examine InterprocessorCommunication.c. There are 3 functions.
    1. Main – Dynamically creates the application task, calls Ipc_start to synchronize the processors, and then calls Bios_start. Nothing magical here.
    2. findNextCore – This functions is very simple. It just generates a random number between 0 and MAX_NUM_CORES -1. It ensures that the generated number is different than the current core so that none of the cores are passing the token to themselves. *Note: There is no reason that a core can't send a message to itself in this manner. In fact we do that at the end with the "Done" message. But we prevent the token passing from doing this so that the output shows a more interesting token path.*
    3. task_fxn - This is where all of the magic happens. The Master Core does the initialization, and creates the shared heap, and all of the slave cores connect to it. All cores register the heap with MessageQ, and they create a "Local" MessageQ where their messages will be received. Each core creates a small lookup table that associates the core number with the appropriate MessageQ Id. Finally, Core 0 creates the token message, passes it to a random core, and then all cores just wait for messages to be received and acted on.

Task 2: Add IPC API's

6. Some of the APIs have been left out of task_fxn. Each location where source needs to be added is marked with the comment "TODO: IPC #<x> -" followed by a description of the task. The <x> holds the task number. Each element of the code that needs to be added is a single function call.

TIP: CCSv5 has a Tasks window that will give shortcuts to these tasks. One way to get to it is from the Window->Show View->Other menu in CCS. When that window opens, look under the "General" folder and double click "Tasks". You can then double click on any of the tasks and you will immediately be taken to the source where that task is located.

Hints:

The following are hints about the code to be added. Try to add the code using only the descriptions contained within the source code. If you need more clues, use the hints below.

1. Allocate Memory for the Token Message
   - The memory for a MessageQ message is allocated by calling the API MessageQ_alloc(). The parameters are the ID of the heap that the memory will be allocated from, and the size of the message to be allocated.
2. Pass the token to the destination core.
   - MessageQ messages are passed by calling the MessageQ_put() API. The parameters to the function are the destination Queue Id, and the pointer to the message.
3. Get a Message from the local queue.
   - Messages are retrieved from the MessageQ with the MessageQ_get() API. The MessageQ_get() function is a blocking call. The parameters it accepts are the handle of the MessageQ to check, the *address* of the pointer to the message, and an enumerated parameter that specifies how long to block until a timeout occurs. MessageQ_FOREVER specifies that a timeout will never occur. The return value is zero if the message is successfully retrieved.
4. Get the Reply Queue ID for the token.
   - The reply queue Id is obtained via the MessageQ_getReplyQueue() API. The only parameter needed is a pointer to the message. The return value is of type MessageQ_QueueId.
5. Allocate the acknowledge message
   - See Hint #1
6. Send the acknowledge message.
   - See Hint #2
7. Free the memory used by the token message.
   - The memory is freed by the API MessageQ_free. The only parameter is the pointer to the message.
8. Note that the loop that sends the "Done" message will send one of those messages to itself.

Set the Reply Queue for the Token Message

- This is done using the MessageQ_setReplyQueue() API. Parameters are the handle to the MessageQ and the pointer to the message.

Task 3: Build and Run the application

7. Once everything has been added to InterprocessorCommunication.c, build the project. It should build without errors or warnings. If it doesn't build properly, attempt to figure out why. Otherwise, ask the instructor.
8. Launch the Debug Session and Connect to all of the cores.
9. Load InterprocessorCommunication.out to all of the cores.
10. Run All of the Cores

Task 4: Verify the Output

11. When the device runs, you should see printf output in the console window that looks like the output below. The order of your output is expected to be different since the token is being passed randomly.

```
[C66xx_1] Token Received - Count = 1
[C66xx_0] Ack Received
[C66xx_5] Token Received - Count = 2
[C66xx_1] Ack Received
[C66xx_2] Token Received - Count = 3
[C66xx_1] Token Received - Count = 4
[C66xx_5] Ack Received
[C66xx_5] Token Received - Count = 5
[C66xx_1] Ack Received
[C66xx_2] Ack Received
[C66xx_3] Token Received - Count = 6
```

12. Ensure that the application behaves as expected, by checking the following items.
    1. Since Core 0 is passes the token first, the first Ack received message should be by Core 0. This is the case with the example above.
    2. In general, you should see the pattern of how the token was passed. We know that Core 0 passes the first token. From the output above, it looks like The order is 0 -> 1 -> 5 -> 1 -> 2 -> 1 -> 5 -> 3, because that's the order of the Token receive messages. The Ack Received messages should follow a similar, pattern, but it likely won't look identical. (This can be attributed to the non-realtime nature of the printf function, but in reality, the pattern would be identical.) In this case, the pattern is identical for the first few passes.
    3. The last Token Message should have a count of 100.
    4. There should be a "Done Received" message from each core in use near the very end.
    5. Each core should halt at the C$$EXIT symbol when the demo is complete.

5B – Multicore Navigator Transport

NOTE - Due to a bug in the current release, the maximum number of cores in this demo is limited to 2.

With some manipulation (see the appendix) we can increase the number of cores to 4. This fix will be part of next release

An IR was submitted to fix the problem and enable all 8 cores.

Task 1 – Add the OS Abstraction Layer Functions

1. Add the file IPC/common/src/Ipc_osal.c to your project via a link relative to the project location. These are functions that the user has to generate that are specific to whatever OS they are using. The QMSS and CPPI Low Level Drivers will make calls to these functions.
2. Add a #include to Ipc.c that points to /common/include/Ipc_osal.h. You can use the relative path `"../common/include/Ipc_common.h"`

Task 2 – Add the Initialization Functions

3. Add the file IPC/common/src/Ipc_MulticoreNavigatorInit.c to your project via a link relative to the project location. This file provides initialization functions for the QMSS and CPPI.
4. Add a #include to Ipc.c that points to /common/include/Ipc_MulticoreNavigatorInit.h. You can use a relative path similar to the one used in step 2.

Task 3 – Add additional module #includes

5. Add a #include for all of the following SYS/Bios modules from Ipc.c

```
#include <ti/sysbios/family/c66/Cache.h>
#include <ti/drv/qmss/qmss_drv.h>
#include <ti/drv/qmss/qmss_firmware.h>
#include <ti/transport/ipc/qmss/transports/TransportQmss.h>
#include <ti/drv/cppi/cppi_drv.h>
```

Task 4 – Add a call to systemInit()

6. Call the systemInit function from main, but only on the master core (typically core 0). This function returns 0 if it is successful. Check the return parameter and report an error initializing QMSS if the return value is not 0.

Task 5 – Update the .cfg File.

7. There are additional modules that need to be added to the .cfg file. Add the following lines appropriately. First add the QMSS/CPPI Memory Settings

```
/* QMSS/CPPI Memory Settings */

var Cppi = xdc.loadPackage('ti.drv.cppi');
var Qmss = xdc.loadPackage('ti.drv.qmss');

Program.sectMap[".qmss"] = new Program.SectionSpec();
Program.sectMap[".qmss"] = "MSMCSRAM";
```

8. Add the configurations for the Transport

```
/* Use Multicore Navigator IPC */
var TransportQmss = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmss');
MessageQ.SetupTransportProxy= xdc.module(Settings.getMessageQSetupDelegate());
var TransportQmssSetup = xdc.useModule('ti.transport.ipc.qmss.transports.TransportQmssSetup');
MessageQ.SetupTransportProxy = TransportQmssSetup;

TransportQmssSetup.descMemRegion = 0;
Program.global.descriptorMemRegion = TransportQmssSetup.descMemRegion;
Program.global.numDescriptors = 8192;
Program.global.descriptorSize = cacheLineSize;

TransportQmss.numDescriptors = Program.global.numDescriptors;
TransportQmss.descriptorIsInSharedMem = true;
TransportQmss.descriptorSize = Program.global.descriptorSize;
TransportQmss.useAccumulatorLogic = false;
TransportQmss.pacingEnabled = false;
TransportQmss.intThreshold = 1;
TransportQmss.timerLoadCount = 0;
TransportQmss.accuHiPriListSize = 2100;
```

9. Remove the following lines that configure the Notify Module, as it is not needed anymore. Also remove the SharedMemoryTransport line, as it has been replaced with a QMSS equivalent above.

```
var Notify              = xdc.module('ti.sdo.ipc.Notify');
Notify.SetupProxy       = xdc.module(Settings.getNotifySetupDelegate());
MessageQ.SetupTransportProxy = xdc.module('ti.sdo.ipc.transports.TransportShmSetup');
```

Task 5 – Build and Run

These should be all of the steps needed to update the existing project from using the Shared Memory transport to use the MulticoreNavigator transport. Notice that we changed *NONE* of the MessageQ code, but only some configuration options in the .cfg file and a bit of initialization code in the source. So, you

can write code for one transport and not have to care which transport will eventually be used in the end. This really enables great source code reuse without modifications across platforms.

Appendix – instructions how to have four cores

Here are instructions for incorporating the fix:

1) In your c6678 PDK replace packages/ti/transport/ipc/qmss/transports/TransportQmss.c with the attached file and transportQmss.xdc.
2) Open a command window and navigate to the packages/ti/transport/ipc/qmss/transports directory
3) Setup your build environment by setting the following paths, where xx_yy_zz are component versions you're using

```
set XDCPATH=c:\ti\bios_6_xx_yy_zz\packages\

set XDCPATH=%XDCPATH%;c:\ti\ipc_1_xx_yy_zz\packages\


set XDCCGROOT=c:\ti\ccsv5\tools\compiler\c6000


set PATH=%PATH%;c:\ti\xdctools_3_xx_yy_zz\
```

4) Clean the transport

```
>xdc clean
```

5) Rebuild the transport

```
>xdc –PR .
```

6) Clean and rebuild your CCS project. You'll need to do the clean in order to pull in the rebuilt qmss transport library.