# TMS320TCI66x

*Keystone Multicore Workshop*

# Lab Manual

**TEXAS INSTRUMENTS**

# Copyright and Contact Information

### Document Copyright

Publication Title:TMS320TCI66x Keystone Multicore Workshop Lab Manual
Publication Number: SPRP820
Revision: A

© 2014 Texas Instruments Incorporated
All Rights Reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

### Software Copyright

Product Name: TMS320TCI66x
Product Release: MCSDK 2.x

© 2014 Texas Instruments Incorporated
All Rights Reserved.

## Contact Information

### General

20450 Century Boulevard
Germantown, MD 20874
Voice: 301.515.8580
Fax: 301.515.7687
Web: www.ti.com (Broadband/Voice over IP)

### Sales Information

E-mail: mktgsupport@list.ti.com

### Applications Engineering

For Registered Customers Only:
E-mail: tech_support@ti.com

The Telogy Software Applications Engineering group is available to all customers who need technical assistance with a Telogy product, technology, or solution. Inquiries are categorized according to the urgency of the issue, as follows:

***Priority Level 4 (P4)*** —You need information or assistance about Telogy product capabilities, product installation, or basic product configuration.

***Priority Level 3 (P3)*** —Your network performance is degraded. Network functionality is noticeably impaired, but most business operations continue.

***Priority Level 2 (P2)*** —Your production network is severely degraded, affecting significant aspects of business operations. No workaround is available.

***Priority Level 1 (P1)*** —Your production network is down, and a critical impact to business operations will occur if service is not restored quickly. No workaround is available.

# Notices and Trademarks

### Important Notice

Texas Instruments Incorporated reserves the right to make changes to its products or discontinue any product or service without notice, and to advise customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied upon is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

Customers are responsible for their applications using Texas Instruments Software.

### Notice of Proprietary Information

Information contained herein is subject to the terms of the Non-disclosure Agreement between Texas Instruments Incorporated and your company, and is of a highly sensitive nature and is confidential and proprietary to Texas Instruments Incorporated. It shall not be distributed, reproduced or disclosed orally or in written form, in whole or in part, to any party other than the direct recipients without the express written consent of Texas Instruments Incorporated.

Telogy Software, VLYNQ, PIQUA Software, wONE, PBCC, Uni-DSL, Dynamic Adaptive Equalization, Telinnovation, TurboDSL Packet Accelerator, interOps Test Labs, TurboDOX, and INCA are trademarks of Texas Instruments Incorporated.

All other brand names and trademarks mentioned in this document are the property of Texas Instruments Incorporated or their respective owners, as applicable.

## Release History

| Release | Date | Chapter/Topic | Description/Comments |
|---------|------|---------------|----------------------|
| SPRP820 | April 2014 | All | Initial release |

# Preface

## About this Document

This document is part of a document set prepared in support of the Texas Instruments/Telogy Software TNETVxxxx x.x product release.

## About the Document Set

Various books in the document set will be of interest to developers of voice-over-packet products according to their role:

- Project Managers
- Hardware Engineers
- Software Engineers
- Test Engineers
- Application Developers

## Texas Instruments Silicon and Telogy Software Documents

The following Texas Instruments/Telogy Software-produced documents are provided with TNETVxxxx Release x.x:

- *DSP Module Software Architecture*

- *MXP Operating Environment Reference GuideMXP Operating Environment User's Manual*

## Document Conventions

This document uses the following conventions:

- Commands and keywords are in **boldface** font.
- Arguments for which you supply values are in *italic* font.
- Terminal sessions and information the system displays are in `screen font`.
- Information you must enter is in **`boldface screen font`**.
- Elements in square brackets ([ ]) are optional.

Notes use the following conventions:

**NOTE**—Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

**CAUTION**—Indicates the possibility of service interruption if precautions are not taken.

**WARNING**—Indicates the possibility of damage to equipment if precautions are not taken.

# Contents

**Chapter 7**

**Chapter 8**

## List of Tables

## List of Figures

## List of Procedures

# Preparations

## 1.1  Introduction

The exercises in this manual were developed for delivery in a classroom environment but have been adapted for use in a personal lab environment. Variations in network configuration, software installation, software version, and Host PC configuration could result in different outcomes.

## 1.2  Software

The exercises in this manual require the following software installations:

- Multicore Software Development Kit (MCSDK) Version 2.x or later
- Code Composer Studio (CCS) Version 5.x or later.

For more information, refer to the BIOS MCSDK Getting Started Guide
http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_Getting_Started_Guide

## 1.3  EVM Configuration

All exercises in this manual were designed for use on the TMS320C6678 Lite evaluation modules from Texas Instruments. While these procedures have not been tested on the TMS320C6657 and the EVMK2H, they should also work on those devices.

For labs requiring a mezzanine card with a trace emulator, the TMS320C6678LE is recommended.

Before you begin, set the EVM to 'no boot' mode as shown in Table 1-1.

**Table 1-1      No Boot Dipswitch Settings**

| 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) |
| **End of Table 1-1** | | | | |

**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

**Procedure 1-1      Create a New Target in CCS**

**Step - Action**

**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:

**2a**   Select the CCS menu option *View → Target Configurations*.

**2b**   Select *User Defined*.

**2c**   Right-click and select *New Target Configuration*.

**3**   Enter the name of the new target configuration in the *File Name:* text box.

**3a**   Set the File name based on the EVM model, *<model>.ccxml*
For example, 'EVM6678LE.ccxml.'

**3b**   Leave the *Location* the default value:
"C:\Documents_and_Settings\student\ti\CCSTargetConfigurations"

**3c**   Click the *Finish* button. The .ccxml file will now open in a GUI-based view with the *Basic* tab active.

**4**   Define the new target configuration by selecting the connection type in the *Basic* Tab.

**4a**   Locate your EVM model Table 1-2 and set the properties accordingly.

**Table 1-2      EVM Emulator Types**

| 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 |
| **End of Table 1-2** | | |

**4b**   The *Connection* drop-down menu identifies the emulator type, as shown in the table above. For example, 'Blackhawk XDS560v2-USB Mezzanine Card.'

**4c**   *Board or Device* identifies the TI processor device, as shown in the table above. For example, 'TMS320C6678.'

**4d**   Under *Save Configuration*, click the *Save* button.

**5** Configure setup on the *Advanced* Tab

**5a** Click the *Advanced* tab at the bottom of the screen.

**5b** 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*

**5c** You will now see a sub-window called *Cpu Properties* that allows you to choose an *initialization script*.

**5d** 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

**5e** Click the *Save* button.

**End of Procedure 1-1**

**Chapter 2**

# CCS Basics (SRIO Loopback)

## 2.1 Purpose

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

## 2.2 Instructions

In this exercise, you will import a sample project from the MCSDK into CCS, build the sample application code, connect to the EVM, load the code, run the application, and verify the results.

The list of processes used in this example are as follows:
- Procedure 2-1 "Import the Example Project"
- Procedure 2-2 "Verify and Set Project Properties"
- Procedure 2-3 "Build the Project"
- Procedure 2-4 "Connect to the Target EVM"
- Procedure 2-5 "Load and Run the Program"

**Procedure 2-1          Import the Example Project**

**Step  -  Action**

**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.

**3a**    Select the CCS menu option *Window →Preferences*

**3b**    In the pop up window that appears, select *Code Composer Studio* ➔ *RTSC* ➔ Products

**3c**    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.

**3d**    For example, select C:\ti\MCSDK_2_1_2_6 to add in the *Tool Discovery Path*.

**3e**    Click *Refresh* to update the discovered tools list.

**4**  Import the example project as follows:

**4a**  Select the CCS menu option *Project* → *Import Existing CCS Eclipse Project*

**4b**  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

**4c**  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.

**4d**  Place a check mark on *Copy projects into workspace*.

**4e**  Click the *Finish* button.

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

**End of Procedure 2-1**

---

**Procedure 2-2       Verify and Set Project Properties**

**Step - Action**

**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.

**6**  Select *Build* properties.

**7**  Choose *C6000 Compiler* → *Optimization* and set/verify the following properties:

- 'Optimization level = 0'
- 'Optimize for code size = 0'

**8**  Select *Build* properties

**9**  Choose *C6000 Compiler* → *Debug Options* and set/verify the following properties:

- 'Debugging model = Full symbolic debug'

**10**   Select *Build* properties, choose *C6000 Compiler* → *Include Options* and ensure that include paths are setup as shown in Figure 2-1.

**Figure 2-1      C6000 Compiler Include Options**
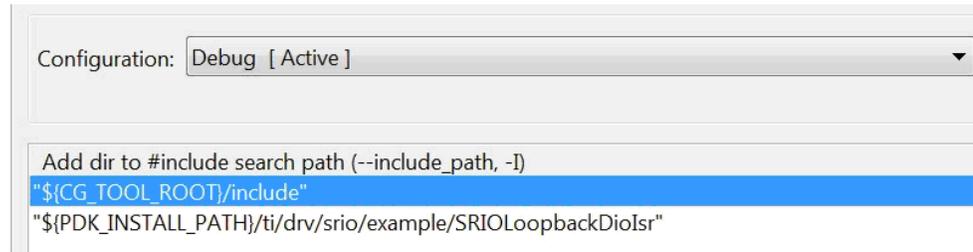


```
Configuration:   Debug  [ Active ]                                              ▼

  Add dir to #include search path (--include_path, -I)
"${CG_TOOL_ROOT}/include"
"${PDK_INSTALL_PATH}/ti/drv/srio/example/SRIOLoopbackDioIsr"
```

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

**End of Procedure 2-2**

---

**Procedure 2-3         Build the Project**

**Step  -  Action**

**1**   In *Project Explorer*, select the *SRIO_LoopbackDioIsrexampleproject* project.

**2**   Build the project:

- 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

**3**   CCS will now attempt to compile and link the project. This may take a few minutes to complete.

**4**   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?

> **Note**—From the *CCS Edit* perspective, check the *Binaries* or *Debug* directory. From the *CCS Debug* perspective, check the *Console*.

**End of Procedure 2-3**

**Procedure 2-4        Connect to the Target EVM**

**Step  -  Action**

**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.

**4**    Select the CCS menu option *View → Target Configurations*. Your newly-created .ccxml target configuration file should be available under *User Defined* target configurations.

**5**    If more than one target is configured, select the target that you defined, right click and set it as default

**6** Launch the target configuration as shown in Figure 2-2:

    **6a** Select the target configuration .ccxml file.

    **6b** Right click and select *Launch Selected Configuration*.

**Figure 2-2**      **Launch Target Configuration**
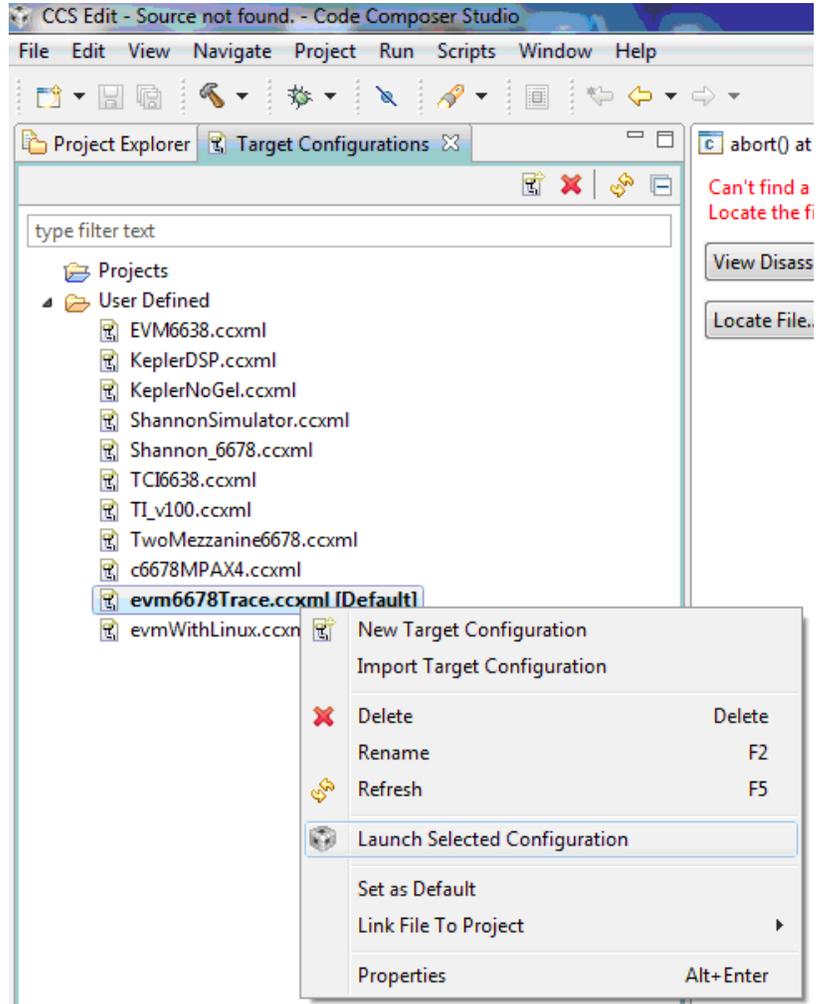
**7**      This will bring up the *Debug* window as shown in Figure 2-3.

**Figure 2-3          CCS Debug Window for evm6678Trace.ccxml**
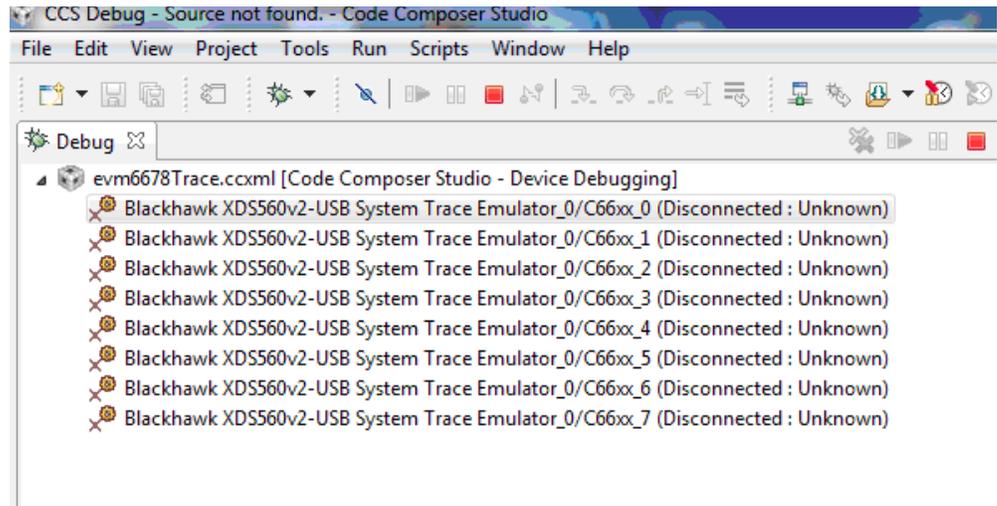


   **7a**   Select Core 0 (*C66x_0*)

   **7b**   Right click and select *Connect Target*.

**End of Procedure 2-4**

---

**Procedure 2-5          Load and Run the Program**

**Step  -  Action**

**1**   Select Core 0 and load the .out file created earlier in the lab.

   **1a**   Select the CCS menu option *Run → Load → Load Program*.

   **1b**   Click *Browse project…*

   **1c**   A pop-up will appear with the projects names.

   **1d**   Select *SRIO_LoopbackDioIsrexampleproject*

   **1e**   Click Debug, then select *SRIO_LoopbackDioIsrexampleproject.out* and click *OK*.

   **1f**   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.

**3**   Once the program completes successfully, you will see the message *"Debug(Core 0): DIO with Interrupts example completed successfully."*

**4**  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.
```

**End of Procedure 2-5**

**QUESTIONS:**

1. Using a text editor, look at the CFG file and determine how the project includes the SRIO module.
   a. What other modules are needed?
      Hint: QMSS and CPPI are needed for SRIO


   b. The CFG file specifies CORE 0 and CORE 1. Is this important to the execution of the application?


2. 3. Load the OUT file and run it on Core 1.
   a. Does it run?

   b. Look at the main function and explain why.

# HyperLink Communication

## 3.1 Purpose

The purpose of this exercise is to demonstrate how to build an application that uses the HyperLink interface on KeyStone C66x devices.

> **Note—**Not all KeyStone devices include HyperLink. Refer to the data manual for your device before proceeding.

> **Note—**Not all KeyStone EVMs include a HyperLink interface on the board. An expansion module may be required. Refer to the Quick Start Guide for your EVM before proceeding.

## 3.2 Instructions

Begin by importing HyperLink example code from the MCSDK and running it in loopback mode on a single C66x EVM. In this example, 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 exercise.

The second part demonstrates the HyperLink connection between two C66x EVMs. As a result, this lab requires a second C66x EVM, a HyperLink cable (HL5CABLE), and an optional EVM breakout card (CI2EVMBOC). One C66xx EVM acts as the sender (Tx) and the other acts as the receiver (Rx).

The list of processes used in this example are as follows:
- Procedure 3-1 "Import the Example Project"
- Procedure 3-2 "Verify and Set Project Properties"
- Procedure 3-3 "Loopback Mode"
- Procedure 3-4 "Build the Project"
- Procedure 3-5 "Connect to the EVM"
- Procedure 3-6 "Load and Run the Program"
- Procedure 3-7 "Board-to-board HyperLink"

**Procedure 3-1          Import the Example Project**

**Step  -  Action**

**1**     Move to the *CCS Edit* Perspective.

**2**     Import the example project as follows:

   **2a**   Select the CCS menu option *Project* → *Import Existing CCS Eclipse Project*

   **2b**   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

   **2c**   From the list of *Discovered projects*, place a check mark in the box next to *hyplnk_exampleProject*

   **2d**   Place a check mark on *Copy projects into workspace*.

   **2e**   Click the *Finish* button.

   **2f**   The *hyplnk_exampleProject* should now appear in the CCS *Project Explorer* on the left-hand side of your screen.

**End of Procedure 3-1**

**QUESTIONS:**

Expand the *hyplnk_exampleProject* folder and double click on *hyplnkLLDCfg.h* to view the file and answer the following:

   1.   How many lanes are configured?

   2.   What is the baud rate? (HINT: 01p250 means 1.25GBaud)

**Procedure 3-2          Verify and Set Project Properties**
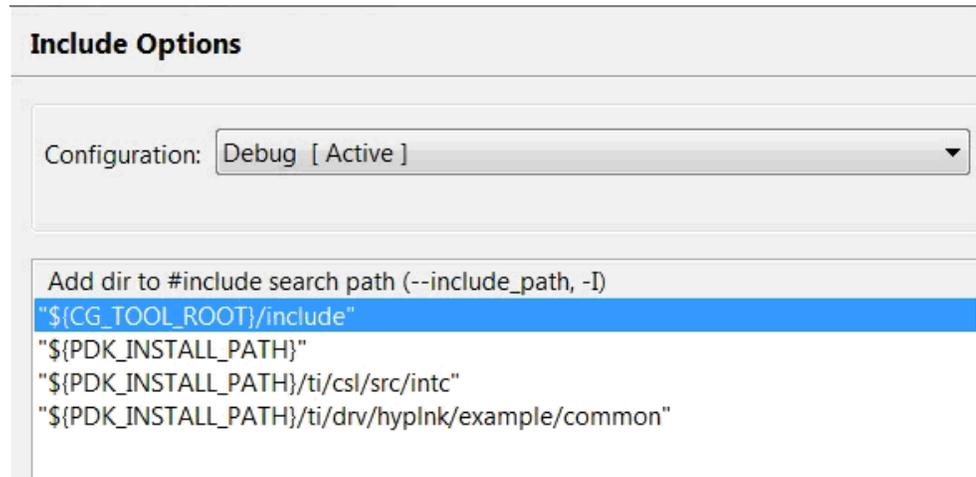
**Step  -  Action**

**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 located in the *main* tab

**6**     Select *Build* properties.

**7**     Choose *C6000 Compiler* → *Optimization* and set/verify the following properties:

   - 'Optimization level = 0'
   - 'Optimize for code size = 0'

**8**     Select *Build* properties

**9**     Choose *C6000 Compiler* → *Debug Options* and set/verify the following properties:

   - 'Debugging model = Full symbolic debug'

**10**     Select *Build* properties, choose *C6000 Compiler* → *Include Options* and ensure that include paths are setup as shown in Figure 3-1.

**Figure 3-1          C6000 Compiler Include Options**



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

**End of Procedure 3-2**

---

**Procedure 3-3          Loopback Mode**

**Step  -  Action**

**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.

**End of Procedure 3-3**

---

**Procedure 3-4          Build the Project**

**Step  -  Action**

**1**     In *Project Explorer*, select the *hyplnk_exampleProject* project.

**2**     Build the project:

**2a**     Select the CCS menu option *Project* → *Build Project*

OR

**2b**     Right-click on the project in *Project Explorer* and select *Build Project*

**3**     CCS will now attempt to compile and link the project. This may take a few minutes to complete.

**4**     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 ****
```

**End of Procedure 3-4**

**QUESTIONS:**

Was the file hyplnk_exampleproject.out generated?

> **Note**—From the *CCS Edit* perspective, check the *Binaries* or *Debug* directory. From the *CCS Debug* perspective, check the *Console*.

**Procedure 3-5      Connect to the EVM**

**Step  -  Action**

**1**    Click the *Open Perspective* (available right top corner of the *CCS*).

**2**    Switch to the Debug Perspective by selecting the CCS menu option

**3**    *Window → Open Perspective → CCS Debug*.

**4**    Select the CCS menu option *View → Target Configurations*. Select the target configuration you created

**5**    Launch the target configuration as follows:

    **5a**    Select the target configuration .ccxml file.

    **5b**    Right click and select *Launch Selected Configuration*.

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

    **6a**    Select Core 0 (*C66x_0*)

    **6b**    Right click and select *Connect Target*.

**End of Procedure 3-5**

**Procedure 3-6      Load and Run the Program**

**Step  -  Action**

**1**    Select Core 0 and load the .out file created earlier in the lab.

    **1a**    Select the CCS menu option *Run → Load → Load Program*

    **1b**    Click *Browse project…*

    **1c**    A pop-up will appear with the projects names.

    **1d**    Select *hyplnk_exampleProject*

    **1e**    Click Debug, then select *hyplnk_exampleProject.out* and click *OK*.

    **1f**    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 ===

. . . .
```

**5**    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*.

**End of Procedure 3-6**

**Procedure 3-7        Board-to-board HyperLink**

**Step  -  Action**

A second C66x EVM is required perform this portion of the lab. You will need a HyperLink cable and (optional) a 2-EVM breakout card.

**1**    Modify the example code for *hyplnk_exampleProject* so it can be run on two EVMs:

    **1a**    Open *hyplnkLLDCfg.h*

    **1b**    Search for `#define hyplnk_EXAMPLE_LOOPBACK`

    **1c**    Comment out this line

    **1d**    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

    **3a**    Open *hyplnkLLDCfg.h*

    **3b**    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 systems must have the same rate!

**End of Procedure 3-7**

**QUESTION:**

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


2.    Look at the errata documentation and identify the theoretical limit of HyperLink transfer. Why is this the case?

**Note—**The maximum length of the board-to-board connection should be 4 inches. Thus, the physical cable connection is not as efficient as a hard wired connection.

# SRIO Type 11

## 4.1 Purpose

The purpose of this exercise is to demonstrate usage of Type 11 SRIO using an example application imported into CCS from the MCSDK.

## 4.2 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

## 4.3 Instructions

First, you will import the example project from MCSDK to CCS. Next, you will build an application from the project and load it to the EVM. Finally, you will run the project and observe the results.

The list of processes used in this example are as follows:

- Procedure 4-1 "Import the Project"
- Procedure 4-2 "Build the Project"
- Procedure 4-3 "Connect to the EVM"
- Procedure 4-4 "Load and Run"

**Procedure 4-1          Import the Project**

**Step  -  Action**

**1**    Open CCS.

**2**    Once CCS starts, verify that the perspective is set to *CCS Edit*.

**3**    Import the project.

    **3a**    Select the CCS menu option *Project* → *Import Existing CCS Eclipse Project*

    **3b**    Set *Select search directory* to point to where the project files for this exercise are located on your computer.

    **3c**    From the list of *Discovered projects*, place a check mark in the box next to *SRIOSingleSRIO*

    **3d**    Place a check mark on *Copy projects into workspace*.

    **3e**    Click the *Finish* button.

**End of Procedure 4-1**

---

**Procedure 4-2          Build the Project**

**Step  -  Action**

**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:

    **3a**    Select the CCS menu option *Project* → *Build Project*

        OR

    **3b**    Right-click on the project in *Project Explorer* and select *Build Project*

**4**    CCS will now attempt to compile and link the project. This may take a few minutes to complete.

**5**   Verify that the executable (.out) was built by looking at the debug directory (assuming the build configuration is debug configuration) as shown in Figure 4-1.

**Figure 4-1**      **Verify Successful .out Build**
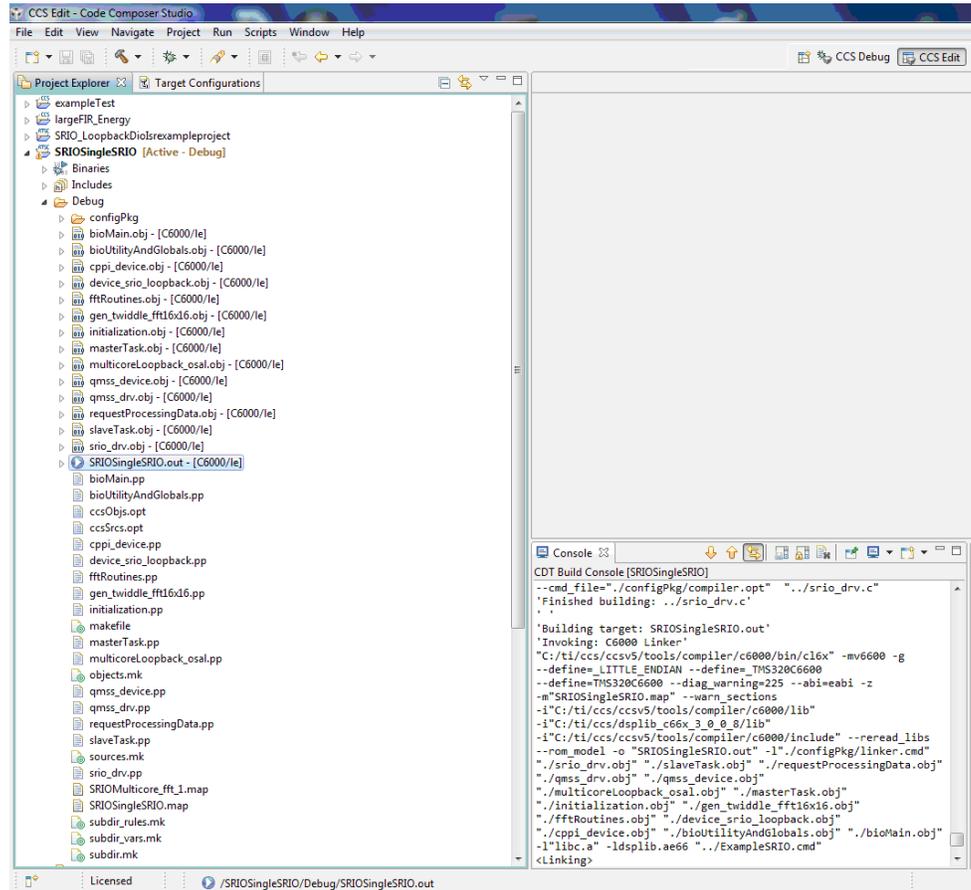


End of Procedure 4-2

---

**Procedure 4-3**      **Connect to the EVM**

**Step  -  Action**

**1**   Cycle 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 previously.

**5**   Launch the target configuration as follows:

   **5a**   Select the target configuration .ccxml file.

   **5b**   Right click and select *Launch Selected Configuration*.

**6**    This brings up the *Debug* window.

**7**   Select all cores, right click, and group them by selecting *Group Core(s)* as shown in Figure 4-2. The default name will be **Group 1**

**Figure 4-2     CCS Debug: Select and Group Cores**
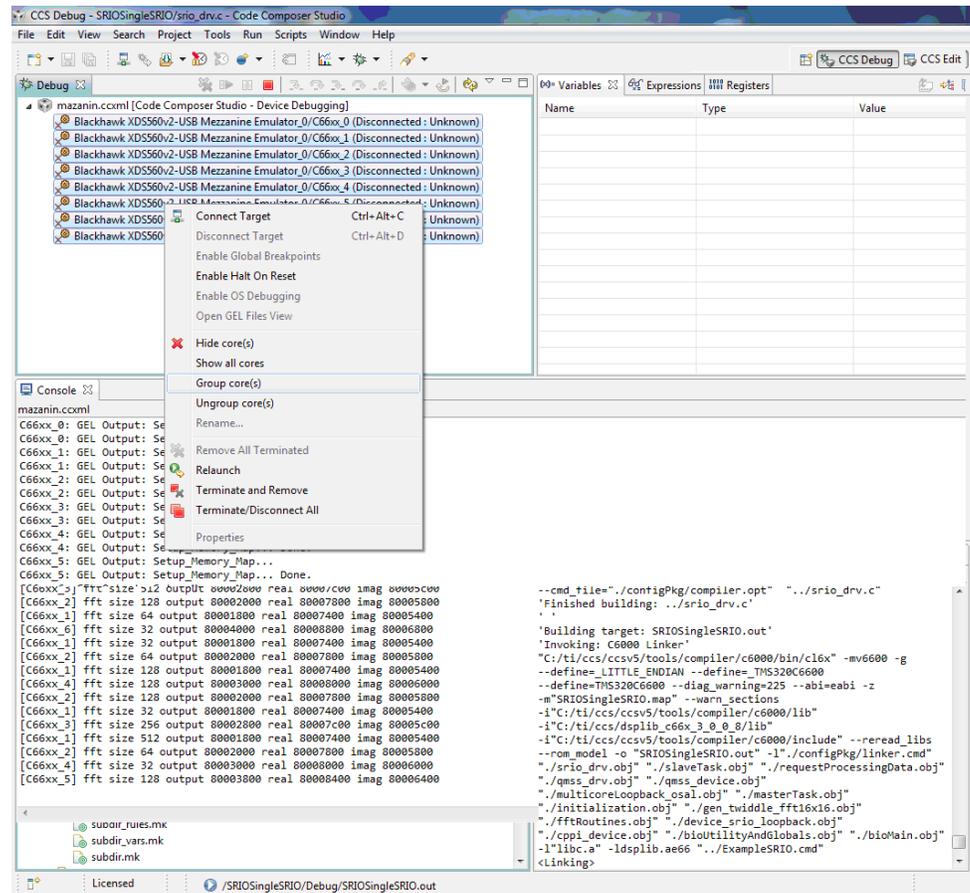


End of Procedure 4-3

**Procedure 4-4     Load and Run**

**Step  -  Action**

**1**    Select Group1 and connect all cores in the group:

    **1a**    From the CCS *Run* menu, select *Connect Target*.

        OR

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

        OR

    **1c**    Click the Connect Target icon.

**2**    Load the *SRIOSingleSRIO.out* to all cores in the group:

    **2a**    From the *Run* menu, select *Load*.

        OR

    **2b**    Click the *Load* icon

**3** Run the code in one of the following ways:

**3a** Press F8

**3b** From the *Run* menu, select *Resume*

**3c** Click on the *Resume* icon (green arrow).

The output results appear as shown in Figure 4-3:

**Figure 4-3** **SRIOSingleSRIO Run Results**
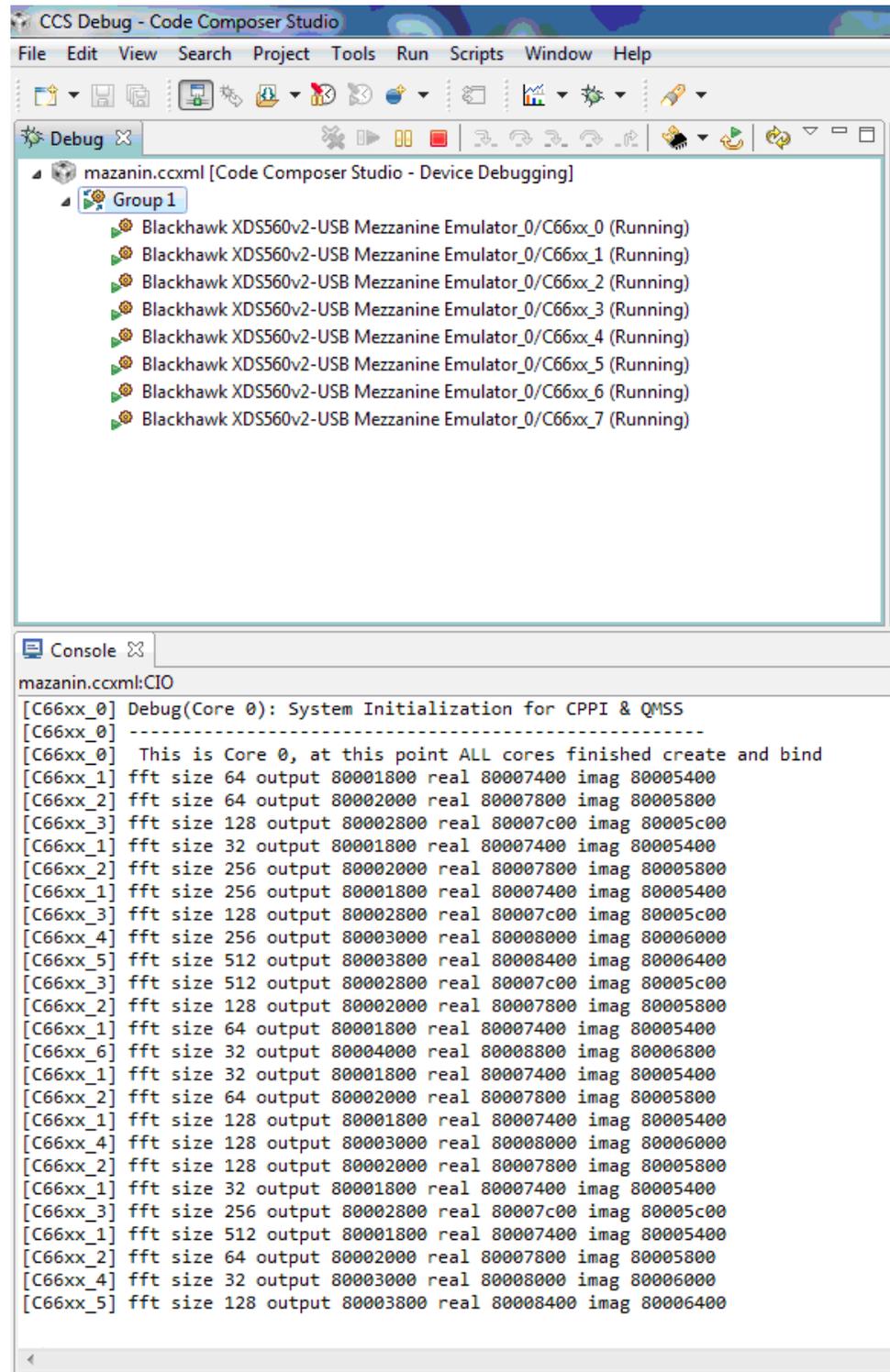
**4**     Observe the results, then suspend the run:

   **4a**   From the *Run* menu, choose *Suspend*.

           OR

   **4b**   Click on the *Suspend* icon (the yellow "pause" lines)

**End of Procedure 4-4**

# Optimization

## 5.1  Purpose

The goal of this exercise is to demonstrate some basic optimization techniques. This exercise works on any KeyStone EVM board. It may also be used with the simulator in conjunction with the estimated cycle count.

## 5.2  Project Files

The following files are used in this lab:
- firMain.c
- intrinsicCFilters.c
- linker.cmd
- naturalCFilters.c
- test.h
- utilities.c

## 5.3  Instructions

In the first part of the exercise, you will build, load and run a project on the EVM without optimization. In the second part, you will enable optimization and analyze the results.

The list of processes used in this example are as follows:
- Procedure 5-1 "Build and Run the Project"
- Procedure 5-2 "Connect to the EVM"
- Procedure 5-3 "Load and Run the Program"
- Procedure 5-4 "Compiler Optimization"
- Procedure 5-5 "Enable Software Pipelining"
- Procedure 5-6 "Align the Data"
- Procedure 5-7 "Enable the MUST_ITERATE Pragma"
- Procedure 5-8 "Cache Considerations"

**Procedure 5-1        Build and Run the Project**

**Step - Action**

**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 in Figure 5-1:

**Figure 5-1        CCS Project Settings**
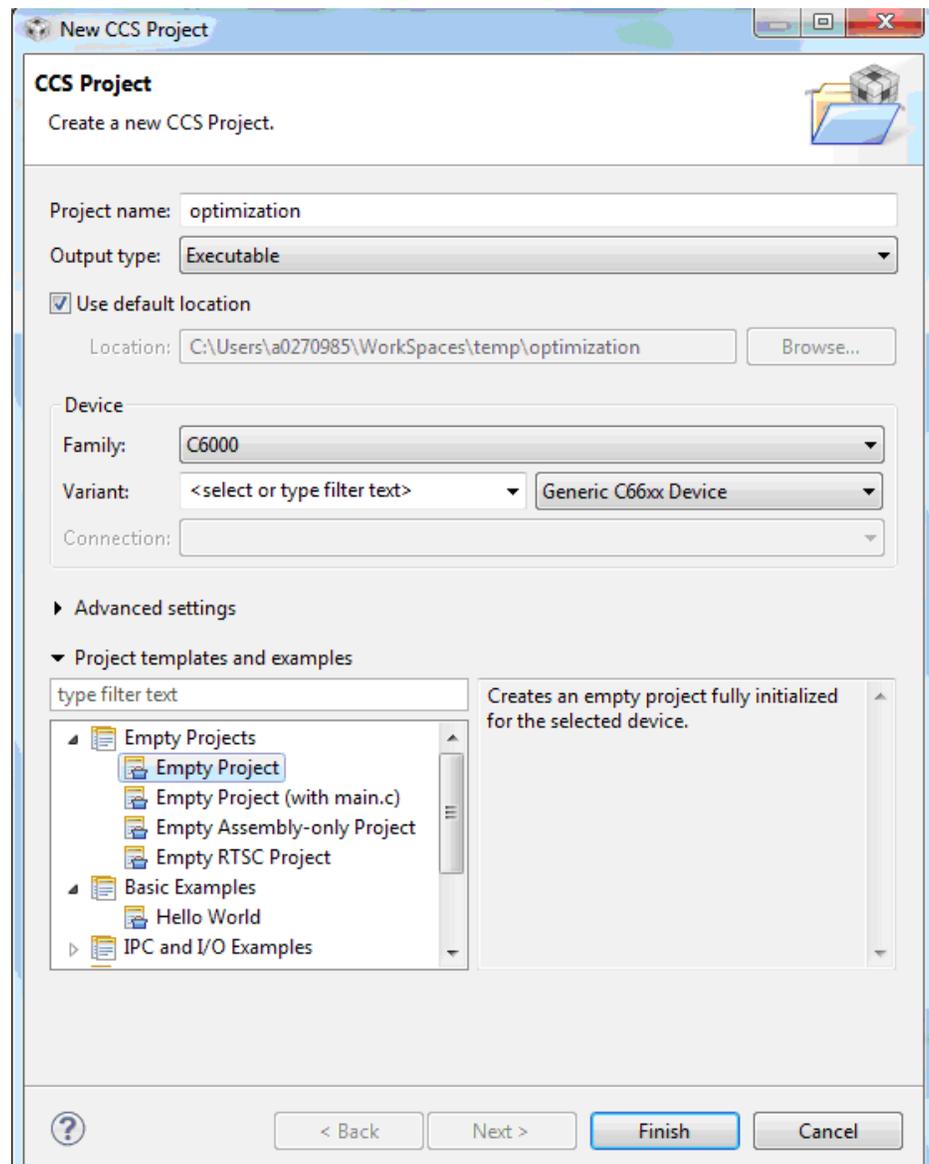


**Note—**The screen shots may reflect different location for the project.

**6**    Then press *Finish* to create the new project.

**7**    Then in the *Project Explorer* view, right-click on the newly-created *optimization* project, and click on *Add Files…*

**8**    Browse to the project directory you created for this exercise and select all required files as outlined at the beginning of this exercise, then click *Open*.

**9** 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.

**10** 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.

**11** Set the properties for the *Debug* configuration. Right-click on the project. Select *Properties*.

 **11a** 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'

 **11b** Choose *C6000 Compiler → Optimization* and set/verify the following properties:

  • 'Optimization level = 0'

  • 'Optimize for code size = 0'

 **11c** Choose *C6000 Compiler → Debug Options* and set/verify the following properties:

  • 'Debugging model = Full symbolic debug'

 **11d** 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.

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

**13** 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 ****
```

**End of Procedure 5-1**

---

**Procedure 5-2      Connect to the EVM**

**Step - Action**

**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:

 **4a** Select the target configuration .ccxml file.

 **4b** Right click and select *Launch Selected Configuration*.

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

 **5a** Select Core 0 (*C66x_0*)

 **5b** Right click and select *Connect Target*.

**End of Procedure 5-2**

**Procedure 5-3      Load and Run the Program**

**Step  -  Action**

**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.

    **2a**    Select the CCS menu option *Run → Load → Load Program*

    **2b**    Click *Browse project…*

    **2c**    Select *optimization.out* by unwrapping the *OptimizationDebug* and click *OK*.

    **2d**    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)

**End of Procedure 5-3**

---

**Procedure 5-4      Compiler Optimization**

**Step  -  Action**

**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.

    **2a**    Right-click on the *Optimization* project. Select *Build Configurations → Set Active → Release*

**3**    Right-click on the *Optimization* Project. Select Properties.

    **3a**    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'

    **Note—**The path and platform may differ for your local devices.

    **3b**    Select *C6000 Compiler → Optimization* and set/verify the following properties:

       •    'Optimization level = 3'

    **3c**    Choose *C6000 Compiler → Debug Options* and ensure that:

       •    'Debugging model = Suppress all symbolic debug generation'

    **3d**    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.

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

**5**    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 ****
```

**6**    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).

**7**    Select Core 0 and load the .out file created earlier in the lab.

   **7a**   Select the CCS menu option *Run → Load → Load Program*

   **7b**   Click *Browse project…*

   **7c**   Select *optimization.out* by unwrapping the *OptimizationRelease* and click *OK.*

   **7d**   Click *OK* to load the application to the target (Core 0).

**8**    Run the application by selecting the CCS menu option *Run → Resume*.

**9**    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
```

**End of Procedure 5-4**

**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?

   **Note—**Consider how intrinsic functions utilize the processor.

**Procedure 5-5          Enable Software Pipelining**

**Step  -  Action**

**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.

**QUESTIONS:**

Open the 'intrinsicCFilter.asm' file and answer the following questions:

- Was the compiler able to schedule the software pipeline?

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

📝
**Note**—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?

📝
**Note**—Think about the inline function.

**4**    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)

**5**    Rebuild the code, load, and run.

**6**    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.

**End of Procedure 5-5**

**Procedure 5-6          Align the Data**

**Step  -  Action**

**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)?

> **Note—**Find the pragma that aligns the data. Consider other ways 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.

**End of Procedure 5-6**

**Procedure 5-7          Enable the MUST_ITERATE Pragma**

**Step  -  Action**

**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

**End of Procedure 5-7.**

**Procedure 5-8          Cache Considerations**

**Step  -  Action**

1    In *test.h*, change the number of elements to 2K, 4K, 8K and 16K

2    In Table 5-1, record the cycle counts for each case.

**Table 5-1          Clock Values & Cycle Counts**

| Size | Multiply for 32K | Clock Value | Cycles for 32K |
|---|---|---|---|
| 32K | 1 | | |
| 16K | 2 | | |
| 8K | 4 | | |
| 4K | 8 | | |
| 2K | 16 | | |
| **End of Table 5-1** | | | |

**QUESTION:**

Why the non-linear jump in performance?

**Note—**Think about cache trashing.

**End of Procedure 5-8**

## 5.3.1  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?

**Note—**To understand better the cache issue, you should consider performing the cache debug exercise in this document.

## 5.3.2  Change the Code to Speed Up to 32K

Change the code to increase the speed to 32K and take full advantage of the cache.

**Note—**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.

# Using Advanced Debug

## 6.1 Purpose

In the optimization exercise, 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 into 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 a cache miss. The same is true for the third filter and the fourth.

The elements in the optimization exercise are floating-point complex, single-precision numbers. So each element requires 8 bytes; Single-precision is 32-bits, equal to 4 bytes, and floating point complex doubles the number of bytes to 8.

In this exercise, you will use the debug features in Code Composer Studio (CCS) to better understand the cache behavior and determine why the higher results start at 8K and not at 4K.

### 6.1.1 Why the Debug Version is Used

For the purpose of reading data from 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, for this exercise, the debug version is used with non-optimization and full symbolic debug turned on.

## 6.2 Instructions

First, you will look at the cache behavior when the number of elements is 4K. Next, you will do the same with 16K elements. Lastly, you will look at the 8K elements use the information derived from the previous two tasks to draw a conclusion.

The list of processes used in this example are as follows:
- Procedure 6-1 "View the 4K Case"
- Procedure 6-2 "Looking at the Cache Lines for 4K Case"
- Procedure 6-3 "View the Cache Lines for 16K Case"
- Procedure 6-4 "View the Cache Lines for 8K Case"

**Procedure 6-1        View the 4K Case**

**Step - Action**

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

```
#define NUMBER_OF_ELEMENTS    4096
```

**2** Rebuild the code.

**3** 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).

**4** From the *Run* menu in the *Debug* perspective, enable the *Clock*.

**5** Open the file intrinsicCfilters.c and put a breakpoint after the first filter.

**6** Run the code and verify that it stopped at this breakpoint, which should appear as shown in Figure 6-1:

**Figure 6-1** **CCS Debug: IntrinsicCFilters Breakpoint**
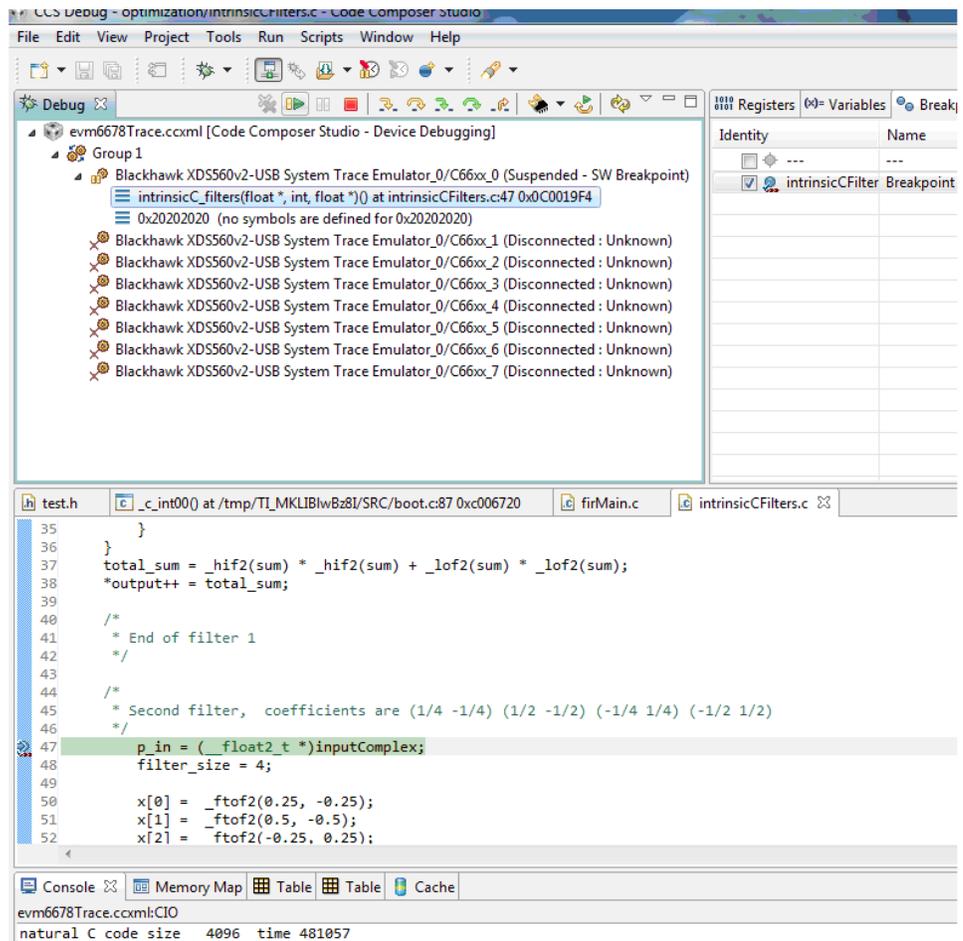


**End of Procedure 6-1**

**Procedure 6-2        Looking at the Cache Lines for 4K Case**

**Step  -  Action**

**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 in Figure 6-2.

**Figure 6-2        CCS Debug: Show View**
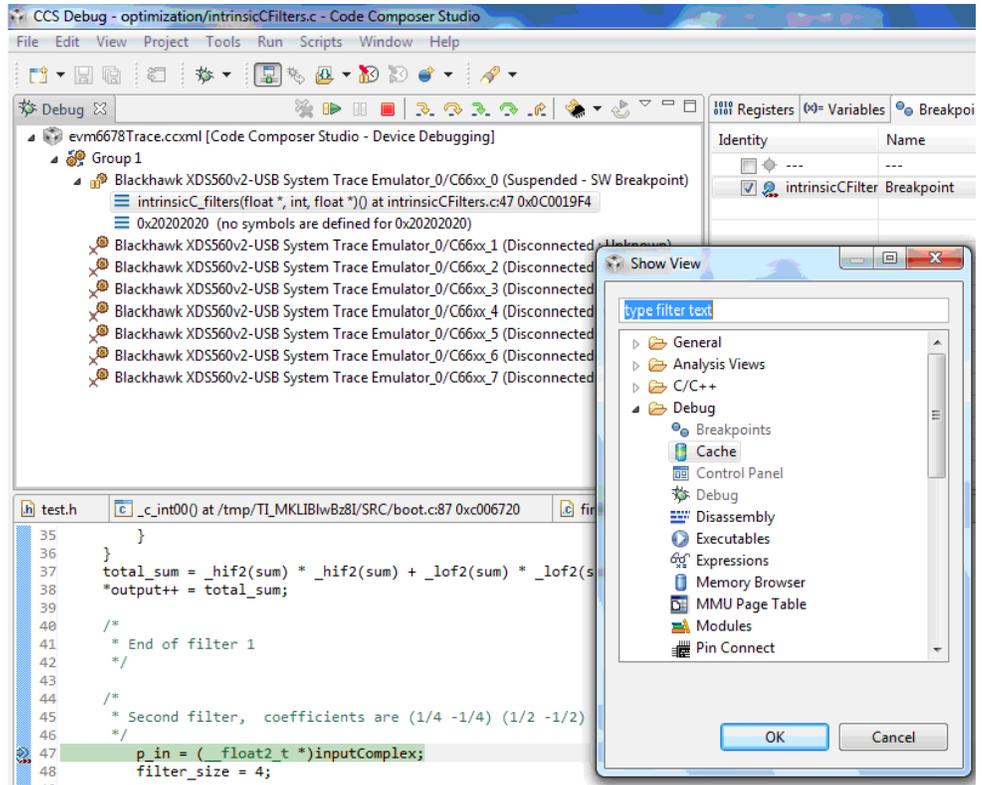
**2**  Select *Cache* and double click to open the *Cache* window, as shown in Figure 6-3.

**Figure 6-3**  **CCS Debug: IntrinsicCFilters Cache (4K)**

**3** Double-click on the *Cache* tab to enlarge the window. Then double-click on any line of the L1D to display all L1D lines, as shown in Figure 6-4.

**Figure 6-4** **CCS Debug: IntrinsicCFilters L1D Cache Lines (4K)**



**4** 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. Notice that the valid flag is set for this cache line.

**5**  Next, look at the last L1D cache line as shown in Figure 6-5:

- 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.

**Figure 6-5      CCS Debug: IntrinsicCFilters L1D Cache (Last Line, 4K)**



**End of Procedure 6-2**

**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?

**Procedure 6-3          View the Cache Lines for 16K Case**

**Step  -  Action**

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

```
#define NUMBER_OF_ELEMENTS       16384
```

**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. The cache lines should be the same or similar to those shown in Figure 6-6:

**Figure 6-6          CCS Debug: IntrinsicCFilters L1D Cache Lines (16K)**



**End of Procedure 6-3**

**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?

**Procedure 6-4**      **View the Cache Lines for 8K Case**

**Step - Action**

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.

**End of Procedure 6-4**

**QUESTIONS:**

Examine the cache.

- 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?

**Finding the Bug**

You should now understand that the number of elements as 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.

# Using MPAX to Define Private Core Memory in DDR

## 7.1 Purpose

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

Downloading the same execution code for multiple cores is very efficient. However, since each core has its own configuration and 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 exercise 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.

## 7.2 Overview

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

The C66x CorePac User's Guide (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). In this example, translation registers are used to assign the different physical address of each core to the same logical address. This 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.

### 7.2.2 Coherency Discussion

For cases where L1D is configured as a cache, there is hardware coherency between L2 data and L1 D inside the C66x 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 guarantee does not hold 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 Chapter 4.4.

### 7.2.3 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 MPAX registers that correspond to 16 privilege IDs. Each set has 8 registers. 8 more MPAX 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 initiates the EDMA transform, the EDMA privilege ID 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.

### 7.2.4  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, you will reduce the size of DDR to 1G. You will then use the other 1G for private memories.

The default settings of the MPAX registers for all cores are as follows:

- 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:
  - Core 0 ->  8 1000 0000 to 8 10ff ffff
  - Core 1 ->  8 1100 0000 to 8 11ff ffff
  - Core 2 ->  8 1200 0000 to 8 12ff ffff
  - Core 3 ->  8 1300 0000 to 8 13ff ffff
  - Core 4 ->  8 1400 0000 to 8 14ff ffff
  - Core 5 ->  8 1500 0000 to 8 15ff ffff
  - Core 6 ->  8 1600 0000 to 8 16ff ffff
  - 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, full permission values are given 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

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

### 7.2.5  MAR Registers

Next, 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 has 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).

## 7.3 Instructions

The list of processes used in this example are as follows:

- Procedure 7-1 "Run the Example Code"
- Procedure 7-2 "Connect to the Non-debuggable Devices (Esp. CCTMS_0)"
- Procedure 7-3 "Load the Code to the 8 Cores"
- Procedure 7-4 "Configure the CSSTM_0 Trace Control"
- Procedure 7-5 "Add and Configure the Trace Location"
- Procedure 7-6 "Start Display"
- Procedure 7-7 "Enable the Trace Point and Run"

**Note—**This exercise requires a mezzanine card with a trace emulator on the target platform.

**Procedure 7-1      Run the Example Code**

**Step - Action**

**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** 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:

    **7a** Select a single core.

    **7b** Open a memory view window (View-> memory).

    **7c** Look at address 0x9000 0000 for all cores (This is external memory).

    **7d** Core zero will have values start at 0, 1, 2, etc.

    **7e** Core 1 will have values 0x0002 0000 and incrementing by 1.

    **7f** Core 2 will have the first value 0x0004 0000 and incrementing by 1.

    **7g** Core 3, 4, 5, 6, and 7 will have first values 0x0006 0000, 0x0008 0000, 0x000a 0000, 0x000c 0000 and 0x000e 0000, respectively.

**Conclusion:** The same logical addresses (0x9000 0000 and on) have a different physical address for each core.

**End of Procedure 7-1**

### 7.3.1 Using Trace to Verify the Write Physical Address

The next section describes how to use the trace facility to verify that each core actually writes to a different physical address.

**Procedure 7-2        Connect to the Non-debuggable Devices (Esp. CCTMS_0)**

**Step  -  Action**

**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.

**4**    The debug window should appear as shown in Figure 7-1.

**Figure 7-1        CCS Debug: MPAX Utilities**
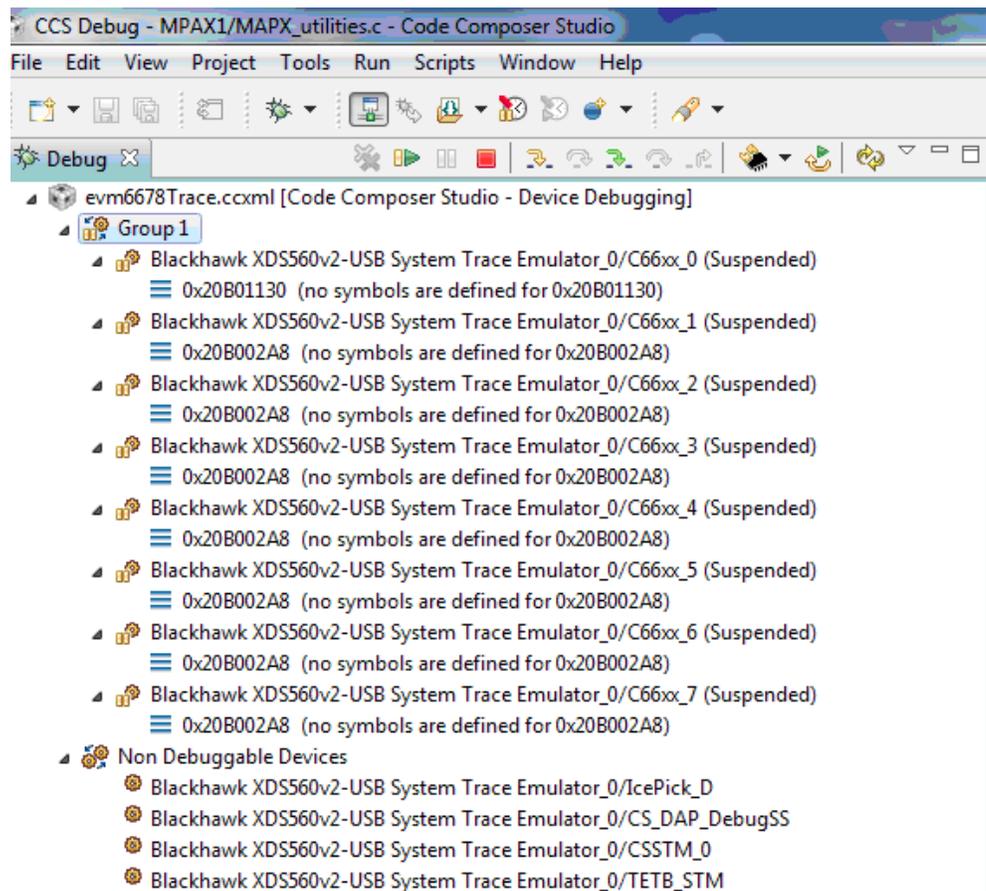


**End of Procedure 7-2**

**Procedure 7-3      Load the Code to the 8 Cores**

**Step  -  Action**

**1**    To make the results easier, change the number of elements (Line 171 in testMPAX1.c) to 10.

**2**    Build the project again and load it o the 8 cores.

**End of Procedure 7-3**

**Procedure 7-4      Configure the CSSTM_0 Trace Control**

**Step  -  Action**

**Note—**These instructions are for CCS V5.3. For CCS V5.4 follow the instructions in Chapter 8 on page 8-1.

**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**    As shown in Figure 7-2, 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.

**Figure 7-2      Trace System Control Setting**
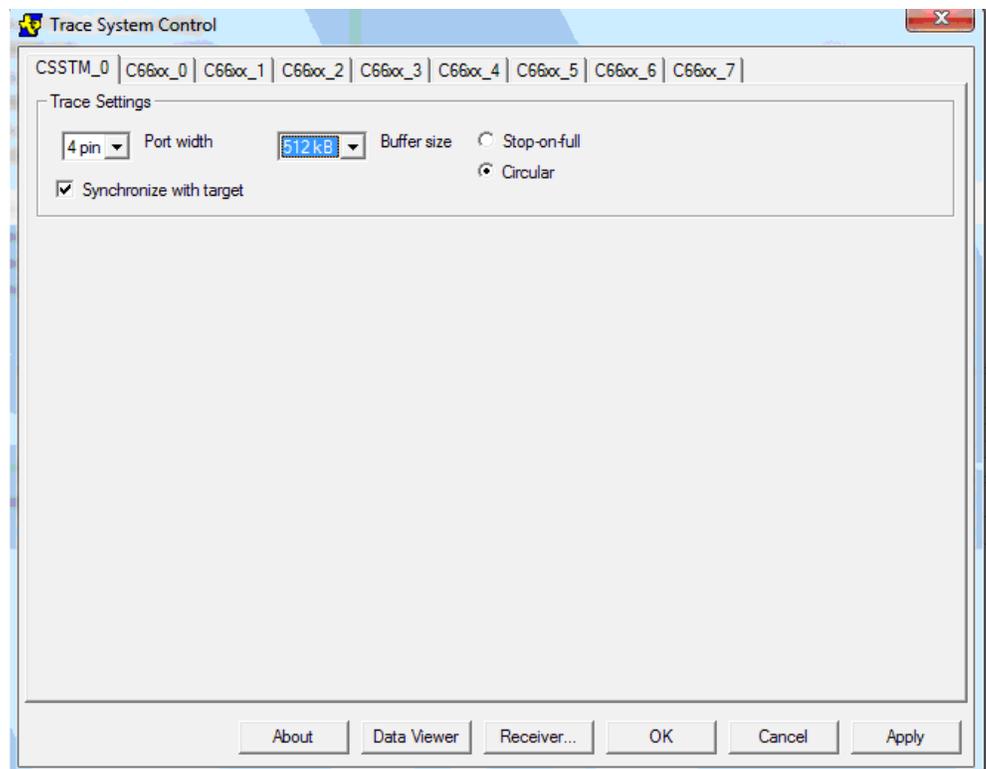
**4**     Click on the *Receiver…* button to open the Select Receiver window as shown in Figure 7-3

    **4a**     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.

    **4b**     Click *OK* on the dialogue box and then click *Apply* and wait for the programming to be done.

**Figure 7-3          Trace System Control Settings: Select Receiver**
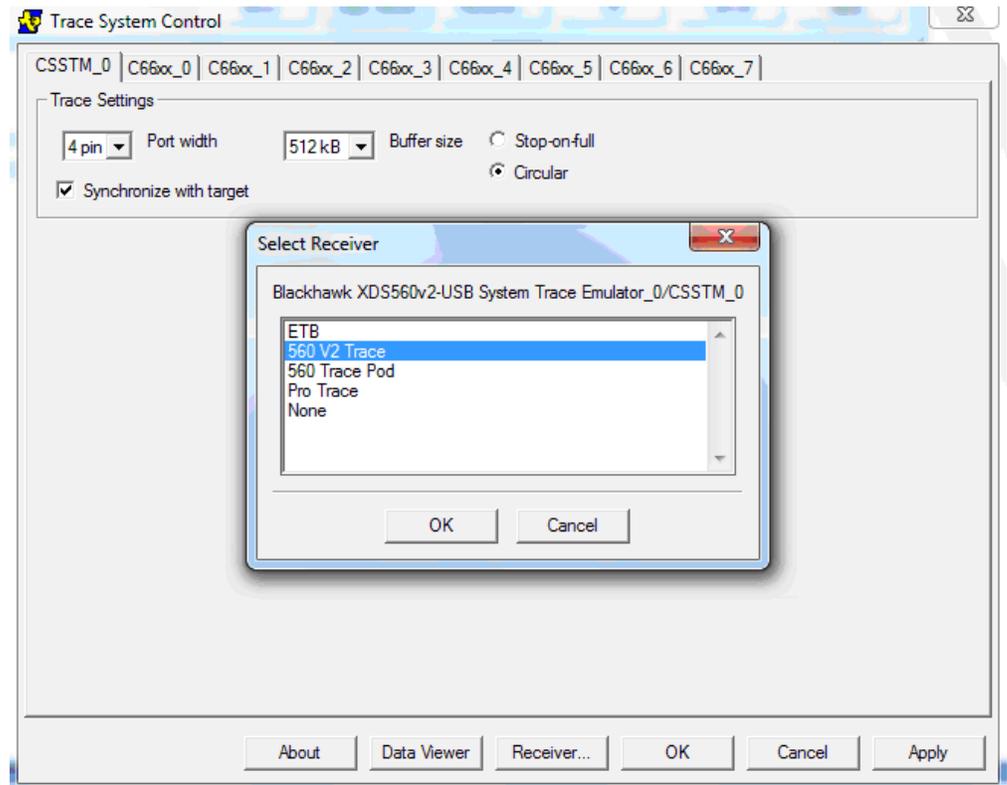


**End of Procedure 7-4**

**Procedure 7-5          Add and Configure the Trace Location**

**Step  -  Action**

**1**    From the *View* tab, open a *Breakpoint* window.

  **Note—**Trace points are currently (CCS5 version 5.3.x) defined from the *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 *Breakpoint Properties* window is opened as shown in Figure 7-4:

**Figure 7-4          Breakpoint Properties: Default Configuration**
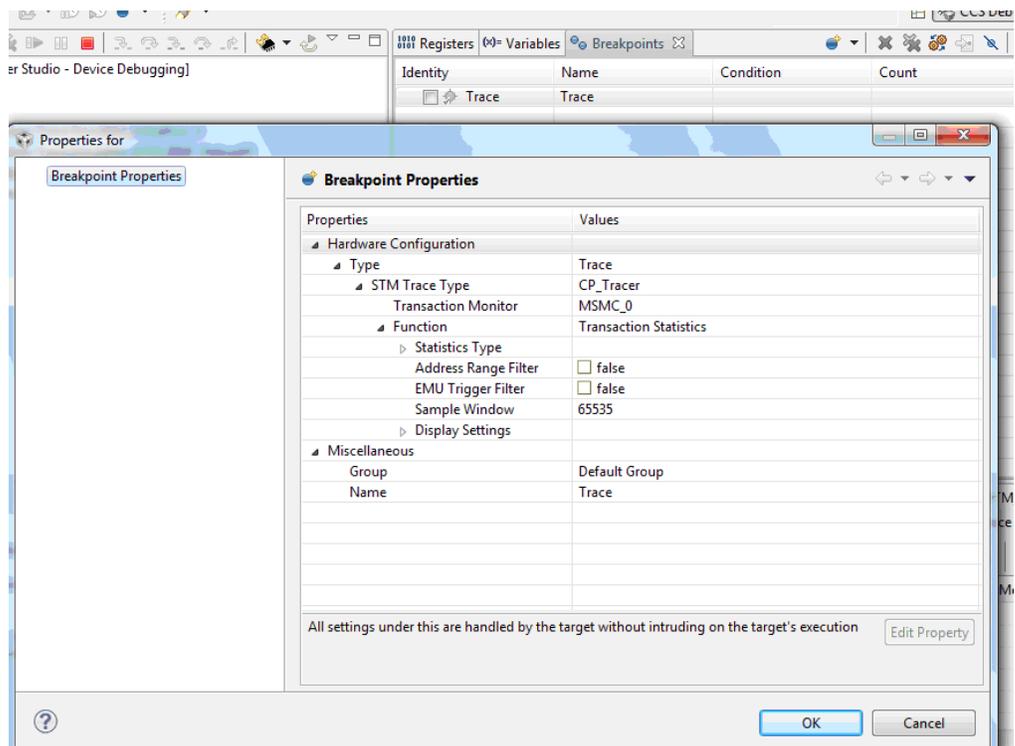
**7**    To configure a Breakpoint Properties value, click on the value line to the right of the each value to access a pull-down menu of options. The trace for this example is configured as follows:

    **7a**    STM Trace Type: CP_Tracer

    **7b**    Transaction Monitor: DDR3

    **7c**    Function: Transaction Logging (This will open a new dialogue box to choose what to log)

    **7d**    Transaction Master ->GEM (Select only the GEM tab, this will open the next level – which GEM to follow)

    **7e**    All GEMs should remain active. So select all the GEM from 0 to 7

    **7f**    At this point, the window appears as shown in Figure 7-5.

**Figure 7-5**        **Breakpoint Properties: Example Configuration**
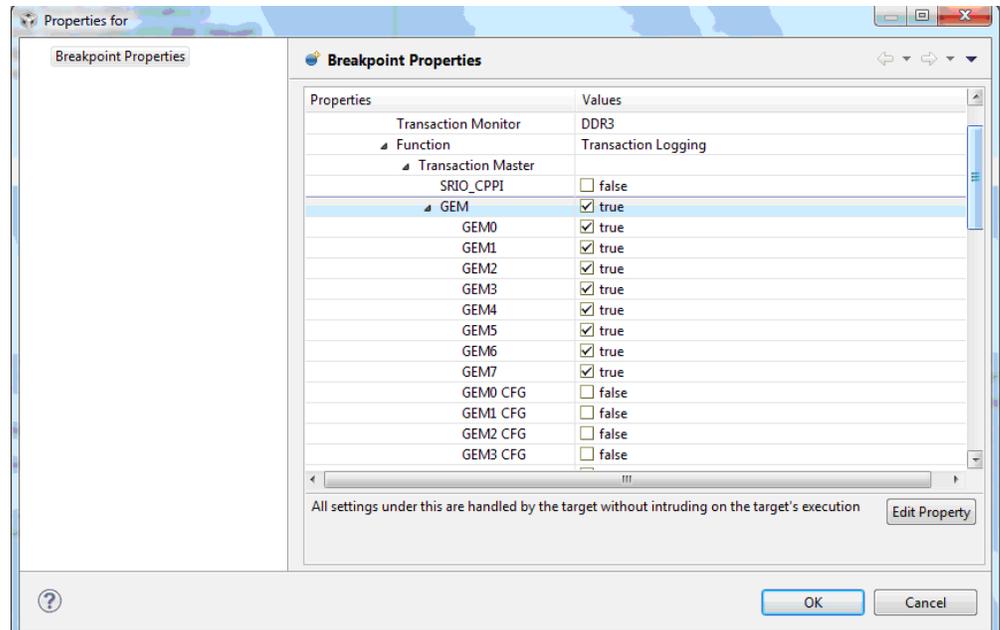


**8**    Scroll down in the window to *Event Filter*, *Transaction Type*, and set *Only CPU access* to *true*.

**9**    *Export Configuration* should be set to *true* for *Status* and *New Requests*.

**10** Address mask: In this example, you want to see the physical addresses. The trace can show only 10 bits of address. So you want to choose the right bits.

Of course, you can run the trace multiple times with each trace covering a different set of bits. 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 you 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.

Click on *Export Bits*. In the pull-down menu (see Figure 7-6), choose 29:20:

**Figure 7-6        Breakpoint Properties: Address Mask Export Bits**
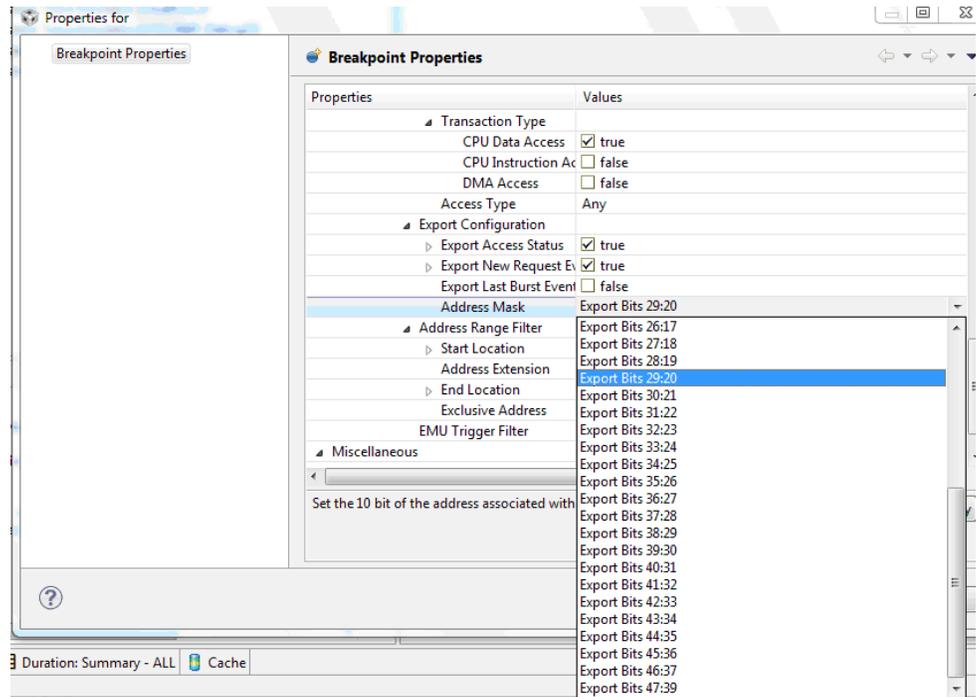
**11**      No other filtering will be used. So both the Address Range Filter and EMU Trigger Filter should be marked false. Click OK.

**12**      The Breakpoint Properties window should now be configured as shown in Figure 7-7

**Figure 7-7        Breakpoint Properties: Example Configuration Complete**
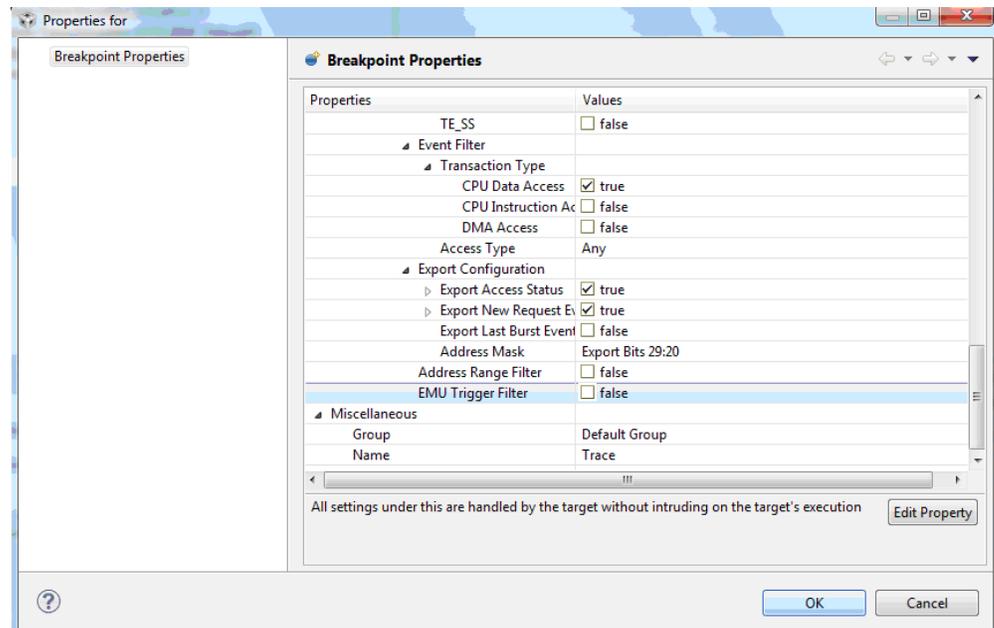


**End of Procedure 7-5**

---

**Procedure 7-6          Start Display**

**Step  -  Action**

**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 shown in Figure 7-8:

**Figure 7-8        Choosing the Trace Analyzer**
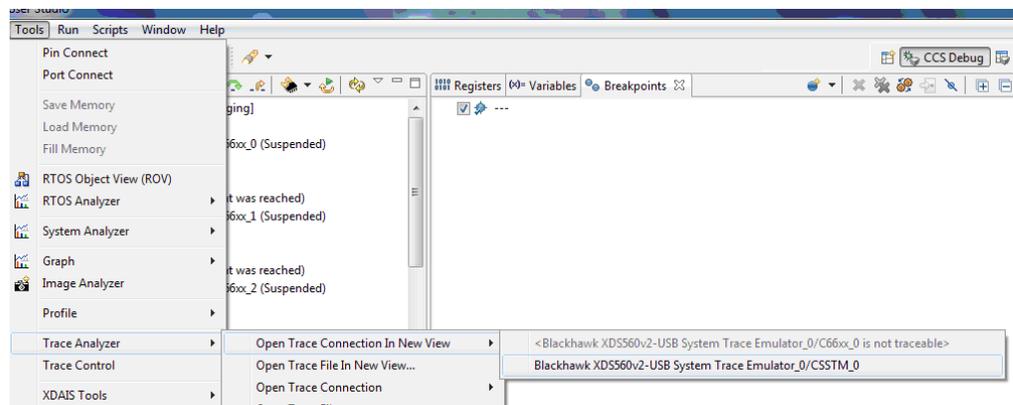


**2**      The Trace window will appear.

**End of Procedure 7-6**

**Procedure 7-7          Enable the Trace Point and Run**

**Step  -  Action**

**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. Once it is there, the results will appear as shown in Figure 7-9.

**Figure 7-9          Trace Results**
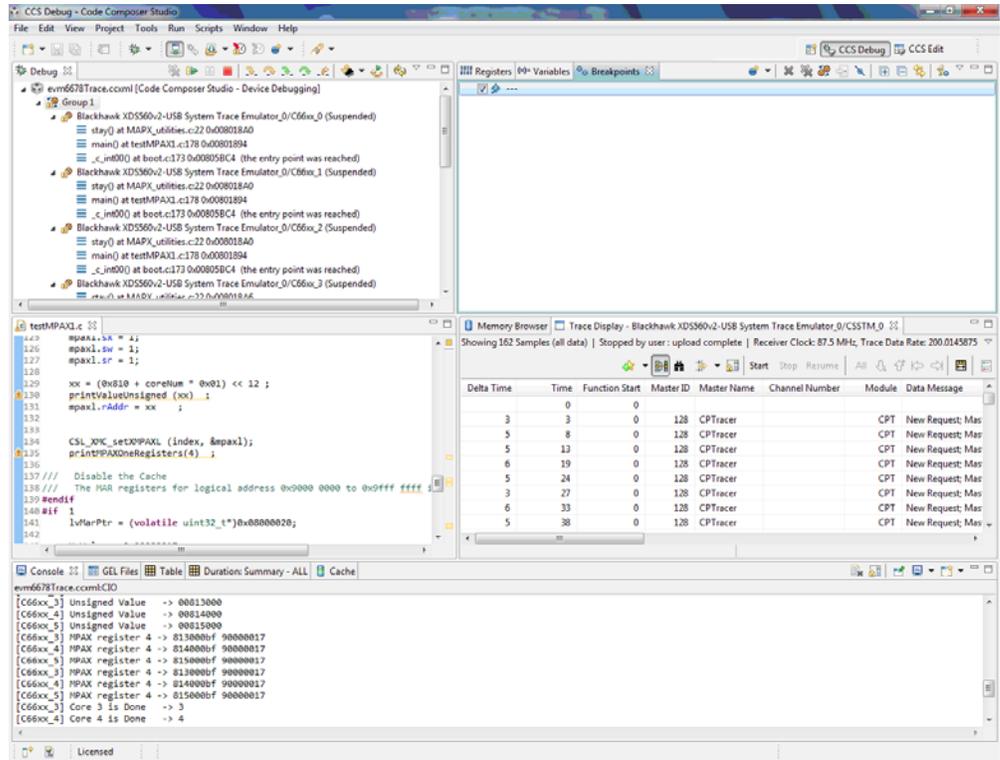


**5**     Double-click on the *Trace Display* window to enlarge it as shown in Figure 7-10. Focus on the central part of the window and notice how each core writes to a different physical address:

**Figure 7-10**      **Trace Display Results: Core Details**

```
Master=GEM0; Write Access; XID=0x0; Addr=0x100
Master=GEM0; Write Access; XID=0x1; Addr=0x100
Master=GEM0; Write Access; XID=0x2; Addr=0x100
Master=GEM0; Write Access; XID=0x3; Addr=0x100
Master=GEM0; Write Access; XID=0x4; Addr=0x100
Master=GEM0; Write Access; XID=0x5; Addr=0x100
Master=GEM0; Write Access; XID=0x6; Addr=0x100
Master=GEM0; Write Access; XID=0x7; Addr=0x100
Master=GEM0; Write Access; XID=0x8; Addr=0x100
Master=GEM0; Write Access; XID=0x9; Addr=0x100
Master=GEM0; Write Access; XID=0xa; Addr=0x100
Master=GEM0; Write Access; XID=0xb; Addr=0x100
Master=GEM0; Write Access; XID=0xc; Addr=0x100
Master=GEM0; Write Access; XID=0xd; Addr=0x100
Master=GEM0; Write Access; XID=0xe; Addr=0x100
Master=GEM0; Write Access; XID=0xf; Addr=0x100
Master=GEM0; Write Access; XID=0x0; Addr=0x100
Master=GEM0; Write Access; XID=0x1; Addr=0x100
Master=GEM0; Write Access; XID=0x2; Addr=0x100
Master=GEM0; Write Access; XID=0x3; Addr=0x100
Master=GEM1; Write Access; XID=0x2; Addr=0x110
Master=GEM1; Write Access; XID=0x3; Addr=0x110
Master=GEM1; Write Access; XID=0x4; Addr=0x110
Master=GEM1; Write Access; XID=0x5; Addr=0x110
Master=GEM1; Write Access; XID=0x6; Addr=0x110
Master=GEM1; Write Access; XID=0x7; Addr=0x110
Master=GEM1; Write Access; XID=0x8; Addr=0x110
Master=GEM1; Write Access; XID=0x9; Addr=0x110
Master=GEM1; Write Access; XID=0xa; Addr=0x110
Master=GEM1; Write Access; XID=0xb; Addr=0x110
Master=GEM1; Write Access; XID=0xc; Addr=0x110
Master=GEM1; Write Access; XID=0xd; Addr=0x110
```

**6** The full Trace Display should appear as shown in Figure 7-11:

**Figure 7-11 Trace Display Results: Full Frame**



**End of Procedure 7-7**

## 7.3.2 Additional Considerations

- 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.

- 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.

- Play a little more with the CCSTM configuration to see what other values can be traced.

- Play with the address bits to see how they can affect the trace values.

- 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).

# STM Library and System Trace

## 8.1 Purpose

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

## 8.2 Project Files

The following files are used in this exercise:

- Initialization.c
- StmMain.c
- System_trace.c
- System_trace.h
- TraceNoRTSC_l2.cmd

## 8.3  Instructions

The list of processes used in this example are as follows:

- Procedure 8-1 "Build and Run the Project"
- Procedure 8-2 "Connect to the EVM"
- Procedure 8-3 "Load the Program and Configure the Trace"
- Procedure 8-4 "Run the Program"

**Note—**This exercise requires a mezzanine card with a trace emulator on the target platform.

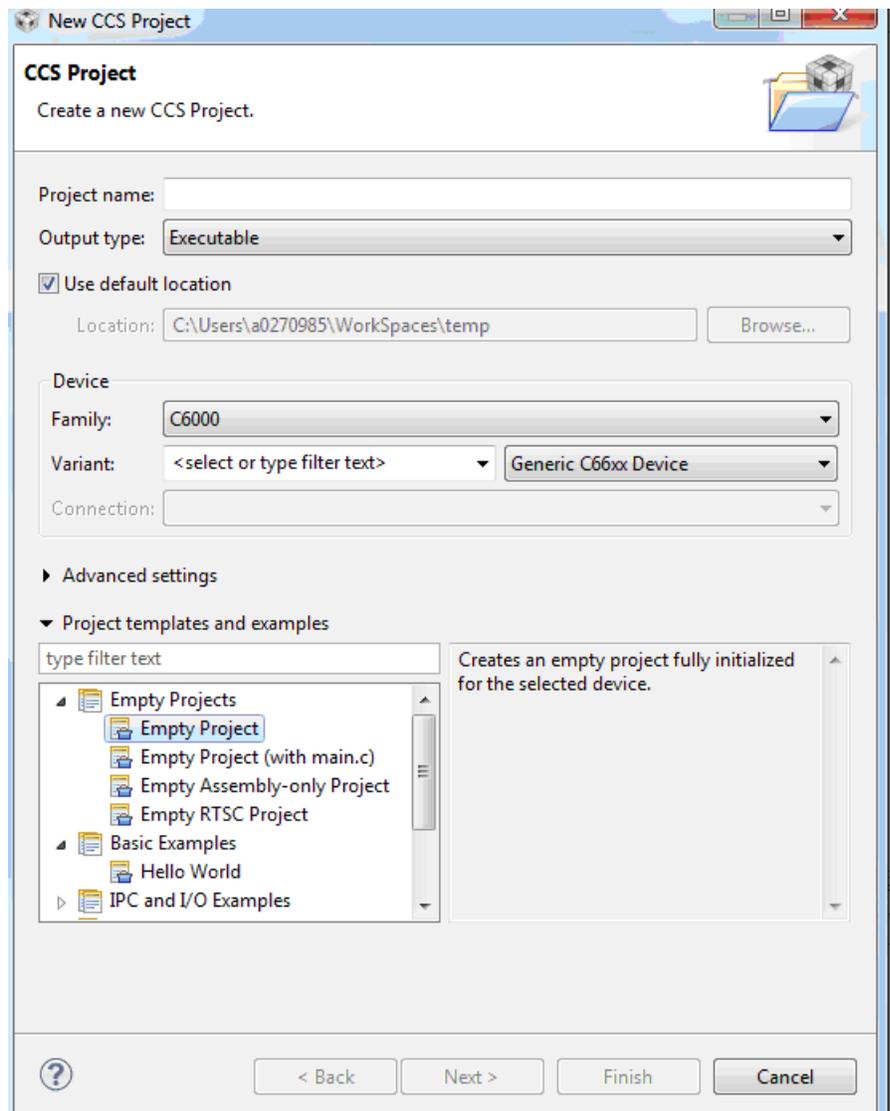**Procedure 8-1        Build and Run the Project**

**Step  -  Action**

**1**     Open CCS.

**2**     Create new project through the CCS menu item

**3**     *File → New → CCS Project.*

**4**     Enter *stm_example* as a *Project Name.*

**5**     Click the check box to Use default location.

**6**     Set the *Family to C6000* and *Variant* to *Generic C66xxx Device* as shown in Figure 8-1:

**Figure 8-1  CCS Project Properties**



**7** Press *Finish* to create the new project.

**8** In the *Project Explorer* view, right-click on the newly-created *stm_example* project, and click on *Add Files…*

**9** 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*.

**10** If a file main.c was generated when you created the new project, remove the main.c file

**11** Examine the code in 'StmMain.c' to understand the code.

**12** Location of the STM library

- 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

- 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'

- After installing the STM library, the directory 'C:\tiMCSDK_3_0_0_11\ctoolslib_1_0_0_3\packages\ti\STMLib' includes multiple sub-directories, as shown in Figure 8-2.

**Figure 8-2**       **STM Lib Directory Structure**
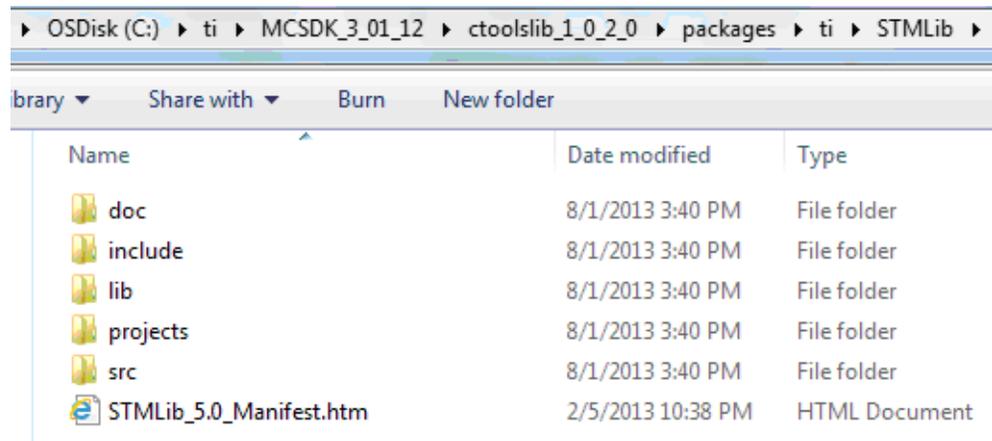


**13** Set the properties for the *Debug* configuration. Right-click on the project. Select *Properties*.

    **13a** 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'

    **13b** Choose *C6000 Compiler* → *Optimization* and set/verify the following properties:

- 'Optimization level = 0'
- 'Optimize for code size = 0'

    **13c** Choose *C6000 Compiler* → *Debug Options* and set/verify the following properties:

- 'Debugging model = Full symbolic debug'

    **13d** 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.

**14** In the linker tab, open file search path.

**15** Add the following path ${STM_ROOT}\lib to the search path and the library stm.c66xx_elf.lib to the library list

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

**17**      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 ****
```
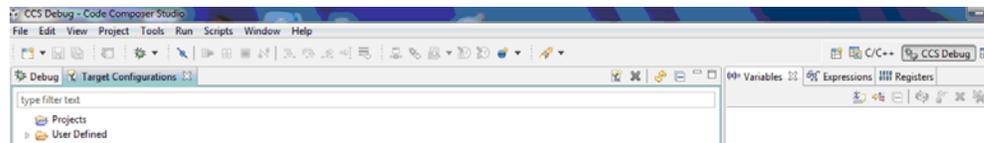
**End of Procedure 8-1**

---

**Procedure 8-2         Connect to the EVM**

**Step  -  Action**

**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* as shown in Figure 8-3.

**Figure 8-3         CCS Target Configurations**



**4**      Use the target configuration that you created in "Preparations" on page 1-1. To create a new
target configuration, follow "Create a New Target in CCS" on page 1-2.

**5**      Launch the target configuration as follows:

    **5a**   Select the target configuration .ccxml file.

    **5b**   Right click and select *Launch Selected Configuration*.

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

    **6a**   Select Core 0 (*C66x_0*)

    **6b**   Right click and select *Connect Target*.

**End of Procedure 8-2**

---

**Procedure 8-3         Load the Program and Configure the Trace**

**Step  -  Action**

**Note—**These instructions are for CCS V5.4. For CCS V5.3 follow the instructions in Chapter 7
"Using MPAX to Define Private Core Memory in DDR" on page 7-1.

**1**      Select Core 0 and load the .out file created earlier in this exercise.

    **1a**   Select the CCS menu option *Run → Load → Load Program*

    **1b**   Click *Browse project*…

    **1c**   Select by unwrapping the *stmSimpleExample→Debug* and click *OK*.

    **1d**   Click *OK* to load the *StmSimpleExample.out* application to the target (Core 0).
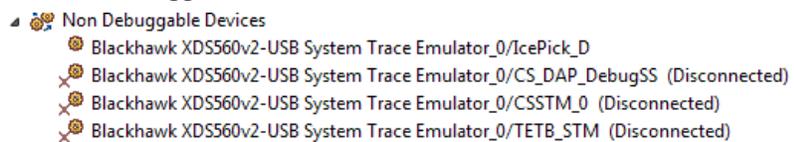
**2**    As shown in Figure 8-4 select the top line in the debug window, right click, and select *Show all cores*.

**Figure 8-4**      **CCS Debug Trace Configuration: Show All Cores**
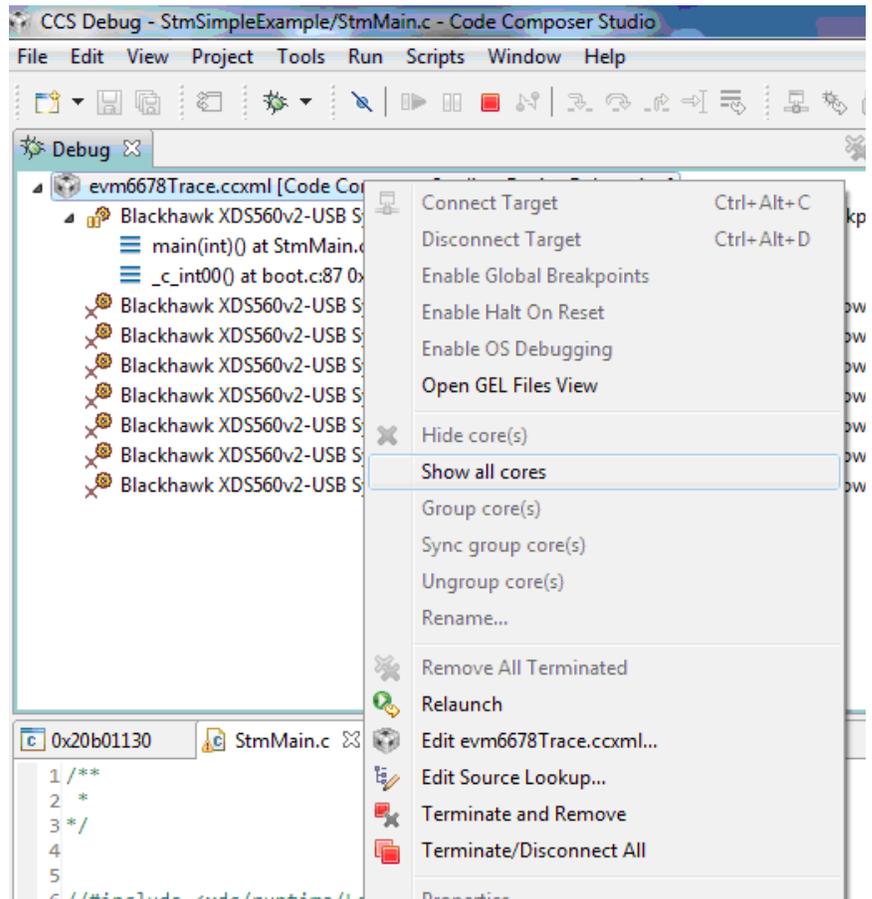


**3**    Non debuggable devices will appear as shown in Figure 8-5.

**Figure 8-5**      **Non Debuggable Devices: Not Connected**



**4**    Select the Non Debuggable device group, right click, and connect Target. You notice that all devices are connected as shown in Figure 8-6.
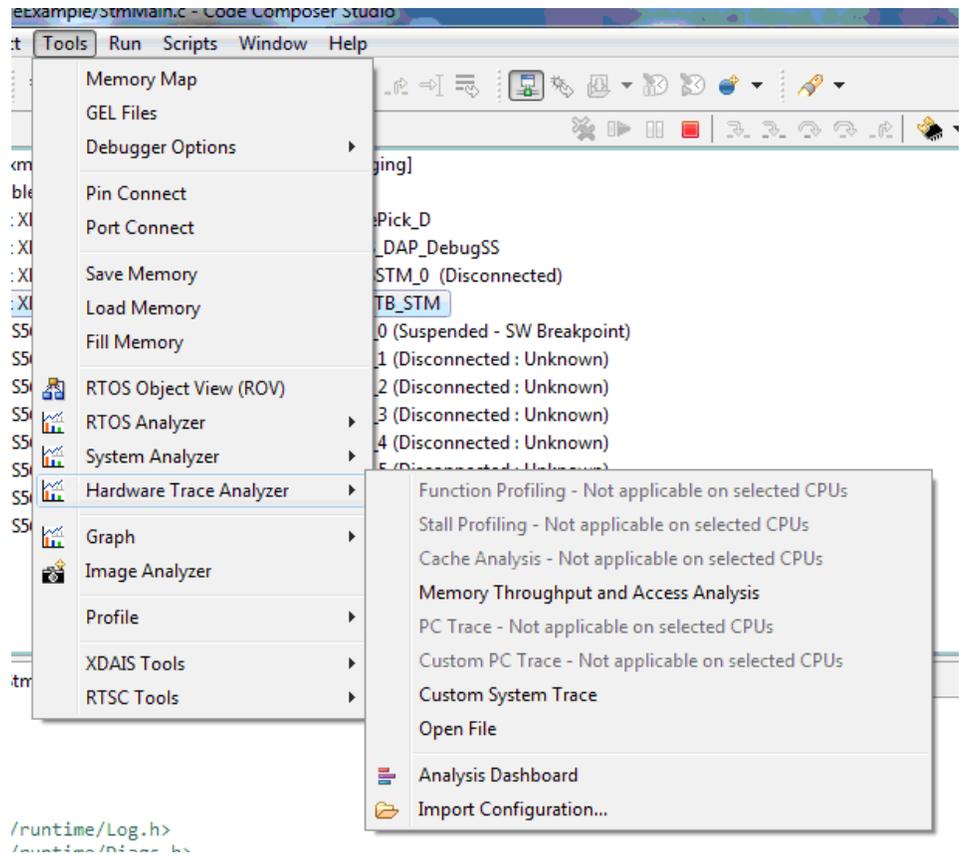
**Figure 8-6**      **Non Debuggable Devices: Connected**

**5** Next, enable the Hardware Trace Analyzer as shown in Figure 8-7
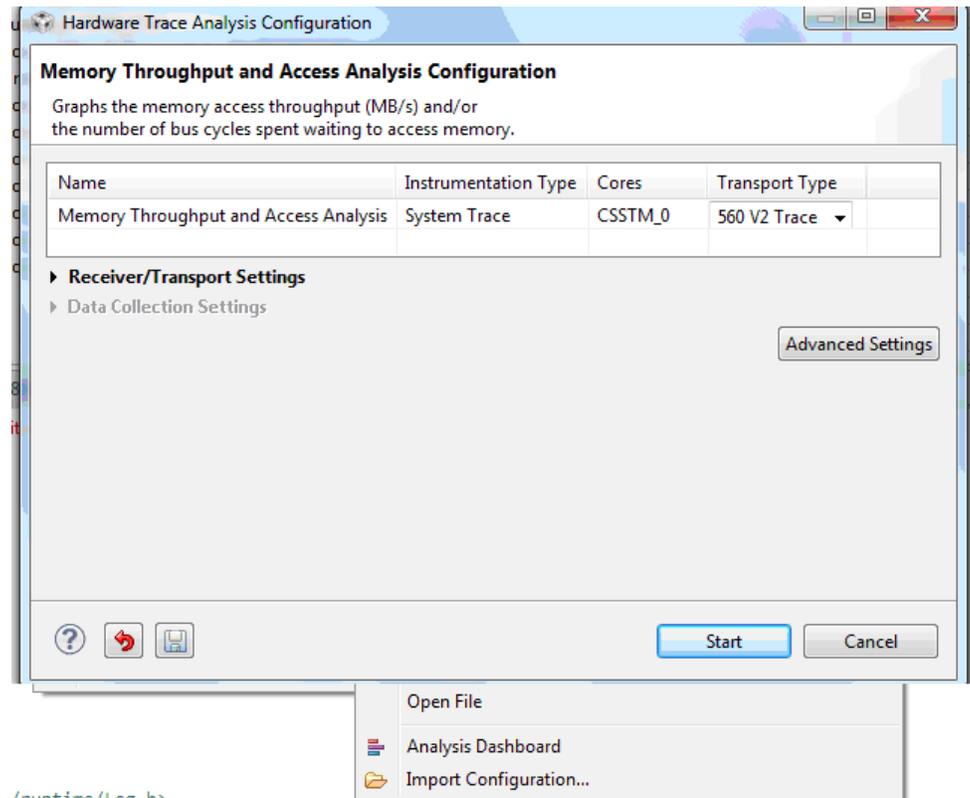
**Note—**These operations are different in CCSV5.3. The screen shots in this example are from CCSv5.4

**Figure 8-7** **Enable Hardware Trace Analyzer**

**6**     The Hardware Trace Analysis Configuration window appears, as shown in Figure 8-8.

**Figure 8-8          Hardware Trace Analysis Configuration**



**7**     Chose *560 V2 Trace* as the transport type, and click *Start*. Shortly, several windows will open. Enlarge the *Trace Viewer CSSTM_0* window.

**End of Procedure 8-3**

---

**Procedure 8-4          Run the Program**

**Step  -  Action**

**1**     Run the application by selecting the CCS menu option *Run → Resume*.

**2**     The *Trace Viewer CSSTM_0* window appears as shown in Figure 8-9.

**Figure 8-9          Trace Viewer CSSTM_0**



**End of Procedure 8-4**

# Revision History

# IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have *not* been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

| Products | | Applications | |
|---|---|---|---|
| Audio | www.ti.com/audio | Automotive and Transportation | www.ti.com/automotive |
| Amplifiers | amplifier.ti.com | Communications and Telecom | www.ti.com/communications |
| Data Converters | dataconverter.ti.com | Computers and Peripherals | www.ti.com/computers |
| DLP® Products | www.dlp.com | Consumer Electronics | www.ti.com/consumer-apps |
| DSP | dsp.ti.com | Energy and Lighting | www.ti.com/energy |
| Clocks and Timers | www.ti.com/clocks | Industrial | www.ti.com/industrial |
| Interface | interface.ti.com | Medical | www.ti.com/medical |
| Logic | logic.ti.com | Security | www.ti.com/security |
| Power Mgmt | power.ti.com | Space, Avionics and Defense | www.ti.com/space-avionics-defense |
| Microcontrollers | microcontroller.ti.com | Video and Imaging | www.ti.com/video |
| RFID | www.ti-rfid.com | | |
| OMAP Applications Processors | www.ti.com/omap | **TI E2E Community** | e2e.ti.com |
| Wireless Connectivity | www.ti.com/wirelessconnectivity | | |