|  |
| --- |
| Texas Instruments |
| Keystone Multicore Workshop |
| Lab Manual |

Contents

[Lab Preparations 1](#_Toc364151618)

[EVM Configuration 1](#_Toc364151619)

[Create a new target in CCS 1](#_Toc364151620)

[Lab 1: CCS Basics (SRIO Loopback) 2](#_Toc364151621)

[I. Purpose 2](#_Toc364151622)

[Task 1: Import the Example Project 2](#_Toc364151623)

[Task 2: Verify and Set Project Properties 3](#_Toc364151624)

[Task 3: Build the Project 5](#_Toc364151625)

[Task 4: Connect to the Target EVM 5](#_Toc364151626)

[Task 5: Load and Run the Program 6](#_Toc364151627)

[Lab 2: Hyperlink 8](#_Toc364151628)

[I. Purpose 8](#_Toc364151629)

[Task 1: Import the Example Project 8](#_Toc364151630)

[Task 2: Verify and Set Project Properties 8](#_Toc364151631)

[Task 3: Loopback Mode 10](#_Toc364151632)

[Task 4: Build the Project 10](#_Toc364151633)

[Task 5: Connect to the EVM 11](#_Toc364151634)

[Task 6: Load and Run the Program 11](#_Toc364151635)

[Task 7: Board-to-board Hyperlink Example 12](#_Toc364151636)

[Lab 3: SRIO Type 11 13](#_Toc364151637)

[I. Purpose 13](#_Toc364151638)

[II. Project Files 13](#_Toc364151639)

[Task 1: Import the Project 13](#_Toc364151640)

[Task 2: Build the Project 13](#_Toc364151641)

[Task 3: Connect to the EVM 16](#_Toc364151642)

[Task 4: Load and Run 17](#_Toc364151643)

[Lab 4: Optimization 19](#_Toc364151644)

[I. Purpose 19](#_Toc364151645)

[II. Project Files 19](#_Toc364151646)

[Task 1: Build and Run the Project 19](#_Toc364151647)

[Task 2: Connect to the EVM 21](#_Toc364151648)

[Task 3: Load and Run the Program 22](#_Toc364151649)

[Task 4: Compiler Optimization 23](#_Toc364151650)

[Task 5: Enable Software Pipelining 25](#_Toc364151651)

[Task 6: Align the Data 26](#_Toc364151652)

[Task 7: Enable the MUST\_ITERATE Pragma 26](#_Toc364151653)

[Task 8: Cache Considerations 26](#_Toc364151654)

[Task 9: Cache Analysis 27](#_Toc364151655)

[Task 10: Change the Code to Speed Up to 32K 27](#_Toc364151656)

[Partial List of Intrinsics (Compiler User’s Guide) 28](#_Toc364151657)

[Lab 5: Using Advanced Debug to Understand Strange Results 29](#_Toc364151658)

[I. Understanding the Problem 29](#_Toc364151659)

[Why the Debug Version is Used 29](#_Toc364151660)

[II. Lab Instructions 29](#_Toc364151661)

[Task 1: View the 4K Case 29](#_Toc364151662)

[Task 2: Looking at the Cache Lines for 4K Case 31](#_Toc364151663)

[Task 3: View the Cache Lines for 16K Case 35](#_Toc364151664)

[Task 4: View the Cache Lines for 8K Case 36](#_Toc364151665)

[Task 5: Finding the Bug 37](#_Toc364151666)

[Lab 6: Inter Processor Communication (IPC) 38](#_Toc364151667)

[Shared Memory Transport 38](#_Toc364151668)

[I. Purpose 38](#_Toc364151669)

[II. Project Details 38](#_Toc364151670)

[III. Lab Instructions 38](#_Toc364151671)

[Task 1: Import & Examine the Skeleton Project 38](#_Toc364151672)

[Task 2: Add IPC APIs 40](#_Toc364151673)

[Task 3: Build the Project 42](#_Toc364151674)

[Task 4: Connect to the EVM 42](#_Toc364151675)

[Task 5: Load and Run the Program 42](#_Toc364151676)

[Lab 7: Using MPAX to Define Private Core Memory in DDR 45](#_Toc364151677)

[I. Purpose 45](#_Toc364151678)

[II. Overview 45](#_Toc364151679)

[Short Description of MPAX (Memory Protection and Extension) 45](#_Toc364151680)

[Coherency Discussion 46](#_Toc364151681)

[Usage of EDMA to Move Data to and from Private Memory 46](#_Toc364151682)

[Platform Configuration and the Memory Map 46](#_Toc364151683)

[III. Lab Instructions 48](#_Toc364151684)

[Task 1: Run the Example Code 48](#_Toc364151685)

[Task 2: Using Trace to Verify the Write Physical Address 49](#_Toc364151686)

[Additional Considerations 60](#_Toc364151687)

[Lab 8: STM library and system Trace 61](#_Toc364151688)

[I. Purpose 61](#_Toc364151689)

[II. Project Files 61](#_Toc364151690)

[Task 1: Build and Run the Project 61](#_Toc364151691)

[Task 2: Connect to the EVM 64](#_Toc364151692)

[Task 3: Load the Program and configure the trace 65](#_Toc364151693)

[Task 3: Run the Program 68](#_Toc364151694)

# Lab Preparations

### EVM Configuration

1. Set the EVM to ‘no boot’ mode as shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Boot Mode** | **DIP SW3**  **(Pin 1, 2, 3, 4)** | **DIP SW4**  **(Pin 1, 2, 3, 4)** | **DIP SW5**  **(Pin 1, 2, 3, 4)** | **DIP SW6**  **(Pin 1, 2, 3, 4)** |
| No boot | (off, on, on, on) | (on, on, on, on) | (on, on, on, on) | (on, on, on, on) |

NOTE: 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>

## Create a new target in CCS

1. Launch CCS by double-clicking the icon on the desktop.  
     
   NOTE: As CCS initializes, a pop-up will appear with a default workspace. Replace the default workspace with “C:/ti/workspace.”
2. Create a new target configuration:
   1. Select the CCS menu option *View 🡪 Target Configurations*.
   2. Select *User Defined*.
   3. Right-click and select *New Target Configuration*.
3. Enter the name of the new target configuration in the *File Name:* text box.
   1. Set the File name based on the EVM model, *<model>.ccxml*  
      For example, ‘EVM6678LE.ccxml’
   2. Leave the *Location* the default value:  
      “C:\Documents and Settings\student\ti\CCSTargetConfigurations”
   3. Click the *Finish* button. The .ccxml file will now open in a GUI-based view with the *Basic* tab active.
4. First step to define the new target configuration is to select the connection type in the *Basic* Tab.
5. For selecting the correct connection, identify your EVM in the table below and set the properties accordingly:

|  |  |  |
| --- | --- | --- |
| **EVM Model** | **Emulator Type** | **Device** |
| EVM6657L | Texas Instruments XDS100v2 USB Emulator | TMS320C6657 |
| EVM6657LE | Blackhawk XDS560v2-USB Mezzanine Card | TMS320C6657 |
| EVM6678L | Texas Instruments XDS100v1 USB Emulator | TMS320C6678 |
| EVM 6678LE | Blackhawk XDS560v2-USB Mezzanine Card | TMS320C6678 |

1. The *Connection* drop-down menu identifies the emulator type, as shown in the table above. For example, ‘Blackhawk XDS560v2-USB Mezzanine Card’
2. *Board or Device* identifies the TI processor device, as shown in the table above. For example, ‘TMS320C6678’
   1. Under *Save Configuration*, click the *Save* button.
3. Second step is to configure setup in *Advance* Tab
   1. Click the *Advanced* tab at the bottom of the screen.
   2. Select Core 0 on the target device:
      * *TMS320C6657\_0* 🡪 *IcePick\_C\_0* 🡪 *Subpath\_1* 🡪 *C66XP\_0*

OR

* + - *TMS320C6678\_0* 🡪 *IcePick\_D* 🡪 *subpath\_0* 🡪 *C66x\_0*
  1. You will now see a sub-window called *Cpu Properties* that allows you to choose an *initialization script*.
  2. Locate the appropriate GEL file, then click *Open*:
     + For EVM6657L/LE, select:  
       C:\ti\ccsv5\ccs\_base\emulation\boards\evmc6657l\gel\evmc6657l.gel”
     + For EVM6678L/LE, select:  
       C:\ti\ccsv5\ccs\_base\emulation\boards\evmc6678l\gel\evmc6678l.gel”
  3. Click the *Save* button.

# Lab 1: CCS Basics (SRIO Loopback)

## I. Purpose

The purpose of this lab is to demonstrate how to build and run a very basic Code Composer Studio v5 (CCS) project on the C6678 EVM. The Direct IO Loopback example delivered with the MCSDK is used to help illustrate these concepts.

### Task 1: Import the Example Project

1. Launch CCS by double-clicking the icon on the desktop.  
     
   NOTE: As CCS initializes, a pop-up will appear with a default workspace. Replace the default workspace with “C:/ti/workspace” .
2. Once CCS starts, verify that the perspective is set to *CCS Edit*.
3. Discover the new packages installed in folders other than C:\ti.
   1. Select the CCS menu option *Window* 🡪*Preferences*
   2. In the pop up window that appears, select *Code Composer Studio* 🡪 *RTSC* 🡪 Products
   3. Add the master folder into *Tool Discover Path* by clicking the *Add* in the upper right corner of the pop-up window and select the master folder.
   4. For example, select C:\ti\MCSDK\_2\_1\_2\_6 to add in the  *Tool Discovery Path.*
   5. Click *Refresh* to update the discovered tools list.
4. Import the example project as follows:
   1. Select the CCS menu option *Project* 🡪 *Import Existing CCS Eclipse Project*
   2. Set *Select search directory* to locate the example projects available for your EVM:
      * (C6657L/LE) C:\ti\mcsdk\pdk\_C6657\_1\_1\_2\_5\packages\ti\drv\exampleProjects
      * (C6678L/LE) C:\ti\mcsdk\pdk\_C6678\_1\_1\_2\_5\packages\ti\drv\exampleProjects
   3. From the list of *Discovered projects*, place a check mark in the box next to *SRIO\_LoopbackDioIsrexampleproject***.**

NOTE: There are multiple SRIO projects with similar names. Verify that the project you import matches exactly with the name as shown above.

* 1. Place a check mark on *Copy projects into workspace*.
  2. Click the *Finish* button.

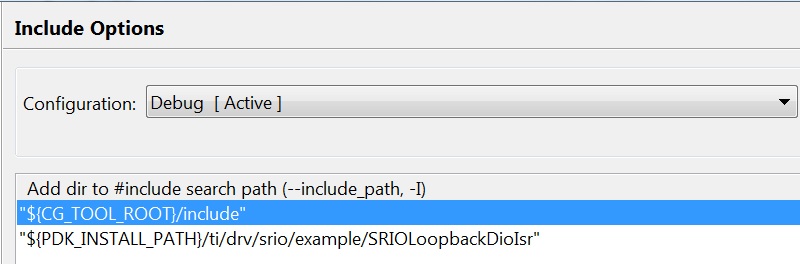
The *SRIO\_LoopbackDioIsrexampleproject* project should now appear in the CCS *Project Explorer* on the left-hand side of your screen.

### Task 2: Verify and Set Project Properties

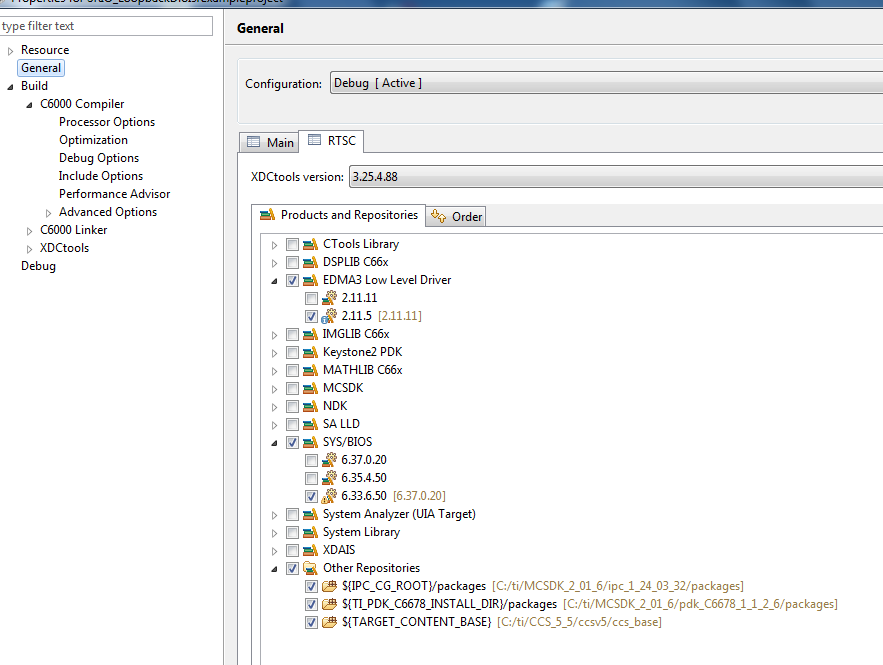
1. In *Project Explorer*, right click on *SRIO\_LoopbackDioIsrexampleproject* and select *Properties*.
2. Select *General* properties.
3. Choose the *Main* tab and set/verify the *Device* properties as follows:
   * ‘Family = C6000’
   * ‘Variant = Generic C66x Device’
4. Select *Build* properties.
5. Choose *C6000 Compiler* 🡪 *Processor Options* and set/verify the following properties:
   * ‘Configuration = Debug’
   * ‘Target processor version = 6600’
   * ‘Application binary interface = eabi’

Note – different version of CCS may have slightly different GUI. The Application binary interface tab may be part of the *main* window and not the *processor option* window.

1. Select *Build* properties.
2. Choose *C6000 Compiler 🡪 Optimization* and set/verify the following properties:
   * ‘Optimization level = 0’
   * ‘Optimize for code size = 0’
3. Select *Build* properties
4. Choose *C6000 Compiler 🡪 Debug Options* and set/verify the following properties:
   * ‘Debugging model = Full symbolic debug’
5. Select *Build* properties, choose *C6000 Compiler 🡪 Include Options* and ensure that include paths are setup as shown in the snapshot below:

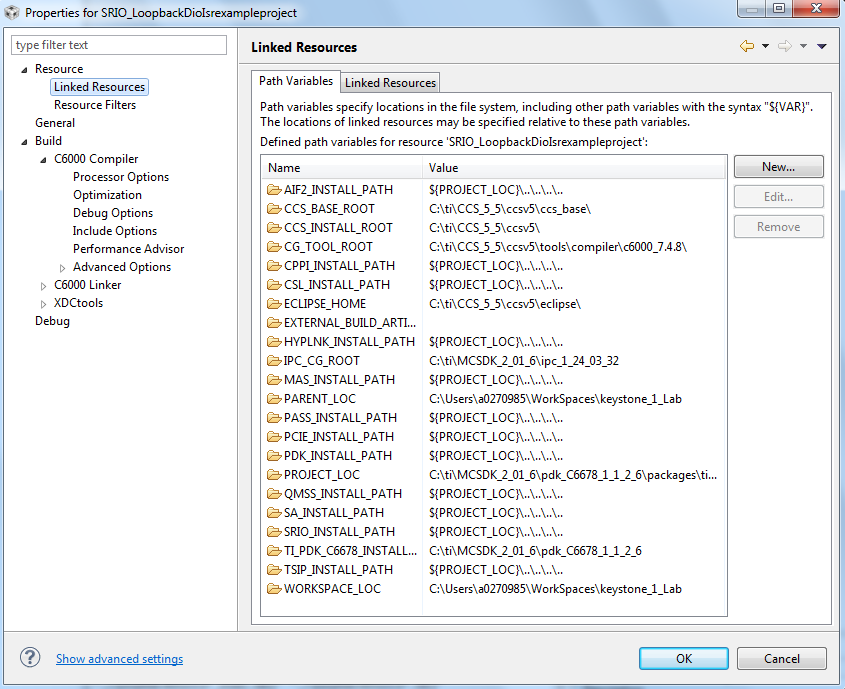


1. Review RTSC setting
   1. Make sure that ONLY the repository that are defined in the screen shot below are checked.



* 1. If there is a red sign next to one of the repositories, and it says that the system could not resolve the location, you have to edit the path to the repository. The path was defined by the example developer and may not agree with your setting.

To do so select Resources->Linked resources in the properties page for the project. If the path (either ${IPC\_CG\_ROOT} or ${TI\_PDK\_C6678\_INSTALL\_DIR} does not exist, add a new path and use the folder tab to find the correct path. If the path exists but does not reflect the location in your Laptop, select the incorrect path and edit it to the correct location.



1. Click the *OK* button to save the project properties and close the *Properties* window.

### Task 3: Build the Project

1. In *Project Explorer*, select the *SRIO\_LoopbackDioIsrexampleproject* project.
2. Build the project:
   1. Select the CCS menu option *Project* 🡪 *Build Project*

OR

Right-click on the project in *Project Explorer* and select *Build Project*

OR

Click on the hammer icon

1. CCS will now attempt to compile and link the project. This may take a few minutes to complete.
2. Please direct your attention to the CCS *Console*. On a successful build, you will see no errors generated in the *Problems* window (NOTE: There may be warnings) and the following message should display in the *Console* window:

'Finished building target: SRIO\_LoopbackDioIsrexampleproject.out'

\*\*\*\* Build Finished \*\*\*\*

**QUESTIONS:**

Was the file SRIO\_LoopbackDioIsrexampleproject.out generated? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

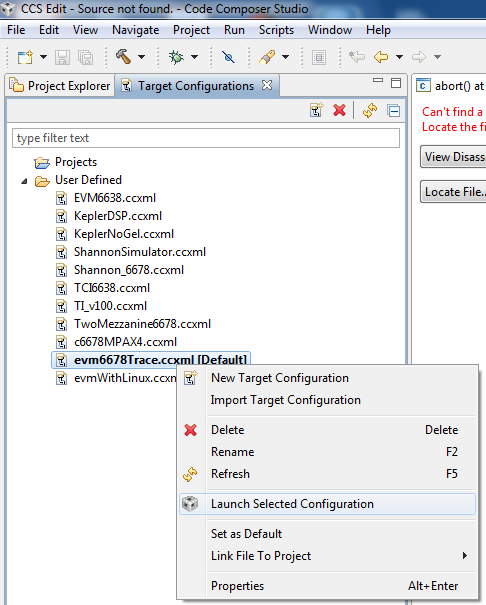
1. From the *CCS Edit* perspective, check the *Binaries* or *Debug* directory. From the *CCS Debug* perspective, check the *Console*.

### Task 4: Connect to the Target EVM

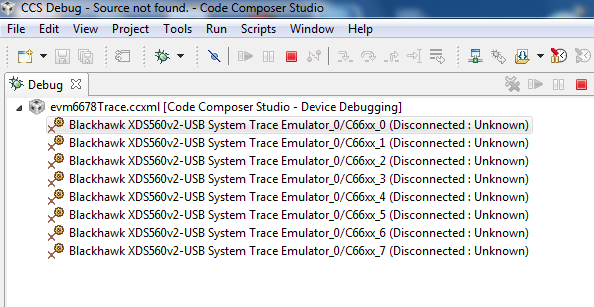
1. Click the *Open Perspective* (available right top corner of the *CCS*).
2. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
3. Connect the power adapter to your EVM, then connect your laptop to the emulator port on the EVM using the provided USB cable. If you are using the XDS560v2, wait till the solid red light appears before proceeding to the next step.

NOTE: The Windows “Found New Hardware Wizard” may pop-up when you first connect the emulator via USB to the laptop. Select “Yes, this time only” 🡪 Next 🡪 “Install the software automatically” 🡪 Next, and allow the drivers to install on your system. Then click “Finish.” You may need to restart CCS at this point.

1. Select the CCS menu option *View* 🡪 *Target Configurations*. Your newly-created .ccxml target configuration file should be available under *User Defined* target configurations.
2. If more than one target is configured, select the target that you defined, right click and set it as default
3. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.



1. This will bring up the *Debug* window.



* 1. Select Core 0 (*C66x\_0*)
  2. Right click and select *Connect Target*.

### Task 5: Load and Run the Program

1. Select Core 0 and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*.
   2. Click *Browse project…*
   3. A pop-up will appear with the projects names.
   4. Select *SRIO\_LoopbackDioIsrexampleproject*
   5. Click Debug, then select *SRIO\_LoopbackDioIsrexampleproject.out* and click *OK*.
   6. Click *OK* to load the application to the target (Core 0).
2. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.

OR

Click on the green arrow

1. Once the program completes successfully, you will see the message *“Debug(Core 0): DIO with Interrupts example completed successfully.”*
2. Then select the CCS menu option *Run* 🡪 *Suspend*.

OR

Click on the two yellow bars next to the green arrow

The expected output on the console should appear as follows:

[C66xx\_0] Executing the SRIO DIO example on the DEVICE

[C66xx\_0] Debug(Core 0): System Initialization for CPPI & QMSS

[C66xx\_0] Debug(Core 0): Queue Manager and CPPI are initialized.

[C66xx\_0] Debug(Core 0): Host Region 0x8268f0

[C66xx\_0] Debug(Core 0): SRIO Driver has been initialized

[C66xx\_0] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[C66xx\_0] \*\*\*\*\*\*\* DIO Socket Example with Interrupts (Core 0) \*\*\*\*\*\*\*\*

[C66xx\_0] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[C66xx\_0] Debug(Core 0): Starting the DIO Data Transfer - Src(0) 0x@1081b100 Dst(0) 0x@1081b200

. . .

[C66xx\_0] Debug(Core 0): Starting the DIO Data Transfer - Src(8) 0x@1081c100 Dst(8) 0x@1081c200

[C66xx\_0] Debug(Core 0): DIO Socket (0) Send for iteration 0

[C66xx\_0] Debug(Core 0): ISR Count: 1

. . .

[C66xx\_0] Debug(Core 0): DIO Socket (2) Send for iteration 2

[C66xx\_0] Debug(Core 0): ISR Count: 9

[C66xx\_0] Debug(Core 0): Transfer Completion without Errors - 9

[C66xx\_0] Debug(Core 0): Transfer Completion with Errors - 0

[C66xx\_0] Debug(Core 0): DIO Transfer Data Validated for all iterations

[C66xx\_0] Debug(Core 0): DIO Data Transfer (WRITE) with Interrupts Example Passed

[C66xx\_0] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[C66xx\_0] \*\*\*\*\*\*\* DIO Socket Example with Interrupts (Core 0) \*\*\*\*\*\*\*\*

[C66xx\_0] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[C66xx\_0] Debug(Core 0): Starting the DIO Data Transfer - Src(0) 0x@1081b100 Dst(0) 0x@1081b200

. . .

[C66xx\_0] Debug(Core 0): Starting the DIO Data Transfer - Src(8) 0x@1081c100 Dst(8) 0x@1081c200

[C66xx\_0] Debug(Core 0): DIO Socket (0) Send for iteration 0

[C66xx\_0] Debug(Core 0): ISR Count: 10

[C66xx\_0] Debug(Core 0): DIO Socket (1) Send for iteration 0

[C66xx\_0] Debug(Core 0): ISR Count: 11

[C66xx\_0] Debug(Core 0): DIO Socket (2) Send for iteration 0

[C66xx\_0] Debug(Core 0): ISR Count: 12

. . .

[C66xx\_0] Debug(Core 0): DIO Socket (0) Send for iteration 2

[C66xx\_0] Debug(Core 0): ISR Count: 16

[C66xx\_0] Debug(Core 0): DIO Socket (1) Send for iteration 2

[C66xx\_0] Debug(Core 0): ISR Count: 17

[C66xx\_0] Debug(Core 0): DIO Socket (2) Send for iteration 2

[C66xx\_0] Debug(Core 0): ISR Count: 18

[C66xx\_0] Debug(Core 0): Transfer Completion without Errors - 9

[C66xx\_0] Debug(Core 0): Transfer Completion with Errors - 0

[C66xx\_0] Debug(Core 0): DIO Transfer Data Validated for all iterations

[C66xx\_0] Debug(Core 0): DIO Data Transfer (READ) with Interrupts Example Passed

[C66xx\_0] Debug(Core 0): Allocation Counter : 81

[C66xx\_0] Debug(Core 0): Free Counter : 72

[C66xx\_0] Debug(Core 0): DIO with Interrupts example completed successfully.

# Lab 2: Hyperlink

## I. Purpose

The purpose of this lab is to demonstrate how to leverage the Hyperlink interface on KeyStone.

We start out with a Hyperlink example from the MCSDK, and run it in loopback mode, i.e. the same C66x EVM acts as both the sender and the receiver of packets. Only one C66x EVM is required for this part of the lab.

The second part of the lab demonstrates the HyperLink connection between two C66x EVMs. As a result, this lab requires collaboration with a neighboring student. One C66xx EVM acts as the sender (Tx) and the other will act as the receiver (Rx). A Hyperlink cable or connector board is required to connect the two EVMs.

### Task 1: Import the Example Project

1. Move to the *CCS Edit* Perspective.
2. Import the example project as follows:
   1. Select the CCS menu option *Project* 🡪 *Import Existing CCS Eclipse Project*
   2. Set *Select search directory* to locate the example projects available for your EVM:
      * (C6657L/LE) C:\ti\mcsdk\pdk\_C6657\_1\_1\_2\_5\packages\ti\drv\exampleProjects
      * (C6678L/LE) C:\ti\mcsdk\pdk\_C6678\_1\_1\_2\_5\packages\ti\drv\exampleProjects
   3. From the list of *Discovered projects*, place a check mark in the box next to *hyplnk\_exampleProject*
   4. Place a check mark on *Copy projects into workspace*.
   5. Click the *Finish* button.

The *hyplnk\_exampleProject* should now appear in the CCS *Project Explorer* on the left-hand side of your screen.

1. Expand the *hyplnk\_exampleProject*folder and double click on *hyplnkLLDCfg.h* to view the file and answer the following:

**QUESTIONS:**

How many lanes are configured?

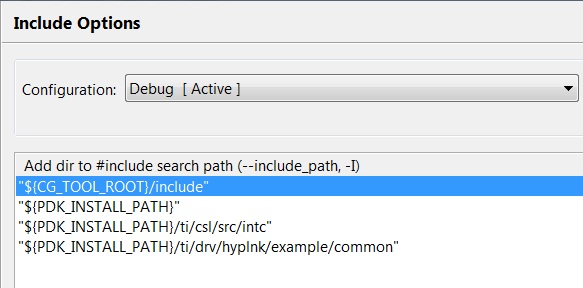
What is the baud rate? (HINT: 01p250 means 1.25GBaud) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Task 2: Verify and Set Project Properties

1. In *Project Explorer*, right click on *hyplnk\_exampleProject* and select *Properties*.
2. Select *General* properties.
3. Choose the *Main* tab and set/verify the *Device* properties as follows:
   * ‘Family = C6000’
   * ‘Variant = Generic C66x Device’
4. Select *Build* properties.
5. Choose *C6000 Compiler* 🡪 *Processor Options* and set/verify the following properties:
   * ‘Configuration = Debug’
   * ‘Target processor version = 6600’
   * ‘Application binary interface = eabi’

Note, in different versions of CCS the Application binary interface tab is in the *main* tab

1. Select *Build* properties.
2. Choose *C6000 Compiler 🡪 Optimization* and set/verify the following properties:
   * ‘Optimization level = 0’
   * ‘Optimize for code size = 0’
3. Select *Build* properties
4. Choose *C6000 Compiler 🡪 Debug Options* and set/verify the following properties:
   * ‘Debugging model = Full symbolic debug’
5. Select *Build* properties, choose *C6000 Compiler 🡪 Include Options* and ensure that include paths are setup as shown in the snapshot below:



1. Click the *OK* button to save the project properties and close the *Properties* window

### Task 3: Loopback Mode

1. Look for the following line in *hyplnkLLDCfg.h* and verify that it is uncommented.

**#define** hyplnk\_EXAMPLE\_LOOPBACK

When uncommented, this #define ensures that the example runs in loopback mode on a  
single EVM.

### Task 4: Build the Project

1. In *Project Explorer*, select the *hyplnk\_exampleProject* project.
2. Build the project:
   1. Select the CCS menu option *Project* 🡪 *Build Project*

OR

* 1. Right-click on the project in *Project Explorer* and select *Build Project*

1. CCS will now attempt to compile and link the project. This may take a few minutes to complete.
2. Please direct your attention to the CCS *Console*. On a successful build, you will see no errors generated in the *Problems* window (NOTE: There may be warnings) and the following message should display in the *Console* window:

<Linking>

'Finished building target: hyplnk\_exampleProject.out'

\*\*\*\* Build Finished \*\*\*\*

**QUESTIONS:**

Was the file hyplnk\_exampleproject.out generated? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. From the *CCS Edit* perspective, check the *Binaries* or *Debug* directory. From the *CCS Debug* perspective, check the *Console*.

### Task 5: Connect to the EVM

1. Click the *Open Perspective* (available right top corner of the *CCS*).
2. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
3. Select the CCS menu option *View* 🡪 *Target Configurations*. Select the target configuration you created
4. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
5. This will bring up the *Debug* window.
   1. Select Core 0 (*C66x\_0*)
   2. Right click and select *Connect Target*.

### Task 6: Load and Run the Program

1. Select Core 0 and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. A pop-up will appear with the projects names.
   4. Select *hyplnk\_exampleProject*
   5. Click Debug, then select *hyplnk\_exampleProject.out* and click *OK*.
   6. Click *OK* to load the application to the target (Core 0)
2. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.
3. The program attempts to send and receive tokens via the Hyperlink interface in loopback mode. So the same device acts as both the send and receive side.
4. A successful run should produce the following console output for each iteration:

. . .  
[C66xx\_0] Checking statistics

[C66xx\_0] About to pass 65536 tokens; iteration = 1

[C66xx\_0] === this is not an optimized example ===

[C66xx\_0] Link Speed is 4 \* 6.25 Gbps

[C66xx\_0] Passed 65536 tokens round trip (read+write through hyplnk) in 9278 Mcycles

[C66xx\_0] Approximately 141574 cycles per round-trip

[C66xx\_0] === this is not an optimized example ===

. . . .

1. Multiple iterations are performed and the program will go on indefinitely until manually stopped. Once you have verified that the program has executed successfully, select the CCS menu option *Run* 🡪 *Suspend*.

### Task 7: Board-to-board Hyperlink Example

Pair up with your neighbor to perform this portion of the lab. You will need a HyperLink cable or a connector board.

1. Modify the example code for *hyplnk\_exampleProject* so it can be run on two EVMs:
   1. Open *hyplnkLLDCfg.h*
   2. Search for **#define** hyplnk\_EXAMPLE\_LOOPBACK
   3. Comment out this line
   4. Ensure that the baud rate is set to 6.25 Gbaud, i.e. the line #define hyplnk\_EXAMPLE\_SERRATE\_06p250 is uncommented.
2. Build the code, load to both targets, and run the generated out file on Core 0 only.
3. Modify the example code for hyplnk\_exampleProject
   1. Open *hyplnkLLDCfg.h*
   2. Change the *Baud Rate* to a higher rate.
4. Build the code, load to both targets, and run on Core 0 only.

Note – The two system must have the same rate!

**QUESTION:**

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

# Lab 3: SRIO Type 11

## I. Purpose

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

## II. Project Files

The following files are used in this lab:

* bioInclude.h
* bioMain.c
* bioUtilityAndGlobals.c
* device\_srio\_loopback.c
* ExampleSRIO.cmd
* fftRoutines.c
* gen\_twiddle\_fft16x16.c
* initialization.c
* masterTask.c
* multicoreLoopback\_osal.c
* requestProcessingData.c
* slaveTask.c
* SRIOMulticore\_fft\_1.cfg

### Task 1: Import the Project

1. Open CCS.
2. Once CCS starts, verify that the perspective is set to *CCS Edit*.
3. Import the project.
   1. Select the CCS menu option *Project* 🡪 *Import Existing CCS Eclipse Project*
   2. Set *Select search directory* to: “C:\ti\labs\code\srio\_type11”
   3. From the list of *Discovered projects*, place a check mark in the box next to *SRIOSingleSRIO*
   4. Place a check mark on *Copy projects into workspace.*
   5. Click the *Finish* button.

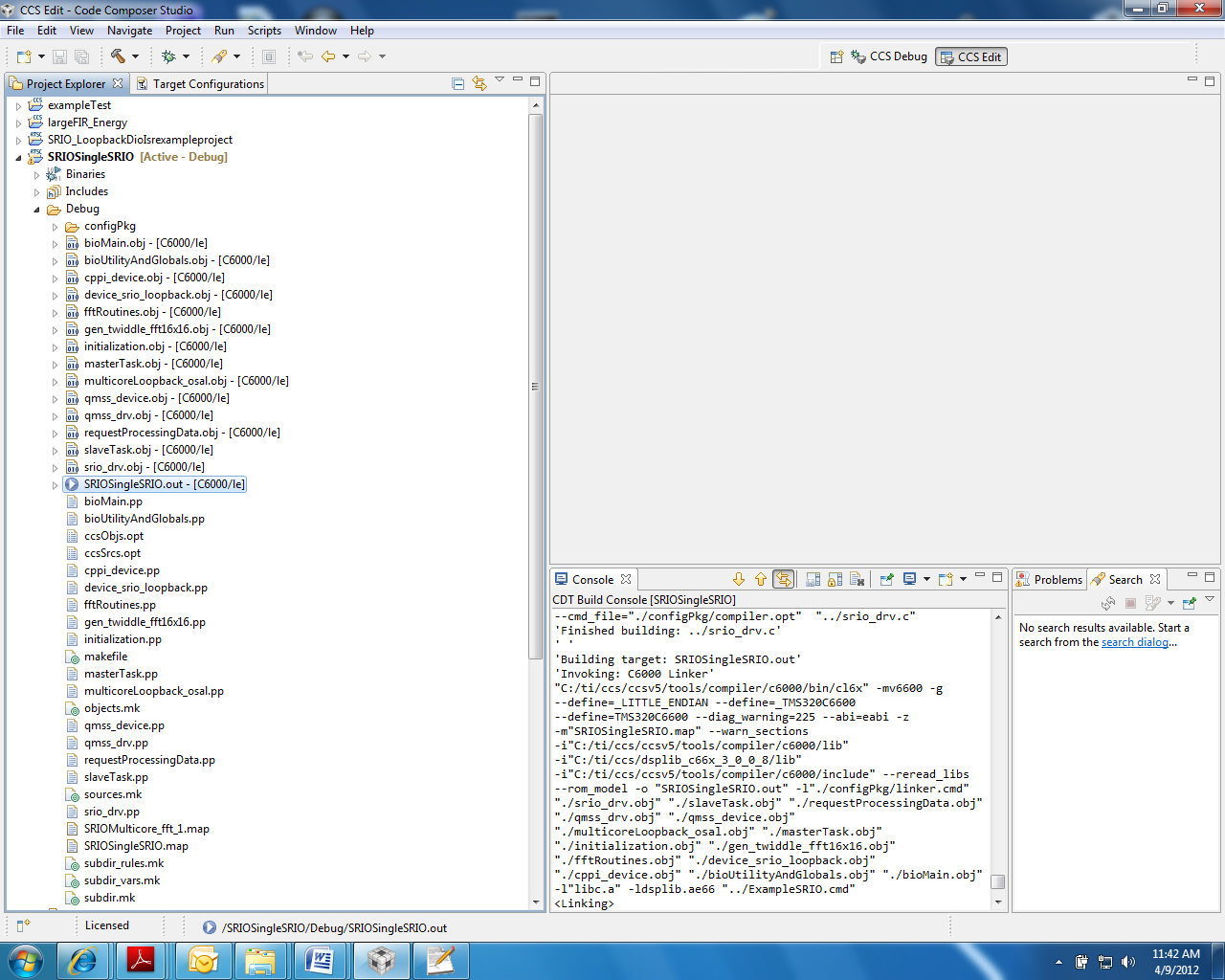
### Task 2: Build the Project

1. In *Project Explorer*, select the *SRIOSingleSRIO* project.
2. Clean the project by right-clicking on the project and selecting *Clean Build.*
3. Build the project:
   1. Select the CCS menu option *Project* 🡪 *Build Project*

OR

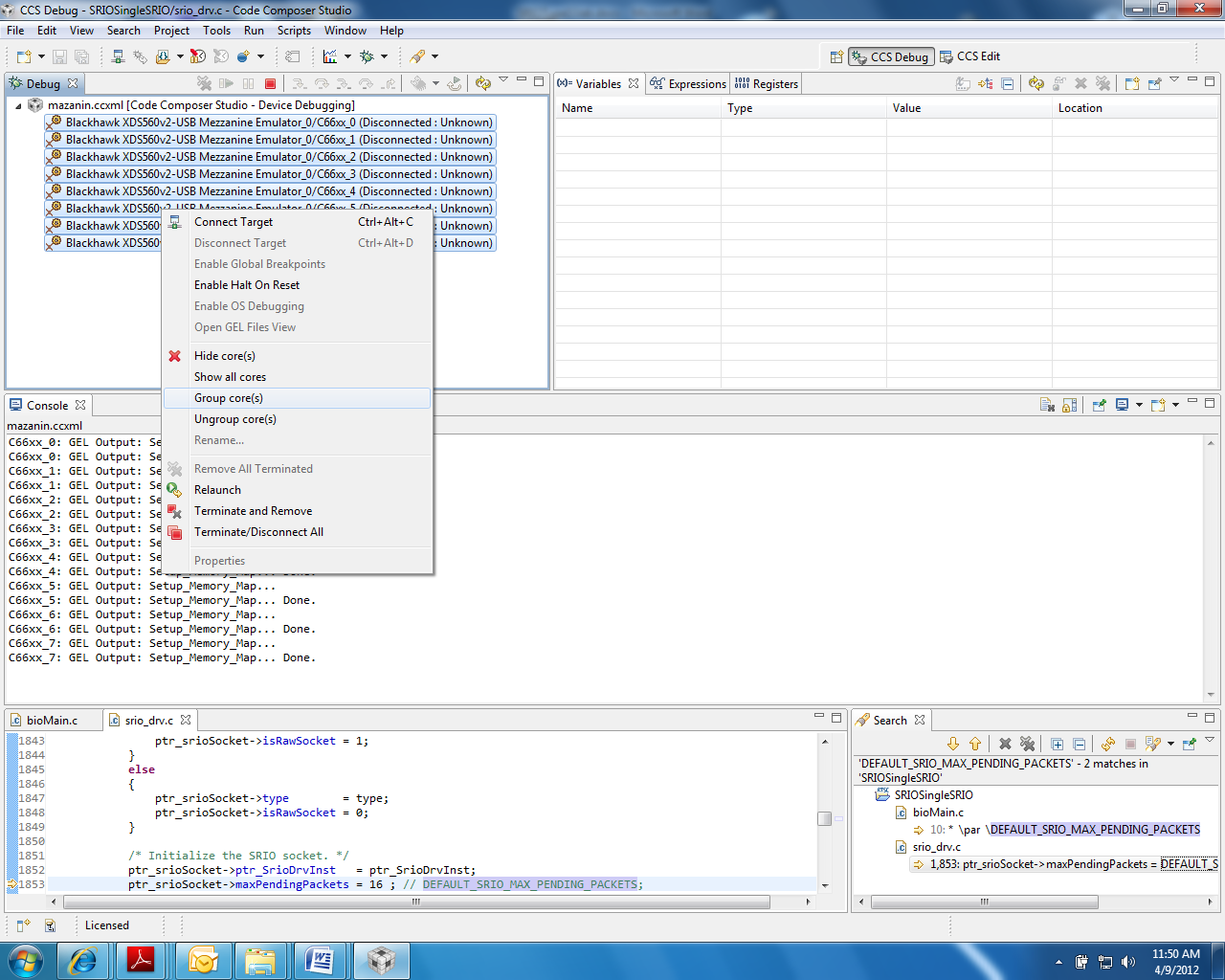
* 1. Right-click on the project in *Project Explorer* and select *Build Project*

1. CCS will now attempt to compile and link the project. This may take a few minutes to complete.
2. Verify that the executable (.out) was built by looking at the debug directory (assuming the build configuration is debug configuration).



### Task 3: Connect to the EVM

1. Recycle power on the EVM
2. Click the *Open Perspective* (available right top corner of the *CCS*).
3. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
4. Use the target configuration that you created
5. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
6. This will bring up the *Debug* window.
7. Select all cores, right click, and group them by selecting *Group Core(s)*: the default name will be ***Group 1***



### Task 4: Load and Run

1. Select Group1 and connect all cores in the group:
   1. From the CCS *Run* menu, select *Connect Target*.

OR

* 1. Right click on the group name and choose *Connect Target*.

OR

* 1. Click the Connect Target icon.

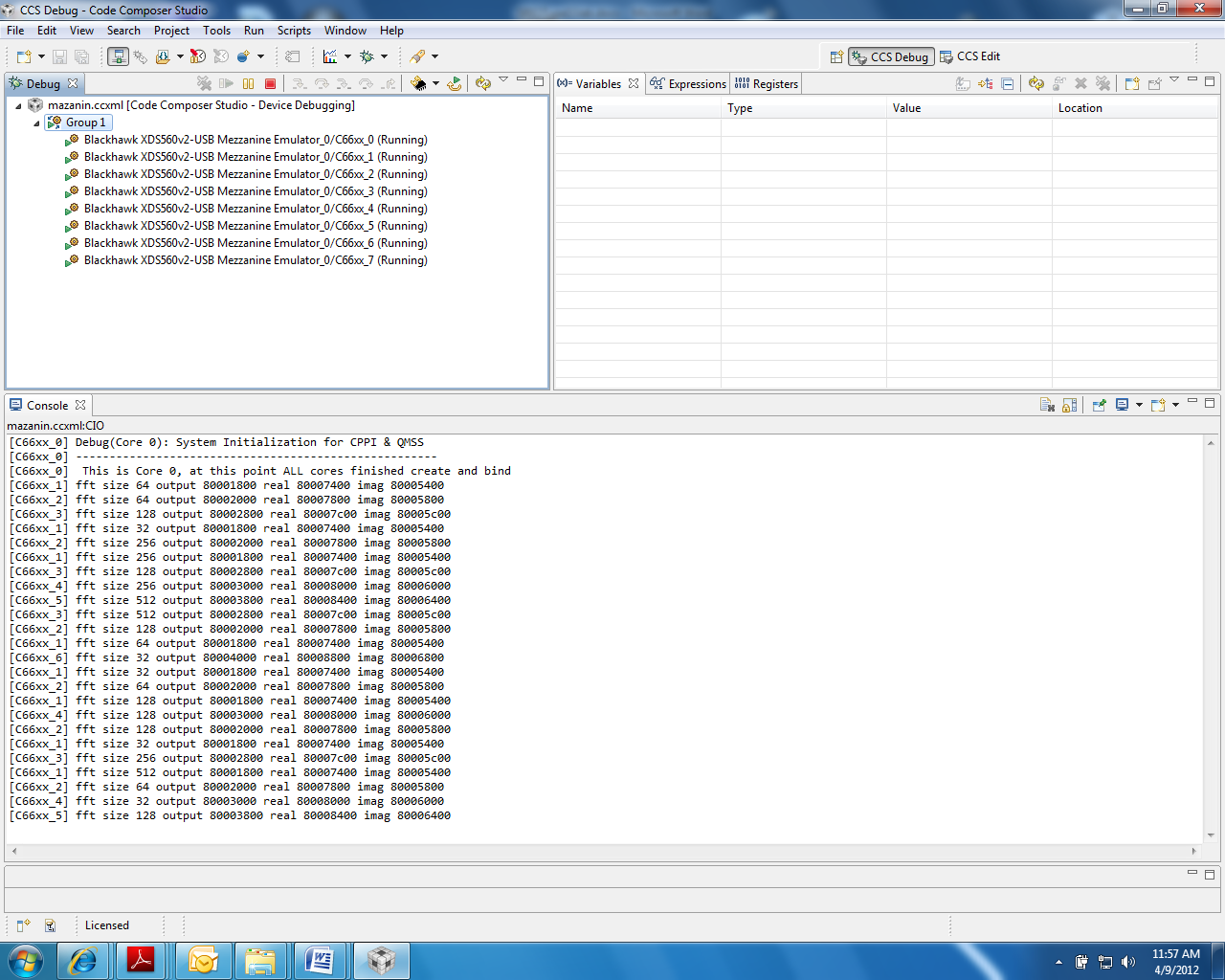
1. Load the *SRIOSingleSRIO.out* to all cores in the group:
   1. From the *Run* menu, select *Load*.

OR

* 1. Click the *Load* icon

1. Run the code in one of the following ways:
   1. Press F8
   2. From the *Run* menu, select *Resume*
   3. Click on the *Resume* icon (green arrow).

The output results appear as follows:



1. Observe the results, then suspend the run:
   1. From the *Run* menu, choose *Suspend*.

OR

* 1. Click on the *Suspend* icon (the yellow “pause” lines)

# Lab 4: Optimization

## I. 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.

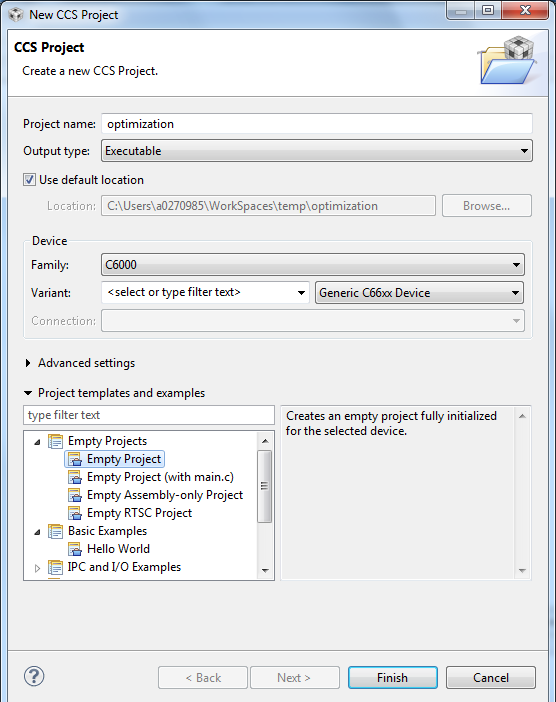
## II. Project Files

The following files are used in this lab:

1. firMain.c
2. intrinsicCFilters.c
3. linker.cmd
4. naturalCFilters.c
5. test.h
6. utilities.c

### Task 1: Build and Run the Project

1. Open CCS.
2. Create new project through the CCS menu item *File* 🡪 *New* 🡪 *CCS Project*.
3. Enter *Optimization* as a *Project Name*.
4. Click the check box to *Use default location.*
5. Set the *Family to C6000* and *Variant* to *Generic C66xxx Device* as shown below:



Note – the screen shots reflect different location for the project.

1. Then press *Finish* to create the new project.
2. Then in the *Project Explorer* view, right-click on the newly-created *optimization* project, and click on *Add Files…*
3. Browse to ‘C:\ti\labs\code\optimization,’ select all the files in this directory, and click *Open*. When prompted how files should be imported into the project, leave it as default of *Copy File.* Remove the main.c file that is created by default when you created the new project.
4. 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. Functions *naturalCFilters* and *intrinsicCfilters* execute filters on the generated data. The former is implemented completely in C, while the latter takes advantage of compiler intrinsics.
5. We will first set the properties for the *Debug* configuration. Right-click on the project. Select *Properties.*
   1. Choose *Build*, click on the *Environment* tab, and click the *Add…*button to add the path to add a variable with *Name* as ‘PDK\_ROOT’ and *Value* as ‘C:\ti\mcsdk\pdk\_C6678\_1\_1\_2\_5’
   2. Choose *C6000 Compiler 🡪 Optimization* andset/verify the following properties:
      * ‘Optimization level = 0’
      * ‘Optimize for code size = 0’
   3. Choose *C6000 Compiler 🡪 Debug Options* and set/verify the following properties:
      * ‘Debugging model = Full symbolic debug’
   4. Choose *C6000 Compiler 🡪 Include Options*. Under the “Add dir to #include search path” add the following two paths:
      * "${PDK\_ROOT}/packages/ti/csl"
      * "${PDK\_ROOT}/packages”

NOTE: This ensures that any include references in the project’s source files to header files located at these paths will be interpreted accurately.

1. Click the *OK* button to save the project properties and close the *Properties* window.
2. Right-click on the project and select *Build Project*. A successful build will generate the following output on the console:

. . .

<Linking>

'Finished building target: optimization.out'

\*\*\*\* Build Finished \*\*\*\*

### Task 2: Connect to the EVM

1. Click the *Open Perspective* (available right top corner of the *CCS*).
2. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
3. Select the CCS menu option *View* 🡪 *Target Configurations*. Select the target configuration you created
4. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
5. This will bring up the *Debug* window.
   1. Select Core 0 (*C66x\_0*)
   2. Right click and select *Connect Target*.

### Task 3: Load and Run the Program

1. Enable the Clock by selecting the CCS menu option *Run* 🡪 *Clock* 🡪 *Enable*
2. Select Core 0 and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. Select *optimization.out* by unwrapping the *Optimization*🡪*Debug* and click *OK.*
   4. Click *OK* to load the application to the target (Core 0).
3. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.
4. A successful run should produce a console output as shown below. Record the cycles time for both natural C and intrinsic C versions:

[C66xx\_0] natural C code size 32768 time 3889442

[C66xx\_0] intrinsic C code size 32768 time 2809073

[C66xx\_0] no error was found !!!

[C66xx\_0]

[C66xx\_0]

[C66xx\_0] DONE

Note – If the time shows zero, you have not enabled the clock (see above)

### Task 4: Compiler Optimization

1. Move back to the *CCS Edit* perspective.
2. You will now set the properties for the *Release* configuration. This suppresses all debug features and enables the highest time optimization.
   1. Right-click on the *Optimization* project. Select *Build Configurations 🡪 Set Active 🡪 Release*
3. Right-click on the *Optimization* Project. Select Properties.
   1. Choose *Build*, click on the *Environment* tab, and click the *Add…*button to add the path to add a variable with *Name* as ‘PDK\_ROOT’ and *Value* as ‘C:\ti\mcsdk\pdk\_C6678\_1\_1\_2\_5’
   2. Select *C6000 Compiler 🡪 Optimization* andset/verify the following properties:
      * ‘Optimization level = 3’
   3. Choose *C6000 Compiler 🡪 Debug Options* and ensure that:
      * ‘Debugging model = Suppress all symbolic debug generation’
   4. Choose *C6000 Compiler 🡪 Include Options*. Under the “Add dir to #include search path” add the following two paths:
      * "${PDK\_ROOT}/packages/ti/csl"
      * "${PDK\_ROOT}/packages”

NOTE: This ensures that any include references in the project’s source files to header files located at these paths will be interpreted accurately.

1. Click the *OK* button to save the project properties and close the *Properties* window.
2. Right-click on the project and select *Build Project*. A successful build will generate the following output on the console:

. . .

<Linking>

'Finished building target: optimization.out'

\*\*\*\* Build Finished \*\*\*\*

1. Enable the Clock by selecting the CCS menu option *Run* 🡪 *Clock* 🡪 *Enable* (if you have done in the previous section of this lab, you can ignore this step).
2. Select Core 0 and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. Select *optimization.out* by unwrapping the *Optimization*🡪*Release* and click *OK.*
   4. Click *OK* to load the application to the target (Core 0).
3. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.
4. A successful run should produce a console output as shown below. Record the optimized cycle times for both natural C and intrinsic C versions:

[C66xx\_0] natural C code size 32768 time 228698

[C66xx\_0] intrinsic C code size 32768 time 1282213

[C66xx\_0] no error was found !!!

[C66xx\_0]

[C66xx\_0]

[C66xx\_0] DONE

**QUESTIONS:**

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?

1. Do intrinsic functions better utilize the processor?

### Task 5: Enable Software Pipelining

1. In the CCS *Project Explorer* go to *Build 🡪 C6000 Compiler 🡪 Advanced Options 🡪 Assembler Options* and check the box that says *Keep the generated assembly language (.asm) file*
2. Rebuild the code.
3. The generated assembly file will be located within the *Release* directory since you are building the project’s release configuration. Open the ‘intrinsicCFilter.asm’ file and answer the following questions:

**QUESTIONS:**

Was the compiler able to schedule the software pipeline?

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

1. Think about cases that can cause randomness in the execution timing.

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

1. Think about the inline function.
2. Replace the regular function with the intrinsic function in all the loops. (Look at the definition of the regular function and see what intrinsic it uses)
3. Rebuild the code, load, and run.
4. Look at the *intrinsicCFilter.asm*.  
     
   **QUESTIONS:**

Did the compiler schedule the software pipeline? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

### Task 6: Align the Data

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

**QUESTIONS:**

What is the alignment of the input data? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the alignment of the filter coefficients (in the stack)? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Find the pragma that aligns the data. What other ways are there to align the data on a 64-bit boundary?
2. Change the code to tell the compiler that the data is loaded from aligned memory. (the \_amemX intrinsic tells the compiler that the data is aligned on 64 bit)
3. Rebuild the code, load, and run.
4. Record the optimized project cycle time for natural C function and for intrinsic function with software pipeline and aligned load.

### Task 7: Enable the MUST\_ITERATE Pragma

1. Uncomment the code to enable the pragmas that tell the compiler the minimum number of iterations and the divisor.
2. Rebuild the code, load, and run
3. Record the optimized project cycle time for natural C function and for intrinsic function with software pipeline, aligned load, and MUST\_ITERATE pragma.

### Task 8: Cache Considerations

1. In *test.h*, change the number of elements to 2K, 4K, 8K and 16K
2. Record the cycle counts for each case.

|  |  |  |  |
| --- | --- | --- | --- |
| **Size** | **Multiply for 32K** | **Clock Value** | **Cycles for 32K** |
| 32K | **1** |  |  |
| 16K | **2** |  |  |
| 8K | **4** |  |  |
| 4K | **8** |  |  |
| 2K | **16** |  |  |

**QUESTION:**

Why the non-linear jump in performance? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Think about cache trashing.

### Task 9: Cache Analysis

**QUESTION:**  Think about the data size and the fact that float complex requires 8 bytes (single precision). What is the actual size of the data?

**HINT:** To understand better the cache issue, you should start the next Lab – cache debug

### Task 10: Change the Code to Speed Up to 32K

1. Change the code to take full advantage of the cache.
2. Break the data into chunks and call each routine multiple times. Make sure to keep the sum between calls as well as the pointer to the data.

NOTE: This may take a long time and is left as homework to the student

# Lab 5: Using Advanced Debug to Understand Strange Results

## I. Understanding the Problem

In the optimization lab, the number of cycles is non-linear with the number of elements. When the number of elements is 8K, the code is much faster than if the number of elements is 16K; Per element, the numbers are normalized.

Recall that the data resides in L2 memory and that L1 D is configured as all cache. The data is read from L2 memory and is put in the L1 cache for reusability.

The obvious explanation is that when the number of elements is 16K, the cache is trashed during the first filter. So the second filter has cache miss. The same is true for the third filter and the fourth. But here is the strange thing: The elements in the optimization lab are floating-point complex, single-precision numbers. So each element requires 8 bytes (floating-point, single-precision is 32-bits, equal to 4 bytes, and complex doubles the number of bytes).

We will use some of the debug features in CCS to understand the cache behavior and determine why the higher results start at 8K and not at 4K.

### Why the Debug Version is Used

For the purpose of reading data from the L1 cache, the optimization is not important. This would not be the case if the optimized code in the release writes out intermediate results (and then reads it later) and the debug version does not. But this is not the case. Both versions read and write the same information. Thus the debug version is chosen with non-optimization and full symbolic debug turned on.

## II. Lab Instructions

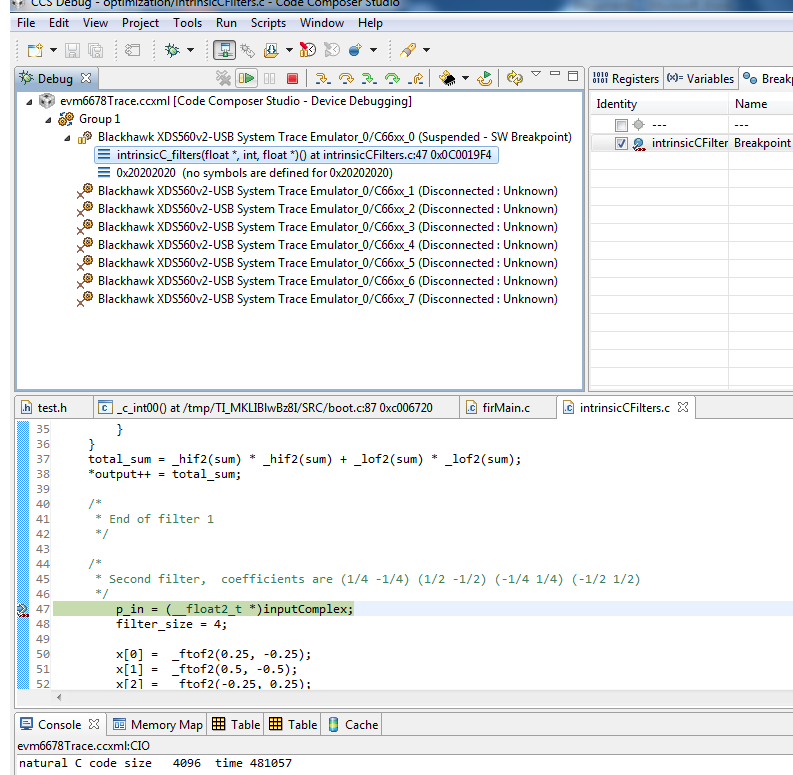
First, look at the cache behavior when the number of elements is 4K. Next, do the same with 16K elements. Lastly, will look at the 8K elements and draw a conclusion.

### Task 1: View the 4K Case

1. Change the number of elements in the test.h file as follows:

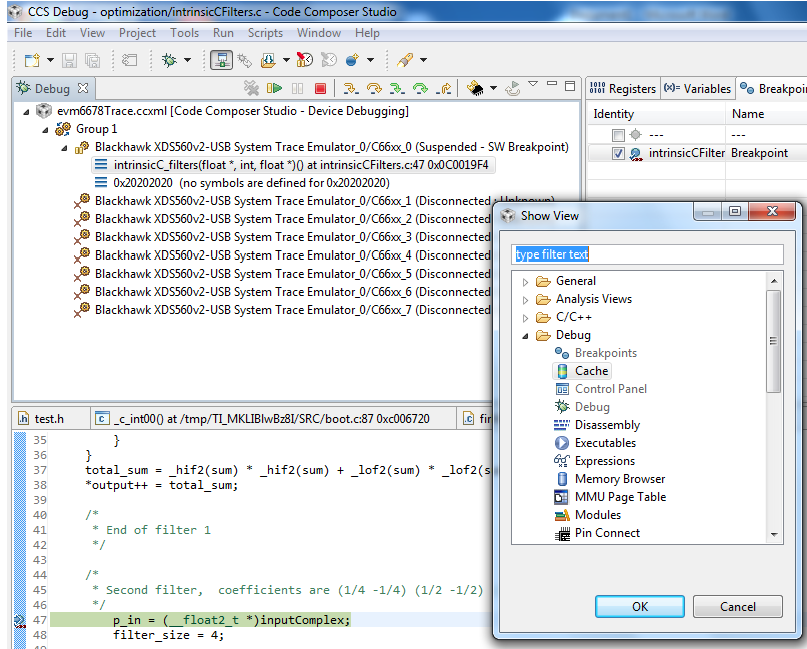
#define NUMBER\_OF\_ELEMENTS 4096

1. Rebuild the code.
2. Launch the debugger, connect Core 0 to the emulator and load the code from the debug configuration (the one with no-optimization and full symbolic debug).
3. From the *Run* menu in the *Debug* perspective, enable the *Clock*.
4. Open the file intrinsicCfilters.c and put a breakpoint after the first filter.
5. Run the code and verify that it stopped at this breakpoint. The screen shot looks as follows:

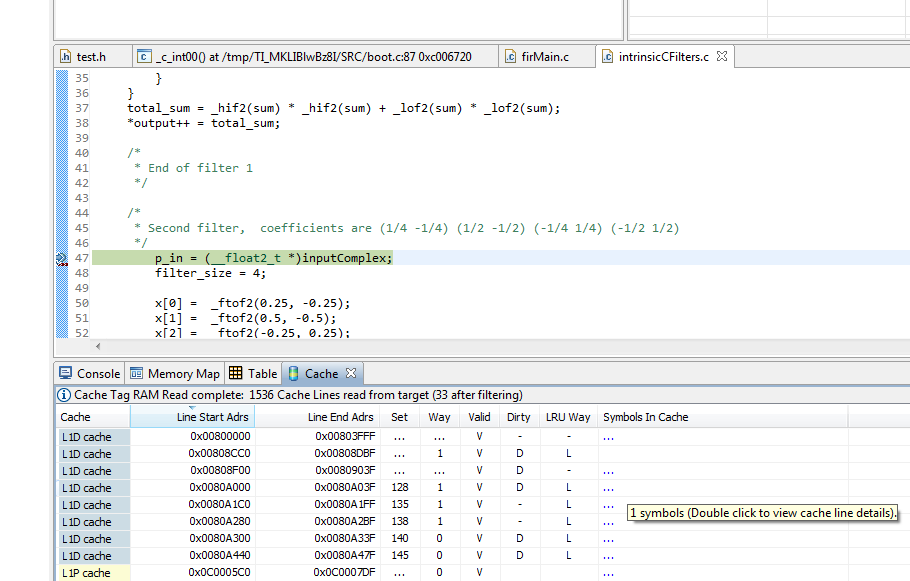


### Task 2: Looking at the Cache Lines for 4K Case

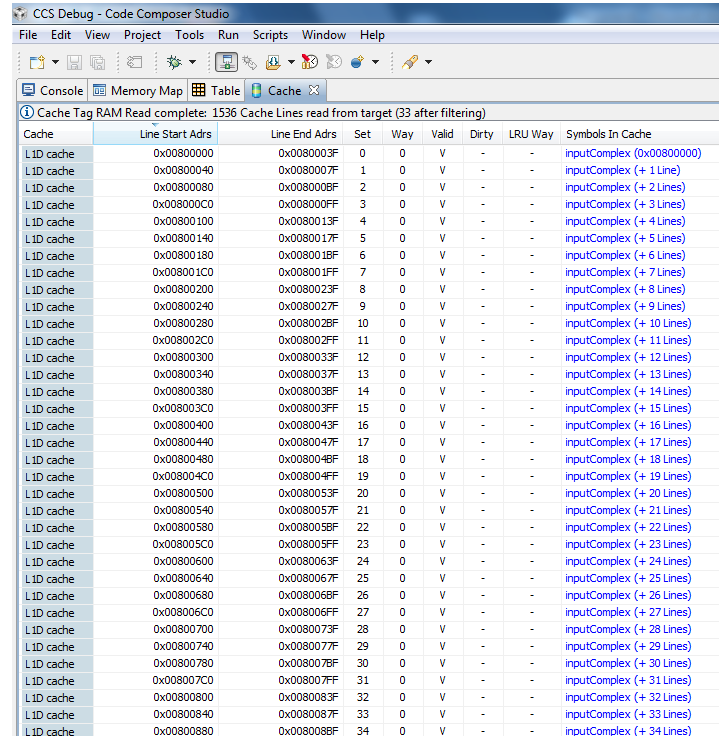
1. Left-click on the *View* tab in the *Debug* perspective. In the pull-down menu, choose *Other*. A new window will open, as shown below.



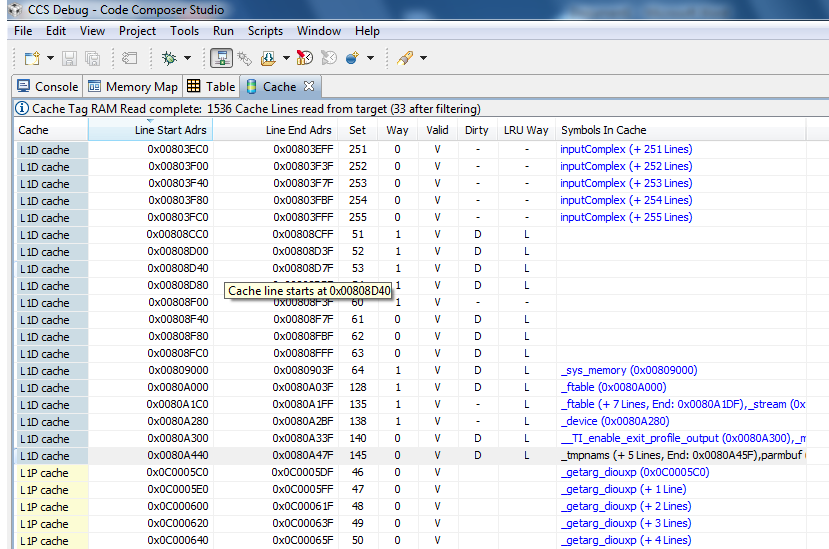
1. Select *Cache* and double click to open the *Cache* window, as shown below.



1. Double-click on the *Cache* tab to enlarge the window. Then double-click on any line of the L1D to show all L1D lines. It will look like the following:



1. Now examine the first L1D line. The address of the first L1D line is 0x0080 0000, which is the first line of the L2 memory and where the input vector resides. Note that the valid flag is set for this cache line.
2. Now look at the last L1D cache line:
   * The last line of the vector *inputComples* is line 255 with a starting address of 0x0080 3fC0.
   * After this, there are several DIRTY lines, which indicate where the code changed some values.
   * Remember that L1D cache line has 64 bytes (0x40). So the last byte of *inputComplex* in the cache is byte 0x00803fc0 +0x40 -1 = 0x00803fff. This is the Line End address of line 255.



**QUESTIONS:**

What is the last cache line that has the *inputComplex* vector?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How many bytes were read from the *inputComplex* vector?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

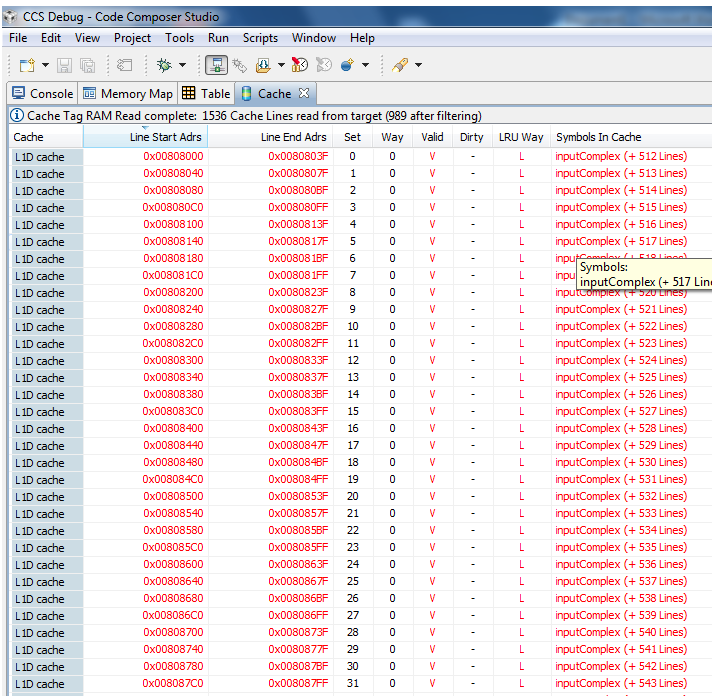
What is the number of elements that were read from the *inputComplex* vector?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Task 3: View the Cache Lines for 16K Case

1. Change the number of elements in the test.h file to 16K:

#define NUMBER\_OF\_ELEMENTS 16384

1. Repeat all the previous steps as defined in Task 1 and Task 2.
2. Build, load, and run the code to the break point. Looking at the cache lines, you will see:



**QUESTIONS:**

Why does the first entry address start with 0x00808000?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the last cache line that has the inputComplex vector?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How many bytes were read from the inputComplex vector? How do you know?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the number of elements that were read from the inputComplex vector?  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Task 4: View the Cache Lines for 8K Case

1. Change the number of elements in the test.h file to 8K.
2. Repeat all the previous steps as defined in Task 1 and Task 2.
3. Build, load, and run the code to the break point. Examine the cache.

**QUESTIONS:**

What is the first entry address? What does it mean?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the last cache line that has the *inputComplex* vector?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How many bytes were read from the *inputComplex* vector? How do you know?  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the number of elements that were read from the *inputComplex* vector?  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Task 5: Finding the Bug

By now you should know that the number of elements as it is defined in *test.h* is not the number of elements that are actually read from the input vector.

Can you suggest a bug fix in *firMain.c* that will fix the problem? Write your answer below.

# Lab 6: Inter Processor Communication (IPC)

## Shared Memory Transport

## I. Purpose

The goal of this lab is to become familiar with how to use the Inter Processor Communication (IPC) software module to communicate between applications running on different cores. You will build a project that leverages IPC *MessageQ* APIs to pass messages between arbitrary cores on the C66x EVM. Initially, shared memory is used to pass data between the cores.

## II. Project Details

The project generates 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 a hundred 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 acknowledgment 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 its *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.

## III. Lab Instructions

### Task 1: Import & Examine the Skeleton Project

1. In the CCS Edit perspective, click on the CCS menu option *Project* 🡪 *Import Existing CCS Eclipse Project*.
2. In the *Select search-directory* box browse to ‘C:\ti\labs\code\ipc\sharedMem’
3. Select the *IpcSharedMem* project from the list of *Discovered projects.*
4. Check the box that says *Copy projects into workspace.*
5. In the Project Explorer, expand the Src folder and open the file *IpcSharedMem.cfg* as follows:
   1. Right-click on the file in the CCS Project Explorer.
   2. Select *Open With* and *XDCScript Editor.*
6. Examine the following lines in the *IpcSharedMem.cfg* file that are necessary for leveraging IPC
   1. The *MultiProc* module handles the management of the various processor IDs. The *Ipc* module is used to initialize the subsystems of IPC. 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** MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');

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

procNameList = ["CORE0", "CORE1", "CORE2", "CORE3", "CORE4", "CORE5", "CORE6", "CORE7"];  
MultiProc.setConfig(procName, procNameList);

* 1. These lines define the shared memory location.

Program.global.shmBase = 0x0C000000;

Program.global.shmSize = 0x00200000;

* 1. 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.translate = false;

SharedRegion.setEntryMeta(0,

{ base: Program.global.shmBase,

len: Program.global.shmSize,

ownerProcId: 0,

isValid: true,

cacheEnable: cacheEnabled,

/\* Aligns allocated messages to a cache line \*/

cacheLineSize: cacheLineSize,

name: "internal\_shared\_mem",

});

1. In the Project Explorer, expand the *include* directory and open *Ipc.h*.  
     
   Notice the enumeration for the different message types and specifically the *myMsg* structure. The exact *myMsg* structure definition is configurable, except for the requirement that a *MessageQ\_MsgHeader* element should be the first item. In this case, the message consists only of a message header, a message type, and an integer value that counts the number of times that the token has been passed.
2. Open and examine *Ipc.c*. There are three functions here:
   1. *main( )*

This function dynamically creates the application task, calls *Ipc\_start*( ) to synchronize the processors, and then calls *BIOS\_start*.

* 1. *findNextCore( )*

This function 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 cores aren’t passing tokens to themselves*.* However, please note that it ***is*** possible for a core to send a message to itself in this manner. In fact we do that at the end with the “Done” message. But in our example, for the sake of showing a more interesting token path, we pass tokens between different cores.

* 1. *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 APIs

1. Some of the API calls required for this example have been left out of the function *task\_fxn*( ). The objective of this part of the exercise is to fill in the blanks and add the relevant code. We have marked each line in the code where your input is required; 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 provides shortcuts for these tasks. To reach the Tasks window, select from the menu *Window*🡪*Show View*🡪*Other*. 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
   1. The memory for a *MessageQ* message is allocated by calling the API *MessageQ\_alloc( )*.
   2. 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.
   1. *MessageQ* messages are passed by calling the *MessageQ\_put( )* API.
   2. The parameters to the function are the destination Queue Id, and the pointer to the message.
3. Get a Message from the local queue.
   1. Messages are retrieved from the *MessageQ* with the *MessageQ\_get( )* API. The *MessageQ\_get( )* function is a blocking call.
   2. 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.
   3. *MessageQ\_FOREVER* specifies that a timeout will never occur.
   4. The return value is zero if the message is successfully retrieved.
4. Get the Reply Queue for the token.
   1. 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
   1. See Hint #1
6. Send the acknowledge message.
   1. See Hint #2
7. Free the memory used by the token message.
   1. The memory is freed by the API *MessageQ\_free*.
   2. 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.
9. Set the Reply Queue for the Token Message
   1. This is done using the *MessageQ\_setReplyQueue( )* API.
   2. Parameters are the handle to the *MessageQ* and the pointer to the message.
10. The instructor might provide you with a solution file just in case you need it

### Task 3: Build the Project

1. Once everything has been added to *Ipc.c,* build the project. To do this, right-click on the project and select *Build Project*.
2. The project should build without errors or warnings. If it doesn’t build properly, attempt to figure out why. Otherwise, ask the instructor.

### Task 4: Connect to the EVM

1. Switch to the Debug CCS Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
2. Connect the power adapter to your EVM, and connect your laptop to the emulator port on the EVM using the provided USB cable. If you are using the XDS560v2, wait till the red and orange lights appear and are stable before proceeding to the next step.  
     
   NOTE: The Windows “Found New Hardware Wizard” may pop-up when you first connect the emulator via USB to the laptop. Select “Yes, this time only” 🡪 Next 🡪 “Install the software automatically” 🡪 Next, and allow the drivers to install on your system. Then click “Finish.” You might need to restart CCS at this point.
3. Select the CCS menu option *View* 🡪 *Target Configurations*. Your newly-created .ccxml target configuration file should be available under *User Defined* target configurations.
4. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
5. This will bring up the *Debug* window.
   1. Select Core 0 (*C66x\_0*)
   2. Right click and select *Connect Target*.

### Task 5: Load and Run the Program

1. Select all cores and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. Select *shmIpcBenchmark.out* and click *OK*
   4. Click *OK* to load the application to the target (Core 0)
2. Now run the application. To do this, select the CCS menu option *Run* 🡪 *Resume*
3. A successful run should produce a console output as shown below. You should see *printf* outputs in the console window similar to the output shown 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  
. . .

1. Ensure that the application behaves as expected by checking the following items:
   1. Since Core 0 passes the token first, the first *Ack Received* message should be from Core 0. This is the case with the example above.
   2. In general, you should see the pattern of how the token are passed. We know that Core 0 passes the first token. Based on the order of the *Token Received* messages in the console output shown above, the order is 0 🡪 1 🡪 5 🡪 1 🡪 2 🡪 1 🡪 5 🡪 3. The *Ack Received* messages should follow a similar, pattern, but it likely won’t look identical. (This can be attributed to the non-real-time 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 Received* 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.

# Lab 7: Using MPAX to Define Private Core Memory in DDR

## I. Purpose

A typical multicore scenario uses multiple cores run the same code, but uses different data structures and different configuration structures.

Downloading the same execution for multiple cores is very efficient. However, since each core has its own configuration structures and its own data structures, these values must reside in the private memory of each core; Namely, L2 SRAM or in L1D SRAM (if the complete L1 D is not configured as cache).

However, the size of L2 is limited to 512K bytes for C6678, and 1024K bytes for C6670. In addition, L2 is usually partitioned into L2 cache and L2 RAM, so that the available L2 RAM size is smaller. In many cases, the amount of private data does not fit into the available L2 RAM.

This lab presents a simple way to configure part of the external DDR as a private memory such that each core sees only its own data and structures.

The lab covers the following:

1. A short description of the MPAX registers part of the XMC (external memory controller).
2. Coherency discussion and the MAR registers.
3. Usage of EDMA to move data to and from the private memory.
4. Platform configuration and the memory map.
5. A set of utility functions to configure the MPAX registers .
6. A demo program that demonstrates the DDR private memory system.
7. A complete project is attached.

## II. Overview

### Short Description of MPAX (Memory Protection and Extension)

[The C66 CorePac User’s Guide](http://www.ti.com/lit/ug/sprugw0b/sprugw0b.pdf) (Chapter 7.3) describes the MPAX unit. The MPAX registers are used to translate a logical address (32-bit address) into a physical address (36-bit address). Here we use the translation registers to assign a different physical address for each core to the same logical address, so that the same code will load the same structures to different physical locations for each core.

The MPAX registers manipulations used to achieve the different translation will be done in the main () function. The initialization of pre-initialized global variables (for example, a declaration of global variable int x=6) is done before the start of main (). Thus, pre-initialized global variables are not allowed in the DDR private memory. NOTE: There may be way to configure the MPAX registers during boot, but this exercise does not consider it.

### Coherency Discussion

For cases where L1D is configure as a cache, there is hardware coherency between L2 data and L1 D inside the C66 CorePac. The hardware coherency guarantees that a variable that is defined in L2 and is cache by L1D has the same value in both places. This is not true with private memory in DDR. There is no hardware coherency between DDR and the internal memory of the core – L2 and L1D. Thus the user must maintain the coherency (using invalidate, writeback and writeback invalidate).

This example solves the cache coherency problem by disabling the cache. Disabling the cache is done using the MAR registers. The MAR registers are described in the  [C66 CorePac user’s Guide](http://www.ti.com/lit/ug/sprugw0b/sprugw0b.pdf) chapter 4.4.

### Usage of EDMA to Move Data to and from Private Memory

Data can be moved in and out of private memory using the EDMA. When a core reads or writes to the DDR, the data goes via the master port of the core into the MSMC through the core MPAX registers. When EDMA reads or writes data to the DDR, the data goes via the TeraNet port. The TeraNet port that connects to the DDR has 16 sets of MPEX registers correspondent to 16 privileges ID. Each set has 8 registers. (8 more MPEX registers are connected to the shared L2 memory inside the MSMC).

The privilege ID of EDMA transform is inherited from the master who initiates the transform. So if core 0 (with Privilege ID = 0) initiates EDMA transform, the privilege ID is 0. If core 7 initiated EDMA transform, the EDMA privilege is 7.

If the EDMA is used with the DDR private memory, then the MPAX registers of each privilege ID should be configured similarly to the core MPAX. In this example, EDMA is not used.

### Platform Configuration and the Memory Map

Projects that use RTSC must define the platform. The platform defines what memories are used in the execution. In addition to L1, L2 and the MSMC memory, the external memory DDR is defined.

Usually DDR is defined as 2G bytes memory (for the 6678 EVM). In the example, we will reduce the size of DDR to 1G – and use the other 1G for private memories.

The default settings of the MPAX registers for all cores are the following.

Register 0 value is 0000001E 000000BF -> correspondent to 2G mapping of internal memory into itself

Register 1 value is 8000001E 800000BF -> maps the 2G external memory 8000 0000 to ffff ffff into the 36 bit range 8 0000 0000 to 8 7fff ffff

Register 2 value is 2100000b 100000ff -> Maps (again) 4K starting at address 0x21000000 to address 1 0000 0000. This is the DDR EMIF configuration registers. Note that if the same memory range is defined in multiple MPAX registers, the higher MPAX register translation is used.

Registers 3 to 15 values are 00000000 00000080 which basically points to empty memory regions.

In this example, the first DDR are divided into two parts –

A shared 256M DDR starting at logical address 0x8000 0000 to address 0x8fff ffff – physical address 8 0000 0000 to address 8 0fff ffff

A private 16M DDR for each core. The logical address starts at 0x9000 0000 to 0x97ff ffff with physical address depends on the core number as follows:

1. Core 0 -> 8 1000 0000 to 8 10ff ffff
2. Core 1 -> 8 1100 0000 to 8 11ff ffff
3. Core 2 -> 8 1200 0000 to 8 12ff ffff
4. Core 3 -> 8 1300 0000 to 8 13ff ffff
5. Core 4 -> 8 1400 0000 to 8 14ff ffff
6. Core 5 -> 8 1500 0000 to 8 15ff ffff
7. Core 6 -> 8 1600 0000 to 8 16ff ffff
8. Core 7 -> 8 1700 0000 to 8 17ff ffff

To perform the translation MPAX register 4 will be used. The high 32 bits of the register has the base logical address (5 bytes) and one minus the log (base 2) of the size. Since the size of the private memory is 16M (2\*\*24), the size is 23 or 10111b = 0x17.

Thus the high 32-bit of MPAX register number 3 should be 0x90000017 (bit 5 to 11 are reserved).

The law 32-bit of MPAX 3 has the physical address (6 bytes) and 2 bytes of the permission value. In this example we give full permission values to the user and Supervisor, thus the permission value is

0011 1111b or 0x3f.

The physical address part of the law 32-bit depends on the core number. Based on the values from above, the law 32-bits are

1. Core 0 -> 0x810000 3f
2. Core 1-> 0x811000 3f
3. Core 2-> 0x812000 3f
4. Core 3-> 0x813000 3f
5. Core 4-> 0x814000 3f
6. Core 5-> 0x815000 3f
7. Core 6-> 0x816000 3f
8. Core 7-> 0x817000 3f

Note: bits 6 and 7 are not used and may read as non zero.

The example configures the MPAX registers in two ways, either with CSL functions or with direct register manipulations. The user can switch between the two ways (or even use both of them). A printf shows the values of the MPAX registers.

Next we consider the MAR registers. For this example, the logical memory starts on 16M boundary and the size is 16M, so one MAR register controls the cache-ability of the complete private memory. (MAR registers control the logical addresses, not the physical address). If the private memory was larger multiple MAR registers will be used, and if the logical memory boundary was not aligned on 16M bytes, a more complex scheme had to be developed.

For the example case, the MAR registers disable the cache and prefetch for all addresses starting in MAR 140 (0x9000 0000) to MAR 159 (0xa000 0000).

## III. Lab Instructions

### Task 1: Run the Example Code

1. Load MPAX\_registerDemo from c:/temp/lab directory and look at its properties. Make sure that all the relative paths exist in your system symbols
2. Step 2 – choose how you want to configure the MPAX register, using CSL or direct register manipulation or both. Change the #if in two places to 0 or 1
3. Rebuild the project
4. Connect EVM6678 and start the debugger. Connect all the cores and load the code to all the cores
5. Run all the cores. The printing will tell you when each core is done. A delay function staggers the cores.
6. When all cores print the done statement they are all in infinite loop. Pause all the cores.
7. Repeat the following for each of the cores:
   1. Select a single core
   2. Open a memory view window (View-> memory)
   3. Look at address 0x9000 0000 for all cores (This is external memory)
   4. Core zero will have values start at 0, 1, 2, etc.
   5. Core 1 will have values 0x0002 0000 and incrementing by 1
   6. Core 2 will have the first value 0x0004 0000 and incrementing by 1
   7. Core 3, 4, 5, 6, and 7 will have first values 0x0006 0000, 0x0008 0000, 0x000a 0000, 0x000c 0000 and 0x000e 0000 respectively
8. Conclusion – the same logical address (0x9000 0000 and on) have different physical address for each core

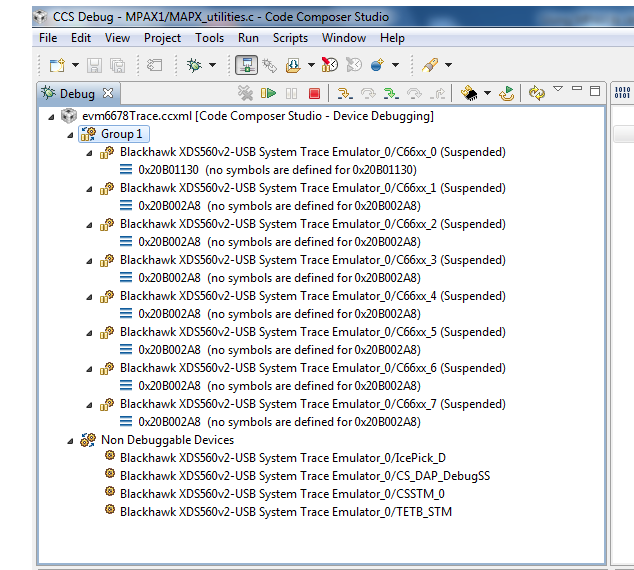
### Task 2: Using Trace to Verify the Write Physical Address

The next task describes how to use the trace facility to verify that each core actually writes to a different physical address. NOTE: This task requires a mezzanine card with trace emulator on the target platform.

#### Step 1: Connect to the Non-debuggable Devices (Esp. CCTMS\_0)

1. After launching the target, right-click on the upper line in the debug window (for example, evm6678Trace.ccsml ) and choose *show all cores*.
2. Go to non-debuggable devices (the bottom of the window), right-click, and connect to target.
3. Connect all the cores as well.

The debug window should appear as follows:



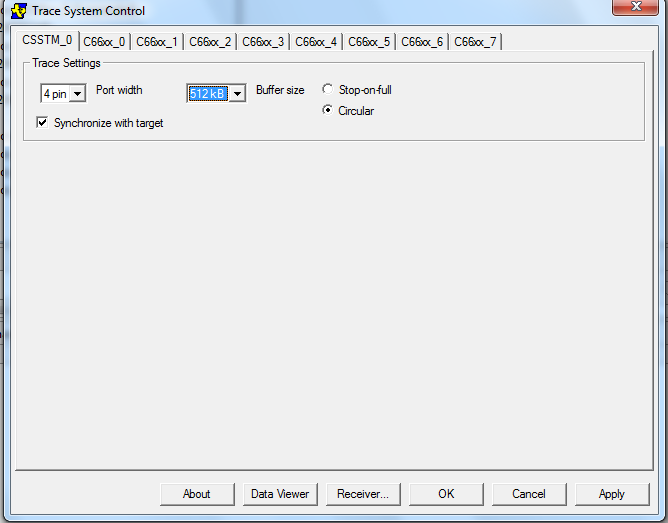
#### Step 2: Load the Code to the 8 Cores

1. To make the results easier, we change the number of elements (Line 171 in testMPAX1.c) to 10.
2. Build the project again and load it o the 8 cores.

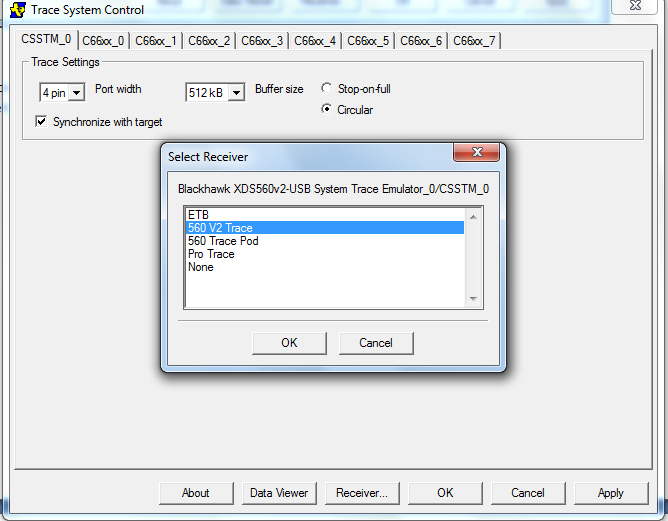
#### Step 3: Configure the CSSTM\_0 Trace Control

Note - These instructions are for CCS V5.3. For CCS V5.4 follow the instructions in Lab 8

1. From the *Tools* menu (in the debug prospective), choose *Trace control*.
2. The Trace System Control window (see below) opens. Click on the CSSTM\_0 tab.
3. Set *Port width* to 4 pin, place a check mark next to *Synchronize with target*, set the buffer size (512kB is sufficient for this example), and select *Circular* buffer.



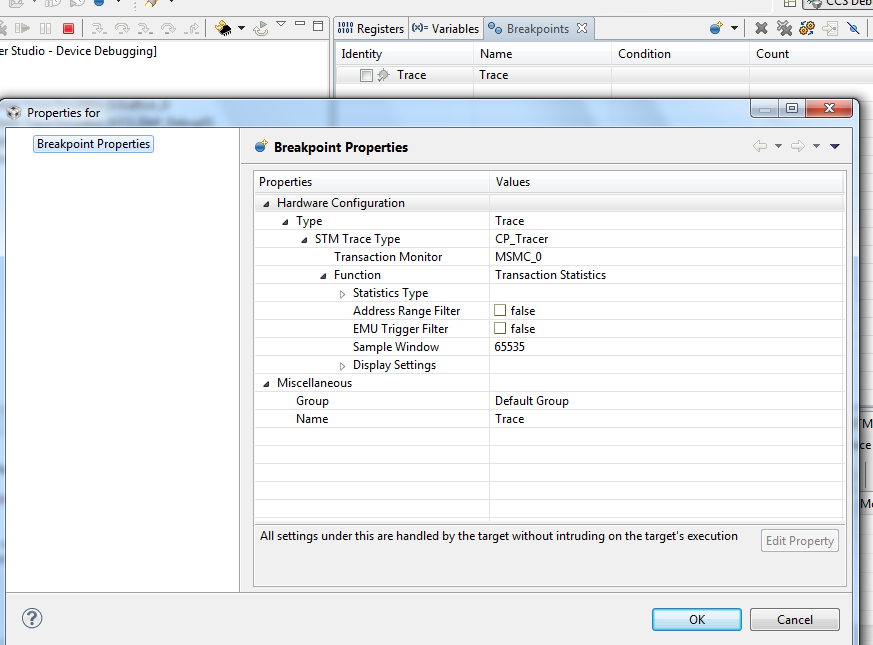
1. Click on the *Receiver…* button to choose where the trace data will go. Choose *560 V2 Trace*.  
   NOTE: You can use the EB as well, but the ETB is small (32K only) and then you have to read it from the ETB.
2. Click *OK* on the dialogue box and then click *Apply* and wait for the programming to be done.



#### Step 4: Add and Configure the Trace Location

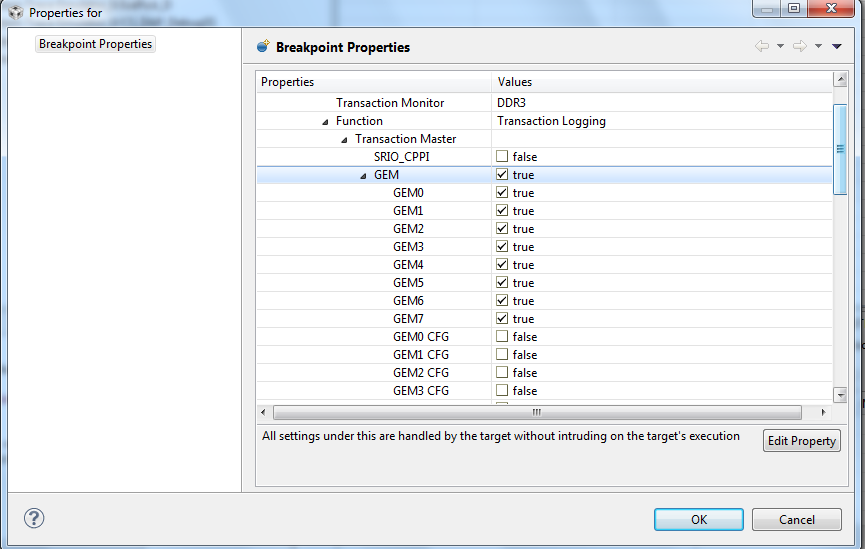
Trace points are currently (CCS5 version 5.3.x) defined from the *Breakpoint* window.

1. From the *View* tab, open a Breakpoint window.
2. Load the code to all the cores.
3. From the *Debug* menu, go back and select the CSSTM\_0 trace.
4. Go to the *Breakpoint* window, right-click, and select *Breakpoint*.
5. You can define either a breakpoint or a trace point. Select *Trace point*. A trace breakpoint will be added to the window.
6. To configure the trace, right-click on the trace and choose *Properties*. The following *Breakpoint Properties* window is opened:



1. To configure a Breakpoint Properties value, click on the value line to the right of the each value to access a pull-down menu of options. The trace for this example is configured as follows:
   1. STM Trace Type: CP\_Tracer
   2. Transaction Monitor: DDR3
   3. Function: Transaction Logging (This will open a new dialogue box to choose what to log)
   4. Transaction Master ->GEM (Select only the GEM tab, this will open the next level – which GEM to follow)
   5. All GEMs should remain active. So select all the GEM from 0 to 7

At this point, the window appears as shown.



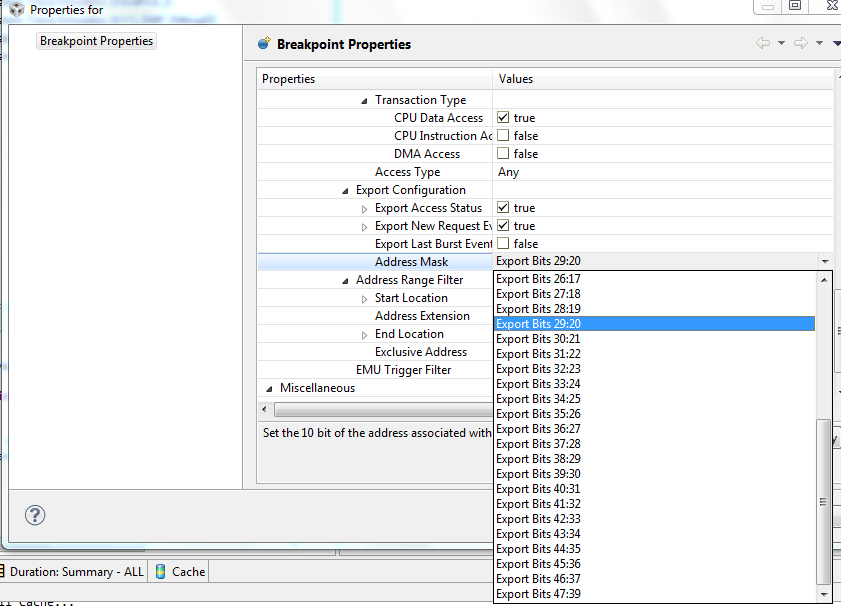
1. Scroll down in the window to *Event Filter*, *Transaction Type*, and set *Only CPU access* to *true*.
2. *Export Configuration* should be set to *true* for *Status* and *New Requests*.
3. Address mask

In this example, we want to see the physical addresses. The trace can show only 10 bits of address. So we want to choose the right bits.

Of course, we can run the trace multiple times, with each trace covering a different set of bits. We know that all of the writes are into DDR3 and that the physical addresses always start at  
8 1N00 0000 (where N is the core number). So the 10 bits that we are going to choose for this example are as follows:

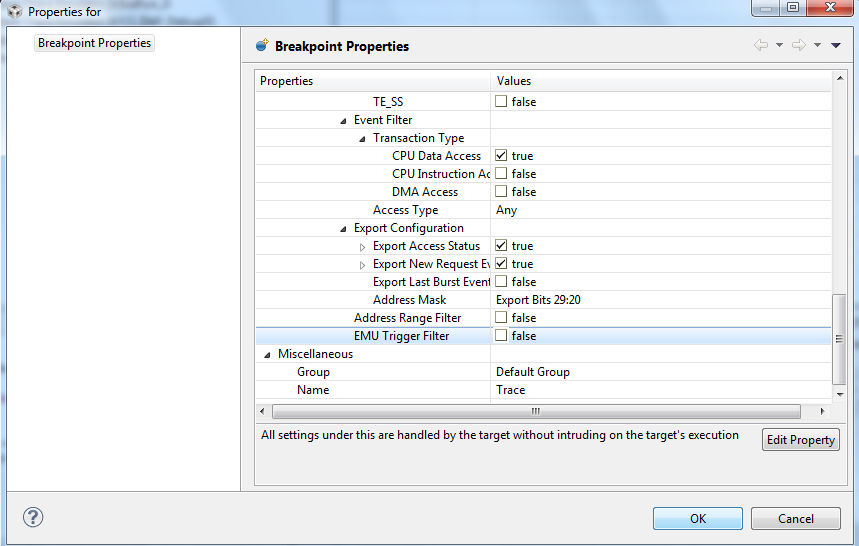
* Bit 29: Expected to be always 0.
* Bit 28: Expected to be always 1.
* Bits 27 to 24: Expected to be the core number.
* Bits 23 to 20: Expected to be all zeros.

1. Click on Export Bits. In the pull-down menu, choose 29:20:



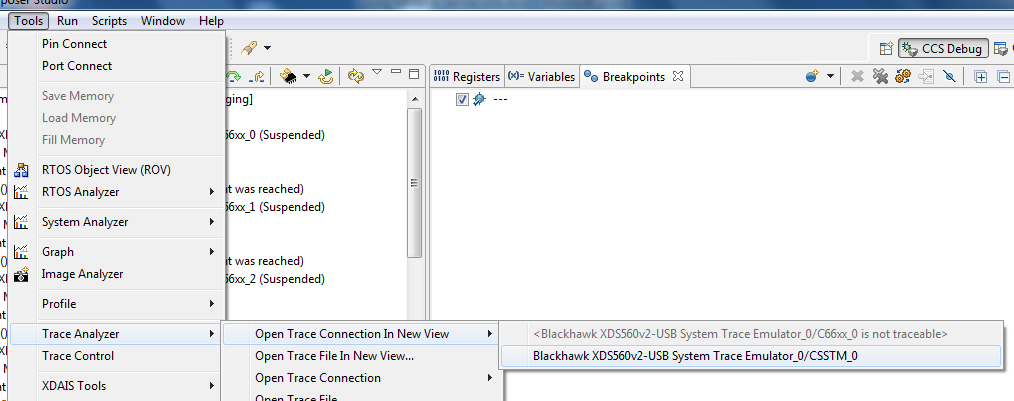
1. We will not use any other filtering. So both the Address Range Filter and EMU Trigger Filter should be marked false. Click OK.

The following Breakpoint Properties window appears.



#### Step 5: Start Display

1. From the *Tools* menu on the *Debug* perspective, choose Trace Analyzer, then *Open Trace Connection*, and select *Blackhawk XDS560v2-USB System Trace* (or your emulator) as follows:

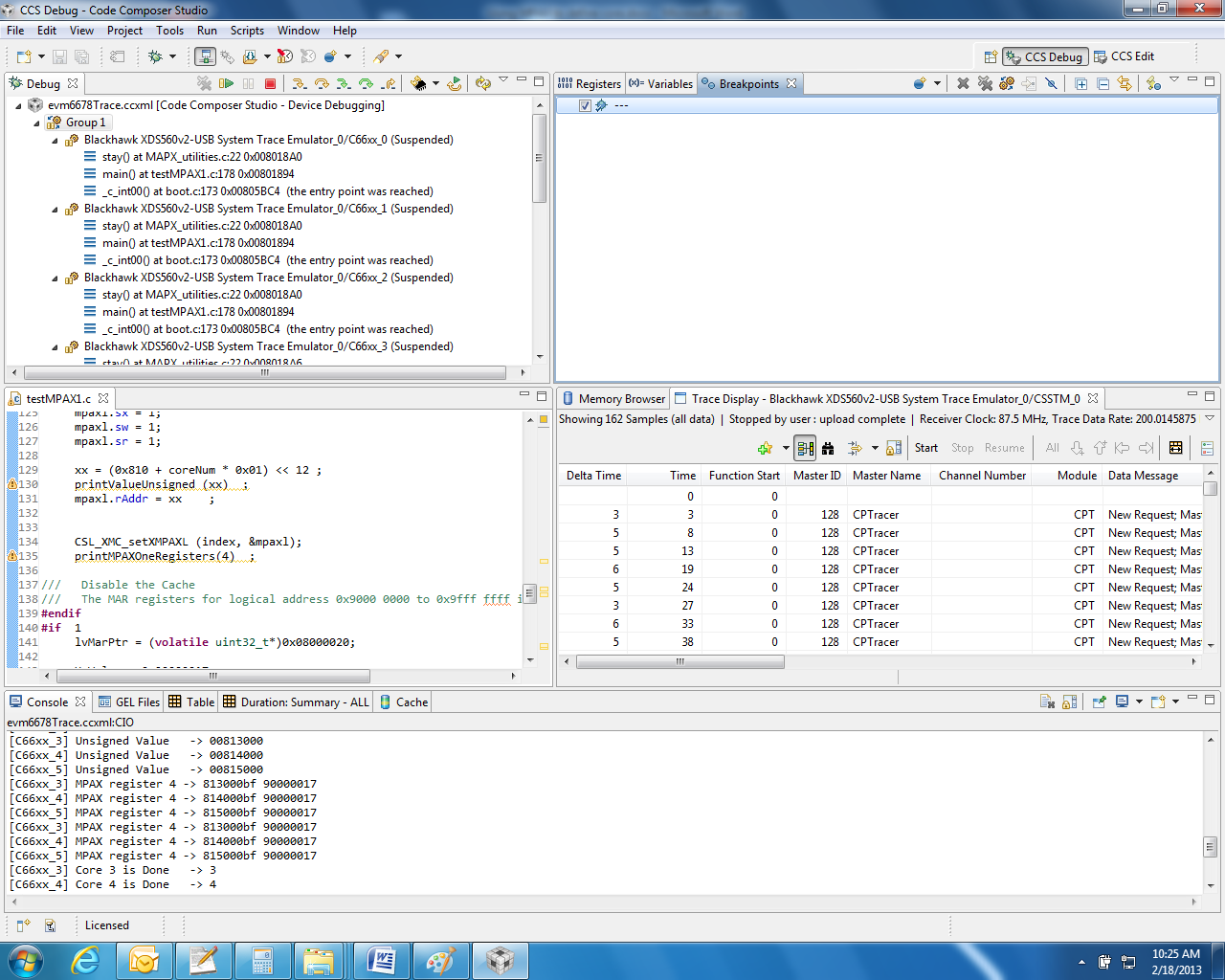


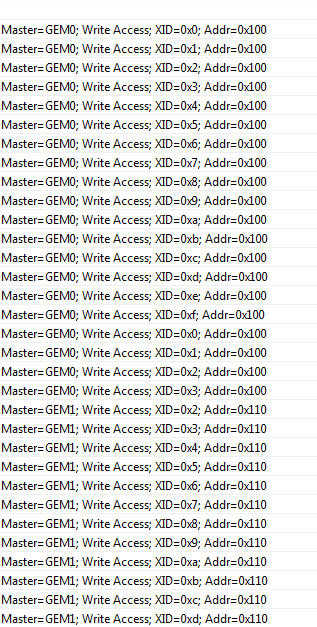
The trace window will appear.

#### Step 6: Enable the Trace Point and Run

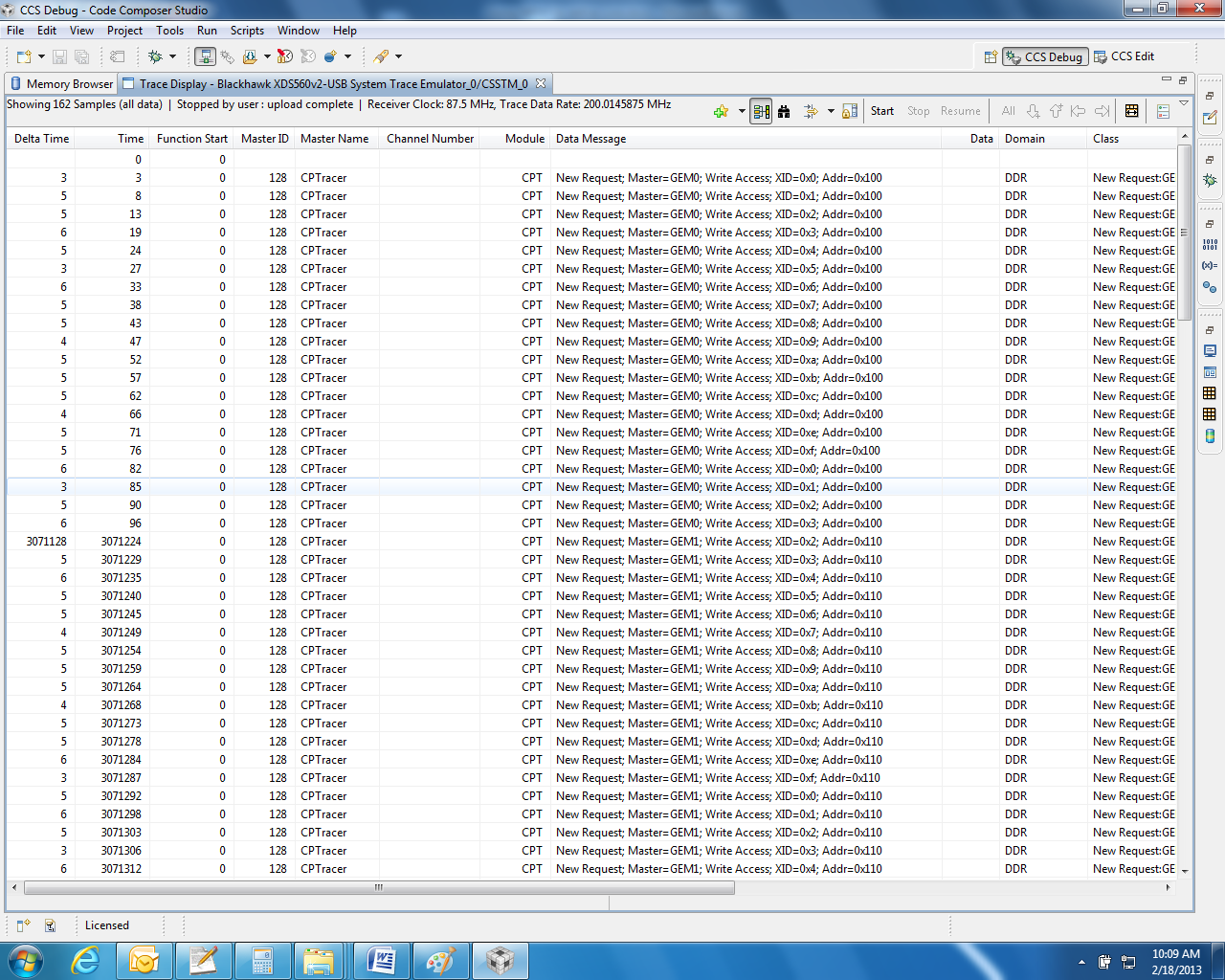
1. If you have not already done so, enable the *Trace point* in the *Breakpoint* window.
2. Run the code on all 8 cores.
3. Wait for the *printf* to verify that all cores are done.
4. *Stop Run* on all of the cores.

It may take several seconds for all the data to get into the CCS, but once it is there, the results will appear as shown below.



1. Double-click on the *Trace* window to enlarge it. Focus on the central part of the window and notice how each core writes to a different physical address:

The start of the page looks as follows:



## Additional Considerations

1. The time unit in the tracer is in tracer ticks driven from an 87.5MHZ clock. So each time unit is 1/87.5 microsecond.
2. If the load is done when the trace is open, the trace will log the loading procedure. To prevent this, load the code before configuring the CCSTM. If you want to run the code again, load it again, and re-configure the CCSTM\_0.
3. Play a little more with the CCSTM configuration to see what other values can be traced.
4. Play with the address bits to see how they can affect the trace values.
5. If the trace file is large, you have to wait until after the system is stopped for the data to reach the console. A percentage value will indicate how much of the trace is already moved to the console (You will not see this in the previous example, since the trace is short).

# Lab 8: STM library and system Trace

## I. Purpose

The goal of this lab is to demonstrate the usage of STM library to collect real-time information into system trace and present it on CCS.

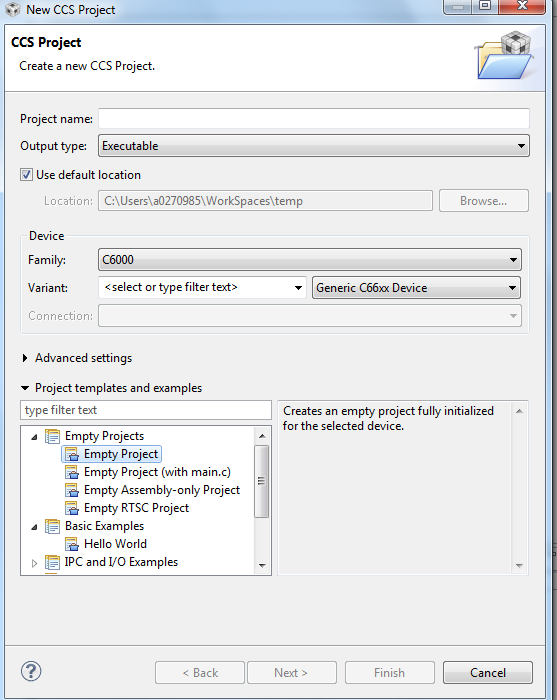
## II. Project Files

The following files are used in this lab:

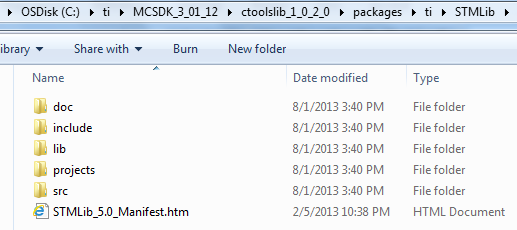
1. Initialization.c
2. StmMain.c
3. System\_trace.c
4. System\_trace.h
5. TraceNoRTSC\_l2.cmd

### Task 1: Build and Run the Project

1. Open CCS.
2. Create new project through the CCS menu item *File* 🡪 *New* 🡪 *CCS Project*.
3. Enter *stm\_example* as a *Project Name*.
4. Click the check box to *Use default location.*

**

1. Set the *Family to C6000* and *Variant* to *Generic C66xxx Device* as shown below:
2. Then press *Finish* to create the new project.
3. Then in the *Project Explorer* view, right-click on the newly-created *stm\_example* project, and click on *Add Files…*
4. Browse to ‘C:\ti\labs\code\STM,’ select all the files in this directory, and click *Open*. When prompted how files should be imported into the project, leave it as default of *Copy File.*
5. If a file main.c was generated when you created the new project, remove the main.c file
6. Examine the code in ‘StmMain.c’ to understand the code.
7. Location of the STM library
   1. The STM library is part of MCSDK 3 release. If you have not installed MCSDK 3, you can get the STM and all advanced debug library from the following address: <https://gforge.ti.com/gf/project/ctoolslib/frs/?action=FrsReleaseBrowse&frs_package_id=92>
   2. If you do not have MCSDK release, load the library from gforge address from above and put it in directory ‘C:\tiMCSDK\_3\_0\_0\_11\ctoolslib\_1\_0\_0\_3\packages\ti\STMLib’
   3. After installing the STM library, the directory ‘C:\tiMCSDK\_3\_0\_0\_11\ctoolslib\_1\_0\_0\_3\packages\ti\STMLib’ has the following sub-directories:



1. We will set the properties for the *Debug* configuration. Right-click on the project. Select *Properties.*
   1. Choose *Build*, click on the *Environment* tab, and click the *Add…*button to add the path to add a variable with *Name* as ‘STM\_ROOT’ and *Value* as ‘C:\tiMCSDK\_3\_0\_0\_11\ctoolslib\_1\_0\_0\_3\packages\ti\STMLib’
   2. Choose *C6000 Compiler 🡪 Optimization* andset/verify the following properties:
      * ‘Optimization level = 0’
      * ‘Optimize for code size = 0’
   3. Choose *C6000 Compiler 🡪 Debug Options* and set/verify the following properties:
      * ‘Debugging model = Full symbolic debug’
   4. Choose *C6000 Compiler 🡪 Include Options*. Under the “Add dir to #include search path” add the following two paths:
      * "${STM\_ROOT}/src"
      * "${STM\_ROOT}/include”

NOTE: This ensures that any include references in the project’s source files to header files located at these paths will be interpreted accurately.

1. In the linker tab open file search path. Add the following path ${STM\_ROOT}\lib to the search path and the library stm.c66xx\_elf.lib to the library list
2. Click the *OK* button to save the project properties and close the *Properties* window.
3. Right-click on the project and select *Build Project*. A successful build will generate the following output on the console:

. . .

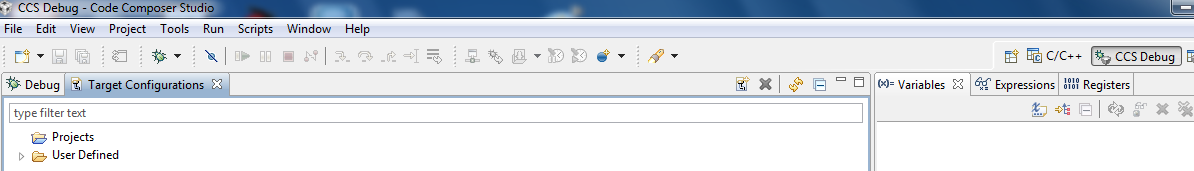
<Linking>

'Finished building target: .out'

\*\*\*\* Build Finished \*\*\*\*

### Task 2: Connect to the EVM

1. Click the *Open Perspective* (available right top corner of the *CCS*).
2. Switch to the Debug Perspective by selecting the CCS menu option *Window* 🡪 *Open Perspective* 🡪 *CCS Debug*.
3. Select the CCS menu option *View* 🡪 *Target Configurations*. Select the target configuration you created in *Lab: CCS Basics*

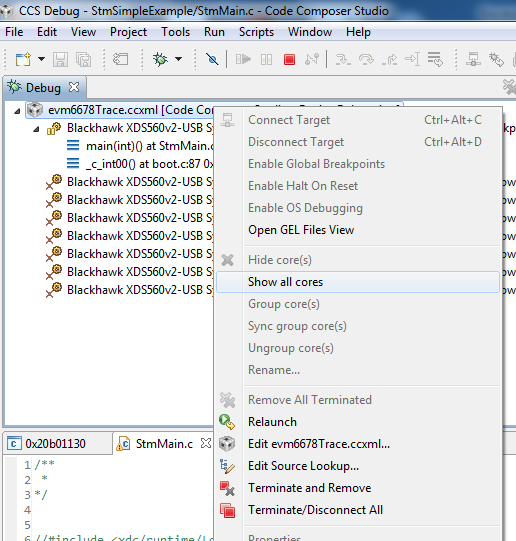


1. Use the target configuration that you created in the *Lab: CCS Basics,* else follow *Task 4* of *Lab: CCS Basics.*
2. Launch the target configuration as follows:
   1. Select the target configuration .ccxml file.
   2. Right click and select *Launch Selected Configuration*.
3. This will bring up the *Debug* window.
   1. Select Core 0 (*C66x\_0*)
   2. Right click and select *Connect Target*.

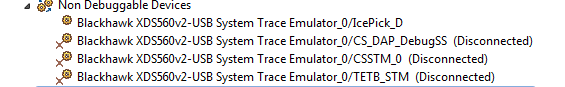
### Task 3: Load the Program and configure the trace

Note - These instructions are for CCS V5.4. For CCS V5.3 follow the instructions in Lab 7

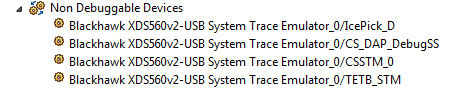
1. Select Core 0 and load the .out file created earlier in the lab.
   1. Select the CCS menu option *Run* 🡪 *Load* 🡪 *Load Program*
   2. Click *Browse project…*
   3. Select by unwrapping the *stmSimpleExample*🡪*Debug* and click *OK.*
   4. Click *OK* to load the *StmSimpleExample.out* application to the target (Core 0).
2. As the picture below shows, select the top line in the debug window and right click on show all cores



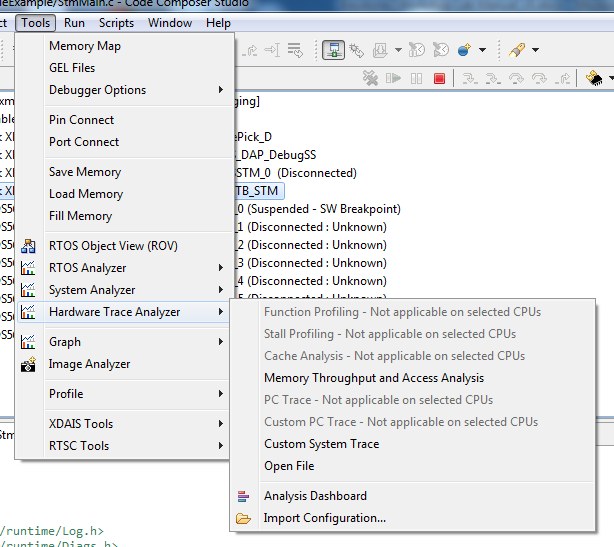
1. Non debuggable devices will appear



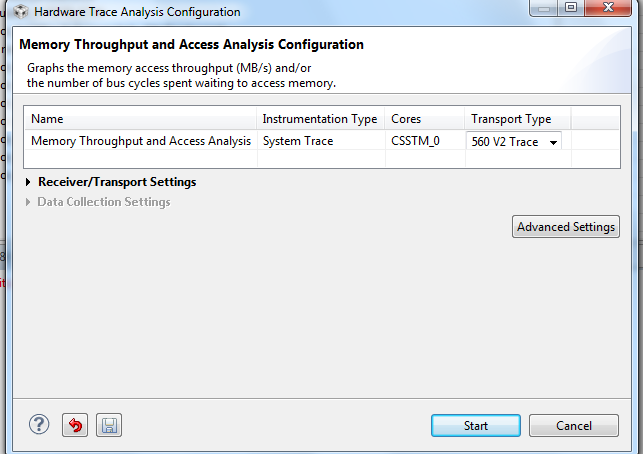
1. Select the Non Debuggable device group, right click and connect Target. You notice that all devices are connected



1. Next we enable the hardware Trace analyzer. Note, these operations are different in CCSV5.3. We present the screen shots of CCSv5.4



1. The following window is open



1. Chose 560 V2 Trace as the transport type, and click start. After couple of seconds several windows will be opened. Enlarge the Trace Viewer CSSTM\_0 window

### Task 3: Run the Program

1. Run the application by selecting the CCS menu option *Run* 🡪 *Resume*.
2. Look at the trace Viewer CSSTM\_0 window you will see the following:

