C55XCSL-LOWPOWER-3.04.00.02

Installation Guide

Texas Instruments

May - 2014

Table of Contents

List of Figures

List of Tables

1. Platforms and Examples

CSL has been tested on Code Composer Studio™ - Ver5.5. The following platforms have been used for testing:

- C5517 EVM
- C5515 EVM
- C5535 eZdsp

This version of CSL will no longer support CCSv3.3 or CCSv4. If CCSv3.3 support is required, refer to CSL version 3.00. If CCSv4 support is required, refer to CSL version 3.01.

This CSL release package contains the following modules. Besides the related CSL functions, each module also contains one or more "example" mini-applications that use and illustrate basic capabilities of the related CSL. These "examples" are listed under each module below.

- **DAT** – Data Buffer Operations **--** creating, filling, copying memory buffers
  - *CSL_DAT_Example*
- **DMA** – DMA Operations **--** polled and interrupt-driven modes, even Ping-Pong buffers
  - *CSL_DMA_IntcExample*
  - *CSL_DMA_PingPongExample*
  - *CSL_DMA_PollExample*
  - *CSL_DMA_StopAPIExample*
  - *CSL_DMA_SDRAM_Example*
- **EMIF** – Control of EMIF for Interfacing with NAND, NOR and SDRAM(C5515 and C5517)
  - *CSL_EMIF_NAND_PollExample*
  - *CSL_EMIF_NAND_DmaExample*
  - *CSL_EMIF_NAND_IntrExample*
  - *CSL_EMIF_NAND_DmaWordSwapExample*
  - *CSL_EMIF_SDRAM_PollExample*
  - *CSL_EMIF_NOR_PollExample*
- **GPIO** – Control of General Purpose IOs
  - *CSL_GPIO_InputPinExample*
  - *CSL_GPIO_OutputPinExample*
- **GPT** – Control of General Purpose Timers
  - *CSL_GPTExample*
  - *CSL_GPTNestedIntrNmiExample (C5517 only)*
  - *CSL_GPT_MultiInstances_Example*
- **I2C** – Control of I2C Ports
  - *CSL_I2C_DmaExample (C5515 and C5517 only)*
  - *CSL_I2C_DmaWordSwapExample (C5515 and C5517 only)*
  - *CSL_I2C_IntcExample (C5515 and C5517 only)*
  - *CSL_I2C_LoopbackExample*
  - *CSL_I2C_PollExample (C5515 and C5517 only)*
  - *CSL_I2C_CodecTestExample*

- **I2S** – Control of I2S Ports
    - *CSL_I2S_DMAExample*
    - *CSL_I2S_INTCExample*
    - *CSL_I2S_PollExamples*
    - *CSL_I2S_IdleLoop*
    - *CSL_I2S_DMA_MultipleInstanceExample*

- **INTC** – Interrupt Control Functions
    - *CSL_INTC_Example*

- **LCD** - LCD Controller Setup & Control – initialize, write, and read LCD display via controller (C5505/15 only)
    - *CSL_LCDC_262kColorModeExample*
    - *CSL_LCDC_65kColorModeExample*
    - *CSL_LCDC_DiagramExample*
    - *CSL_LCDC_DmaIntcExample*
    - *CSL_LCDC_DmaPolledExample*
    - *CSL_LCDC_TextDisplayExample*

- **McBSP** – Control of the Multichannel Buffered Serial Port (C5517 only)
    - *CSL_McBSP_InternalLB*
    - *CSL_McBSP_Master_AIC3204_48kbps_POLL*
    - *CSL_McBSP_Master_AIC3204_48kbps_DMA*

- **McSPI** – Control of Multichannel Serial Peripheral Interface (C5517 only)
    - *CSL_McSPI_MasterFullDuplexTest_Flash*
    - *CSL_McSPI_MasterFullDuplex_Example*
    - *CSL_McSPIMaster_MSP430Slave_FullDuplex*

- **MEMORY** – Basic Memory Control and Modes
    - *CSL_MEMORY_DARAM_PartialRetentionExample*
    - *CSL_MEMORY_DARAM_RetentionExample*
    - *CSL_MEMORY_SARAM_PartialRetentionExample*
    - *CSL_MEMORY_SARAM_RetentionExample*

- **MMC_SD** – Multi Media Card & Secure Data Card Interface Control
    - *CSL_MMCSD_MmcCardExample*
    - *CSL_MMCSD_SdCardExample*
    - *CSL_MMCSD_dmaExample*
    - *CSL_MMCSD_intrExample*
    - *CSL_MMCSD_SdCardFSDirExample*
    - *CSL_MMCSD_SdCardFSExample*
    - *CSL_MMCSD_SdCardFSExtExample*
    - *CSL_MMCSD_extAPIs_Example*
    - *CSL_MMCSD_SdCard_MultiInstance_Example*

- **PLL** – PLL Initialization and Control
    - *CSL_PLL_Example*
    - *CSL_PLL_LDO_Setting_Example*

- **RTC** – Real Time Clock Control
  - *CSL_RTC_Compensation_Example*
  - *CSL_RTC_Example*

- **SAR** – Initialization and Control of SAR "A to D" Inputs (C5515 and C5517 only)
  - *CSL_SAR_DmaExample*
  - *CSL_SAR_IntcExample*
  - *CSL_SAR_PollExample*

- **SPI** – Initialization and Control of SPI Serial Ports
  - *CSL_SPI_Example*

- **UART** – Initialization and Control of UART Serial Ports
  - *CSL_UART_IntExample*
  - *CSL_UART_dmaExample*
  - *CSL_UART_pollExample*

- **UHPI** – Control of the Universal Host Port Interface (C5517 only)
  - *CSL_UHPI_Example*
  - *CSL_UHPI_MSP430_ Example*

- **USB** – USB Port Control – Basic USB operations plus Mass Storage Class (MSC), Human Interface Device class (HID) and CDC ACM support
  - *CSL_USB_CdcExample*
  - *CSL_USB_HidExample*
  - *CSL_USB_DmaExample*
  - *CSL_USB_IntcExample*
  - *CSL_USB_MSC_dmaExamplet*
  - *CSL_USB_MSC_pollExample*
  - *CSL_USB_MSC_fullSpeedExample*
  - *CSL_USB_PollExample*

- **WDTIM** – Watch-Dog Timer Control
  - *CSL_WDT_Example*

2. What is Included in the CSL Package

- Source code of all CSL Modules (as listed above in "Purpose of Release"). Source code is available in the path c55_csl_3.04\c55xx_csl\src and c55_csl_3.04\c55xx_csl\inc.

- Sample applications, or "Examples," which demonstrate basic CSL module functionalities.

- CSL API reference documentation. This documentation is available in the path c55_csl_3.04\c55xx_csl\doc\c55xx_csl_api_html\. To begin, open file index.html with a browser.

- Example application reference documentation. This documentation is available in the path c55_csl_3.04\c55xx_csl\doc\c55xx_csl_examples_html\. To begin, open file index.html with a browser.

3. Installation Guide

Important Notes:

3.1 For Running the Projects on C5504/05/14/15 DSP:

- Make sure that #define CHIP_C5517 in the file c55xx_csl\inc\csl_general.h is **commented out** (e.g., with a beginning "//"). With this line commented out, enables the macro for C5515 device.

- Select the correct platform (C5515_EVM or C5515_EZDSP) in csl_general.h. If using the C5515 eZdsp, make sure that #define C5515_EVM (on file c55xx_csl\inc\csl_general.h is **commented out** and **NOT comment out** the #define C5515_EZDSP. If using the C5535 eZdsp, make sure that #define C5515_EVM (on file c55xx_csl\inc\csl_general.h is **NOT commented out** so the condition will define C5515_EVM.

- To run CCSv5 examples, change the emulator target configuration file (*.ccxml) to use right c55xx_csl\build\C55<XX>.gel as your source of gel commands.
  - Open CCSv5
  - Open Target Configuration for desired target board.
    - Respective CCXML file can be imported from the below path.
      <CSL_INSTALL_DIR>\c55_csl_3.04\c55xx_csl\ ccs_v5.0_examples \Gel_ccxml_Files\ C55<XX>.xxcml
  - Click Target Configuration under the Advanced tab.
  - Click on C55xx under the All Connections window.
  - Under CPU Properties click Browse for the initialization script.
  - Select desired Gel file:
    <CSL_INSTALL_DIR>\c55_csl_3.04\c55xx_csl\ ccs_v5.0_examples \Gel_ccxml_Files\C55<XX>.gel.
  - Save updated configuration.

3.2 For Running the Projects on C5517 DSP:

- Make sure that #define CHIP_C5517 near the top of file c55_csl_3.04\c55xx_csl\inc\csl_general.h. is **NOT commented out** (e.g., with a beginning "//"). The only platform available is the C5517_EVM, and that should be enabled in csl_general.h.

- To run CCSv5 examples, change the emulator target configuration file (*.ccxml) to use c55_csl_3.04\ c55xx_csl \build\c5517.gel as your source of gel commands.
  - Open CCSv5
  - Open Target Configuration for desired target board.
    - Respective CCXML file can be imported from the below path.
      <CSL_INSTALL_DIR>\c55_csl_3.04\c55xx_csl\ ccs_v5.0_examples \Gel_ccxml_Files\ C55<XX>.xxcml
  - Click Target Configuration under the Advanced tab.
  - Click on C55xx under the All Connections window.
  - Under CPU Properties click Browse for the initialization script.
  - Select desired Gel file:
    <CSL_INSTALL_DIR>\c55_csl_3.04\c55xx_csl\ccs_v5.0_examples \Gel_ccxml_Files\C55<XX>.gel.
  - Save updated configuration.

3.3 For Running the Projects on C5535 DSP:

- Make sure that #define CHIP_C5517 near the top of file c55_csl_3.04\c55xx_csl\inc\csl_general.h. is **commented out** (e.g., with a beginning "//").

- Make sure that #define C5515_EVM and #define C5515_EZDSP (on file c55_csl_3.04\c55xx_csl\inc\csl_general.h) are **commented out** so the condition will define C5535_EZDSP.

- To run CCSv5 examples, change the emulator target configuration file (*.ccxml) to use c55_csl_3.04\c55xx_csl\build\c5505evm_pg20.gel as your source of gel commands.
  - Open CCSv5
  - Open Target Configuration for desired target board.
    - Respective CCXML file can be imported from the below path.
      <CSL_INSTALL_DIR>\c55_csl_3.04\ c55xx_csl\ccs_v5.0_examples \Gel_ccxml_Files\ C55<XX>.xxcml
  - Click Target Configuration under the Advanced tab.
  - Click on C55xx under the All Connections window.
  - Under CPU Properties click Browse for the initialization script.
  - Select desired Gel file:
    <CSL_INSTALL_DIR>\c55_csl_3.04\c55xx_csl\ccs_v5.0_examples \Gel_ccxml_Files\C55<XX>.gel.
  - Save updated configuration.

3.4 Building and Running the CCS v5 Projects

- For running CCS v5 example projects connect your Target, via a suitable emulator such as the "XDS510", "XDS100" or the EVM's "Onboard" emulator, to CCS. To use the Onboard emulator, connect a USB A/B cable from your host PC's USB port to port 'EMU USB'(J201) on the EVM. As released, all CCS v5 projects include at least an Onboard_Emulator.ccxml file for using the Onboard emulator. (Other emulators, such as the XDS510, can also be used as well but each requires a *.ccxml file specific to that emulator.).

- You can also run the CCS v5 examples on the C5515 eZdsp USB Stick which has the XDS-100 emulator built in. CCS 5.x supports this emulator. You can generate the ccxml for XDS-100 easily. In CCS 5.x, select Target → New Target Configuration …, then select "Texas Instruments XDS100v1 USB Emulator" for Connection C5515 for Device. Pick "USBSTK5505" or "USBSTK5515" depending on the eZdsp USB Stick you are using.

- Start the CCS5.x IDE and select the c55_csl_3.04 folder as the CCS work space while opening then CCS v5 application.

  *****NOTE*** - You cannot have CCS v4 and v5 workspaces on the same folder. If you want to use the main folder as a workspace for v5 after using it for v4, delete the .metadata folder.**

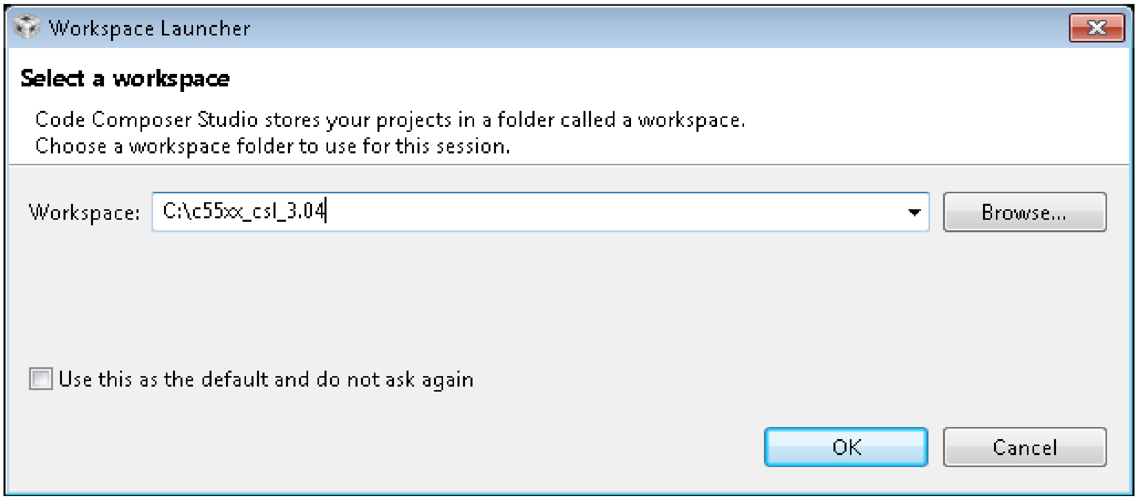

Figure 3-1 Selecting the CCS v5 Workspace

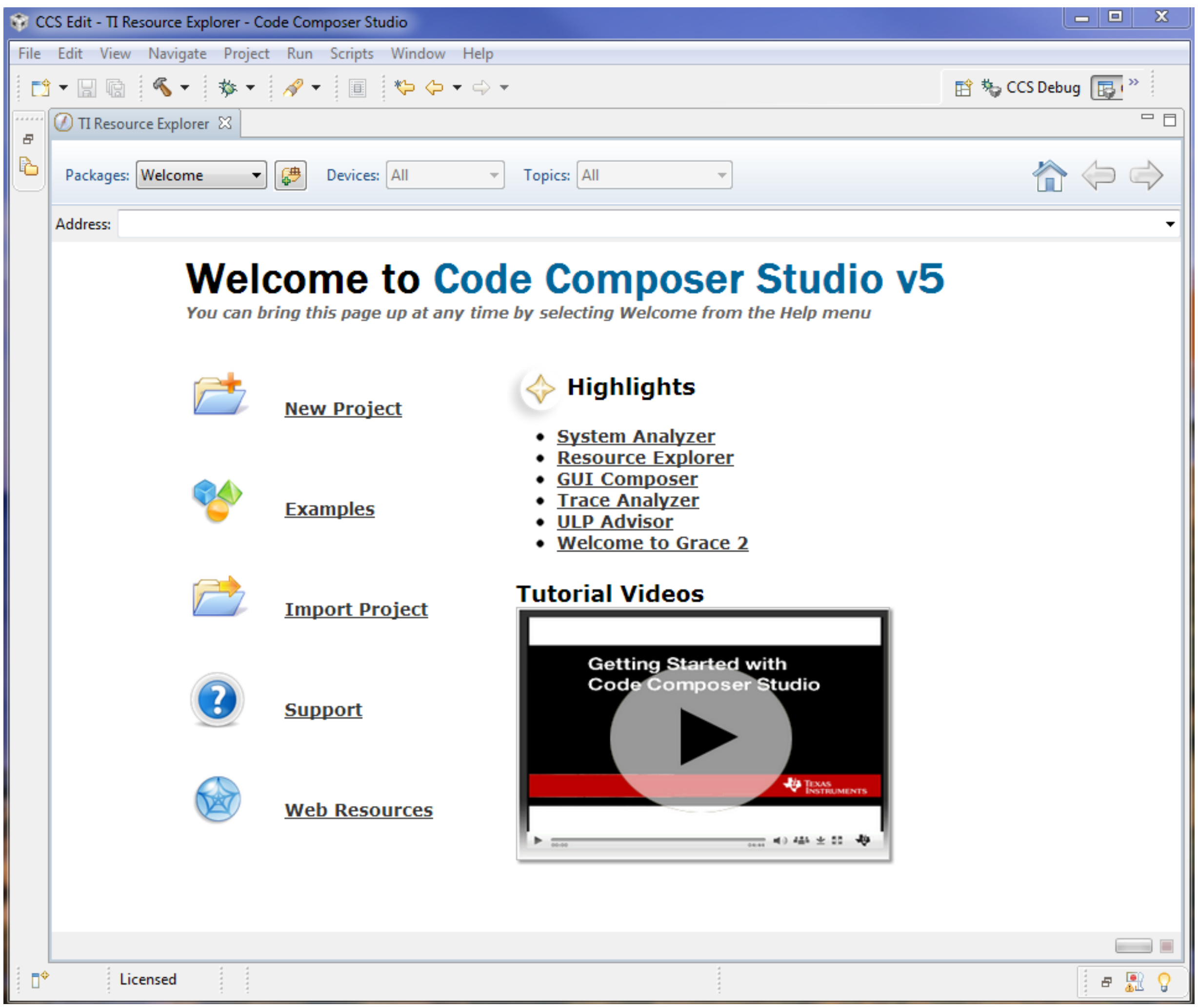

Figure 3-2 Starting the CCS v5 Workbench

- Select the menu **Project→Import Existing CCS/CCE Eclipse Project…**. Browse for the c55_csl_3.04/ c55xx_csl \ccs_v5.0_examples folder and click ok. All the CCS v5 projects will be displayed in the list of projects. Click on "Select All" to select all the projects or manually select the required projects by checking the checkboxes. Leave the "Copy projects into workspace" box unchecked. Click on "Finish". Projects will be loaded to the CCS.

- **To keep the dependency among projects, we highly recommend importing all the projects in one time (by default).**

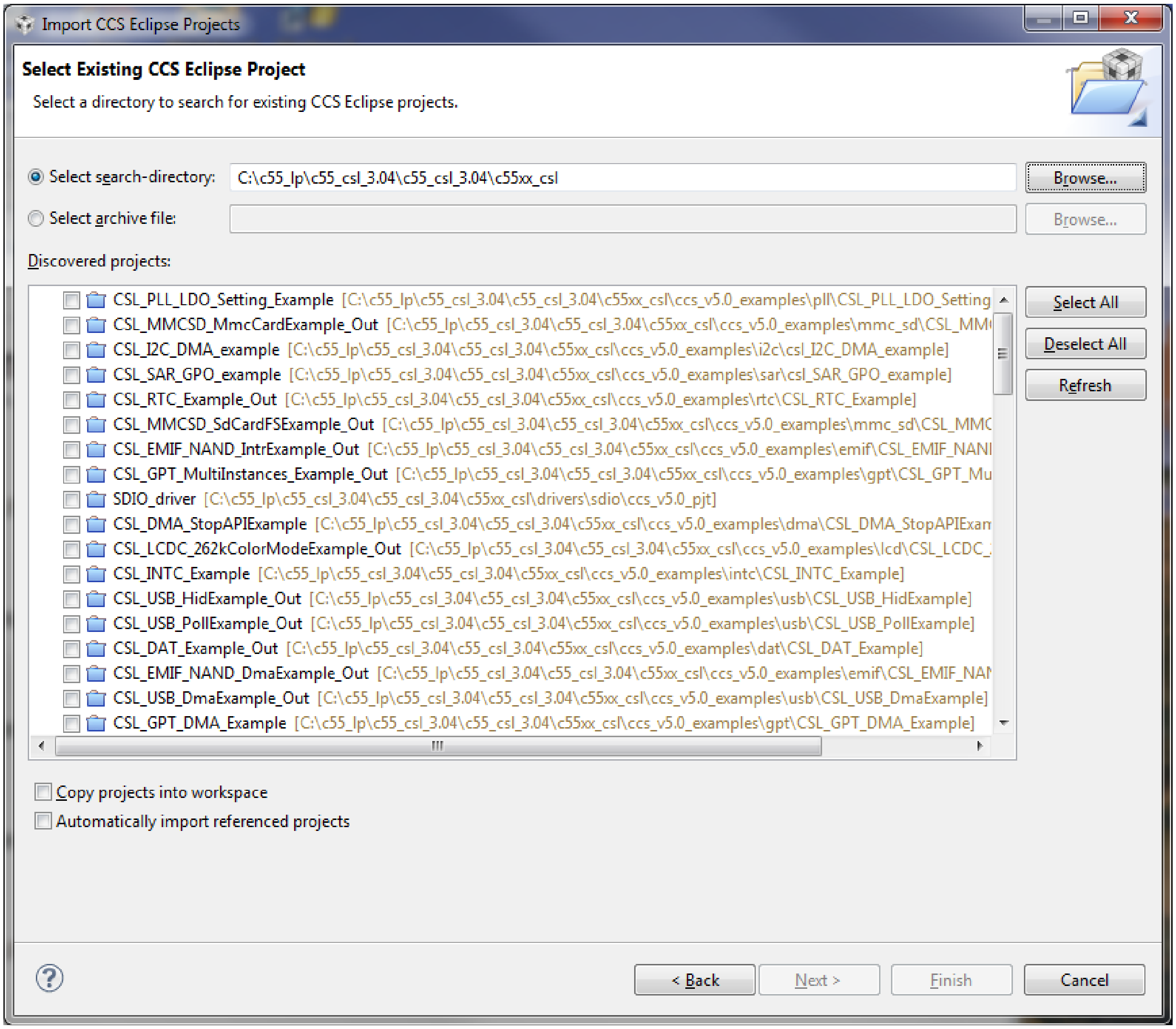

Figure 3-3 Browsing for the CCS v5 Projects

- Click on the project that you want to test and it will become the active project

- Right click on your active project and set the **Active Build Configuration** as either **Debug** or **Release** from **Build Configurations -> Set Active**.
(CCS v5.5 **-** supports building programs in two distinct modes. **Debug** mode is used for building programs with little/no compiler optimization enabled. Resultant executables still retain full symbolic debugging

information on variables and also linkage information between most points in the executable and the line(s) of source code from which each came. This information generally makes the code easier to debug but also makes it bigger and slower. **Release** mode, on the other hand, is used for building programs with high degrees of compiler optimization enabled. This eliminates much of the debug-supportive information described above from the executable but makes it smaller and faster.)

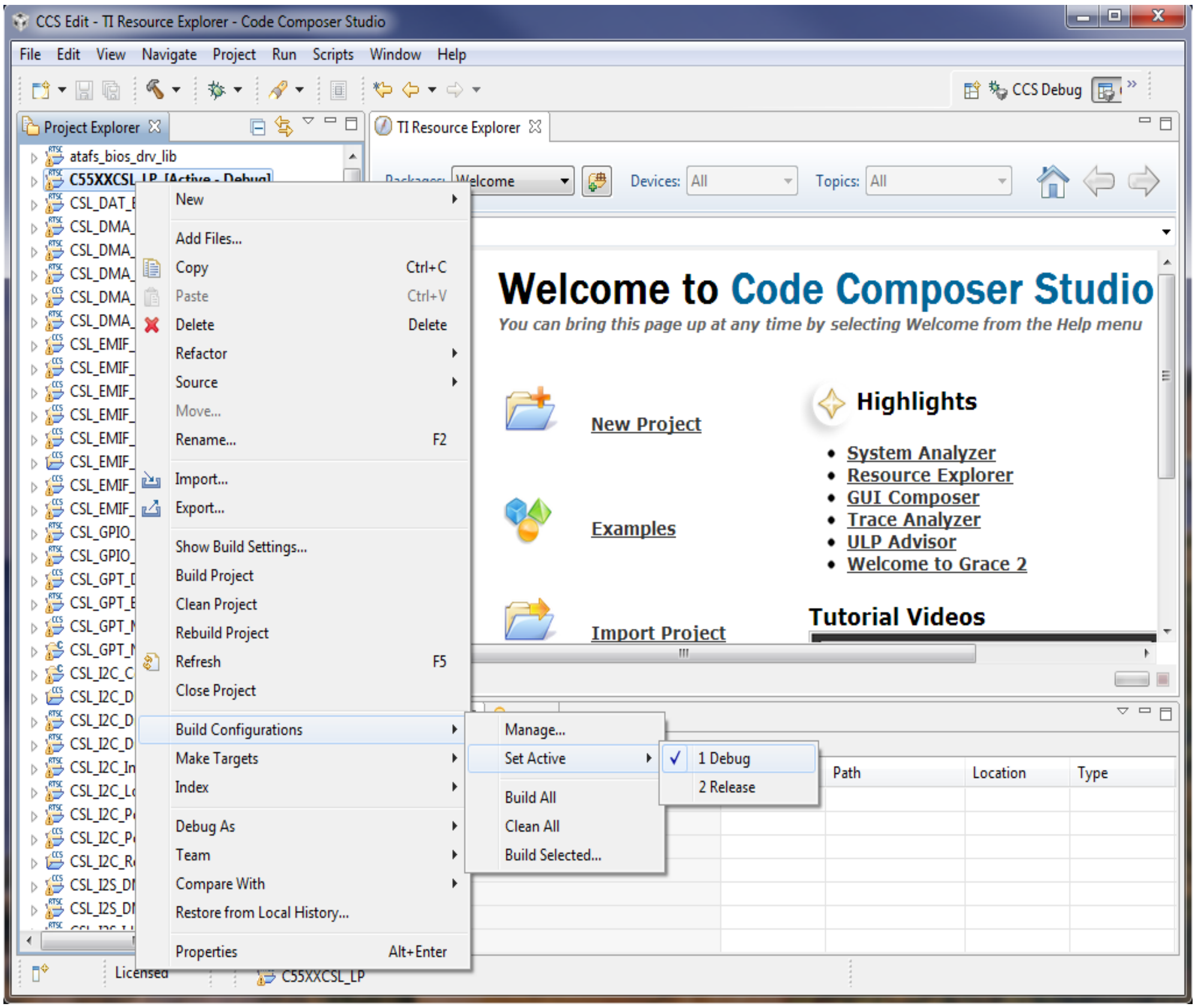

Figure 3-4 Setting Active CCS v5 Build Configuration

- Select the menu **Run→Debug**. Project will be built (if needed) and debugger will be opened.

  (The project will be (re)built here only if needed, as when a piece of involved source code has changed. If a (re)build does occur, you can monitor its progress in a special console sub-window that will open during the build. Any build errors will be reported there for your information. If the build completes without any issues, the program will be loaded to the target with the Debug view opened and the debugger ready to use.

  (Note that the menu **Run→Debug** recommended above includes an automatic project pre/re-build if needed before debug can commence. If you prefer, you can instead build the project in a separate step first by using menu **Project/Build Project**.)

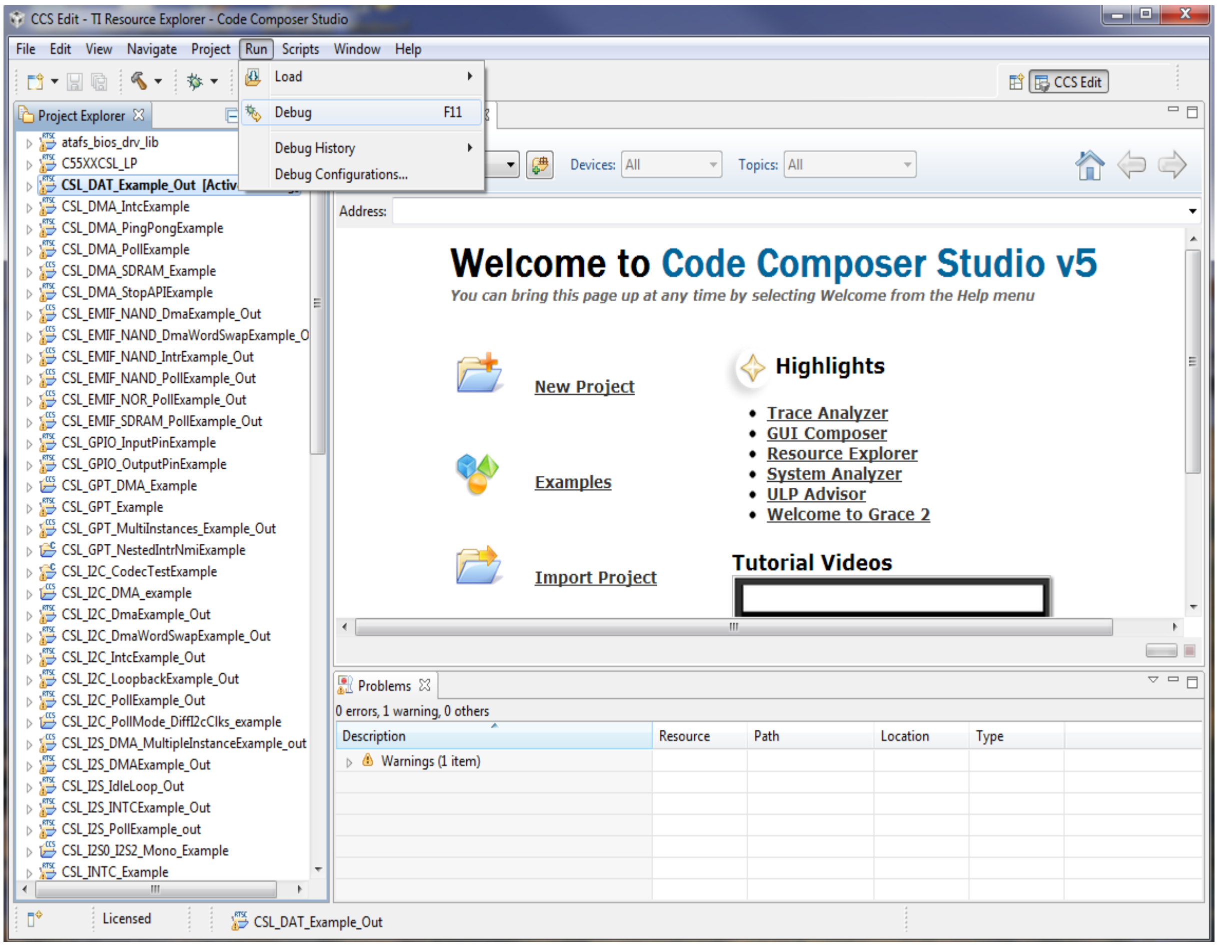

Figure 3-5 Debugging the Active CCS v5 Project

- If using **C5517** → Select **Scripts→ C5517EVM_Configuration** to set the PLL to the desired frequency.

- Similarly If using **C5515/35** → Select **Scripts→ C5515EVM_Configuration** to set the PLL to the desired frequency.
  ******NOTE – You are only able to see the Scripts menu if using select gel files******

- Select menu **Run/Resume** to run the project.

3.5 Building and Running some of the example (McSPI & uHPI) using MSP430.

Some of the McSPI and uHPI example code requires external Host / Master /Slave and for this purpose MSP430 will be used.

Below information illustrates on how to run McSPI code which involves MSP430.

Let's take an example of - *CSL_McSPI_MasterFullDuplex_Example*:

- To run the above example code needs 2 emulators – For C5517 one can use On board emulator and for MSP430 side use MSP430 supported emulator (eg MSP-FET430UIF).

- Respective CCXML and gel file exists in " Gel_ccxml_Files" folder . Following is the Path for this folder - <CSL_PATH> \c55xx_csl\ccs_v5.0_examples\Gel_ccxml_Files – use the respective ccxml & gel files.

- The programs need to be loaded using 2 different CCS sessions.

- The project to load from C5517 will be – "*CSL_McSPI_MasterFullDuplex_Example*" and from MSP430 side – "*CSL_McSPIMaster_MSP430Slave_FullDuplex*" (Projects to be loaded from MSP430 will have MSP430 mentioned in it.

- Then follow the read me that exists in each of these projects. Generally the slave code needs to be run first followed by Master code. In the above example MSP430 is run first and then C5517 code.

Note: While installing CCS, MSP430 device needs to be selected; otherwise the MSP430 related projects will fail to load.

4. Target Requirements for Testing

One important target specific requirement is to use a CSL build that is compatible with your silicon.

For C5517 silicon, ensure #define CHIP_C5517 is **uncommented** near the top of file c55_csl_3.04\c55xx_csl\inc\csl_general.h.

On the other hand, **for C5515/35** silicon, make sure that #define CHIP_C5517 near the top of file c55_csl_3.04\c55xx_csl\inc\csl_general.h. **is commented out** (e.g., with a beginning "//"). With this line commented out, the #ifndef logic in csl_general.h #define's the macro CHIP_C5515 instead. This, in turn, causes your build, by default, to be tailored for C5515/35 silicon.

Since we have different platforms for the C5515/35/17, another important platform specific requirement is to use a CSL build that is compatible with your platform.

- **For C5515 EVM and C5535 eZdsp**, **comment out** #define C5517_EVM in Part 3 of file c55_csl_3.04\c55xx_csl\inc\csl_general.h.

- **For C5515 eZdsp USB Stick**, make sure that #define C5515_EVM and the #define C5517_EVM in Part 3 of file c55_csl_3.04\c55xx_csl\inc\csl_general.h. **<u>are commented out</u>** (e.g., with a beginning "//"). With these lines commented out, the #ifndef logic in csl_general.h #define's the macro C5515_EZDSP instead. This, in turn, causes your build, by default, to be tailored for C5515 eZdsp USB Stick.

- **For C5535 eZdsp**, make sure that #define C5515_EVM, #define C5515_EZDSP and the #define C5517_EVM in Part 3 of file c55_csl_3.04\c55xx_csl\inc\csl_general.h. **<u>are commented out</u>** (e.g., with a beginning "//"). With these lines commented out, the #ifndef logic in csl_general.h #define's the macro C5535_EZDSP instead. This, in turn, causes your build, by default, to be tailored for C5535 eZdsp.

Additionally, it is recommended that you use versions of code gen tools and BIOS that are compatible with those used by us to test the CSL and Examples in this release. In general, we recommend that you use the following, or newer, versions. (If the comments in a particular example cite special tool version requirements, abide by those.)

- CCS Version 5.5.0.00077 using code generation tool v4.3.9 or later and DSP BIOS 5.41.02.14. There is some bug in v4.3.x prior to 4.3.9. Even 4.3.6 has some problem with the MDK Pulse Ox application. The EVM's "Onboard" Emulator is used to interact with the target to load and run the CCS v5 projects thereon.

5. CSL Overview

This section introduces the Chip Support Library, describes its architecture, and provides an overview of the collection of functions, macros, and constants that help you program DSP peripherals.

5.1 Introduction to CSL

CSL is a collection of functions, macros, and symbols used to configure and control on-chip peripherals. It is fully scalable and it does not require the use of DSP/BIOS components to operate.

5.1.1 Benefits of CSL

The benefits of CSL include peripheral ease of use, shortened development time, portability, hardware abstraction, and a level of standardization and compatibility among devices. CSL can be viewed as offering two fundamental levels of peripheral interface to users, a more abstract function-level layer 1 offering a fairly high level of interfaces and protocols, and a lower hardware-detailed register-level layer 2 offering direct symbolic access to all hardware control registers. These two layers are described below.

1. Function Level CSL -- Higher level interfaces and protocols

    - Standard Protocol to Program Peripherals: CSL provides developers with a standard protocol to program on-chip peripherals. This protocol includes data types and macros to define peripheral configurations, and functions to implement various operations of each peripheral.
    - Basic Resource Management: Basic resource management is provided through the use of open and close functions for many of the peripherals. This is especially helpful for peripherals that support multiple channels.

2. Register Level CSL -- Lower level register-manipulation interface

    - Symbolic Peripheral Descriptions: A complete symbolic detailed description of all peripheral registers and register fields has been created. It is suggested that developers use the higher level protocols (of CSL layers b. and c.), as these are less device-specific, thus making it easier to migrate code to newer versions of DSPs.

5.1.2 CSL Architecture

CSL consists of modules that are built and archived into a library file. Each peripheral is covered by a single module while additional modules provide general programming support. This architecture allows for future expansion because new modules can be added as new peripherals emerge.

Users have two levels of access to peripherals using CSL, register level access and function level access. All function CSL files have a name of the form csl_PER.c where PER is a placeholder for the specific peripheral. In a similar fashion, all register level files have a name of the form cslr_PER.h. The function level of CSL is implemented based on register level CSL. Users can use either level of CSL to build their applications. The following Figure 5-1 shows the architecture of CSL and its role in interfacing an application to the DSP hardware on which it executes.

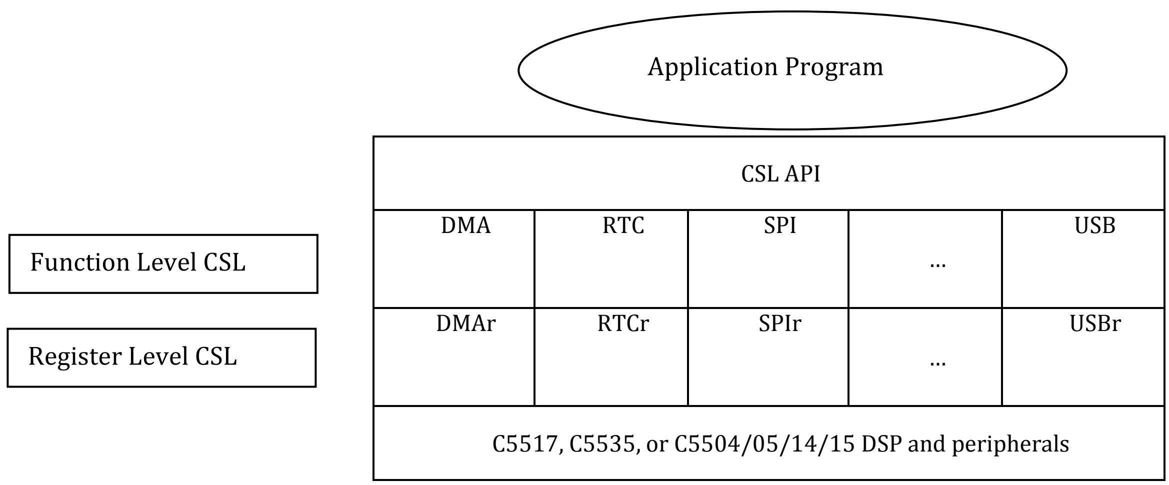

Figure 5-1 CSL Architecture

Table 5-1 lists the key modules and related interface defining files within CSL.

Table 5-1 CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File
DAT	A data copy/fill module based on the DMA C5505	csl_dat.h
DMA	DMA peripheral	csl_dma.h
GPIO	General Purpose I/O	csl_gpio.h
GPT	32-bit General purpose timer	csl_gpt.h
I2C	I2C peripheral	csl_i2c.h
I2S	I2S peripheral	csl_i2s.h
INTC	Interrupt Controller	csl_intc.h
LCDC	LCD Controller	csl_lcdc.h
McBSP	McBSP peripheral	csl_mcbsp.h
McSPI	McSPI peripheral	csl_mcspi.h
MEM	Enable or Disable the Memory Retention Mode for SARAM and DARAM	csl_mem.h

MMC/SD	MMC/SD Controller	csl_mmcsd.h
MMC/SD	ATAFS Interface to MMC/SD driver	csl_mmcsd_at aIf.h
NAND	NAND flash	csl_nand.h
PLL	PLL	csl_pll.h
RTC	Real-time clock	csl_rtc.h
SAR	10 bit SAR ADC	csl_sar.h
SDIO	Secure Data I/O driver	csl_sdio.h
SPI	SPI	csl_spi.h
SYS	System	csl_sysctrl.h
UART	UART	csl_uart.h
UHPI	UHPI	csl_uhpi.h
USB	USB core driver	csl_usb.h
USB MSC	USB MSC driver	csl_msc.h
USB Audio	USB Audio driver	csl_audioClass.h
WDT	Watch Dog Timer	csl_wdt.h

5.2 Naming Conventions

The following conventions are used when naming CSL functions, macros, and data types. Note that PER is used as a placeholder for any of the specific module / peripheral names from Table 5-1 above.

Table 5-2 CSL Naming Conventions

Object Type	Naming Convention
Function	PER_funcName()
Variable	PER_varName
Macro	PER_MACRO_NAME
Typedef	PER_Typename
Function Argument	funcArg
Structure Member	memberName

- All functions, macros, and data types start with PER_ (where PER is the peripheral module name listed in Table 5-1) in uppercase letters.
- Function names use all lowercase letters. Uppercase letters are used only if the function name consists of two separate words. For example, PER_getConfig().
- Macro names use all uppercase letters; for example, DMA_DMPREC_RMK.
- Data types start with an uppercase letter followed by lowercase letters, e.g., DMA_Handle.

5.3 CSL Data Types

CSL provides its own set of data types that all begin with an uppercase letter. Table 5-3 lists CSL data types as defined in the file …/c55xx_csl/inc/tistdtypes.h.

Note: The minimum data unit in C5517, C5504/05/14/15 is 16-bit word, therefore char and bool type will be allocated a 16-bit word (short). It does not support byte operation natively.

Table 5-3 CSL Data Types

Data Type	Description
bool	*short*
int	*short*
Char	*short*
ptr	*void **
String	*char **
Uint32	*unsigned long*
Uint16	*unsigned short*
Uint8	*unsigned char*
Int32	*long*
Int16	*short*
Int8	*char*

5.4 CSL Functions

Table 5-4 provides a description of the most common CSL functions where PER indicates a peripheral module as listed in Table 5-1. Note that not all of the peripheral functions listed in the table are available for all modules / peripherals. Furthermore, some peripheral modules may offer additional peripheral-specific functions not listed in the table. Refer to the documentation in path c55xx_csl\doc\csl_api_html\index.html for a list of CSL functions offered for each module / peripheral.

The following conventions are used in Table 5-4:

- Italics indicate variable names.
- Brackets [...] indicate optional parameters.
  - *[handle]* is required only for handle-based peripherals: such as DAT, DMA, SPI, MMC/SD and USB.

CSL offers two fundamental ways to program peripherals

- Directly write to hardware control registers using the lower CSLR layer
- Use the more abstract functions (Table 5-4) of the higher CSL layer. For example, you can use PER_config() plus any other needed peripheral specific functions. See section 5.4.1 for more detail.

Table 5-4 Generic CSL Functions

Function	Description
PER_**init**(*void*)	This function initializes and activates the SPI module. It has to be called before any function call
*handle = PER_***open**(*...*)	Opens a peripheral channel and then performs the operation indicated by *the parameters*; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls.

PER_**config**([handle,] *configStructure)	Initializes the peripheral based on the functional parameters included in the initialization structure. Functional parameters are peripheral specific. This function may not be supported in all peripherals. Please consult the CSL API document for specific details.
*PER*_**start**(*[handle,] …*)	Starts the peripheral after it has been configured using PER_config().
*PER*_**stop**(*[handle,] …*)	Stops the peripheral after it has been started using PER_start().
PER_**reset**([handle])	Resets the peripheral to its power-on default values.
PER_**close**(handle)	Closes a peripheral channel previously opened with *PER*_open(). The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared.
PER_**read**(handle …)	Read from the peripheral.
PER_**write**(handle …)	Write to the peripheral.

5.4.1 Peripheral Initialization and Programming via Function Level CSL

On top of the register-level CSLR, CSL also provides higher level functions (Table 5-4) to initialize and to control peripherals. Using the CSL functional layer, relatively few function calls, each with appropriate parameters, can be used to control peripherals. This method provides a higher level of abstraction than the direct register manipulation method of CSLR but generally at a cost of larger code size and higher execution cycle count.

Even though each CSL module may offer different parameter-based functions, PER_init() is the most commonly used. PER_init() initializes the parameters in the peripheral that are typically initialized only once in the application as shown in Table 5-5. PER_init() can then be followed by other module functions implementing other common run-time peripheral operations. Other parameter-based functions include module-specific functions such as the PER_config() function shown in Table 5-6.

Table 5-5 Using PER_init()

```
main() {
       ...
       PER_init();

       ...
}
```

Table 5-6 Using PER_config

```
PER_config myConfig = {param_1, ..., param_n};

main() {
             ...
       PER_config (&myConfig);
             ...
}
```

5.4.2 Example of DMA Control via Function Level CSL

The following example illustrates the use of CSL to initialize and use DMA channel 0 to copy a table from address 0x3000 to address 0x2000. Addresses and size of data to be moved are as follows.

Source address: 2000h in data space
Destination address: 3000h in data space
Transfer size: Sixteen 16-bit single words

The example uses CSL functions DMA_init(), DMA_open(…), DMA_config(…), DMA_start(…), DMA_getStatus(…), and DMA_close(…). The next 9 steps illustrate the preparation and use of these functions in exercising control of the DMA operation.

Step 1: Include the header file of the module/peripheral, use <csl_dma.h>. The different header files are shown in Table 2-1.

```
#include "csl_dma.h"
#include <stdio.h>
```

Step 2: Define a DMA_Handle pointer and buffers. DMA_open will initialize this handle when a DMA channel is opened.

```
#define CSL_DMA_BUFFER_SIZE 1024

/* Declaration of the buffer */
Uint16 dmaSRCBuff[CSL_DMA_BUFFER_SIZE];
Uint16 dmaDESTBuff[CSL_DMA_BUFFER_SIZE];

CSL_DMA_Handle          dmaHandle;
CSL_DMA_Config          dmaConfig;
CSL_DMA_Config          getdmaConfig;

CSL_DMA_ChannelObj      dmaObj;
CSL_Status              status;
```

Step 3: Define and initialize the DMA channel configuration structure (see csl_dma.h for other options).

```
dmaConfig.autoMode     = CSL_DMA_AUTORELOAD_DISABLE;

dmaConfig.burstLen     = CSL_DMA_TXBURST_8WORD;

dmaConfig.trigger      = CSL_DMA_SOFTWARE_TRIGGER;

dmaConfig.dmaEvt       = CSL_DMA_EVT_NONE;

dmaConfig.dmaInt       = CSL_DMA_INTERRUPT_DISABLE;

dmaConfig.chanDir      = CSL_DMA_READ;

dmaConfig.trfType      = CSL_DMA_TRANSFER_MEMORY;

dmaConfig.dataLen      = CSL_DMA_BUFFER_SIZE * 2;

dmaConfig.srcAddr      = (Uint32)dmaSRCBuff;
```

```
dmaConfig.destAddr     = (Uint32)dmaDESTBuff;
```

Step 4: Initialize the DMA module driver. It must be done before calling any DMA module API:

```
status = DMA_init();

if (status != CSL_SOK)

{

printf("DMA_init() Failed \n");

}
```

Step 5: For multi-resource peripherals such as McBSP and DMA, call PER_open to reserve resources (SPI_open(), DMA_open()...):

```
 dmaHandle = DMA_open(0,&dmaObj, &status);
 if (dmaHandle == NULL)
 {
     printf("DMA_open() Failed \n");
 }
```

By default, the TMS320C55xx compiler assigns all data symbols word addresses. The DMA however, expects all addresses to be byte addresses. The CSL will convert the word address to a byte address (multiply by 2 or shift left one bit) for the DMA transfer.

Step 6: Configure the DMA channel by calling DMA_config() function and read back the configuration values by calling DMA_getConfig() function:

```
status = DMA_config(dmaHandle, &dmaConfig);

 if (status != CSL_SOK)

 {

     printf("DMA_config() Failed \n");

     break;

 }

status = DMA_getConfig(dmaHandle, &getdmaConfig);

 if (status != CSL_SOK)

 {

     printf("DMA_getConfig() Failed \n");

     break;
```

```
}
```

Step 7: Call DMA_start() to begin DMA transfers:

```
status = DMA_start(dmaHandle);
 if (status != CSL_SOK)
 {
     printf("DMA_start() Failed \n");
 }
```

Step 8: Wait for DMA transfer to complete:

```
// DMA_getStatus will return 0 when the DMA is done
while (DMA_getStatus(dmaHandle));
```

Step 9: Close DMA channel:

```
status = DMA_close(dmaHandle);
 if (status != CSL_SOK)
 {
     printf("DMA_reset() Failed \n");
 }
```

For more detail, refer to example csl_dma_PollExample.c in the following path

- c55xx_csl/ccs_v5.0_examples/dma/CSL_DMA_PollExample/

The path is the CCS v5.5 version of the project.

5.5 CSL Macros

Table 5-7 provides a generic description of the most common CSL macros. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 5-1 (with the exception of the DAT module).
- *REG* indicates a register name (without the channel number).
- *REG#* indicates, if applicable, a register with the channel number. (For example: DMAGCR, TCR0, ...)

- *FIELD* indicates a field in a register.
- *regval* indicates an integer constant, an integer variable, a symbolic constant (*PER_REG*_DEFAULT), or a merged field value created with the *PER*_REG_RMK() macro.
- *fieldval* indicates an integer constant, integer variable, macro, or symbolic constant (*PER_REG_FIELD_SYMVAL*) as explained in section 5.6; all field values are right justified.

CSL also offers equivalent macros to those listed in Table 5-7, but instead of using REG# to identify which channel the register belongs to, it uses the Handle value. The Handle value is returned by the PER_open() function.

Table 5-7 Generic CSL Macros

Macro	Description
CSL_FMK(PER_REG_FIELD, val)	Creates a shifted version of *val* that you could OR with the result of other _FMK macros to initialize register REG. This allows you to initialize few fields in REG as an alternative to the _RMK macro that requires that ALL the fields in the register be initialized.
val = CSL_FEXT(reg, PER_REG_FIELD)	Returns the value of the specified *FIELD* in the peripheral register.
CSL_FINS(reg, PER_REG_FIELD, val)	Insert the value to the specified FIELD in the peripheral register
CSL_FMKR(msb, lsb, val)	Creates a shifted version of *val for the bits between msb and lsb*
CSL_FEXTR(reg, msb, lsb)	Extracts the bits between msb and lsb of the reg
CSL_FINSR(reg, msb, lsb, val)	Set the bits between msb and lsb of the reg to val

All Macros are defined in file …/c55xx_csl/inc/cslr.h.

The following statement will enable the timer interrupt by setting the bit 4 of IER0 to 1:

```
CSL_FINST(CSL_CPU_REGS->IER0, CPU_IER0_TINT, ENABLE);
```

*****Important note*****:**

CSLr macro CSL_FEXT cannot be used to read the status registers which will have ‘read to clear’ property. Some of the status registers will be cleared when they are read. While trying to read a specific bit using CSL_FEXT, it resets other bits also. This is an expected behavior as per the macro implementation but may mislead the users who are not familiar with CSL.

For example, if user tries to read MMCSD status register to check the error condition shown below:

```
if(CSL_FEXT(CSL_MMCSD0_REGS->MMCST0, MMCSD_MMCST0_TOUTRD))
{
    printf(“MMCSD Read Timeout\n”);
}
if(CSL_FEXT(CSL_MMCSD0_REGS->MMCST0, MMCSD_MMCST0_CRCRD))
{
```

```
    printf("MMCSD Read CRC Error\n");
}
...
...
```

In the above case, assume no read timeout occurs but there are some other errors. But the MMCST0 will be cleared by the CSL_FEXT macro in the first 'if' condition and the other errors are not visible to the program.

It is always recommended to read the whole status register and then check each bit for errors.

5.6 CSL Symbolic Constant Values

To facilitate initialization of values in application code, the CSLR register level layer provides symbolic constants for peripheral registers and writable field values as described in Table 5-8. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 5-1 (with the exception of the DAT module, which does not have its own registers).
- *REG* indicates a peripheral register.
- *FIELD* indicates a field in the register.
- *SYMVAL* indicates the symbolic value of a register field.

Table 5-8 Generic CSL Symbolic Constants

Constant	Description
PER_REG_FIELD_SYMVAL	Symbolic constant to specify values for individual fields in the specified peripheral register.
*PER_REG_FIELD_***DEFAULT**	Default value for a field; corresponds to the field value after a reset or to 0 if a reset has no effect.

All Symbolic Constant Values are defined in file …/c55xx_csl/inc/cslr_PER.h and …/c55xx_csl/inc/soc.h.

5.7 Resource Management and the Use of CSL Handles

CSL provides limited support for resource management in applications that involve multiple threads, reusing the same multichannel peripheral device.

Resource management in CSL is achieved through calls to the PER_open and PER_close functions. The PER_open function normally takes a channel/port number as the primary argument and returns a pointer to a Handle structure that contains information about which channel (DMA) or port (SPI) was opened.

When given a specific channel/port number, the open function checks a global flag to determine its availability. If the port/channel is available, then it returns a pointer to a predefined Handle structure for this device. If the device has already been opened by another process, then an invalid Handle is returned with a value equal to the CSL symbolic constant, INV.

Calling PER_close frees a port/channel for use by other processes. PER_close clears the in_use flag and resets the port/channel.

5.7.1 Using CSL Handles

CSL Handle objects are used to uniquely identify an opened peripheral channel/port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open function before calling any other API functions that require a handle object as argument. For example:

```
DMA_Handle myDma; /* Defines a DMA_Handle object, myDma */ … //Once defined,
the CSL Handle object is initialized by a call to PER_open:

myDma = DMA_open(DMA_CHA0,DMA_OPEN_RESET); /* Open DMA channel 0 */


//The call to DMA_open initializes the handle, myDma. This handle can then
be used in calls to other API //functions:

DMA_start(myDma); /* Begin transfer */

DMA_close(myDma); /* Free DMA channel */
```