F28P65x Firmware Development Package

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 5.00.00.00 of this document, last updated on Wed Jul 12 21:12:54 IST 2023.

Table of Contents

Copyright **2**
Revision Information **2**
1 Introduction **9**
1.1 Detailed Revision History 9
2 Getting Started and Troubleshooting **11**
2.1 Introduction 11
2.2 Project Creation 11
2.3 Debugging Dual Core Applications 30
2.4 Project: Adding Bit-field or DriverLib Support 33
2.5 Troubleshooting 34
3 Interrupt Service Routine Priorities **37**
3.1 Interrupt Hardware Priority Overview 37
3.2 PIE Interrupt Priorities 38
3.3 Software Prioritization of Interrupts 39
4 CPU 1 Bit-field Example Applications **43**
4.1 ADC SOC Software Force (adc_soc_software) 43
4.2 ADC ePWM Triggering (adc_soc_epwm) 43
4.3 ADC temperature sensor conversion (adc_soc_epwm_tempsensor) 43
4.4 ADC Synchronous SOC Software Force (adc_soc_software_sync) 44
4.5 ADC Continuous Triggering (adc_soc_continuous) 44
4.6 ADC Continuous Conversions Read by DMA (adc_soc_continuous_dma) 44
4.7 ADC PPB Offset (adc_ppb_offset) 44
4.8 ADC PPB Limits (adc_ppb_limits) 45
4.9 ADC PPB Delay Capture (adc_ppb_delay) 45
4.10 Buffered DAC Enable (buffdac_enable) 45
4.11 Buffered DAC Sine DMA (buffdac_sine_dma) 45
4.12 DMA GSRAM Transfer (dma_gsram_transfer) 46
4.13 EPWM Trip Zone Module (epwm_trip_zone) 46
4.14 EPWM Action Qualifier (epwm_updown_aq) 47
4.15 EPWM Action Qualifier (epwm_up_aq) 47
4.16 EPWM dead band control (epwm_deadband) 47
4.17 HRPWM Duty SFO (hrpwm_duty_sfo_v8) 48
4.18 HRPWM SFO Test (hrpwm_prdupdown_sfo_v8) 49
4.19 External Interrupts (ExternalInterrupt) 49
4.20 SDFM Filter Sync CPU 50
4.21 SDFM Filter Sync CLA 51
4.22 SDFM Filter Sync DMA 52
4.23 SDFM PWM Sync 53
5 Dual Core Bit-field Example Applications **55**
5.1 DMA Transfer Shared Peripheral 55
5.2 Shared RAM management (CPU1) 55
5.3 Shared RAM management (CPU2) 56
6 C28x Driver Library Example Applications **57**
6.1 ADC ePWM Triggering Multiple SOC 57
6.2 ADC Burst Mode 58
6.3 ADC Burst Mode Oversampling 58
6.4 ADC SOC Oversampling 58

6.5 ADC PPB PWM trip (adc_ppb_pwm_trip) . . . 59
6.6 ADC Software Triggering . . . 59
6.7 ADC ePWM Triggering . . . 60
6.8 ADC Temperature Sensor Conversion . . . 60
6.9 ADC Synchronous SOC Software Force (adc_soc_software_sync) . . . 60
6.10 ADC Continuous Triggering (adc_soc_continuous) . . . 61
6.11 ADC Continuous Conversions Read by DMA (adc_soc_continuous_dma) . . . 61
6.12 ADC PPB Offset (adc_ppb_offset) . . . 61
6.13 ADC PPB Limits (adc_ppb_limits) . . . 62
6.14 ADC PPB Delay Capture (adc_ppb_delay) . . . 62
6.15 BGCRC CPU Interrupt Example . . . 62
6.16 BGCRC Example with Watchdog and Lock . . . 63
6.17 CLA-BGCRC Example in CRC mode . . . 63
6.18 CLA-BGCRC Example in Scrub Mode . . . 63
6.19 CAN External Loopback . . . 64
6.20 CAN External Loopback with Interrupts . . . 64
6.21 CAN External Loopback with DMA . . . 65
6.22 CAN Transmit and Receive Configurations . . . 65
6.23 CAN Error Generation Example . . . 66
6.24 CAN Remote Request Loopback . . . 66
6.25 CAN example that illustrates the usage of Mask registers . . . 67
6.26 CLA $arcsine(x)$ using a lookup table (cla_asin_cpu01) . . . 67
6.27 CLA $arctangent(x)$ using a lookup table (cla_atan_cpu01) . . . 68
6.28 CLA background nesting task . . . 68
6.29 Controlling PWM output using CLA . . . 69
6.30 Just-in-time ADC sampling with CLA . . . 69
6.31 Optimal offloading of control algorithms to CLA . . . 70
6.32 Handling shared resources across C28x and CLA . . . 71
6.33 CLB Timer Two States . . . 72
6.34 CLB Interrupt Tag . . . 72
6.35 CLB Output Intersect . . . 72
6.36 CLB PUSH PULL . . . 72
6.37 CLB Multi Tile . . . 72
6.38 CLB Combinational Logic . . . 73
6.39 CLB XBARs . . . 73
6.40 CLB Lock Output Mask . . . 73
6.41 CLB INPUT Pipeline Mode . . . 73
6.42 CLB Clocking and PIPELINE Mode . . . 73
6.43 CLB SPI Data Export . . . 73
6.44 CLB SPI Data Export DMA . . . 74
6.45 CLB Trip Zone Timestamp . . . 74
6.46 CLB GPIO Input Filter . . . 74
6.47 CLB Auxilary PWM . . . 75
6.48 CLB PWM Protection . . . 75
6.49 CLB Signal Generator . . . 75
6.50 CLB State Machine . . . 75
6.51 CLB External Signal AND Gate . . . 75
6.52 CLB Timer . . . 75
6.53 CLB Empty Project . . . 76
6.54 CMPSS Asynchronous Trip . . . 76
6.55 CMPSS Digital Filter Configuration . . . 76
6.56 Buffered DAC Enable . . . 77

6.57 Buffered DAC Random . . . 77
6.58 Buffered DAC Sine (buffdac_sine) . . . 77
6.59 DCC Single shot Clock verification . . . 78
6.60 DCC Single shot Clock measurement . . . 79
6.61 DCC Continuous clock monitoring . . . 79
6.62 DCC Continuous clock monitoring . . . 79
6.63 DCC Detection of clock failure . . . 80
6.64 DCSM Memory partitioning Example . . . 80
6.65 Empty DCSM Tool Example . . . 81
6.66 DMA GSRAM Transfer (dma_ex1_gsram_transfer) . . . 81
6.67 DMA GSRAM Transfer (dma_ex2_gsram_transfer) . . . 82
6.68 eCAP APWM Example . . . 82
6.69 eCAP Capture PWM Example . . . 82
6.70 eCAP APWM Phase-shift Example . . . 82
6.71 eCAP Software Sync Example . . . 83
6.72 Pin setup for EMIF module accessing ASRAM. . . . 83
6.73 EMIF1 ASYNC module accessing 16bit ASRAM. . . . 83
6.74 EMIF1 module accessing 16bit ASRAM as code memory. . . . 83
6.75 EMIF1 module accessing 16bit SDRAM using memcpy_fast_far(). . . . 84
6.76 EMIF1 module accessing 16bit SDRAM then puts into Self Refresh mode before entering Low Power Mode. . . . 84
6.77 EMIF1 module accessing 32bit SDRAM using DMA. . . . 85
6.78 EMIF1 module accessing 16bit SDRAM using alternate address mapping. . . . 85
6.79 Empty Project Example . . . 86
6.80 EPG Generate Serial Data Shift Mode . . . 86
6.81 EPG Generating Synchronous Clocks . . . 86
6.82 EPG Generating Two Offset Clocks . . . 86
6.83 EPG Generating Two Offset Clocks With SIGGEN . . . 86
6.84 EPG Generate Serial Data . . . 87
6.85 ePWM Chopper . . . 87
6.86 EPWM Configure Signal . . . 87
6.87 Realization of Monoshot mode . . . 88
6.88 EPWM Action Qualifier (epwm_up_aq) . . . 88
6.89 ePWM XCMP Mode . . . 88
6.90 ePWM Event Detection . . . 89
6.91 ePWM Diode Emulation . . . 90
6.92 ePWM Minimum Deadband and ICL . . . 91
6.93 ePWM Trip Zone . . . 92
6.94 ePWM Up Down Count Action Qualifier . . . 92
6.95 ePWM Synchronization . . . 92
6.96 ePWM Digital Compare . . . 93
6.97 ePWM Digital Compare Event Filter Blanking Window . . . 93
6.98 ePWM Valley Switching . . . 94
6.99 ePWM Digital Compare Edge Filter . . . 95
6.100ePWM Deadband . . . 95
6.101ePWM DMA . . . 96
6.102Frequency Measurement Using eQEP . . . 96
6.103Position and Speed Measurement Using eQEP . . . 97
6.104Frequency Measurement Using eQEP via unit timeout interrupt . . . 98
6.105Motor speed and direction measurement using eQEP via unit timeout interrupt . . . 99
6.106ERAD Profile Function . . . 99
6.107ERAD Profile Function . . . 100

6.108ERAD HWBP Monitor Program Counter . . . 100
6.109ERAD HWBP Monitor Program Counter . . . 101
6.110ERAD HWBP Stack Overflow Detection . . . 101
6.111ERAD HWBP Stack Overflow Detection . . . 101
6.112ERAD Profiling Interrupts . . . 102
6.113ERAD Profiling Interrupts . . . 102
6.114ERAD MEMORY ACCESS RESTRICT . . . 103
6.115ERAD INTERRUPT ORDER . . . 104
6.116ERAD AND CLB . . . 104
6.117ERAD PWM PROTECTION . . . 104
6.118ERAD Profiling Interrupts . . . 105
6.119ERAD Profile Function . . . 106
6.120ERAD Stack Overflow . . . 107
6.121ERAD Profile Interrupts CLA . . . 108
6.122FSI daisy chain topology, lead device example . . . 109
6.123FSI daisy chain topology, node device example . . . 110
6.124FSI Loopback:CPU Control . . . 111
6.125FSI Loopback CLA control . . . 112
6.126FSI DMA frame transfers:DMA Control . . . 113
6.127FSI data transfer by external trigger . . . 114
6.128FSI data transfers upon CPU Timer event . . . 115
6.129FSI and SPI communication(fsi_ex6_spi_main_tx) . . . 116
6.130FSI and SPI communication(fsi_ex7_spi_remote_rx) . . . 116
6.131FSI P2Point Connection:Rx Side . . . 116
6.132FSI P2Point Connection:Tx Side . . . 118
6.133Device GPIO Setup . . . 119
6.134Device GPIO Toggle . . . 119
6.135Device GPIO Interrupt . . . 119
6.136External Interrupt (XINT) . . . 119
6.137HRCAP Capture and Calibration Example . . . 120
6.138HRPWM Duty Control with SFO . . . 120
6.139HRPWM Slider . . . 121
6.140HRPWM Period Control . . . 121
6.141HRPWM Duty Control with UPDOWN Mode . . . 121
6.142HRPWM Slider Test . . . 122
6.143HRPWM Duty Up Count . . . 122
6.144HRPWM Period Up-Down Count . . . 123
6.145I2C Digital Loopback with FIFO Interrupts . . . 123
6.146I2C EEPROM . . . 124
6.147I2C Digital External Loopback with FIFO Interrupts . . . 124
6.148I2C EEPROM . . . 125
6.149I2C controller target communication using FIFO interrupts . . . 125
6.150I2C EEPROM . . . 126
6.151External Interrupts (ExternalInterrupt) . . . 126
6.152Multiple interrupt handling of I2C, SCI & SPI Digital Loopback . . . 127
6.153CPU Timer Interrupt Software Prioritization . . . 128
6.154EPWM Real-Time Interrupt . . . 128
6.155F28P65X LaunchPad Out of Box Demo Example . . . 129
6.156LED Blinky Example . . . 130
6.157LIN Internal Loopback with Interrupts . . . 130
6.158LIN SCI Mode Internal Loopback with Interrupts . . . 130
6.159LIN SCI MODE Internal Loopback with DMA . . . 131

6.160LIN Internal Loopback without interrupts(polled mode) . . . 131
6.161LIN Internal Loopback with Interrupts using Sysconfig . . . 131
6.162LIN Incomplete Header Detection . . . 132
6.163LIN External Loopback without interrupts(polled mode) . . . 132
6.164MCAN Loopback with Interrupts Example Using SYSCONFIG Tool . . . 133
6.165Correctable & Uncorrectable Memory Error Handling . . . 133
6.166Tune Baud Rate via UART Example . . . 133
6.167SCI FIFO Digital Loop Back . . . 134
6.168SCI Digital Loop Back with Interrupts . . . 134
6.169SCI Echoback . . . 135
6.170stdout redirect example . . . 135
6.171SDFM Filter Sync CPU . . . 136
6.172SDFM Filter Sync CLA . . . 137
6.173SDFM Filter Sync DMA . . . 138
6.174SDFM PWM Sync . . . 139
6.175SDFM Type 1 Filter FIFO . . . 140
6.176SDFM Filter Sync CLA . . . 140
6.177SD FATFS Library Example . . . 141
6.178SD FATFS Library Example with exFAT Support . . . 141
6.179SPI Digital Loopback . . . 142
6.180SPI Digital Loopback with FIFO Interrupts . . . 142
6.181SPI Digital External Loopback without FIFO Interrupts . . . 143
6.182SPI Digital External Loopback with FIFO Interrupts . . . 143
6.183SPI Digital Loopback with DMA . . . 144
6.184SPI EEPROM . . . 144
6.185SPI DMA EEPROM . . . 144
6.186Missing clock detection (MCD) . . . 145
6.187XCLKOUT (External Clock Output) Configuration . . . 145
6.188CPU Timers . . . 146
6.189UART Echoback . . . 146
6.190USB HUB Host example . . . 147
6.191USB CDC serial example . . . 147
6.192USB HID Mouse Device . . . 147
6.193USB Device Keyboard . . . 148
6.194USB Generic Bulk Device . . . 148
6.195USB HID Mouse Host . . . 148
6.196USB HID Keyboard Host . . . 149
6.197USB Mass Storage Class Host . . . 149
6.198USB Dual Detect . . . 149
6.199USB Throughput Bulk Device Example (usb_ex9_throughput_dev_bulk) . . . 149
6.200Watchdog . . . 150
6.201Flash Programming with AutoECC, DataAndECC, DataOnly and EccOnly . . . 150
6.202Flash Programming with AutoECC, DataAndECC, DataOnly and EccOnly . . . 151

7 Dual Core Driver Library Example Applications . . . 153
7.1 DMA Transfer Shared Peripheral . . . 153
7.2 IPC basic message passing example with interrupt . . . 154
7.3 IPC basic message passing example with interrupt . . . 154
7.4 IPC message passing example with interrupt and message queue . . . 154
7.5 IPC message passing example with interrupt and message queue . . . 155
7.6 IPC basic message passing example with interrupt . . . 155
7.7 IPC basic message passing example with interrupt . . . 156
7.8 IPC message passing example with interrupt and message queue . . . 156

7.9 IPC message passing example with interrupt and message queue . . . 157
7.10 LED Blinky Example . . . 157
7.11 LED Blinky Example . . . 158
7.12 Shared RAM Management (CPU1) . . . 158
7.13 Shared RAM Management (CPU2) . . . 159
7.14 NMI handling . . . 159
7.15 Watchdog Reset . . . 160
8 Device APIs for examples . . . 161
8.1 Introduction . . . 161
8.2 API Functions . . . 161
IMPORTANT NOTICE . . . 164

1 Introduction

The Texas Instruments® F28P65x Firmware development library is a group of example applications and helper libraries that demonstrate the basics of getting started with a F28P65x device.

The following chapter (chapter 2) provides a step by step guide for from scratch project creation for each core as well as debug. It is highly recommended that users new to the F28P65x family of devices start by reading this section first.

Because the F28P65x devices have two cores the example applications have been broken up to distinguish which examples run on each core.

- The driver library example applications which run exclusively on the C28x CPU core can be found in the `~/driverlib/f28p65x/examples/c28x` directory.- The driver library example applications which require both C28x CPU Cores to run can be found in the `~/driverlib/f28p65x/examples/c28x_dual` directory. directory.

The examples provided are built for controlCARD compatibility.

As users move past evaluation, and get started developing their own application, TI recommends they maintain a similar project directory structure to that used in the example projects. Example projects have a heirarchy as follows:

- Main project directory
  - C28x project folder (c28x)
    - C28x project sources (∗.c, ∗.h)
    - CCS folder (ccs)
      - CCS project specific files
  - C28x Dual project folder (c28x_dual)
    - CPU 2 project sources (∗.c, ∗.h)
    - CCS folder (ccs)
      - CCS project specific files

1.1 Detailed Revision History

V5.00.00.00

- This version is the first release (packaged with development tools) of the F28P65x header files, bitfield commons, drivers and examples.

2 Getting Started and Troubleshooting

Project Creation 11 Project: Adding Bit-field or DriverLib Support 33 Debugging Dual Core Applications 30
Troubleshooting 34

2.1 Introduction

Because of the sheer complexity of the F28P65x devices, it is not uncommon for new users to have trouble bringing up the device their first time. This guide aims to give you, the user, a step by step guide for how to create and debug projects from scratch. This guide will focus on the user of a F28P65x controlCARD, but these same ideas should apply to other boards with minimal translation.

2.2 Project Creation

A typical F28P65x application consists of two separate CCS projects: one for CPU 1 and one for CPU2. The two projects are completely independent and have no real linking between them as far as CCS is concerned.

CPU 1 Subsystem Project Creation

1. From the main CCS window select File -> New -> CCS Project. Select your Target as "TMS320F28P650DK9" and use the "C28XX [C2000]" Tab. Name your project and choose a location for it to reside. Click Finish and your project will be created.

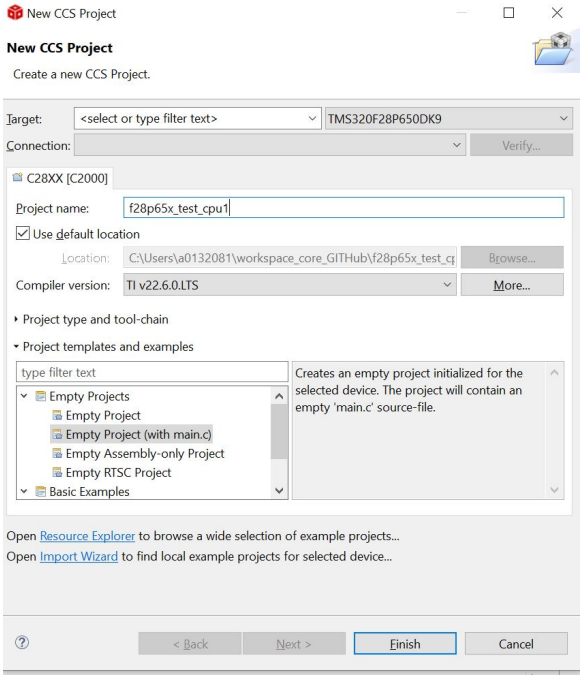

Figure 2.1: Creating a new C28 project

2. Before we can successfully build a project we need to setup some build specific settings. Right click on your project and select Properties. Look at the Processor Options and ensure they match the below image:

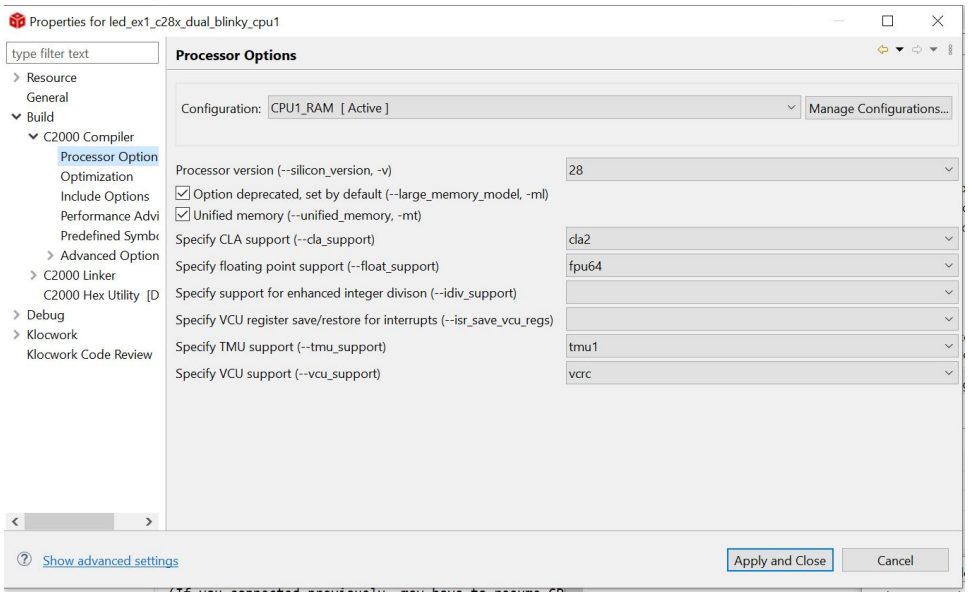

Figure 2.2: Project configuration dialog box

3. In the C2000 Compiler entry look for and select the Include Options. Click on the add directory icon to add a directory to the search path. Click the File System button to browse to the `common\include` folder of your C2000Ware installation (typically `C:\ti\c2000\C2000Ware_X_XX_XX_XX\device_support\f28p65x\common\include`). Replace the 'X's with your current C2000Ware version installation. Click ok to add this path, and repeat this same process to add the `headers\include` directory and the driverlib directory

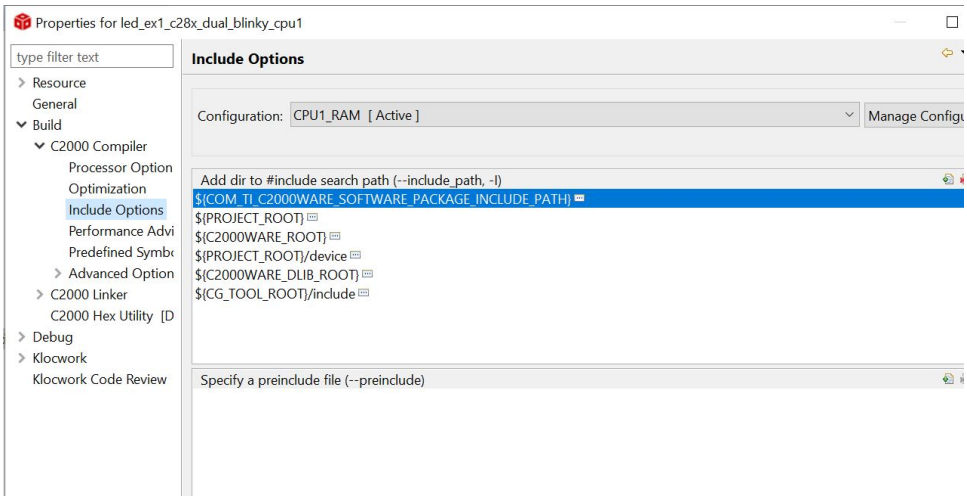

Figure 2.3: Project configuration dialog box

4. Expand the Advanced Options and look for the Predefined Symbol entry. Add a Pre-define NAME called "CPU1". This ensures that the header files build correct for this CPU.

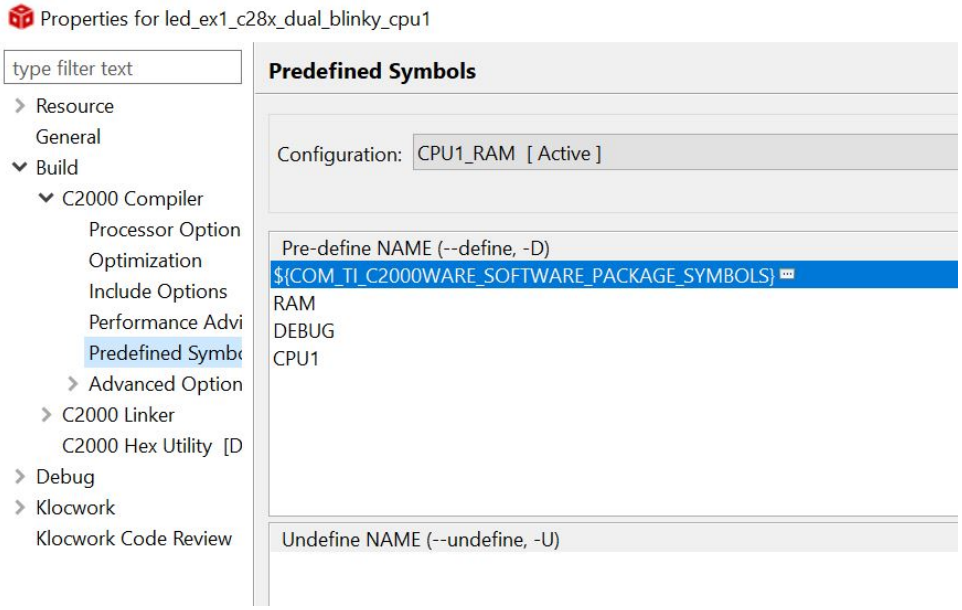

Figure 2.4: Project configuration dialog box

5. Click on the Linker File Search Path. Add these directories to the search path: `common\cmd` and `headers\cmd`. Then you'll also want to add the lib and linker command files

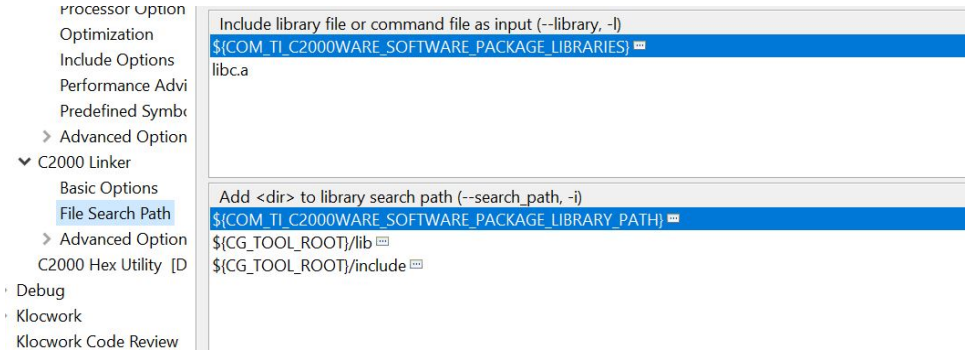

Figure 2.5: Project configuration dialog box

6. While you have this window open select the Symbol Management options under C2000 Linker Advanced Options. Specify the program entry point to be `code_start`. Select ok to close out of the Build Properties.

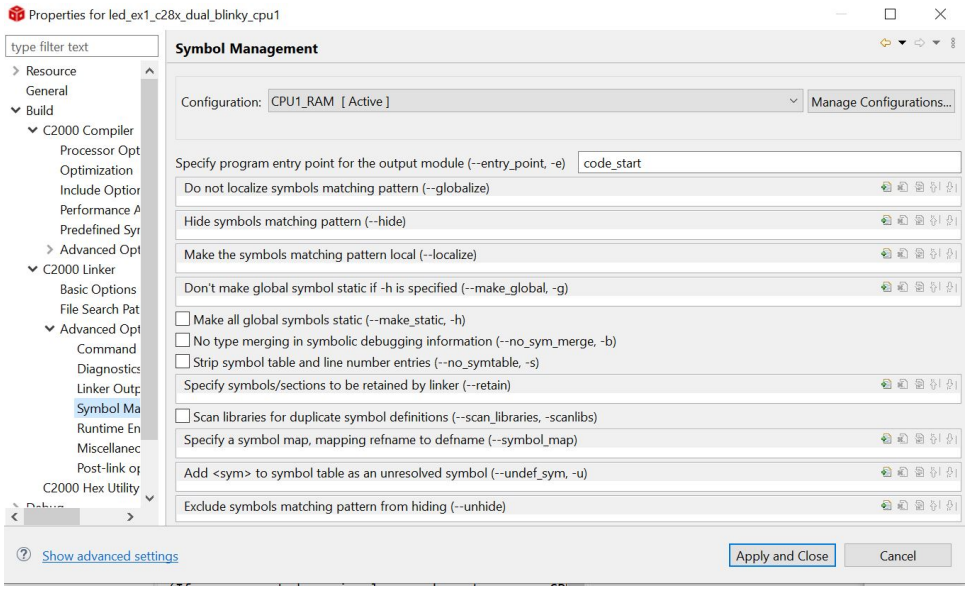

Figure 2.6: Symbol Management options

7. In the project explorer, check that no linker command file got added during project setup. If so, remove the linker command file that got added.
8. Next we need to link in a few files which are used by the header files. To do this right click on your project in the workspace and select Add Files...

   At this point your project workspace should look like the following:

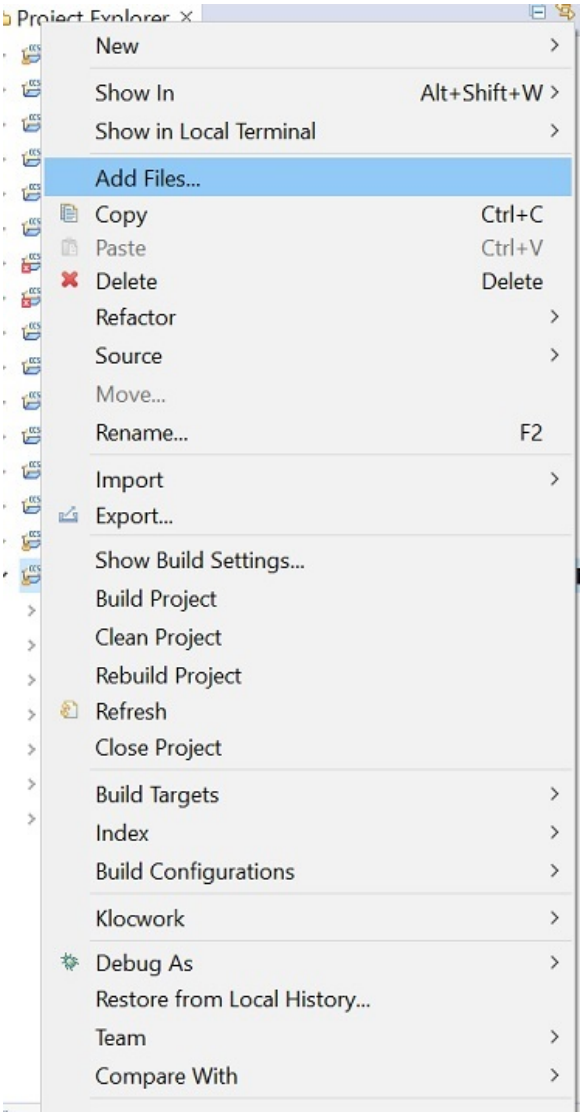

Figure 2.7: Linking files to project

9. Create a new file by right clicking on the project and selecting New -> File. Name this file main.c and copy the following code into it:

```
//
// Included Files
//
#include "driverlib.h"
#include "device.h"

//
// Main
//
void main(void)
{
    //
    // Initialize device clock and peripherals
    //
    Device_init();

    //
    // Initialize GPIO and configure the GPIO pin as a push-pull output
    //
    Device_initGPIO();
    GPIO_setPadConfig(DEVICE_GPIO_PIN_LED1, GPIO_PIN_TYPE_STD);
    GPIO_setDirectionMode(DEVICE_GPIO_PIN_LED1, GPIO_DIR_MODE_OUT);

    //
    // Initialize PIE and clear PIE registers. Disables CPU interrupts.
    //
    Interrupt_initModule();

    //
    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    //
    Interrupt_initVectorTable();

    //
    // Enable Global Interrupt (INTM) and realtime interrupt (DBGM)
    //
    EINT;
    ERTM;

    //
    // Loop Forever
    //
    for(;;)
    {
        //
        // Turn on LED
        //
        GPIO_writePin(DEVICE_GPIO_PIN_LED1, 0);
```

```
            //
            // Delay for a bit.
            //
            DEVICE_DELAY_US(500000);

            //
            // Turn off LED
            //
            GPIO_writePin(DEVICE_GPIO_PIN_LED1, 1);

            //
            // Delay for a bit.
            //
            DEVICE_DELAY_US(500000);
        }
}

//
// End of File
//
```

10. Save main.c, update it with the content needed and then attempt to build the project by right click on it and selecting Build Project. Assuming the project builds try debugging this project on a f28p65x device. When the code runs you should see LED1 toggle.

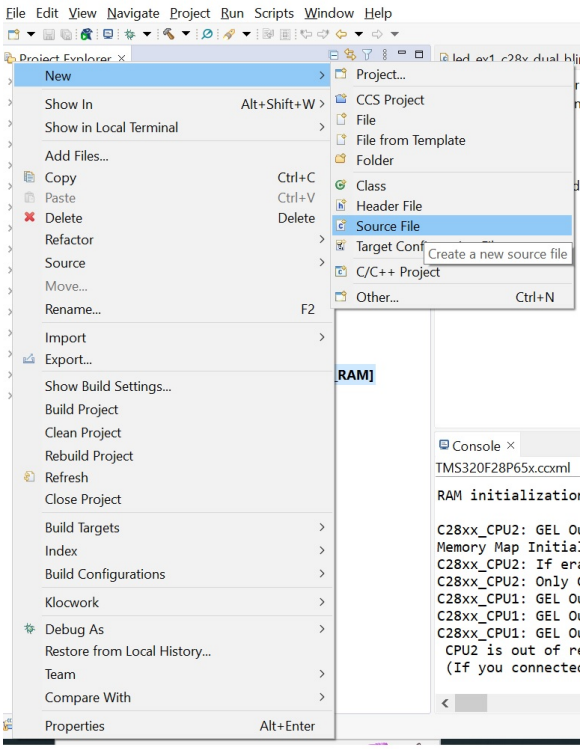

Figure 2.8: Create a new file

CPU 2 Subsystem Project Creation

1. From the main CCS window select File -> New -> CCS Project. Select your Target as "TMS320F28P650DK9" and use the "C28XX [C2000]" Tab. Name your project and choose a location for it to reside. Click Finish and your project will be created.

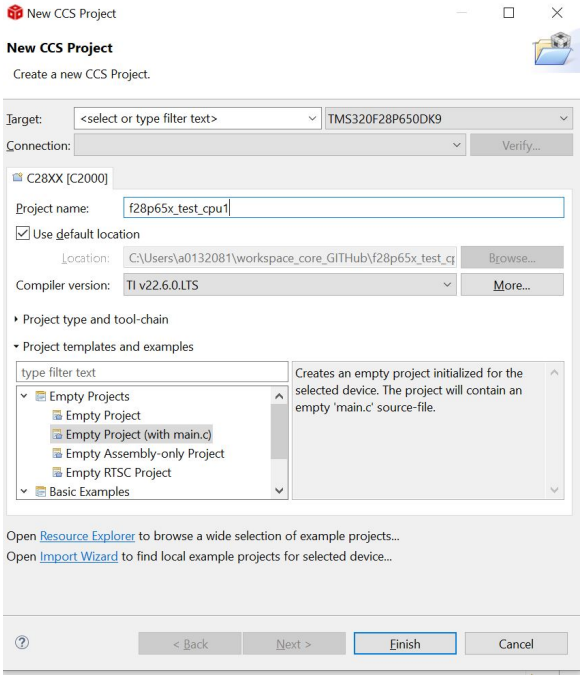

Figure 2.9: Creating a new C28 project

2. Before we can successfully build a project we need to setup some build specific settings. Right click on your project and select Properties. Look at the Processor Options and ensure they match the below image:

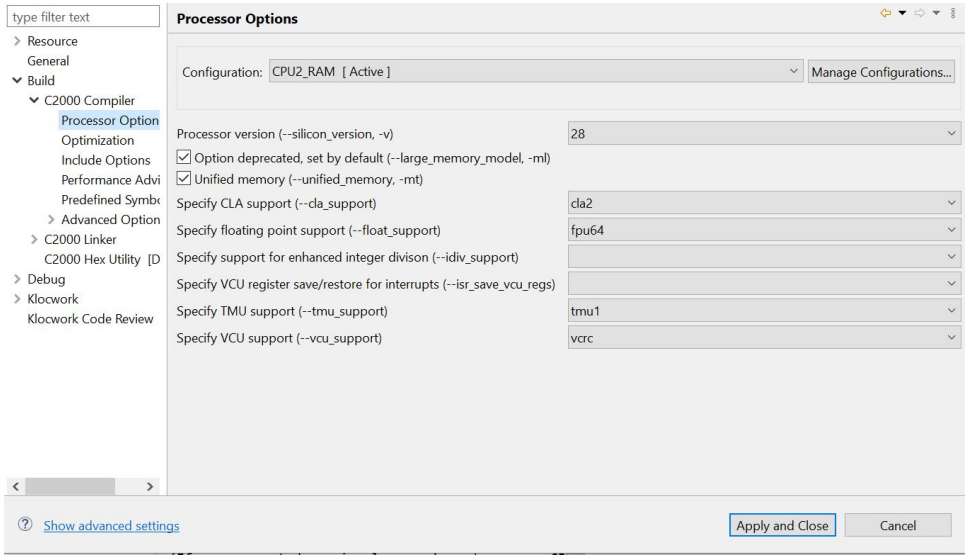

Figure 2.10: Project configuration dialog box

3. In the C2000 Compiler entry look for and select the Include Options. Click on the add directory icon to add a directory to the search path. Click the File System button to browse to the `common\include` folder of your C2000Ware installation (typically `C:\ti\c2000\C2000Ware_X_XX_XX_XX\device_support\f28p65x\common\include`). Replace the 'X's with your current C2000Ware version installation. Click ok to add this path, and repeat this same process to add the `headers\include` directory and the driverlib directory

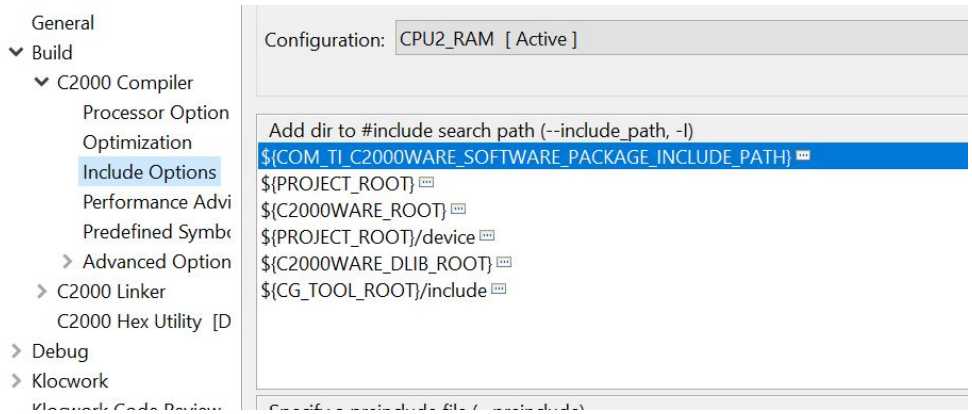

Figure 2.11: Project configuration dialog box

4. Expand the Advanced Options and look for the Predefined Symbol entry. Add a Pre-define NAME called "CPU2". This ensures that the header files build correct for this CPU.

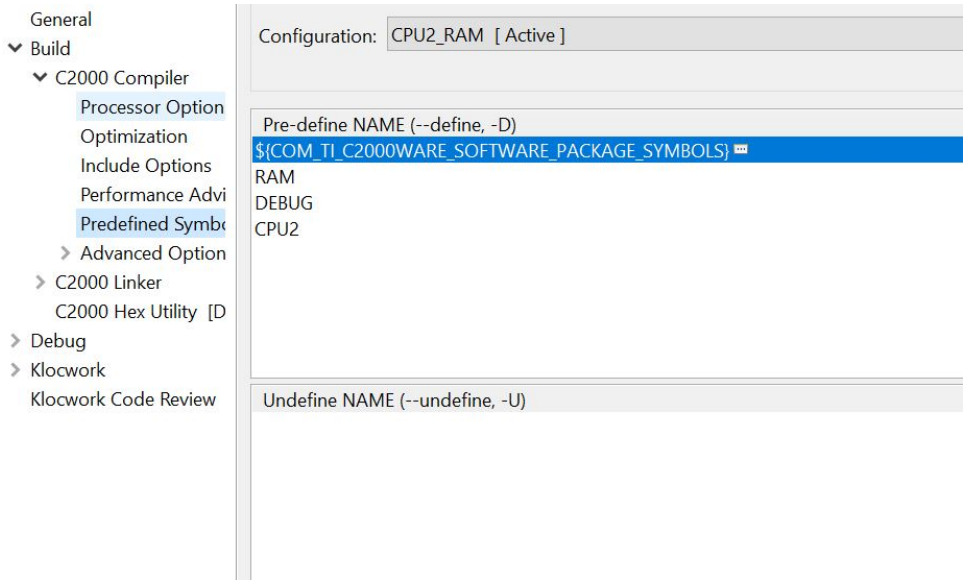

Figure 2.12: Project configuration dialog box

5. Click on the Linker File Search Path. Add these directories to the search path: `common\cmd` and `headers\cmd`. Then you'll also want to add the lib and linker command files

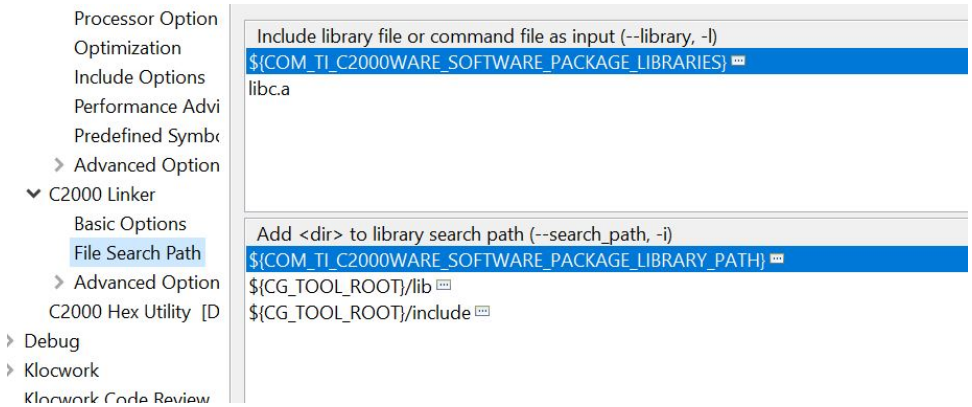

Figure 2.13: Project configuration dialog box

6. While you have this window open select the Symbol Management options under C2000 Linker Advanced Options. Specify the program entry point to be `code_start`. Select ok to close out of the Build Properties.

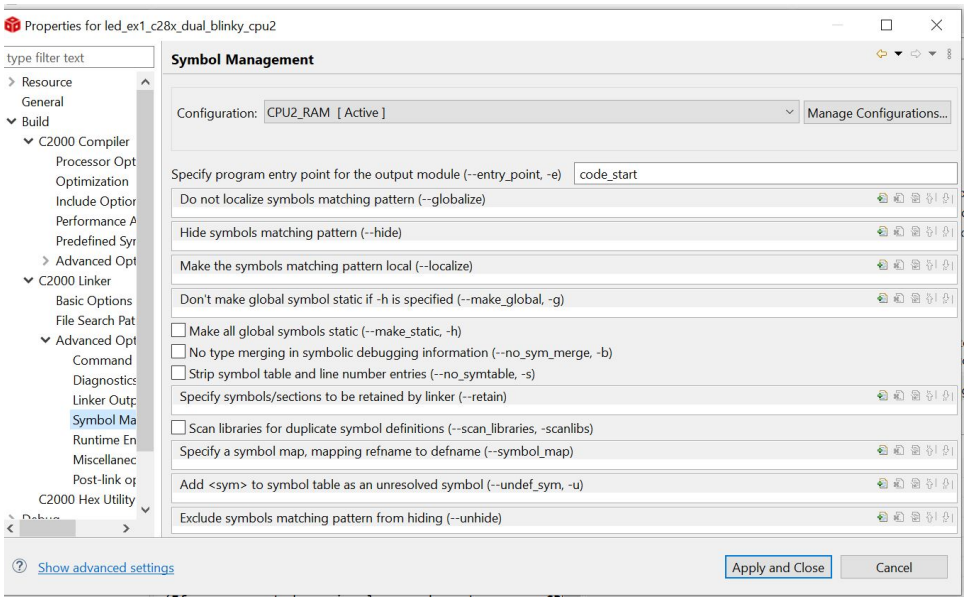

Figure 2.14: Symbol Management options

7. In the project explorer, check that no linker command file got added during project setup. If so, remove the linker command file that got added.
8. Next we need to link in a few files which are used by the header files. To do this right click on your project in the workspace and select Add Files...

   At this point your project workspace should look like the following:

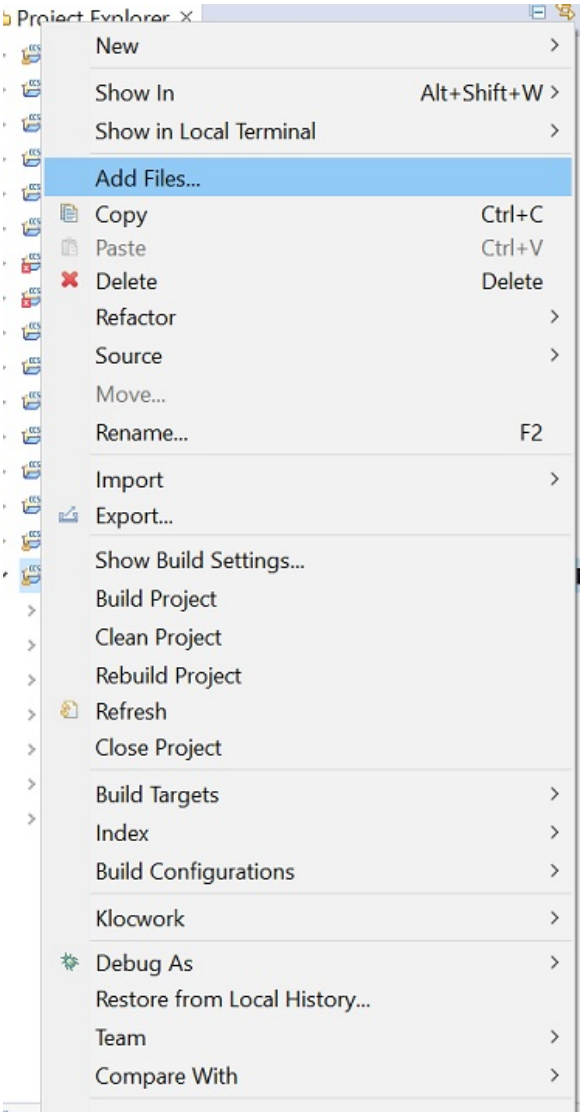

Figure 2.15: Linking files to project

9. Create a new file by right clicking on the project and selecting New -> File. Name this file main.c and copy the following code into it:

```
//
// Included Files
//
#include "driverlib.h"
#include "device.h"

//
// Main
//
void main(void)
{
    //
    // Initialize device clock and peripherals
    //
    Device_init();

    //
    // Initialize GPIO and configure the GPIO pin as a push-pull output
    //
    // This is configured by CPU1

    //
    // Initialize PIE and clear PIE registers. Disables CPU interrupts.
    //
    Interrupt_initModule();

    //
    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    //
    Interrupt_initVectorTable();

    //
    // Enable Global Interrupt (INTM) and realtime interrupt (DBGM)
    //
    EINT;
    ERTM;

    //
    // Loop Forever
    //
    for(;;)
    {
        //
        // Turn on LED
        //
        GPIO_writePin(DEVICE_GPIO_PIN_LED2, 0);

        //
        // Delay for a bit.
```

```
            //
            DEVICE_DELAY_US(500000);

            //
            // Turn off LED
            //
            GPIO_writePin(DEVICE_GPIO_PIN_LED2, 1);

            //
            // Delay for a bit.
            //
            DEVICE_DELAY_US(500000);
        }
}

//
// End of File
//
```

10. Save main.c,update it with the content needed and then attempt to build the project by right click on it and selecting Build Project. Assuming the project builds try debugging both these projects simultaneously on a f28p65x device, otherwise carefully examine the error and the above steps to determine what could have gone wrong. When the code runs you should see LED2 toggle.

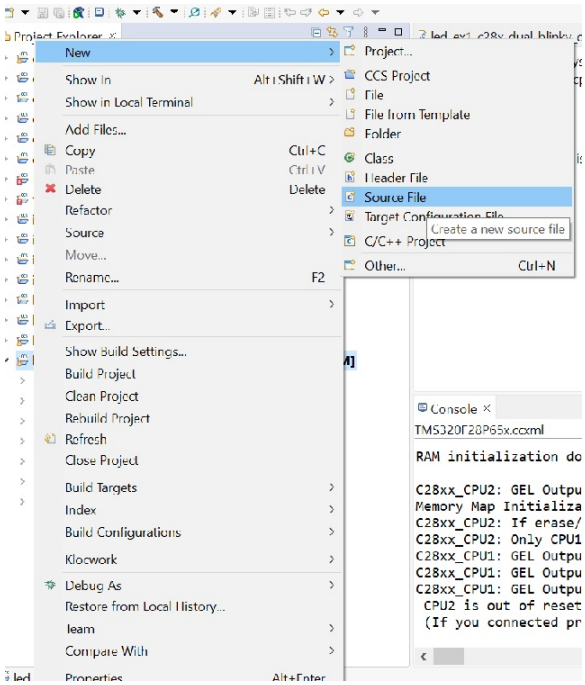

Figure 2.16: Create a new file

2.3 Debugging Dual Core Applications

1. Ensure CCS version 12.4 or newer is installed and up to date. You should have C2000 Code Generation Tools version 22.6.0.LTS and TI ARM Compiler 20.2.7.LTS.
2. Connect a USB C cable from the computer to the USB port on the left hand side of the controlCARD. Windows will enumerate and try to install drivers. As long as CCS is installed, Windows should automatically find and install drivers for the emulator.
3. Apply power either via USB or the 5V DC in jack on the docking station. While the emulator on the board is powered from the host computer's USB port, the rest of the board is not. The reason for this is that the JTAG connection on the F28P65x controlCARDs is completely electrically isolated. Because of the typical applications these devices will be used in, it is neccessary to isolate the JTAG connection. However, for bench debug and evaluation (with low voltages), both halves of the board can be powered from the same supply (i.e. USB). Each power domain has an associated power LED which can be used to ensure that each domain has power.
4. Launch CCS and pick the workspace you would like to debug in.
5. Create a new target configuration. Click File -> New -> Target Configuration File and name the file appropriately (i.e. F28P65x_xds110.ccxml). Select the emulator you intend to use (XDS110 USB Debug Probe) from the drop down list, and then select the device variant present on your board (f28p65x controlCARDs have a f28p65x). Save the target configuration and close the window.

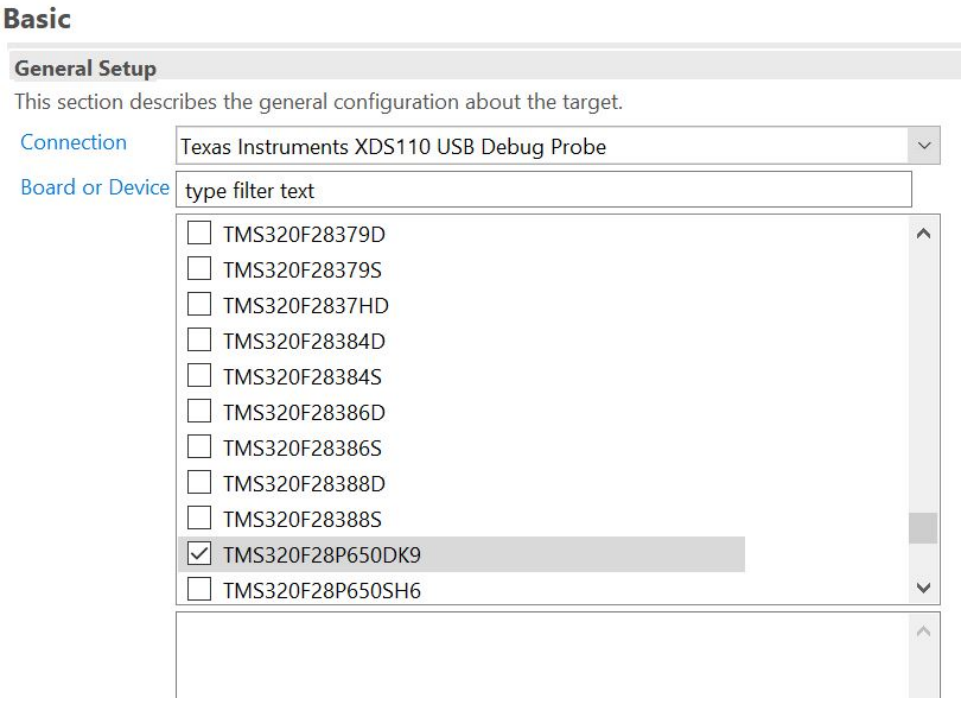

Figure 2.17: F28P65x Card Target Configuration Setup

6. Import the desired example projects (or skip this step if you are using projects you created in the Project Creation section). Click File -> Import, and in the CCS folder select Existing

CCS/CCE Eclipse Projects before clicking Next. With the "Select search-directory" radio button checked, browse to the root of your C2000Ware installation. Device specific software as well as examples are stored in the `driverlib/device_variant` folders. Navigate to the `F28P65x` directory, and then to the `examples/c28x` directory. Click OK and CCS will parse all of the projects in this directory. Import any projects you wish to run into the workspace.

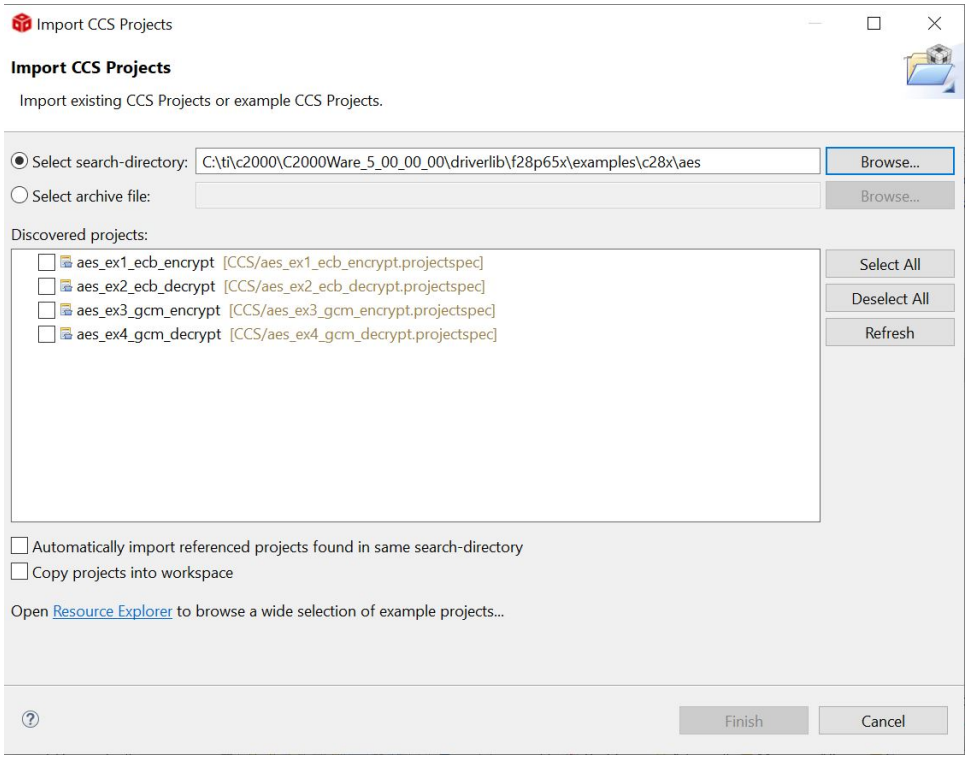

Figure 2.18: Importing f28p65x Projects

7. Build each of the example projects. Right click on each project title and select build project.

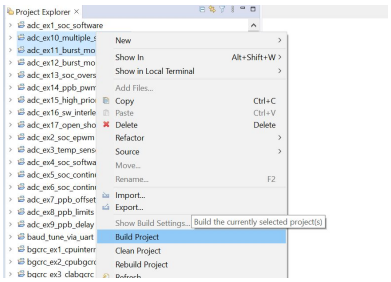

Figure 2.19: Building f28p65x Projects

8. Launch the previously created target configuration. Click View -> Target Configurations. In the window that opens, find the target configuration you created previously, right click on it and select "Launch Target Configuration".

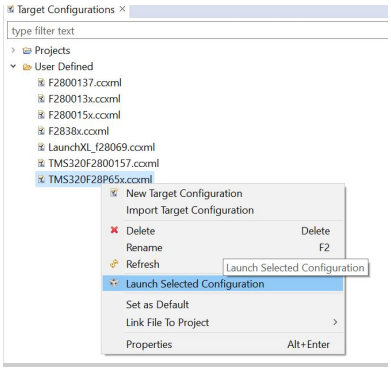

Figure 2.20: Launching a CCS Target Configuration

9. Connect to the device. Right click on each core in the debug window and select "Connect Target. This will connect CCS to the device and will allow you to load code and debug applications.

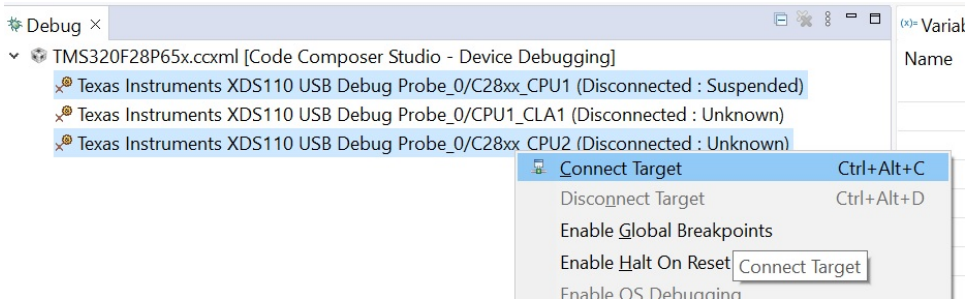

Figure 2.21: Connecting to a Target

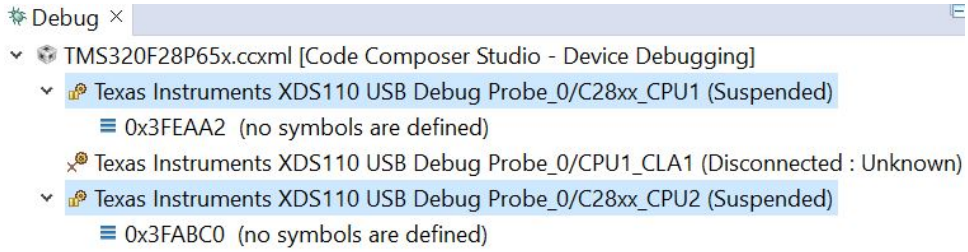

Figure 2.22: After connection to both cores

10. Load code on each of the cores. Select one of the cores in the debug window and then click Target -> Load Program. A dialog box is display which will allow you to select a program to load. Be careful to ensure that you load the appropriate out file on the appropriate core. Repeat this process for the other core by selecting it and following these same steps.
11. At this point both cores should have code loaded and be halted at main. From this point, users should be able to debug code just as they are used to with CCS. Please keep in mind that any

action you take in CCS only has an effect on the core you currently have selected in the debug window. For instance if CPU 1 is selected, the memory window will display the memory map of of the system as seen by CPU 1. The opposite would be true if CPU 2 were selected and similarly for CM.

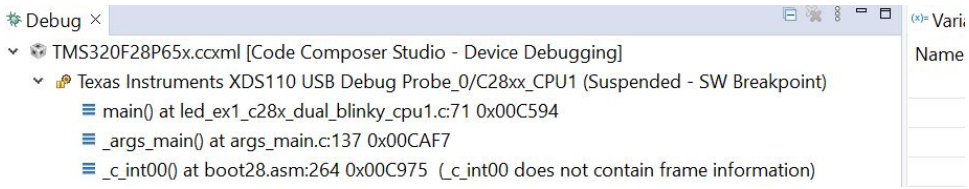

Figure 2.23: Projects loaded on each core

START$_{C}2000WARE_{O}NLY$

2.4 Project: Adding Bit-field or DriverLib Support

F28P65x devices support two types of development software, driver library APIs and bit-field structures. Each have their advantages and are implemented to be compatible together within the same user application. This section details how to add driverlib support to a bit-field project as well as how to add bit-field support to a driverlib project.

When combining bit-field and driverlib support, add a pre-defined symbol within the project properties called "_DUAL_HEADERS". This is required to avoid having conflicting definitions (in enums/structs/macros) which share the exact same names in both bit-field and driverlib headers.

Adding DriverLib Support

1. Add the following include directory path to the project:
   `driverlib\f28p65x\driverlib`
2. Include the following header file in the project main source file:
   `device_support\f28p65x\common\include\driverlib.h`
3. Add or link the `driverlib.lib` library to the project. Location of file:
   `driverlib\f28p65x\driverlib\ccs\Debug`

Adding Bit-field Support

1. Add the following include directory path to the project:
   `device_support\f28p65x\headers\include`
2. Include the following header file in the project main source file:
   `device_support\f28p65x\headers\include\f28p65x_device.h`
3. Add or link the `f28p65x_globalvariabledefs.c` file to the project. Location of file:
   `device_support\f28p65x\headers\source`
4. Add or link the `f28p65x_Headers_nonBIOS.cmd` file to the project. Location of file:
   `device_support\f28p65x\headers\cmd`

END$_{C}2000WARE_{O}NLY$

2.5 Troubleshooting

There are a number of things that can cause the user trouble while bringing up a debug session the first time. This section will try to provide solutions to the most common problems encountered with the Delfino devices.

"I get a managed make error when I import the example projects"

This occurs when one imports a project for which he or she doesn't have the code generation tools for. Please ensure that you have at least C2000 Code Generation Tools version 22.6.0.LTS and TI ARM Compiler 20.2.7.LTS or later.

"I cannot build the example projects"

This is caused by linked resources not being where the project expects them to be. For instance, if you imported the projects and selected "Copy projects to workspace", the projects would no longer build because the files they reference aren't a part of your workspace. Always build and run the examples directly in the C2000Ware directory tree.

"My F28P65x device isn't in the target configuration selection list"

The list of available device for debug is determined based on a number of factors, including drivers and tools chains available on the host system. If you system has previously been used only for development on previous C2000 devices, you may not have the required CCS device files. In CCS click on "Help, Check for updates" and follow the dialog boxes to update your CCS installation.

"I cannot connect to the target"

This is most often times caused by either a bad target configuration, or simply the emulator being physically disconnected. If you are unable to connect to a target check the following things:

1. Ensure the target configuration is correct for the device you have.
2. Ensure the emulator is plugged in to both the computer and the device to be debugged.
3. Ensure that the target device is powered.

"I cannot load code"

This is typically caused by an error in the GEL script or improperly linked code. If you are having trouble loading code, check the linker command files and maps to ensure that they match the device memory map. If these appear correct, there is a chance there is something wrong in one of your GEL scripts.

"When a core gets an interrupt, it faults"

Ensure that the interrupt vector table is where the interrupt controller thinks it is. On both cores the interrupt vector table may be mapped to either RAM or flash. Please ensure that your vector table is where the interrupt controller thinks it is.

"When the CPU1 comes up, it is not fresh out of reset"

F28P65x devices support several boot modes, several of which allow program code to be loaded into and executed out of RAM via one of the device many serial peripherals. If the boot mode pins are in the wrong state at power up, one of these peripheral boot modes may be entered accidentally before the debugger is connected. This leaves the chip in an unclean state with potentially several of the peripherals configured as well as the interrupt vector table setup. If you are seeing strange behavior check to ensure that the "Boot to Flash" or "Boot to RAM" boot mode is selected.

"\"Fapi_Error_InvalidHclkValue\" is returned afte execution of Fapi_setActiveFlashBank(Fapi_FlashBank0) function." Occurs when using the Flash APIs in the code.

Please ensure that the correct frequency is passed as an input to the Fapi_initializeAPI function and the wait

states are correctly configured.

3 Interrupt Service Routine Priorities

Interrupt Hardware Priority Overview 37
f28p65x PIE Interrupt Priorities 38
Software Prioritization of Interrupts - The Example 39

3.1 Interrupt Hardware Priority Overview

With the PIE block enabled, the interrupts are prioritized in hardware by default as follows:
Global Priority (CPU Interrupt level):

CPU Interrupt	Hardware Priority
Reset	1(Highest)
INT1	5
INT2	6
INT3	7
INT4	8
INT5	9
INT6	10
INT7	11
...	...
INT12	16
INT13	17
INT14	18
DLOGINT	19(Lowest)
RTOSINT	20
reserved	2
NMI	3
ILLEGAL	-
USER1	-(Software Interrupts)
USER2	-
...	...

CPU Interrupts INT1 - INT14, DLOGINT and RTOSINT are maskable interrupts. These interrupts can be enabled or disabled by the CPU Interrupt enable register (IER).

Group Priority (PIE Level):
If the Peripheral Interrupt Expansion (PIE) block is enabled, then CPU interrupts INT1 to INT12 are connected to the PIE. This peripheral expands each of these 12 CPU interrupt into 8 interrupts. Thus the total possible number of available interrupts in the PIE is 96. Note, not all of the 96 are used on a 2803x device.

Each of the PIE groups has its own interrupt enable register (PIEIERx) to control which of the 8 interrupts (INTx.1 - INTx.8) are enabled and permitted to issue an interrupt.

CPU Interrupt	PIE Group	PIE Interrupts							
		Highest——————Hardware Priority Within the Group——————-Lowest							
INT1	1	INT1.1	INT1.2	INT1.3	INT1.4	INT1.5	INT1.6	INT1.7	INT1.8
INT2	2	INT2.1	INT2.2	INT2.3	INT2.4	INT2.5	INT2.6	INT2.7	INT2.8
INT3	3	INT3.1	INT3.2	INT3.3	INT3.4	INT3.5	INT3.6	INT3.7	INT3.8
... etc ...									
... etc ...									
INT12	12	INT12.1	INT12.2	INT12.3	INT12.4	INT12.5	INT12.6	INT12.7	INT4.8

Table 3.1: PIE Group Hardware Priority

3.2 PIE Interrupt Priorities

The PIE block is organized such that the interrupts are in a logical order. Interrupts that typically require higher priority, are organized higher up in the table and will thus be serviced with a higher priority by default.

The interrupts in a control subsystem can be categorized as follows (ordered highest to lowest priority):

1. **Non-Periodic, Fast Response**
   These are interrupts that can happen at any time and when they occur, they must be serviced as quickly as possible. Typically these interrupts monitor an external event.

   On the f28p65x devices, such interrupts are allocated to the first few interrupts within PIE Group 1 and PIE Group 2. This position gives them the highest priority within the PIE group. In addition, Group 1 is multiplexed into the CPU interrupt INT1. CPU INT1 has the highest hardware priority. PIE Group 2 is multiplexed into the CPU INT2 which is the 2nd highest hardware priority.

2. **Periodic, Fast Response**
   These interrupts occur at a known period, and when they do occur, they must be serviced as quickly as possible to minimize latency. The A/D converter is one good example of this. The A/D sample must be processed with minimum latency.

   On the f28p65x devices, such interrupts are allocated to the group 1 in the PIE table. Group 1 is multiplexed into the CPU INT1. CPU INT1 has the highest hardware priority

3. **Periodic**
   These interrupts occur at a known period and must be serviced before the next interrupt. Some of the PWM interrupts are an example of this. Many of the registers are shadowed, so the user has the full period to update the register values.

   In the f28p65x device's PIE modules, such interrupts are mapped to group 2 - group 5. These groups are multiplexed into CPU INT3 to INT5 (the ePWM and eCAP), which are the next lowest hardware priority.

4. **Periodic, Buffered**
   These interrupts occur at periodic events, but are buffered and hence the processor need only service such interrupts when the buffers are ready to filled/emptied. All of the serial ports (SCI / SPI / I2C / CAN) either have FIFOs or multiple mailboxes such that the CPU has plenty

of time to respond to the events without fear of losing data.

In the f28p65x device, such interrupts are mapped to INT6, INT8, and INT9, which are the next lowest hardware priority.

3.3 Software Prioritization of Interrupts

The user will probably find that the PIE interrupts are organized where they should be for most applications. However, some software prioritization may still be required for some applications.

Recall that the basic software priority scheme on the C28x works as follows:

- **Global Priority**
  This priority can be managed by manipulating the CPU IER register. This register controls the 16 maskable CPU interrupts (INT1 - INT16).

- **Group Priority**
  This can be managed by manipulating the PIE block interrupt enable registers (PIEIERx). There is one PIEIERx per group and each control the 8-interrupts multiplexed within that group.

The F28 software prioritization of interrupt example demonstrates how to configure the Global priority (via IER) and group priority (via PIEIERx) within an ISR in order to change the interrupt service priority based on user assigned levels. The steps required to do this are:

1. **Set the global priority**
   Modify the IER register to allow CPU interrupts with a higher user priority to be serviced.

2. **Set the Group priority**
   Modify the appropriate PIEIERx register to allow group interrupts with a higher user set priority to be serviced.

3. **Enable interrupts**

The software prioritized interrupts example provides a method using mask values that are configured during compile time to allow you to manage this easily.

To setup software prioritization for the example, the user must first assign the desired global priority levels and group priority levels.

This can be done as follows:

1. *User assigns global priority levels*
   INT1PL - INT16PL
   These values are used to assign a priority level to each of the 16 interrupts controlled by the CPU IER register. A value of 1 is the highest priority while a value of 16 is the lowest. More then one interrupt can be assigned the same priority level. In this case the default hardware priority would determine which would be serviced first. A priority of 0 is used to indicate that the interrupt is not used.

2. *User assigns PIE group priority levels*
   GxyPL (where x = PIE group number 1 - 12 and y = interrupt number 1 - 8)

   These values are used to assign a priority level to each of the 8 interrupts within a PIE group. A value of 1 is the highest priority while a value of 8 is the lowest. More then one interrupt can be assigned the same priority level. In this case the default hardware priority would determine which would be serviced first. A priority of 0 is used to indicate that the interrupt is not used.

Once the user has defined the global and group priority levels, the compiler will generate mask values that can be used to change the IER and PIEIERx registers within each ISR. In this manner the interrupt software prioritization will be changed. The masks that are generated at compile time are:

- **IER mask values**
  MINT1 - MINT16

  The user assigned INT1PL - INT16PL values are used at compile time to calculate an IER mask for each CPU interrupt. This mask value will be used within an ISR to allow CPU interrupts with a higher priority to interrupt the current ISR and thus be serviced at a higher priority level.
- **PIEIERxy mask values**
  MGxy (where x = PIE group number 1 - 12 and y = interrupt number 1 - 8)

  The assigned group priority levels (GxyPL) are used at compile time to calculate PIEIERx masks for each PIE group. This mask value will be used within an ISR to allow interrupts within the same group that have a higher assigned priority to interrupt the current ISR and thus be serviced at a higher priority level.

3.3.1 Using the IER/PIEIER Mask Values

Within an interrupt service routine, the global and group priority can be changed by software to allow other interrupts to be serviced. The procedure for setting an interrupt priority using the mask values created in the F28_SWPrioritizedIsrLevels.h is the following:

1. **Set the global priority**
   - Modify IER to allow CPU interrupts from the same PIE group as the current ISR.
   - Modify IER to allow CPU interrupts with a higher user defined priority to be serviced.
2. **Set the group priority**
   - Save the current PIEIERx value to a temporary register.
   - The PIEIER register is then set to allow interrupts with a higher priority within a PIE group to be serviced.
3. **Enable interrupts**
   - Enable all PIE interrupt groups by writing all 1's to the PIEACK register
   - Enable global interrupts by clearing INTM
4. **Execute ISR.** Interrupts that were enabled in steps 1-3 (those with a higher software priority) will be allowed to interrupt the current ISR and thus be serviced first.
5. **Restore the PIEIERx register**
6. **Exit**

3.3.2 Example Code

The sample C code below shows an EV-A Comparator 1 Interrupt service routine software prioritization written in C. This interrupt is connected to PIE group 2 interrupt 1.

```
// Connected to PIEIER2_1 (use MINT2 and MG21 masks):
#if (G21PL != 0)
interrupt void EPWM1_TZINT_ISR(void)   // EPWM1 Trip Zone
{
    // Set interrupt priority:
    volatile Uint16 TempPIEIER = PieCtrlRegs.PIEIER2.all;
    IER |= M_INT2;
    IER &= MINT2;                      // Set "global" priority
    PieCtrlRegs.PIEIER2.all &= MG21;   // Set "group"  priority
    PieCtrlRegs.PIEACK.all = 0xFFFF;   // Enable PIE interrupts
    asm(" NOP");
    EINT;

    // Insert ISR Code here.......
    // for now just insert a delay
    for(i = 1; i <= 10; i++) {}

    // Restore registers saved:
    DINT;
    PieCtrlRegs.PIEIER2.all = TempPIEIER;

    //  Add ISR to Trace
    ISRTrace[ISRTraceIndex] = 0x0021;
    ISRTraceIndex++;
}
#endif

CMP1INT_ISR:
            ASP
            ADDB    SP,#1
            CLRC    OVM,PAGE0
            MOVW    DP,#0x0033
            MOV     AL,@36
            MOV     *-SP[1],AL
            OR      IER,#0x0002
            AND     IER,#0x0002
            AND     @36,#0x000E
            MOV     @33,#0xFFFF
            CLRC    INTM

            User code goes here...

            SETC    INTM
            MOV     AL,*-SP[1]
            MOV     @36,AL
            SUBB    SP,#1
```

```
                    NASP
                    IRET
```

The interrupt latency is approx 22 cycles.

/*!

4 CPU 1 Bit-field Example Applications

These example applications show how to make use of various peripherals of a F28P65x device. These applications are intended for demonstration and as a starting point for new applications.

All these examples contain two build configurations which allow you to build each project to run from either RAM or Flash. To change how the project is built simply right click on the project and select "Build Configurations". Then, move over to set the active build configuration, either RAM or Flash.

The examples provided are built for controlCARD compatibility.

Because CPU 1 is ultimately in control of the entire F28P65x device and these applications contain no CPU 2 dependencies, these examples may be run completely on their own without any associated CPU2 program.

All of these examples reside in the `device_support/f28p65x/examples/cpu1` subdirectory of the C2000Ware package.

4.1 ADC SOC Software Force (adc_soc_software)

This example converts some voltages on ADCA and ADCB based on a software trigger.

After the program runs, the memory will contain:

- **AdcaResult0** : a digital representation of the voltage on pin A2
- **AdcaResult1** : a digital representation of the voltage on pin A3
- **AdcbResult0** : a digital representation of the voltage on pin B2
- **AdcbResult1** : a digital representation of the voltage on pin B3

Note: The software triggers for the two ADCs happen sequentially, so the two ADCs will run asynchronously.

4.2 ADC ePWM Triggering (adc_soc_epwm)

This example sets up the ePWM to periodically trigger the ADC.

After the program runs, the memory will contain:

- **AdcaResults** : A sequence of analog-to-digital conversion samples from pin A0. The time between samples is determined based on the period of the ePWM timer.

4.3 ADC temperature sensor conversion (adc_soc_epwm_tempsensor)

This example sets up the ePWM to periodically trigger the ADC. The ADC converts the internal connection to the temperature sensor, which is then interpreted as a temperature by calling the GetTemperatureC function.

After the program runs, the memory will contain:

- **sensorSample** : The raw reading from the temperature sensor.
- **sensorTemp** : The interpretation of the sensor sample as a temperature in degrees Celsius.

4.4 ADC Synchronous SOC Software Force (adc_soc_software_sync)

This example converts some voltages on ADCA and ADCB using input 5 of the input X-BAR as a software force. Input 5 is triggered by toggling GPIO0, but any spare GPIO could be used. This method will ensure that both ADCs start converting at exactly the same time.

After the program runs, the memory will contain:

- **AdcaResult0** : a digital representation of the voltage on pin A2
- **AdcaResult1** : a digital representation of the voltage on pin A3
- **AdcbResult0** : a digital representation of the voltage on pin B2
- **AdcbResult1** : a digital representation of the voltage on pin B3

4.5 ADC Continuous Triggering (adc_soc_continuous)

This example sets up the ADC to convert continuously, achieving maximum sampling rate.

After the program runs, the memory will contain:

- **AdcaResults** : A sequence of analog-to-digital conversion samples from pin A0. The time between samples is the minimum possible based on the ADC speed.

4.6 ADC Continuous Conversions Read by DMA (adc_soc_continuous_dma)

This example sets up two ADC channels to convert simultaneously. The results will be transferred by the DMA into a buffer in RAM.

After the program runs, the memory will contain:

- **adcData0** : a digital representation of the voltage on pin A3
- **adcData1** : a digital representation of the voltage on pin B3

4.7 ADC PPB Offset (adc_ppb_offset)

This example software triggers the ADC. Some SOCs have automatic offset adjustment applied by the post-processing block.

After the program runs, the memory will contain:

- **AdcaResult** : a digital representation of the voltage on pin A0
- **AdcaResult_offsetAdjusted** : a digital representation of the voltage on pin A0, minus 100 LSBs of automatically added offset

- **AdcbResult** : a digital representation of the voltage on pin B0
- **AdcbResult_offsetAdjusted** : a digital representation of the voltage on pin B0 plus 100 LSBs of automatically added offset

4.8 ADC PPB Limits (adc_ppb_limits)

This example sets up the ePWM to periodically trigger the ADC. If the results are outside of the defined range, the post-processing block will generate an interrupt.

The default limits are 1000LSBs and 3000LSBs. With VREFHI set to 3.3V, the PPB will generate an interrupt if the input voltage goes above about 2.4V or below about 0.8V.

4.9 ADC PPB Delay Capture (adc_ppb_delay)

This example demonstrates delay capture using the post-processing block.

Two asynchronous ADC triggers are setup:

- ePWM1, with period 2048, triggering SOC0 to convert on pin A0
- ePWM1, with period 9999, triggering SOC1 to convert on pin A1

Each conversion generates an ISR at the end of the conversion. In the ISR for SOC0, a conversion counter is incremented and the PPB is checked to determine if the sample was delayed.

After the program runs, the memory will contain:

- **conversion** : the sequence of conversions using SOC0 that were delayed
- **delay** : the corresponding delay of each of the delayed conversions

4.10 Buffered DAC Enable (buffdac_enable)

This example generates a voltage on the buffered DAC output, DACOUTA/ADCINA0 (HSEC Pin 9) and uses the default DAC reference setting of VDAC.

When the DAC reference is set to VDAC, an external reference voltage must be applied to the VDAC pin. This can accomplished by connecting a jumper wire from 3.3V to ADCINB0 (HSEC pin 12).

4.11 Buffered DAC Sine DMA (buffdac_sine_dma)

This example generates a sine wave on the buffered DAC output using the DMA to transfer sine values stored in a sine table in GSRAM to DACVALS, DACOUTA/ADCINA0 (HSEC Pin 9) and uses the default DAC reference setting of VDAC.

When the DAC reference is set to VDAC, an external reference voltage must be applied to the VDAC pin. This can accomplished by connecting a jumper wire from 3.3V to ADCINB0 (HSEC pin 12).

Run the included .js file to add the watch variables.

outputFreq_hz = (samplingFreq_hz/SINE_TBL_SIZE)∗tableStep

The generated waveform can be adjusted with the following variables/macros but require recompile:

- **waveformGain** : Adjust the magnitude of the waveform. Range is from 0.0 to 1.0. The default value of 0.8003 centers the waveform within the linear range of the DAC.
- **waveformOffset** : Adjust the offset of the waveform. Range is from -1.0 to 1.0. The default value of 0 centers the waveform.
- **samplingFreq_hz** : Adjust the rate at which the DAC is updated. Range - Bounded by cpu timer maximum interrupt rate.
- **tableStep** : The sine table step size. Range - Bounded by sine table size, should be much less than sine table size to have good resolution.
- **REFERENCE** : The reference for the DAC. Range - REFERENCE_VDAC, REFERENCE_VREF
- **CPUFREQ_MHZ** : The cpu frequency. This does not set the cpu frequency. Range - See device data manual
- **DAC_NUM** : The DAC to use. Range - DACA, DACC

4.12 DMA GSRAM Transfer (dma_gsram_transfer)

This example uses one DMA channel to transfer data from a buffer in RAMGS0 to a buffer in RAMGS1. The example sets the DMA channel PERINTFRC bit repeatedly until the transfer of 16 bursts (where each burst is 8 16-bit words) has been completed. When the whole transfer is complete, it will trigger the DMA interrupt.

Watch Variables

- **sdata** - Data to send
- **rdata** - Received data

4.13 EPWM Trip Zone Module (epwm_trip_zone)

This example configures ePWM1 and ePWM2 as follows

- ePWM1 has TZ1 as one shot trip source
- ePWM2 has TZ1 as cycle by cycle trip source

Initially tie TZ1 high. During the test, monitor ePWM1 or ePWM2 outputs on a scope. Pull TZ1 low to see the effect.

External Connections

- EPWM1A is on GPIO0
- EPWM2A is on GPIO2
- TZ1 is on GPIO12

This example also makes use of the Input X-BAR. GPIO12 (the external trigger) is routed to the input X_BAR, from which it is routed to TZ1.

The TZ-Event is defined such that EPWM1A will undergo a One-Shot Trip and EPWM2A will undergo a Cycle-By-Cycle Trip.

4.14 EPWM Action Qualifier (epwm_updown_aq)

This example configures ePWM1, ePWM2, ePWM3 to produce an waveform with independent modulation on EPWMxA and EPWMxB.

The compare values CMPA and CMPB are modified within the ePWM's ISR.

The TB counter is in up/down count mode for this example.

View the EPWM1A/B(PA0_GPIO0 & PA1_GPIO1), EPWM2A/B(PA2_GPIO2 & PA3_GPIO3) and EPWM3A/B(PA4_GPIO4 & PA5_GPIO5) waveforms via an oscilloscope.

4.15 EPWM Action Qualifier (epwm_up_aq)

This example configures ePWM1, ePWM2, ePWM3 to produce an waveform with independent modulation on EPWMxA and EPWMxB.

The compare values CMPA and CMPB are modified within the ePWM's ISR.

The TB counter is in up count mode for this example.

View the EPWM1A/B(PA0_GPIO0 & PA1_GPIO1), EPWM2A/B(PA2_GPIO2 & PA3_GPIO3) and EPWM3A/B(PA4_GPIO4 & PA5_GPIO5) waveforms via an oscilloscope.

4.16 EPWM dead band control (epwm_deadband)

During the test, monitor ePWM1, ePWM2, and/or ePWM3 outputs on a scope.

- ePWM1A is on GPIO0
- ePWM1B is on GPIO1
- ePWM2A is on GPIO2
- ePWM2B is on GPIO3
- ePWM3A is on GPIO4
- ePWM3B is on GPIO5

This example configures ePWM1, ePWM2 and ePWM3 for:

- Count up/down
- Deadband

3 Examples are included:

- ePWM1: Active low PWMs
- ePWM2: Active low complementary PWMs
- ePWM3: Active high complementary PWMs

Each ePWM is configured to interrupt on the 3rd zero event. When this happens the deadband is modified such that 0 <= DB <= DB_MAX. That is, the deadband will move up and down between 0 and the maximum value.

View the EPWM1A/B, EPWM2A/B and EPWM3A/B waveforms via an oscilloscope

4.17 HRPWM Duty SFO (hrpwm_duty_sfo_v8)

This program requires the header files, which include the following files required for this example: sfo_v8.h and SFO_v8_fpu_lib_build_c28_eabi.lib

Monitor ePWM1-ePWM7 A/B pins on an oscilloscope. DESCRIPTION:

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up-Down count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

==========
NOTE: For more information on using the SFO software library, see the High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

1. $**$!!IMPORTANT!!$**$
2. Run this example at maximum SYSCLKOUT
3. Activate Real time mode
4. Run the code
5. Watch ePWM A / B channel waveforms on a Oscilloscope
6. In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA & ePWMxB output with HRPWM capabilities (default) Observe the period/frequency of the waveform changes in fine MEP steps
7. In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA & ePWMxB output without HRPWM capabilities Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

4.18 HRPWM SFO Test (hrpwm_prdupdown_sfo_v8)

This program requires the header files, which include the following files required for this example: sfo_v8.h and SFO_v8_fpu_lib_build_c28_eabi.lib

Monitor ePWM1-ePWM7 A/B pins on an oscilloscope. DESCRIPTION:

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up-Down count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

==========
NOTE: For more information on using the SFO software library, see the High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

1. **!!IMPORTANT!!**
2. Run this example at maximum SYSCLKOUT
3. Activate Real time mode
4. Run the code
5. Watch ePWM A / B channel waveforms on an Oscilloscope
6. In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA & ePWMxB output with HRPWM capabilities (default) Observe the period/frequency of the waveform changes in fine MEP steps
7. In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA & ePWMxB output without HRPWM capabilities Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

4.19 External Interrupts (ExternalInterrupt)

This program sets up GPIO0 as XINT1 and GPIO1 as XINT2. Two other GPIO signals are used to trigger the interrupt (GPIO30 triggers XINT1 and GPIO31 triggers XINT2). The user is required to externally connect these signals for the program to work properly.

XINT1 input is synced to SYSCLKOUT.

XINT2 has a long qualification - 6 samples at 510∗SYSCLKOUT each.

GPIO34 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope.

Each interrupt is fired in sequence - XINT1 first and then XINT2

External Connections

- Connect GPIO30 to GPIO0. GPIO0 will be assigned to XINT1
- Connect GPIO31 to GPIO1. GPIO1 will be assigned to XINT2

Monitor GPIO34 with an oscilloscope. GPIO34 will be high outside of the ISRs and low within each ISR.

Watch Variables

- Xint1Count for the number of times through XINT1 interrupt
- Xint2Count for the number of times through XINT2 interrupt
- LoopCount for the number of times through the idle loop

4.20 SDFM Filter Sync CPU

In this example, SDFM filter data is read by CPU in SDFM ISR routine. The SDFM configuration is shown below:

- SDFM1 is used in this example. For using SDFM2, few modifications would be needed in the example.

Input control mode selected - MODE0

- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - HLT = 0x7FFF (Higher threshold setting)
  - LLT = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Main Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled

- All the 4 filter will generate interrupt when a new filter data is available.

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **Filter1_Result** - Output of filter 1
- **Filter2_Result** - Output of filter 2
- **Filter3_Result** - Output of filter 3
- **Filter4_Result** - Output of filter 4

4.21 SDFM Filter Sync CLA

In this example, SDFM filter data is read by CLA in Cla1Task1. The SDFM configuration is shown below:

- SDFM1 used in this example.For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - HLT = 0x7FFF (Higher threshold setting)
  - LLT = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Main Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled
  - All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **Filter1_Result** - Output of filter 1
- **Filter2_Result** - Output of filter 2
- **Filter3_Result** - Output of filter 3
- **Filter4_Result** - Output of filter 4

4.22 SDFM Filter Sync DMA

In this example, SDFM filter data is read by DMA. The SDFM configuration is shown below:

- SDFM1 used in this example. For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - HLT = 0x7FFF (Higher threshold setting)
  - LLT = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Main Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled
  - All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31

- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **Filter1_Result** - Output of filter 1
- **Filter2_Result** - Output of filter 2
- **Filter3_Result** - Output of filter 3
- **Filter4_Result** - Output of filter 4

4.23 SDFM PWM Sync

In this example, SDFM filter data is read by CPU in SDFM ISR routine. The SDFM configuration is shown below:

- SDFM1 is used in this example. For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - HLT = 0x7FFF (Higher threshold setting)
  - LLT = 0x0000(Lower threshold setting)

Data filter settings

- All the 4 filter modules enabled
- Sinc3 filter selected
- OSR = 256
- All the 4 filters are synchronized by using PWM (Main Filter enable bit)
- Filter output represented in 16 bit format
- In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256

Interrupt module settings for SDFM filter

- All the 4 higher threshold comparator interrupts disabled
- All the 4 lower threshold comparator interrupts disabled
- All the 4 modulator failure interrupts disabled
- All the 4 filter will generate interrupt when a new filter data is available **External Connections**

SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31

- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **Filter1_Result** - Output of filter 1
- **Filter2_Result** - Output of filter 2
- **Filter3_Result** - Output of filter 3
- **Filter4_Result** - Output of filter 4

5 Dual Core Bit-field Example Applications

These example applications show how to make use of f28p65x device functions which span both the CPU 1 and CPU 2. All of these examples contain two example projects: one for CPU 1 and one for CPU 2.

Like the CPU1 only projects, these projects also contain different build configurations for RAM and Flash builds. All of the projects contain RAM and Flash build configurations with debugger support.

The examples provided are built for controlCARD compatibility.

To run one of these examples after compiling it, load the appropriate programs on each of the two cores. Then, for more example specific instructions please refer to the documentation regarding the example you wish to run on the following pages or in the comments of the example sources.

All of these examples can be found in the

`device_support/f28p65x/examples/dual` subdirectory of the C2000Ware package.

5.1 DMA Transfer Shared Peripheral

This example shows how to initiate a DMA transfer on CPU1 from a shared peripheral which is owned by CPU2. In this specific example, a timer ISR is used on CPU2 to initiate a SPI transfer which will trigger the CPU1 DMA. CPU1's DMA will then in turn update the EPWM1 CMPA value for the PWM which it owns. The PWM output can be observed on the GPIO pins configured in the InitEPwm1Gpio() function.

Watch Pins

- GPIO0 and GPIO1 - ePWM output can be viewed with oscilloscope

5.2 Shared RAM management (CPU1)

This example shows how to assign shared RAM for use by both the CPU02 and CPU01 core. Shared RAM regions are defined in both the CPU02 and CPU01 linker files. In this example GS0 and GS4 are assigned to/owned by CPU02. The remaining shared RAM regions are owned by CPU01. In this example:

A pattern is written to c1_r_w_array and then IPC flag is sent to notify CPU02 that data is ready to be read. CPU02 then reads the data from c2_r_array and writes a modified pattern to c2_r_w_array. Once CPU02 acknowledges the IPC flag to , CPU01 reads the data from c1_r_array and compares with expected result.

A Timed ISR is also serviced in both CPUs. The ISRs are copied into the shared RAM region owned by the respective CPUs. Each ISR toggles a GPIO. Watch GPIO31 and GPIO34 on oscilloscope. If using the control card watch LED1 and LED2 blink at different rates.

Following are the memory allocation details of CPU Timer interrupt ISRs & read(R)/read write(RW) arrays in CPU1 & CPU2 as configured in the example.

- c1_r_w_array[] is mapped to shared RAM GS1
- c1_r_array[] is mapped to shared RAM GS0
- c2_r_array[] is mapped to shared RAM GS1
- c2_r_w_array[] is mapped to shared RAM GS0
- cpu_timer0_isr in CPU02 is copied to shared RAM GS4 , toggles GPIO31

- cpu_timer0_isr in CPU01 is copied to shared RAM GS3 , toggles GPIO34

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

Watch Variables

- error Indicates that the data written is not correctly received by the other CPU.

5.3 Shared RAM management (CPU2)

This example shows how to assign shared RAM for use by both the CPU02 and CPU01 core. Shared RAM regions are defined in both the CPU02 and CPU01 linker files. In this example GS0 and GS4 are assigned to/owned by CPU02. The remaining shared RAM regions are owned by CPU01. In this example:

A pattern is written to c1_r_w_array and then IPC flag is sent to notify CPU02 that data is ready to be read. CPU02 then reads the data from c2_r_array and writes a modified pattern to c2_r_w_array. Once CPU02 acknowledges the IPC flag to , CPU01 reads the data from c1_r_array and compares with expected result.

A Timed ISR is also serviced in both CPUs. The ISRs are copied into the shared RAM region owned by the respective CPUs. Each ISR toggles a GPIO. Watch GPIO31 and GPIO34 on oscilloscope. If using the control card watch LED1 and LED2 blink at different rates.

Following are the memory allocation details of CPU Timer interrupt ISRs & read(R)/read write(RW) arrays in CPU1 & CPU2 as configured in the example.

- c1_r_w_array[] is mapped to shared RAM GS1
- c1_r_array[] is mapped to shared RAM GS0
- c2_r_array[] is mapped to shared RAM GS1
- c2_r_w_array[] is mapped to shared RAM GS0
- cpu_timer0_isr in CPU02 is copied to shared RAM GS4 , toggles GPIO31
- cpu_timer0_isr in CPU01 is copied to shared RAM GS3 , toggles GPIO34

Watch Variables

- error Indicates that the data written is not correctly received by the other CPU.

6 C28x Driver Library Example Applications

These example applications show how to make use of various peripherals of a F28P65x device. These applications are intended for demonstration and as a starting point for new applications.

All these examples are setup using the Code Composer Studio (CCS) "projectspec" format. Upon importing the "projectspec", the example project will be generated in the CCS workspace with copies of the source and header files included.

All these examples contain two build configurations which allow you to build each project to run from either RAM or Flash. To change how the project is built simply right click on the project and select "Build Configurations". Then, move over to set the active build configuration, either RAM or Flash.

The examples provided are built for controlCARD compatibility.

There may be a few examples which need either an external hardware or device not present on the controlCARD like:

- I2C communication with eeprom
- SPI communication with eeprom
- EMIF accessing external memory
- DCC clock failure detection

Because CPU 1 is ultimately in control of the entire F28P65x device and these applications contain no CPU 2, these examples may be run completely on their own without any associated CPU2 .

All of these examples reside in the `driverlib/f28p65x/examples/c28x` subdirectory of the C2000Ware package.

Note that in all the examples, Device_init function assumes that the XTAL frequency is 25MHz. If a 20MHz XTAL is used, please add a predefined symbol "USE_20MHZ_XTAL" in your CCS project. If a different XTAL is used, you need to update the macro DEVICE_SETCLOCK_CFG in device.h file accordingly.

6.1 ADC ePWM Triggering Multiple SOC

This example sets up ePWM1 to periodically trigger a set of conversions on ADCA and ADCC. This example demonstrates multiple ADCs working together to process of a batch of conversions using the available parallelism across multiple ADCs.

ADCA Interrupt ISRs are used to read results of both ADCA and ADCC.

External Connections

- A0, A1, A2 and C2, C3, C4 pins should be connected to signals to be converted.

Watch Variables

- **adcAResult0** - Digital representation of the voltage on pin A0
- **adcAResult1** - Digital representation of the voltage on pin A1
- **adcAResult2** - Digital representation of the voltage on pin A2
- **adcCResult0** - Digital representation of the voltage on pin C2
- **adcCResult1** - Digital representation of the voltage on pin C3
- **adcCResult2** - Digital representation of the voltage on pin C4

6.2 ADC Burst Mode

This example sets up ePWM1 to periodically trigger ADCA using burst mode. This allows for different channels to be sampled with each burst.

Each burst triggers 3 conversions. A0 and A1 are part of every burst while the third conversion rotates between A2, A3, and A4. This allows high importance signals to be sampled at high speed while lower priority signals can be sampled at a lower rate.

ADCA Interrupt ISRs are used to read results for ADCA.

External Connections

- A0, A1, A2, A3, A4

Watch Variables

- **adcAResult0** - Digital representation of the voltage on pin A0
- **adcAResult1** - Digital representation of the voltage on pin A1
- **adcAResult2** - Digital representation of the voltage on pin A2
- **adcAResult3** - Digital representation of the voltage on pin A3
- **adcAResult4** - Digital representation of the voltage on pin A4

6.3 ADC Burst Mode Oversampling

This example is an ADC oversampling example implemented with software. The ADC SOCs are configured in burst mode, triggered by the ePWM SOC A event trigger.

External Connection

- A2

Watch Variables

- **lv_results** - Array of digital values measured on pin A2 (oversampling is configured by Oversampling_Amount)

6.4 ADC SOC Oversampling

This example sets up ePWM1 to periodically trigger a set of conversions on ADCA including multiple SOCs that all convert A2 to achieve oversampling on A2.

ADCA Interrupt ISRs are used to read results of ADCA.

External Connections

- A0, A1, A2 should be connected to signals to be converted.

Watch Variables

- **adcAResult0** - Digital representation of the voltage on pin A0
- **adcAResult1** - Digital representation of the voltage on pin A1
- **adcAResult2** - Digital representation of the voltage on pin A2

6.5 ADC PPB PWM trip (adc_ppb_pwm_trip)

This example demonstrates EPWM tripping through ADC limit detection PPB block. ADCAINT1 is configured to periodically trigger the ADCA channel 2 post initial software forced trigger. The limit detection post-processing block(PPB) is configured and if the ADC results are outside of the defined range, the post-processing block will generate an ADCxEVTy event. This event is configured as EPWM trip source through configuring EPWM XBAR and corresponding EPWM's trip zone and digital compare sub-modules. The example showcases

- one-shot
- cycle-by-cycle
- and direct tripping of PWMs through ADCAEVT1 source via Digital compare submodule.

The default limits are 0LSBs and 3600LSBs. With VREFHI set to 3.3V, the PPB will generate a trip event if the input voltage goes above about 2.9V.

External Connections

- A2 should be connected to a signal to convert
- Observe the following signals on an oscilloscope
  - ePWM1(GPIO0 - GPIO1)
  - ePWM2(GPIO2 - GPIO3)
  - ePWM3(GPIO4 - GPIO5)
-

Watch Variables

- adcA2Results - digital representation of the voltage on pin A2

6.6 ADC Software Triggering

This example converts some voltages on ADCA and ADCC based on a software trigger.

The ADCC will not convert until ADCA is complete, so the ADCs will not run asynchronously. However, this is much less efficient than allowing the ADCs to convert synchronously in parallel (for example, by using an ePWM trigger).

External Connections

- A0, A1, C2, and C3 should be connected to signals to convert

Watch Variables

- **myADC0Result0** - Digital representation of the voltage on pin A0

- **myADC0Result1** - Digital representation of the voltage on pin A1
- **myADC1Result0** - Digital representation of the voltage on pin C2
- **myADC1Result1** - Digital representation of the voltage on pin C3

6.7 ADC ePWM Triggering

This example sets up ePWM1 to periodically trigger a conversion on ADCA.

External Connections

- A0 should be connected to a signal to convert

Watch Variables

- **myADC0Results** - A sequence of analog-to-digital conversion samples from pin A0. The time between samples is determined based on the period of the ePWM timer.

6.8 ADC Temperature Sensor Conversion

This example sets up the ePWM to periodically trigger the ADC. The ADC converts the internal connection to the temperature sensor, which is then interpreted as a temperature by calling the ADC_getTemperatureC() function.

Watch Variables

- **sensorSample** - The raw reading from the temperature sensor
- **sensorTemp** - The interpretation of the sensor sample as a temperature in degrees Celsius.

6.9 ADC Synchronous SOC Software Force (adc_soc_software_sync)

This example converts some voltages on ADCA and ADCC using input 5 of the input X-BAR as a software force. Input 5 is triggered by toggling GPIO0, but any spare GPIO could be used. This method will ensure that both ADCs start converting at exactly the same time.

External Connections

- A2, A3, C2, C3 pins should be connected to signals to convert

Watch Variables

- **myADC0Result0** : a digital representation of the voltage on pin A2
- **myADC0Result1** : a digital representation of the voltage on pin A3
- **myADC1Result0** : a digital representation of the voltage on pin C2
- **myADC1Result1** : a digital representation of the voltage on pin C3

6.10 ADC Continuous Triggering (adc_soc_continuous)

This example sets up the ADC to convert continuously, achieving maximum sampling rate.

External Connections

- A0 pin should be connected to signal to convert

Watch Variables

- **adcAResults** - A sequence of analog-to-digital conversion samples from pin A0. The time between samples is the minimum possible based on the ADC speed.

6.11 ADC Continuous Conversions Read by DMA (adc_soc_continuous_dma)

This example sets up two ADC channels to convert simultaneously. The results will be transferred by the DMA into a buffer in RAM.

External Connections

- A3 & C3 pins should be connected to signals to convert

Watch Variables

- **myADC0DataBuffer** : a digital representation of the voltage on pin A3
- **myADC1DataBuffer** : a digital representation of the voltage on pin C3

6.12 ADC PPB Offset (adc_ppb_offset)

This example software triggers the ADC. Some SOCs have automatic offset adjustment applied by the post-processing block. After the program runs, the memory will contain ADC & post-processing block(PPB) results.

External Connections

- A2, C2 pins should be connected to signals to convert

Watch Variables

- **myADC0Result** : a digital representation of the voltage on pin A2
- **myADC0PPBResult** : a digital representation of the voltage on pin A2, minus 100 LSBs of automatically added offset
- **myADC1Result** : a digital representation of the voltage on pin C2
- **myADC1PPBResult** : a digital representation of the voltage on pin C2 plus 100 LSBs of automatically added offset

6.13 ADC PPB Limits (adc_ppb_limits)

This example sets up the ePWM to periodically trigger the ADC. If the results are outside of the defined range, the post-processing block will generate an interrupt.

The default limits are 1000LSBs and 3000LSBs. With VREFHI set to 3.3V, the PPB will generate an interrupt if the input voltage goes above about 2.4V or below about 0.8V.

External Connections

- A0 should be connected to a signal to convert

Watch Variables

- None

6.14 ADC PPB Delay Capture (adc_ppb_delay)

This example demonstrates delay capture using the post-processing block.

Two asynchronous ADC triggers are setup:

- ePWM1, with period 2048, triggering SOC0 to convert on pin A0
- ePWM2, with period 9999, triggering SOC1 to convert on pin A2

Each conversion generates an ISR at the end of the conversion. In the ISR for SOC0, a conversion counter is incremented and the PPB is checked to determine if the sample was delayed.

After the program runs, the memory will contain:

- **conversion** : the sequence of conversions using SOC0 that were delayed
- **delay** : the corresponding delay of each of the delayed conversions

6.15 BGCRC CPU Interrupt Example

This example demonstrates how to configure and trigger BGCRC from the CPU. BGCRC module is configured for 1 KB of GS0 RAM which is programmed with a known data. The pre-computed CRC value is used as the golden CRC value. Interrupt is generated once the computation is done and checks if no error flags are raised Calculation uses the 32-bit polynomial 0x04C11DB7 and seed value 0x00000000.

External Connections

- None.

Watch Variables

- pass - This should be 1.
- runStatus - BGCRC running status. This will be BGCRC_ACTIVE if the module is running, BGCRC_IDLE if the module is idle

6.16 BGCRC Example with Watchdog and Lock

This example demonstrates how to configure and trigger BGCRC from the CPU. It also showcases how to configure the CRC watchdog and lock the registers after configuring the module. The watchdog is used as a diagnostic to check memory test completion within the expected time window. An error signal is generated if the test does not complete in the specified time window.

The module is configured for 1kB of GS0 RAM which is programmed with random data. The golden CRC value for comparison is computed using software method. Interrupt is generated once the computation is done and checks if no error flags are raised. The NMI is enabled and is triggered if an error is detected.

External Connections

- None.

Watch Variables

- pass
- bgcrcDone

6.17 CLA-BGCRC Example in CRC mode

This example demonstrates how to configure and trigger CLABGCRC from the CPU. It also showcases how to configure the CRC watchdog and lock the registers after configuring the module. The watchdog is used as a diagnostic to check memory test completion within the expected time window. An error signal is generated if the test does not complete in the specified time window.

The module is configured for 1kB of CLA ROM memory. The golden CRC value for comparison is computed using software method. Interrupt is generated once the computation is done and checks if no error flags are raised. The NMI is enabled and is triggered if an error is detected.

External Connections

- None.

Watch Variables

- pass
- bgcrcDone

6.18 CLA-BGCRC Example in Scrub Mode

This example demonstrates how to configure and trigger CLA-BGCRC in Scrub mode. In Scrub mode, CRC of data is not compared with the golden CRC. Error check is done using the ECC/Parity logic. It also showcases how to configure the CRC watchdog and lock the registers after configuring the module. The watchdog is used as a diagnostic to check memory test completion within the expected time window. An error signal is generated if the test does not complete in the specified time window.

The module is configured for 256 bytes of CLA ROM memory. Interrupt is generated once the computation is done and checks if no error flags are raised. The NMI is enabled and is triggered if an error is detected.

External Connections

- None.

Watch Variables

- pass
- bgcrcDone

6.19 CAN External Loopback

This example shows the basic setup of CAN in order to transmit and receive messages on the CAN bus. The CAN peripheral is configured to transmit messages with a specific CAN ID. A message is then transmitted once per second, using a simple delay loop for timing. The message that is sent is a 2 byte message that contains an incrementing pattern.

This example sets up the CAN controller in External Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core. Please refer to details of the External Loopback Test Mode in the CAN Chapter in the Technical Reference Manual. Refer to [Programming Examples and Debug Strategies for the DCAN Module](www.ti.com/lit/SPRACE5) for useful information about this example

External Connections

- None.

Watch Variables

- msgCount - A counter for the number of successful messages received
- txMsgData - An array with the data being sent
- rxMsgData - An array with the data that was received

6.20 CAN External Loopback with Interrupts

This example shows the basic setup of CAN in order to transmit and receive messages on the CAN bus. The CAN peripheral is configured to transmit messages with a specific CAN ID. A message is then transmitted once per second, using a simple delay loop for timing. The message that is sent is a 4 byte message that contains an incrementing pattern. A CAN interrupt handler is used to confirm message transmission and count the number of messages that have been sent.

This example sets up the CAN controller in External Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core. Please refer to details of the External Loopback Test Mode in the CAN Chapter in the Technical Reference Manual. Refer to [Programming Examples and Debug Strategies for the DCAN Module](www.ti.com/lit/SPRACE5) for useful information about this example

External Connections

- None.

Watch Variables

- txMsgCount - A counter for the number of messages sent
- rxMsgCount - A counter for the number of messages received
- txMsgData - An array with the data being sent
- rxMsgData - An array with the data that was received
- errorFlag - A flag that indicates an error has occurred

6.21 CAN External Loopback with DMA

This example sets up the CAN module to transmit and receive messages on the CAN bus. The CAN module is set to transmit a 4 byte message internally. An interrupt is used to assert the DMA request line which then triggers the DMA to transfer the received data from the CAN interface register to the receive buffer array. A data check is performed once the transfer is complete.

This example sets up the CAN controller in External Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core. Please refer to details of the External Loopback Test Mode in the CAN Chapter in the Technical Reference Manual. Please refer to the appnote Programming Examples and Debug Strategies for the DCAN Module (www.ti.com/lit/SPRACE5) for useful information about this example

External Connections

- None.

Watch Variables

- txMsgCount - A counter for the number of messages sent
- rxMsgCount - A counter for the number of messages received
- txMsgData - An array with the data being sent
- rxMsgData - An array with the data that was received

6.22 CAN Transmit and Receive Configurations

This example shows the basic setup of CAN in order to transmit or receive messages on the CAN bus with a specific Message ID. The CAN Controller is configured according to the selection of the define.

When the TRANSMIT define is selected, the CAN Controller acts as a Transmitter and sends data to the second CAN Controller connected externally.If TRANMSIT is not defined the CAN Controller acts as a Receiver and waits for message to be transmitted by the External CAN Controller. Refer to [Programming Examples and Debug Strategies for the DCAN Module](www.ti.com/lit/SPRACE5) for useful information about this example

Note:
CAN modules on the device need to be connected to via CAN transceivers.

Hardware Required

- A C2000 board with CAN transceiver.

External Connections

- ControlCARD CANA is on DEVICE_GPIO_PIN_CANTXA (CANTXA)
- and DEVICE_GPIO_PIN_CANRXA (CANRXA)

Watch Variables Transmit

- MSGCOUNT - Adjust to set the number of messages
- txMsgCount - A counter for the number of messages sent
- txMsgData - An array with the data being sent
- errorFlag - A flag that indicates an error has occurred
- rxMsgCount - Has the initial value as No. of Messages to be received and decrements with each message.

6.23 CAN Error Generation Example

This example demonstrates the ways of handling CAN Error conditions It generates the CAN Packets and sends them over GPIO It is looped back externally to be received in CAN module The CAN Interrupt service routine reads the Error status and demonstrates how different Error conditions can be detected

Change ERR_CFG define to the different Error Scenarios and run the example. The corresponding Error Flag will be set in status variable of canaISR() routine. Uses a CPU Timer(Timer 0) for periodic timer interrupt of CANBITRATE uSec On the Timer interrupt it sends the required CAN Frame type with the specified error conditions

Note:
CAN modules on the device need to be connected to via CAN transceivers.

Please refer to the application note titled "Configurable Error Generator for Controller Area Network" at [Configurable Error Generator for Controller Area Network](https://www.ti.com/lit/pdf/spracq3) for further details on this example

External Connections

- ControlCARD GPIOTX_PIN should be connected to
- DEVICE_GPIO_PIN_CANRXA(CANRXA)

Watch Variables Transmit

- status - variable in canaISR for checking error Status

6.24 CAN Remote Request Loopback

This example shows the basic setup of CAN in order to transmit a remote frame and get a response for the remote frame and store it in a receive Object. The CAN peripheral is configured to transmit remote request frame and a remote answer frame messages with a specific CAN ID. Message object 3 is configured to transmit a remote request. Message object 2 is configured as a remote answer object with filter mask such that it accepts remote frame with any message ID and transmit's remote answer with message ID 7 and data length 8. Message object 1 is configured as a received object with filter message ID 7 so as to store the remote answer data transmitted by message object 2.

This example sets up the CAN controller in External Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core. Please refer to details of the External Loopback Test Mode in the CAN Chapter in the Technical Reference Manual.

External Connections

- None.

Watch Variables

- txMsgData - An array with the data being sent
- rxMsgData - An array with the data that was received

6.25 CAN example that illustrates the usage of Mask registers

This example initializes CAN module A for Reception. When a frame with a matching filter criterion is received, the data will be copied in mailbox 1 and LED will be toggled a few times and the code gets ready for the next frame. If a message of any other MSGID is received, an ACK will be provided Completion of reception is determined by polling CAN_NDAT_21 register. No interrupts are used. Refer to [Programming Examples and Debug Strategies for the DCAN Module](www.ti.com/lit/SPRACE5) for useful information about this example

Hardware Required

- An external CAN node that transmits to CAN-A on the C2000 MCU

Watch Variables

- rxMsgCount - A counter for the number of messages received
- rxMsgData - An array with the data that was received

6.26 CLA $arcsine(x)$ using a lookup table (cla_asin_cpu01)

In this example, Task 1 of the CLA will calculate the arcsine of an input argument in the range (-1.0 to 1.0) using a lookup table.

Note that since this example does not use background CLA task, the compile flag cla_background_task is turned off for this project. Set this flag as on to enable background CLA task. The option is available in Project Properties -> C2000 Build -> C2000 Compiler -> Advanced Options -> Runtime Model Options.

Memory Allocation

- CLA1 Math Tables (RAMLS1)
  - CLAasinTable - Lookup table
- CLA1 to CPU Message RAM
  - fResult - Result of the lookup algorithm
- CPU to CLA1 Message RAM
  - fVal - Sample input to the lookup algorithm

Watch Variables

- fVal - Argument to task 1
- fResult - Result of $arcsin(fVal)$

6.27 CLA $arctangent(x)$ using a lookup table (cla_atan_cpu01)

In this example, Task 1 of the CLA will calculate the arctangent of an input argument using a lookup table.

Note that since this example does not use background CLA task, the compile flag cla_background_task is turned off for this project. Set this flag as on to enable background CLA task. The option is available in Project Properties -> C2000 Build -> C2000 Compiler -> Advanced Options -> Runtime Model Options.

Memory Allocation

- CLA1 Math Tables (RAMLS1)
  - CLAatan2Table - Lookup table
- CLA1 to CPU Message RAM
  - fResult - Result of the lookup algorithm
- CPU to CLA1 Message RAM
  - fNum - Numerator of sample input
  - fDen - Denominator of sample input

Watch Variables

- fVal - Argument to task 1
- fResult - Result of $arctan(fVal)$

6.28 CLA background nesting task

This example configures CLA task 1 to be triggered by EPWM1 running at 2 Hz (period = 0.5s). A background task is configured to be triggered by CPU timer running at .5 Hz (period = 2s). CLA task 1 toggles LED1 at the start and end of the task and the background task toggles LED2 at the start and end of the task. Background task will be preempted by Task1 and hence LED1 will be toggling even while LED2 is ON.

Note that the compile flag cla_background_task is turned on in this project. Enabling background task adds additional context save/restore cycles during task switching thus increasing the overall trigger-to-task latency. If the application does not use the background CLA task, it is recommended to turn this flag off for better performance. The option is available in Project Properties -> C2000 Build -> C2000 Compiler -> Advanced Options -> Runtime Model Options.

External Connections

- None

Watch Variables

- None

##

6.29 Controlling PWM output using CLA

This example showcases how to update PWM signal output using CLA. EPWM1 is configured to generate complementary signals on both of its channels of fixed frequency 100 KHz. EPWM4 is configured to trigger a periodic CLA control task of frequency 10 KHz. The CLA task implements a very simple logic to vary the duty of the EPWM1 outputs by increasing it by 0.1 in every iteration and maintaining it in the range of 0.1-0.9. For actual use-cases, the control logic could be modified to much more complex depending upon the application. The other CLA task (CLA task 8) is triggered by software at beginning to initialize the CLA global variables

External Connections

- Observe GPIO0 (EPWM1A) on oscilloscope
- Observe GPIO1 (EPWM1B) on oscilloscope

Watch Variables

- duty

6.30 Just-in-time ADC sampling with CLA

This example showcases how to utilize early-interrupt feature of ADC in combination with the low interrupt response of CLA to enable faster system response and achieve high frequency control loops. EPWM1 is configured to generate a PWM output signal of frequency 1 MHz and this is also used to trigger the ADC sampling at each cycle. ADCA is configured to sample the input on Channel 0 and to generate the early interrupt at the end of S/H + offset cycles. This interrupt is used to trigger the CLA control task. The CLA task implements the control logic to update the duty of the PWM output based on reading the ADC sample data just-in-time i.e. as soon as the ADC results gets latched.The early interrupt feature and low interrupt latency of CLA allows to do some pre-processing as well before reading the ADC data and still completes updating the PWM output before the next interrupts comes in i.e. data read and PWM update is done within a 1 MHz cycle. For illustration purposes, 3-point moving average filter is used to simulate some processing and few steps of the filtering code are done before reading the ADC result which we consider as pre-processing code. The ADC interrupt offset is programmed based on the cycles consumed by the pre-processing code.

The calculation for interrupt offset value is as follows :- -ADC acquisition cycles programmed = 10 SYSCLKS -Conversion time for 12-bit data = 10.5 ADCCLKS = N = 42 SYSCLKS -CLA task trigger to first instruction in Fetch delay = 4 -Let the interrupt offset value be 'x' -The code inside CLA control task before ADC read takes below cycles : Setting up profiling gpio : 3 cycles Pre-processing : 13 cycles Total = 3 + 13 = 16 cycles

As described in device TRM, in order to read just-in-time the total delay before reading ADC should be (N-2) cycles = 40 i.e. : x + 4 + 16 = 40 : x = 20

NOTE :- The optimization is off for this project and the cycles quoted above corresponds to that case.

GPIO2 is used for profiling purposes. GPIO2 is set at the beginning of CLA task 1 and is reset at the end of the task. Thus ON time of GPIO2 indicates the CLA activity. In order to validate the example functionality , observe the GPIO0 (PWM output) and GPIO2 (profiling GPIO) on CRO. The cycles difference between the rising edge of the GPIO0 and GPIO2 indicate the total delay from the time of ADC trigger to setting up of profiling GPIO inside CLA task which should be around 44 cycles (220 ns) based on the above calculation.

External Connections

- Provide constant DC input on ADCA0 for quick validation. GND -> Should observe PWM

output duty = 0.1 3.3V -> Should observe PWM output duty = 0.9 Can also provide analog input in range 0 - 3.3V upto fs / 10 = 100 KHz for observing continuous duty variations

- Observe GPIO0 on oscilloscope
- Observe GPIO2 on oscilloscope

Watch Variables

- None

6.31 Optimal offloading of control algorithms to CLA

This example showcases how to optimally offload the control algorithms from CPU to CLA in order to meet the system requirements. In this example, two control loops are simulated, the faster one (loop1) running at 200 KHz and the slower one (loop2) running at 20 KHz. Loop1 senses the first parameter at ADCA Channel 0, runs the PI controller to achieve the target and contributes to the duty of EPWM1A output with 80% weightage. Loop2 senses the second parameter at ADCB Channel 2, runs the PI controller and contributes to the duty of EPWM1A output with 20% weightage. It is important to note that since these are just software simulated control loops but there is no actual physical process involved and hence updating the duty is not going to have any affect on sampled inputs. ADCA is configured to oversample the first parameter using SOCs 0-3 to suppress the noise and similarly ADCB is used to oversample the second parameter. EPWM4 and EPWM5 are configured to trigger the ADCA and ADCB sampling at loop1 and loop2 frequencies respectively. Once the conversion of all 4 SOCs complete, a CPU ISR or a CLA task is triggered based on the user-configuration. There is also a background task running in the main loop which disables the entire system including PWM output and the control loops when "system_OFF" is set to 1. The system gets enabled again once "system_OFF" is restored back to 0. By default system_OFF is set to 0 but it's value can be updated dynamically by adding it to expression window and writing to it. DCL library is included in the project to make use of optimal PI controllers used in both the loops. User-configurable pre-defined symbol "run_loop1_cla" has been added to the project options in order to specify whether to run the loop1 on C28x or CLA. GPIO2 and GPIO3 are used to profile the execution of loop1 and loop2.

For run_loop1_cla == 0 i.e. both loops running on CPU

-> Loop1 Utilization = ~77.5% (measured using profiling GPIO2) -> Loop2 Utilization = ~6% (measured using profiling GPIO3) -> Background task in a while loop -> Total CPU utilization is greater than Utilization bound (UB) Hence the system is non-schedulable, lower priority task (Loop2) execution never completes (no toggling observed on GPIO3) and also background task never gets chance to execute

For run_loop1_cla == 1 i.e. high frequency control loop (loop1) is offloaded to CLA while loop2 runs on CPU

-> Loop1 Utilization (CLA) = ~73% -> Loop2 Utilization (CPU)= ~6% -> Total CPU utilization has come down to just ~6% Hence the system is perfectly schedulable, no miss happens for any of the loops and offloading of loop1 to CLA saves CPU bandwidth to execute background tasks as well

For quick inspection of the example functionality, constant DC HIGH/LOW inputs can be provided to the analog channels instead of varying analog voltages. The target value for both the loops are set as some intermediate value i.e. 3500 corresponds to ~2.8V. Now since the sensed inputs are constant and not same as target so the controller outputs will get saturated soon to either 1 or 0. Thus the "duty" variable can take only fixed values based on the equations used in the loops. Infact the duty output would be very intutive, for instance if both inputs are LOW(GND), the controller will try to produce the maximum duty as the target is higher than sensed value hence the duty should be 1.0(0.2 + 0.8) but will get saturated to 0.9(the maximum value defined). Similarly if both inputs are made HIGH, the duty will be 0.1 (the minimum saturation value defined). The final duty table is shown below :

External Connections

- Observe GPIO2 (Loop1 Profiling) on oscilloscope
- Observe GPIO3 (Loop2 Profiling) on oscilloscope
- Observe GPIO0 (EPWM1A Output) on oscilloscope
- Provide constant HIGH(3.3V)/LOW(0V) on both ADCA Ch0 and ADCB Ch2 for quick validation, the following duty value should be observable at EPWM1A for various combinations if the system is perfectly schedulable i.e. both loops gets chance to execute properly :-

A0 B2 duty GND GND 0.9 3.3V GND 0.2 GND 3.3V 0.8 3.3V 3.3V 0.1

Note :- The optimization is OFF for this project and all the profiling data quoted above corresponds to this case.

6.32 Handling shared resources across C28x and CLA

This example showcases how to handle shared resource challenges across C28x and CLA. As the peripherals are shared between CLA and the CPU, overlapping read-modify-write to the registers by them can lead to data race conditions ultimately leading to data violation or incorrect functionality. In this example, CPU ISR and CLA tasks runs independently. CPU ISR gets triggered by EPWM4 and toggles the EPWM1B output via software by controlling CSFB bits of AQCSFRC. CLA task gets triggered by EPWM5 and toggles the EPWM1A output via software by controlling CSFA bits of AQCSFRC. Thus in this process both CPU and CLA do read-modify -write to AQCSFRC register independently at different frequencies so there is chance of race condition and updates due to one of them can get lost/. overwritten. This can be clearly observed by updating "phase_shift_ON" to 0U and probing the EPWM1A and 1B outputs on a scope.

This is a standard critical section problem and can be handled by software handshaking mechanism like mutex etc. But most of the real-time control applications are time-sensitive and cannot afford addition software overhead hence this example suggests an alternative hardware based technique to avoid shared resource conflicts between CPU and CLA. The phase shifting mechanism of the EPWM modules is utilized to schedule the CLA task and CPU ISR as desired. EPWM4 generates a synchronous pulse every ZERO event and provides a phase shift of 20 cycles to EPWM5. This way both CLA task and C28x ISR runs at original frequencies i.e. 100KHz and 10KHz but CLA task leads with a phase offset of 20 cycles wrt CPU ISR. Hence concurrent read-modify-writes to AQCSFRC never happens and the EPWM1A and EPWM1B outputs behave as desired i.e. consistent 50 KHz PWM output on EPWM1A and 5 KHz PWM output on EPWM1B with a duty $\sim$50% on both should be generated. In order to utilize this phase shifting mechanism in this example, please make sure "phase_shift_ON" is set to 1.

External Connections

- Observe GPIO0 (EPWM1A Output) on oscilloscope
- Observe GPIO1 (EPWM1B Output) on oscilloscope
- Observe GPIO2 (CLA Task Profiling) on oscilloscope
- Observe GPIO3 (CPU ISR Profiling) on oscilloscope

Note :- The phase offset value can easily be configured by updating TBPHS register to schedule the CLA task and C28x ISR as desired depending upon the application need so as to avoid overlapping register writes by CPU and CLA

Note :- The optimization is on and set to O2 for the project and all the results quoted correspond to this case.

6.33 CLB Timer Two States

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, the timer is setup the same as the previous example. The difference is the use of the FSM submodule to toggle the output of the CLB which is then exported to a GPIO. The FSM module acts as a single bit memory block. Interrupts are setup in the same format as the previous example. The interrupt delay of the CLB can be seen by comparing the output of the CLB and the GPIO toggled in the ISR.

6.34 CLB Interrupt Tag

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, a timer is setup with two different match values. These two events are used by the HLC submodule to generate interrupts. The interrupt TAG is used to differentiate between the interrupt generated due to the match1 event of the CLB counter and the match2 event of the CLB counter.

6.35 CLB Output Intersect

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, the CLB module is set up the same as the external_AND_gate example. However, instead of the output being exported to the GPIO using Output X-BAR, the output is exported to the GPIO by replacing the output of ePWM1. This is done by configuring the GPIO for EPWM1A output, followed by enabling output intersection.

6.36 CLB PUSH PULL

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, the use of the PUSH-PULL interface is shown. Multiple COUNTER submodules, HLC submodule, FSM submodules, and OUTLUT submodules are used. The PUSH-PULL interface is used alongside the GP register to update the COUNTER submodulesÃćÂĂÂŹ event frequencies.

6.37 CLB Multi Tile

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the output of a CLB TILE is passed to the input of another CLB TILE. The output of the second CLB TILE is then exported to a GPIO, showcasing how two CLB TILEs can be used in series.

6.38 CLB Combinational Logic

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

The objective of this example is to prevent simultaneous high or low outputs on a PWM pair. PWM modules 1 and 2 are configured to generate identical waveforms based on a fixed frequency up-count mode.

6.39 CLB XBARs

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the CLB INPUTXBAR and CLB OUTPUTXBAR are used to take input signals from GPIOs into the CLB TILEs and take output signal from the TILE to GPIOs. The availability of these XBARs are device dependent.

6.40 CLB Lock Output Mask

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the lock outpt mask feature of the CLB is used to lock the selected output signal override settings. This module is only available for CLB types 3 and up.

6.41 CLB INPUT Pipeline Mode

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the CLB Input Pipeline mode is enable to delay the input signal by a clock cycle. This module is only available for CLB types 3 and up.

6.42 CLB Clocking and PIPELINE Mode

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the CLB pipeline mode is enable and affects the behavior of the CLB COUNTERs and HLC. This module is only available for CLB types 3 and up.

6.43 CLB SPI Data Export

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the high speed data export feature of the CLB is used and one of the HLC registers is exported out of the CLB module using the SPI RX buffer. This module is only available for CLB types 3 and up.

6.44 CLB SPI Data Export DMA

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example the high speed data export feature of the CLB is used and one of the HLC registers is exported out of the CLB module using the SPI RX buffer. The data received in the SPI RX buffer is transferred to memory using DMA. This module is only available for CLB types 3 and up.

6.45 CLB Trip Zone Timestamp

This example displays how to timestamp interrupts generated by the CLB. An interrupt is generated when ePWM1 is tripped.

ePWM1 is configured to be interrupted by TZ1 and TZ2, both one shot trip sources.

The CLB is configured as follows:

- COUNTER0 and COUNTER1 continually count when the program begins.
- COUNTER0 timestamps TZ1 and COUNTER1 timestamps TZ2.
- COUNTER2 increments once when COUNTER0/COUNTER1 overflows using LUT2.
- FSM0/1 are configured to sync counters and stop COUNTER0/1 when an interrupt is received.
- TZ1 (GPIO12) and TZ2 (GPIO13) are routed as inputs through CLBXBAR.
- BOUNDARY.boundaryInput0 denotes TZ1. On rising edge, HLC issues an interrupt with tag 12.
- BOUDNARY.in1 denotes TZ2. On rising edge, HLC issues an interrupt with tag 13.
- BOUNDARY.boundaryInput7 serves as a simultaneous enable for COUNTER0/1 to begin counting.

TZ1 is tripped when GPIO12 is connected to GND. TZ2 is tripped when GPIO13 is connected to GND. When an interrupt occurs, the interrupt handler determines the initial trip source and stores this value in a variable 'initialTripZone'.

View these variables in Debug Expressions tab:

initialTripZone: stores the first TZ to have been tripped tz1Counter64bit: stores the counter value at the instant that TZ1 is tripped. tz2Counter64bit: stores the counter value at the instant that TZ2 is tripped.

6.46 CLB GPIO Input Filter

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

This example demonstrates use of finite state machines (FSMs) and counters to implement a simple ÃćÂĂÂŸglitchÃćÂĂÂŹ filter which might, for example, be applied to an incoming GPIO signal to remove unwanted short duration pulses.

6.47 CLB Auxilary PWM

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

This example configures a CLB tile as an auxiliary PWM generator. The example uses combinatorial logic (LUTs), state machines (FSMs), counters, and the high level controller (HLC) to demonstrate the PWM output generation capabilities using CLB.

6.48 CLB PWM Protection

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

This example extends the features of example 1 to ensure an active high complementary pair PWM configuration always operates with a minimum value of dead-band irrespective of how the generating PWM module is configured. The example illustrates the configuration of four separate PWM tiles to implement PWM protection on four PWM modules. The outputs of PWM modules 1 to 4 are operated on by CLB tiles 1 to 4, respectively.

6.49 CLB Signal Generator

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

This example uses CLB1 to generate a rectangular wave and CLB2 to check the rectangular wave generated by CLB1 doesnÃćÂĂÂŹt exceed the defined duty cycle and period limits.

6.50 CLB State Machine

For the detailed description of this example, please refer to: C2000Ware_PATH With the C2000 CLB.pdf This application report describes the process of creating this CLB example and can be used as guidance on desiging custom logic with the CLB. This example uses all submodules inside a CLB TILE in order to implement a complete system.

6.51 CLB External Signal AND Gate

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, two external signals from two GPIOs are passed through the Input X-BAR and the CLB X-BAR to the CLB TILE. Inside the CLB module these two signals are ANDED. The output of the AND gate is then exported to a GPIO, using Output X-BAR.

6.52 CLB Timer

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

In this example, a COUNTER module is used to create timed events. The use of the GP Register is shown. Through setting/clearing the bits in the GP register, the timer is started, stopped or changes direction. The output of the timer event (1-clock cycle) is exported to a GPIO. Interrupts are generated from the timer event using the HLC module. A GPIO is also toggled inside the CLB ISR. The indirect CLB register access is used to update the timerÃćÂĂÂŹs event match value and the active counter register to modify the frequency of the timer.

6.53 CLB Empty Project

For the detailed description of this example, please refer to: C2000Ware_PATH Tool Users Guide.pdf

6.54 CMPSS Asynchronous Trip

This example enables the CMPSS1 COMPH comparator and feeds the asynchronous CTRIPOUTH signal to the GPIO14/OUTPUTXBAR3 pin and CTRIPH to GPIO15/EPWM8B.

CMPSS is configured to generate trip signals to trip the EPWM signals. CMPIN1P is used to give positive input and internal DAC is configured to provide the negative input. Internal DAC is configured to provide a signal at VDD/2. An EPWM signal is generated at GPIO15 and is configured to be tripped by CTRIPOUTH.

When a low input(VSS) is provided to CMPIN1P,

- Trip signal(GPIO14) output is low
- PWM8B(GPIO15) gives a PWM signal

When a high input(higher than VDD/2) is provided to CMPIN1P,

- Trip signal(GPIO14) output turns high
- PWM8B(GPIO15) gets tripped and outputs as high

External Connections

- Give input on CMPIN1P (HSEC Pin 15)
- Outputs can be observed on GPIO14 and GPIO15 using an oscilloscope

Watch Variables

- None

6.55 CMPSS Digital Filter Configuration

This example enables the CMPSS1 COMPH comparator and feeds the output through the digital filter to the GPIO14/OUTPUTXBAR3 pin.

CMPIN1P is used to give positive input and internal DAC is configured to provide the negative input. Internal DAC is configured to provide a signal at VDD/2.

When a low input(VSS) is provided to CMPIN1P,

- GPIO14 output is low

When a high input(higher than VDD/2) is provided to CMPIN1P,

- GPIO14 output turns high

6.56 Buffered DAC Enable

This example generates a voltage on the buffered DAC output, DACOUTA/ADCINA0 and uses the default DAC reference setting of VDAC.

External Connections

- When the DAC reference is set to VDAC, an external reference voltage must be applied to the VDAC pin. This can be accomplished by connecting a jumper wire from 3.3V to ADCINB0.

Watch Variables

- None.

6.57 Buffered DAC Random

This example generates random voltages on the buffered DAC output, DACOUTA/ADCINA0 and uses the default DAC reference setting of VDAC.

External Connections

- When the DAC reference is set to VDAC, an external reference voltage must be applied to the VDAC pin. This can accomplished by connecting a jumper wire from 3.3V to ADCINB0.

Watch Variables

- None.

6.58 Buffered DAC Sine (buffdac_sine)

This example generates a sine wave on the buffered DAC output, DACOUTA/ADCINA0 (HSEC Pin 9) and uses the default DAC reference setting of VDAC.

When the DAC reference is set to VDAC, an external reference voltage must be applied to the VDAC pin. This can accomplished by connecting a jumper wire from 3.3V to ADCINB0.

Run the included .js file to add the watch variables. This example uses the SGEN module. Documentation for the SGEN module can be found in the SGEN library directory.

The generated waveform can be adjusted with the following variables while running:

- **waveformGain** : Adjust the magnitude of the waveform. Range is from 0.0 to 1.0. The default value of 0.8003 centers the waveform within the linear range of the DAC

- **waveformOffset** : Adjust the offset of the waveform. Range is from -1.0 to 1.0. The default value of 0 centers the waveform
- **outputFreq_hz** : Adjust the output frequency of the waveform. Range is from 0 to maxOutputFreq_hz
- **maxOutputFreq_hz** : Adjust the max output frequency of the waveform. Range - See SGEN module documentation for how this affects other parameters

The generated waveform can be adjusted with the following variables/macros but require recompile:

- **samplingFreq_hz** : Adjust the rate at which the DAC is updated. Range - See SGEN module documentation for how this affects other parameters
- **SINEWAVE_TYPE** : The type of sine generated. Range - LOW_THD_SINE, HIGH_PRECISION_SINE

The following variables give additional information about the generated waveform: See SGEN module documentation for details

- **freqResolution_hz**
- **maxOutput_lsb** : Maximum value written to the DAC.
- **minOutput_lsb** : Minimum value written to the DAC.
- **pk_to_pk_lsb** : Magnitude of generated waveform.
- **cpuPeriod_us** : Period of cpu.
- **samplingPeriod_us** : The rate at which the DAC is updated. Note that samplingPeriod_us has to be greater than the DAC settling time.
- **interruptCycles** : Interrupt duration in cycles.
- **interruptDuration_us** : Interrupt duration in uS.
- **sgen** : The SGEN module instance.
- **DataLog** : Circular log of writes to the DAC.

6.59 DCC Single shot Clock verification

This program uses the XTAL clock as a reference clock to verify the frequency of the PLLRAW clock.

The Dual-Clock Comparator Module 0 is used for the clock verification. The clocksource0 is the reference clock (Fclk0 = 20Mhz) and the clocksource1 is the clock that needs to be verified (Fclk1 = 200Mhz). Seed is the value that gets loaded into the Counter.

Please refer to the TRM for details on counter seed values to be set.

External Connections

- None

Watch Variables

- **status/result** - Status of the PLLRAW clock verification

6.60 DCC Single shot Clock measurement

This program demonstrates Single Shot measurement of the INTOSC2 clock post trim using XTAL as the reference clock.

The Dual-Clock Comparator Module 0 is used for the clock measurement. The clocksource0 is the reference clock (Fclk0 = 20Mhz) and the clocksource1 is the clock that needs to be measured (Fclk1 = 10Mhz). Since the frequency of the clock1 needs to be measured an initial seed is set to the max value of the counter.

Please refer to the TRM for details on counter seed values to be set.

External Connections

- None

Watch Variables

- **result** - Status if the INTOSC2 clock measurement completed successfully.
- **meas_freq1** - measured clock frequency, in this case for INTOSC2.

6.61 DCC Continuous clock monitoring

This program demonstrates continuous monitoring of PLL Clock in the system using INTOSC2 as the reference clock. This would trigger an interrupt on any error, causing the decrement/ reload of counters to stop.

The Dual-Clock Comparator Module 0 is used for the clock monitoring. The clocksource0 is the reference clock (Fclk0 = 10Mhz) and the clocksource1 is the clock that needs to be monitored (Fclk1 = 200Mhz). The clock0 and clock1 seed are set to achieve a window of 500us. Seed is the value that gets loaded into the Counter. For the sake of demo a slight variance is given to clock1 seed value to generate an error on continuous monitoring.

Please refer to the TRM for details on counter seed values to be set. Note : When running in flash configuration it is good to do a reset & restart after loading the example to remove any stale flags/states.

External Connections

- None

Watch Variables

- **status/result** - Status of the PLLRAW clock monitoring
- **cnt0** - Counter0 Value measure when error is generated
- **cnt1** - Counter1 Value measure when error is generated
- **valid** - Valid0 Value measure when error is generated

6.62 DCC Continuous clock monitoring

This program demonstrates continuous monitoring of PLL Clock in the system using INTOSC2 as the reference clock. This would trigger an interrupt on any error, causing the decrement/ reload of counters to stop. The Dual-Clock Comparator Module 0 is used for the clock monitoring. The clocksource0 is the reference clock

(Fclk0 = 10Mhz) and the clocksource1 is the clock that needs to be monitored (Fclk1 = 200Mhz). The clock0 and clock1 seed are set automatically by the error tolerances defined in the sysconfig file included this project. For the sake of demo an un-realistic tolerance is assumed to generate an error on continuous monitoring.

Please refer to the TRM for details on counter seed values to be set. Note : When running in flash configuration it is good to do a reset & restart after loading the example to remove any stale flags/states.

External Connections

- None

Watch Variables

- **status/result** - Status of the PLLRAW clock monitoring
- **cnt0** - Counter0 Value measure when error is generated
- **cnt1** - Counter1 Value measure when error is generated
- **valid** - Valid0 Value measure when error is generated

6.63 DCC Detection of clock failure

This program demonstrates clock failure detection on continuous monitoring of the PLL Clock in the system using XTAL as the osc clock source. Once the oscillator clock fails, it would trigger a DCC error interrupt, causing the decrement/ reload of counters to stop. In this examples, the clock failure is simulated by turning off the XTAL oscillator. Once the ISR is serviced, the osc source is changed to INTOSC1 and the PLL is turned off.

The Dual-Clock Comparator Module 0 is used for the clock monitoring. The clocksource0 is the reference clock (Fclk0 = 20Mhz) and the clocksource1 is the clock that needs to be monitored (Fclk1 = 200Mhz). Seed is the value that gets loaded into the Counter.

Note:
In the current example, the XTAL is expected to be a Resonator running in Crystal mode which is later switched off to simulate the clock failure. If an SE Crystal is used, you will need to physically disconnect the clock on the board.

Please refer to the TRM for details on counter seed values to be set. Note : When running in flash configuration it is good to do a reset & restart after loading the example to remove any stale flags/states.

External Connections

- None

Watch Variables

- **status/result** - Status of the clock failure detection

6.64 DCSM Memory partitioning Example

This example demonstrates how to configure and use DCSM. It configures the 1st Zone Select Block in the OTP to change the zone passwords and allocates LS0-LS3 to zone 1 & LS4-LS7 to zone 2.

Zone1 | Zone2 | LS0-LS3 | LS4-LS7 |

In this example, zoning of memories is done by the OTP programming whose values are configured in dcsm_ex1_f28p65x_dcsm_zxotp.asm while the securing functionalities are done through this file. It writes some data in the zones and checks before locking and after locking and matches with the data set . Ideally after locking zone1, the data set stored in zone1 should not be readable(or reads a 0 value) and zone2 that is not secured matches the written data set. It demonstrates how to lock and and unlock zones by showing where to put the password and how to check if it is secured or unsecured.

External Connections

- None.

Watch Variables

- **result** - Status of Secure memory partitioning done through OTP programming.
- **set_error**, error_not_locked ,error_not_unlocked ,error1 - Count of errors occurring during the execution of the example.
- **Zone1_Locked_Array** - Array demonstrating secured memory
- **Unsecure_mem_Array** - Array demonstrating Unsecured memory

Note:
Before running the example, the below configuration is expected to be done through the dcsm_ex1_f28p65x_dcsm_zxotp.asm :

- Allocate LS0-LS3 to zone 1 , LS4-LS7 to zone 2 ZSBx_Z1_GRABRAM1R 0x000AAA55 ZSBx_Z2_GRABRAM1R 0x000A55AA
- Password of zone 1 is 0xFFFFFFFF4D7FFFFFFFFFFFFFFFFFFFFF
- Password of zone 2 is 0xFFFFFFFF1F7FFFFFFFFFFFFFFFFFFFFF

Note:
DCSM_unlockZone∗CSM function should not be called in an actual application, should only be used for once to program the OTP memory. Ensure flash data cache is disabled before calling this function.

6.65 Empty DCSM Tool Example

This example is an empty project setup for DCSM Tool and Driverlib development. For guidance refer to: [C2000 DCSM Security Tool](http://www.ti.com/lit/pdf/spracp8)

6.66 DMA GSRAM Transfer (dma_ex1_gsram_transfer)

This example uses one DMA channel to transfer data from a buffer in PERINTFRC bit repeatedly until the transfer of 16 bursts (where each burst is 8 16-bit words) has been completed. When the whole transfer is complete, it will trigger the DMA interrupt.

Watch Variables

- **sData** - Data to send
- **rData** - Received data

6.67 DMA GSRAM Transfer (dma_ex2_gsram_transfer)

This example uses one DMA channel to transfer data from a buffer in PERINTFRC bit repeatedly until the transfer of 16 bursts (where each burst is 8 16-bit words) has been completed. When the whole transfer is complete, it will trigger the DMA interrupt.

Watch Variables

- **sData** - Data to send
- **rData** - Received data

6.68 eCAP APWM Example

This program sets up the eCAP module in APWM mode. The PWM waveform will come out on GPIO5. The frequency of PWM is configured to vary between 10Hz and 20Hz using the shadow registers to load the next period/compare values.

6.69 eCAP Capture PWM Example

This example configures ePWM3A for:

- Up count mode
- Period starts at 500 and goes up to 8000
- Toggle output on PRD

eCAP1 is configured to capture the time between rising and falling edge of the ePWM3A output.

External Connections

- eCAP1 is on GPIO16
- ePWM3A is on GPIO4
- Connect GPIO4 to GPIO16.

Watch Variables

- **ecap1PassCount** - Successful captures.
- **ecap1IntCount** - Interrupt counts.

6.70 eCAP APWM Phase-shift Example

This program sets up the eCAP1 and eCAP2 modules in APWM mode to generate the two phase-shifted PWM outputs of same duty and frequency value The frequency, duty and phase values can be programmed of choice by updating the defined macros. By default 10 Khz frequency, 50% duty and 30% phase shift values are used. eCAP2 output leads the eCAP1 output by 30% GPIO5 and GPIO6 are used as eCAP1/2 outputs and can be probed using analyzer/CRO to observe the waveforms.

6.71 eCAP Software Sync Example

This example configures ePWM3A for:

- Up count mode
- Period starts at 500 and goes up to 8000
- Toggle output on PRD

eCAP1, eCAP2 and eCAP3 are configured to capture the time between rising and falling edge of the ePWM3A output.

External Connections

- eCAP1, eCAP2, eCAP3 are on GPIO16
- ePWM3A is on GPIO4
- Connect GPIO4 to GPIO16.

Watch Variables

- **ecapPassCount** - Successful captures.
- **ecap3IntCount** - Interrupt counts.

6.72 Pin setup for EMIF module accessing ASRAM.

This example configures pins for EMIF in ASYNC mode.

##

6.73 EMIF1 ASYNC module accessing 16bit ASRAM.

This example configures EMIF1 in 16bit ASYNC mode and uses CS2 as chip enable.

External Connections

- External ASRAM memory (CY7C1041CV33 -10ZSXA) daughter card

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.74 EMIF1 module accessing 16bit ASRAM as code memory.

This example configures EMIF1 in 16bit ASYNC mode and uses CS2 as chip enable. This example enables use of ASRAM as code memory.

External Connections

- External ASRAM memory (CY7C1041CV33 -10ZSXA) daughter card

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.75 EMIF1 module accessing 16bit SDRAM using memcpy_fast_far().

This example configures EMIF1 in 16bit SYNC mode and uses CS0 as chip enable.It will first write to an array in the SDRAM and then read it back using the FPU function, memcpy_fast_far(), for both operations.

The buffer in SDRAM will be placed in the .farbss memory on account of the fact that its assigned the attribute "far" indicating it lies beyond the 22-bit program address space. The compiler will take care to avoid using instructions such as PREAD, which uses the Program Read Bus, or addressing modes restricted to the lower 22-bit space when accessing data with the attribute "far".

Note:
The memory space beyond 22-bits must be treated as data space for load/store operations only. The user is cautioned against using this space for either instructions or working memory.

External Connections

- External SDR-SDRAM memory (MT48LC32M16A2 -75) daughter card

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.76 EMIF1 module accessing 16bit SDRAM then puts into Self Refresh mode before entering Low Power Mode.

This example configures EMIF1 in 16bit SYNC mode and uses CS0 as chip enable. This example puts SDRAM into self refresh before entering standby mode. Watchdog timer is configured to trigger WAKEINT interrupt.

As soon as the watchdog timer expires, the device should wake up, SDRAM should come out of self refresh mode and GPIO11 can be observed to toggle.

External Connections

- External SDR-SDRAM memory (MT48LC32M16A2 -75) daughter card

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.77 EMIF1 module accessing 32bit SDRAM using DMA.

This example configures EMIF1 in 16bit SYNC(SDRAM) mode and uses CS0 as chip enable. It will first write to an array in the SDRAM and then read it back, using the DMA for both operations.

The buffer in SDRAM will be placed in the .farbss memory on account of the fact that its assigned the attribute "far" indicating it lies beyond the 22-bit program address space. The compiler will take care to avoid using instructions such as PREAD, which uses the Program Read Bus, or addressing modes restricted to the lower 22-bit space when accessing data with the attribute "far".

Note:
The memory space beyond 22-bits must be treated as data space for load/store operations only. The user is cautioned against using this space for either instructions or working memory.

External Connections

- External SDR-SDRAM (Micron MT48LC32M16A2 "P -75 C") daughter card.

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.78 EMIF1 module accessing 16bit SDRAM using alternate address mapping.

This example configures EMIF1 in 16bit SYNC mode and uses CS0 as chip enable.It will first write to an array in the SDRAM and then read it back.

The buffer in SDRAM will be placed in the emif_cs0_nonfar memory section which is dual mapped with CS2 memory range. This has been done to keep the SDRAM memory range within 22-bit address range in order to generate optimal code. EMIF1 Async RAM accesses will not be issued at the same time and program space reads & fetches will be allowed to SDRAM in non-far range.

External Connections

- External SDR-SDRAM memory (MT48LC32M16A2 -75) daughter card

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.79 Empty Project Example

This example is an empty project setup for Driverlib development.

6.80 EPG Generate Serial Data Shift Mode

This example generates SPICLK and SPI DATA signals using the SIGGEN module in SHIFT mode. For more information on this example, visit: [Designing With the C2000 Embedded Pattern Generator (EPG)](https://www.ti.com/lit/spracy7)

External Connections

- None. Signal is generated on GPIO 24, 3. Can be visualized through oscilloscope.

6.81 EPG Generating Synchronous Clocks

This example shows how to generate 2 synchronous clocks with edges being offset by 2 clock cycles. It configures Signal Generator to shift a periodic data. Generated Clock has period EPG CLOCK/6.

External Connections

- None. Signal is generated on GPIO 24, 3. Can be visualized through oscilloscope.

Watch Variables

- sigGenActiveData - Active Data of signal generator transform output

6.82 EPG Generating Two Offset Clocks

This example generates two offset clocks using the CLKGEN (CLKDIV) modules. For more information on this example, visit: [Designing With the C2000 Embedded Pattern Generator (EPG)](https://www.ti.com/lit/spracy7)

External Connections

- None. Signal is generated on GPIO 24, 3. Can be visualized through oscilloscope.

6.83 EPG Generating Two Offset Clocks With SIGGEN

This example generates two offset clocks using the SIGGEN module. For more information on this example, visit: [Designing With the C2000 Embedded Pattern Generator (EPG)](https://www.ti.com/lit/spracy7)

External Connections

- None. Signal is generated on GPIO 24, 3. Can be visualized through oscilloscope.

6.84 EPG Generate Serial Data

This example generates SPICLK and SPI DATA signals using the SIGGEN module. For more information on this example, visit: [Designing With the C2000 Embedded Pattern Generator (EPG)](https://www.ti.com/lit/spracy7)

External Connections

- None. Signal is generated on GPIO 24, 3. Can be visualized through oscilloscope.

6.85 ePWM Chopper

This example configures ePWM1, ePWM2, ePWM3 and ePWM4 as follows

- ePWM1 with Chopper disabled (Reference)
- ePWM2 with chopper enabled at 1/8 duty cycle
- ePWM3 with chopper enabled at 6/8 duty cycle
- ePWM4 with chopper enabled at 1/2 duty cycle with One-Shot Pulse enabled

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO2 EPWM2A
- GPIO3 EPWM2B
- GPIO4 EPWM3A
- GPIO5 EPWM3B
- GPIO6 EPWM4A
- GPIO7 EPWM4B

Watch Variables

- None.

6.86 EPWM Configure Signal

This example configures ePWM1, ePWM2, ePWM3 to produce signal of desired frequency and duty. It also configures phase between the configured modules.

Signal of 10kHz with duty of 0.5 is configured on ePWMxA & ePWMxB with ePWMxB inverted. Also, phase of 120 degree is configured between ePWM1 to ePWM3 signals.

During the test, monitor ePWM1, ePWM2, and/or ePWM3 outputs on an oscilloscope.

- ePWM1A is on GPIO0
- ePWM1B is on GPIO1

- ePWM2A is on GPIO2
- ePWM2B is on GPIO3
- ePWM3A is on GPIO4
- ePWM3B is on GPIO5

6.87 Realization of Monoshot mode

This example showcases how to generate monoshot PWM output based on external trigger i.e. generating just a single pulse output on receipt of an external trigger. And the next pulse will be generated only when the next trigger comes. The example utilizes external synchronization and T1 action qualifier event features to achieve the desired output.

ePWM1 is used to generate the monoshot output and ePWM2 is used an external trigger for that. No external connections are required as ePWM2A is fed as the trigger using Input X-BAR automatically.

ePWM1 is configured to generated a single pulse of 0.5us when received an external trigger. This is achieved by enabling the phase synchronization feature and configuring EPWMxSYNCI as EXTSYNCIN1. And this EPWMxSYNCI is also configured as T1 event of action qualifier to set output HIGH while "CTR = PRD" action is used to set output LOW.

ePWM2 is configured to generate a 100 KHz signal with a duty of 1% (to simulate a rising edge trigger) which is routed to EXTSYNCIN1 using Input XBAR.

Observe GPIO0 (EPWM1A : Monoshot Output) and GPIO2(EPWM2 : External Trigger) on oscilloscope.

NOTE : In the following example, the ePWM timer is still running in a continuous mode rather than a one-shot mode thus for more reliable implementation, refer to CLB based one shot PWM implementation demonstrated in "clb_ex17_one_shot_pwm" example

6.88 EPWM Action Qualifier (epwm_up_aq)

This example configures ePWM1, ePWM2, ePWM3 to produce an waveform with independent modulation on EPWMxA and EPWMxB.

The compare values CMPA and CMPB are modified within the ePWM's ISR.

The TB counter is in up count mode for this example.

View the EPWM1A/B(GPIO0 & GPIO1), EPWM2A/B(GPIO2 & GPIO3) and EPWM3A/B(GPIO4 & GPIO5) waveforms via an oscilloscope.

6.89 ePWM XCMP Mode

(Note - base frequency and duty cycle of all ePWM's are 50 KHz and 50% respectively. Value of TBPRD = 1999) This example configures ePWM1, ePWM2, ePWM4, ePWM6 and ePWM8 as follows

- ePWM1A is allocated all XCMP1-8 registers. ePWM1B has no output.
  - New duty cycle = 50%, new frequency = 200 KHz

    - No Shadow registers used
- ePWM2A is allocated XCMP1-4 and ePWM2B is allocated XCMP5-8 registers.
    - A and B waveforms are complimentary
    - New duty cycle = 50%, new frequency = 100 KHz
    - No Shadow registers used
- ePWM4 is configured same as ePWM2 with Minimum Deadband.
    - Minimum Deadband of 200 SYSCLK cycles provided (200 $*$ (1/200 MHz) = 1 micro second)
    - This implies dead band of 1 us is visible on output of ePWM4A and ePWM4B after their falling edge
    - New duty cycle = 40%, new frequency = 100 KHz
- ePWM6A is allocated XCMP1-4 registers.
    - 3 Shadow register sets used with LOADMULTIPLE mode
    - Shadow set 2 repeated 2 times, Shadow set 3 repeated 4 times
    - ISR to update all Shadow registers with new values after they repeat
    - This means Shadow3 is active for 5 periods, Shadow2 is active for 3 periods and Shadow1 is active for 1 period before their new values are visible in output
- ePWM8A is allocated XCMP1-4 registers.
    - 3 Shadow register sets used with LOADONCE mode
    - Only Shadow set 3 is loaded from every period
    - ISR updates Shadow 3 register with new values every 5 periods

External Connections

- ePWM1A is on GPIO0
- ePWM2A is on GPIO2 and ePWM2B is on GPIO3
- ePWM4A is on GPIO6 and ePWM4B is on GPIO7
- ePWM6A is on GPIO10
- ePWM8A is on GPIO14
- Monitor GPIO24 for ePWM6A new Shadow register value loading
- Monitor GPIO25 for ePWM8A new Shadow register value loading

Shadow register updations for ePWM6 and ePWM8:

- Only XCMP1 and XCMP4 are updated
- Update values are +/- 20 TBCTR steps depending on direction of updation
- GPIO24 is toggled every 9 cycles for ePWM6 after cycling through all Shadow buffers and loading new values Similarly GPIO25 is toggled every 5 cycles for ePWM8

6.90 ePWM Event Detection

(Note - base frequency and duty cycle of all ePWM's are 50 KHz and 50% respectively. Value of TBPRD = 1999) This example configures ePWM1 and ePWM2 in identical fashion with XCMP1-8 allocated to channel A. No shadow registers are used.

- XCMP1 = 250, XMP2 = 500, XCMP3 = 750 ..... XCMP 8 = 1750.
- In ePWM1, XMIN = 300 and XMAX = 400.
  - This window has no edge and generates a CAPEVT pulse every period.
- In ePWM2, XMIN = 300 and XMAX = 600.
  - This window has an edge and doesn't generate CAPEVT pulse signal.

External Connections

- ePWM1A is on GPIO0
- ePWM2A is on GPIO2
- ePWM1 Tripout is on GPIO24
- ePWM2 Tripout is on GPIO25
- LED1 is on GPIO31 (For control card)
- LED2 is on GPIO34 (For control card)

CAPIN and CAPGATE signals are both sourced as Trip4 for ePWM1 and Trip5 for ePWM2. Trip4 and Trip5 are routed from the INPUT X-BAR through EPWM-XBAR which feeds into the Digital Compare submodule. CAPEVT signal is used as Tripout and Trip-Zone interrupt source.

For ePWM1, Trip-Zone ISR configured to make LED1 blink 1 second on/off. For ePWM2, Trip-Zone ISR configured to make LED2 turn on if an interrupt ever occurs.

6.91 ePWM Diode Emulation

This example configures ePWM1, ePWM2 and ePWM4 as follows

- All ePWM's are 50 KHz, 50% duty cycle Type 4 generated signals

Buffered DAC1 configuration used as positive input to CMPSS1 High and Low:

- Reference voltage source = internal ADC VREFHI = 2.5V
- Gain = 1

CMPSS1 configuration for TRIPH/TRIPL signal inputs to enter DE mode:

- Positive input = DACA_OUT (from DAC1) for both CMPSS1 High, Low
- Negative input = 825 mV from internal DAC for CMPSS1 High 3.3V from internal DAC for CMPSS1 Low
- This means only CMPSSTRIPH1 is triggered when DACA_OUT $>$ 825 mV CMPSSTRIPL1 is always low

Diode Emulation configuration:

- For all ePWM's, TripH = CMPSSTRIPH1 and TRIPL = CMPSSTRIPL1 Since a TRIPH_OR_TRIPL triggers DE entry, only TripH is used for demo
- ePWM1A is set to active high and 1B set to low during DE
- ePWM2 is configured same as ePWM1 with a re-entry delay

  - Re-entry delay = 250 EPWMSYNCPER cycles = 5 ms
- ePWM4 is configured same as ePWM1 with DE monitor
  - Increment/ Decrement on EPWMSYNCPER = +1 when TRIPH_OR_TRIPL high -1 when TRIPH_OR_TRIPL low
  - Threshold value = 250 In this case, 5 ms after entering DE, DEMONTRIP is generated

External Connections

- ePWM1A is on GPIO0 and ePWM1B is on GPIO1
- ePWM2A is on GPIO2 and ePWM2B is on GPIO3
- ePWM4A is on GPIO6 and ePWM4B is on GPIO7
- CMPSS1_TRIPH is on GPIO25 and CMPSS1_TRIPL is on GPIO26
- ePWM1_DEACTIVE is on GPIO24
- ePWM4_DEMONTRIP is on GPIO15
- DAC1 output can be seen on P1 (ADC-A0) pin

For DAC1, DACVALS -> DACVALA shadow to active load happening on SYSCLK and a 2 us delay is provided after loading. Shadow to active load can be configured to load on EPWMSYNCPER for better control.

6.92 ePWM Minimum Deadband and ICL

This example configures ePWM2, ePWM3, ePWM4 and ePWM5 as follows (Note: ePWM2 and ePWM3 are configured for demonstration purpose - i.e they are original ePWM4 and ePWM5 signals respectively)

Minimum Deadband:

- ePWM2 is 50 KHz signal Type-4 signal with 55% duty cycle. TBPRD = 2000
  - There is an overlap of active high states in A and B between TBCNT = 900 and 1100
- ePWM4 applies a minimum deadband with 200 SYSCLK cycles delay (1 us).
  - The new duty cycle changes to 40%.

Illegal Combo Logic:

- ePWM3 is 100 KHz signal Type-4 signal with 50% duty cycle. TBPRD = 1000.
- ePWM5 applies ICL look up table (LUT) logic to perform AND operation between 5A & 4A and 5B & 4B.

External Connections

- ePWM2A is on GPIO2 and ePWM2B is on GPIO3
- ePWM3A is on GPIO4 and ePWM3B is on GPIO5
- ePWM4A is on GPIO6 and ePWM4B is on GPIO7
- ePWM5A is on GPIO8 and ePWM5B is on GPIO9

For LUT1, ePWM4A is input from ICL X-BAR For LUT2, ePWM4B is input from ICL X-BAR ICL LUT and associated decision qualifiers can be visualized in the following truth table:

LUT1:	ePWM4A	ePWM5A	ePWM5B	ePWM5A	<– 5A output
	0	0	X	0	0
	0	1	X	0	1
	1	0	X	0	1
	1	1	X	1	1

LUT2:	ePWM4B	ePWM5B	ePWM5A	ePWM5B	<– 5B output
	0	0	X	0	0
	0	1	X	0	1
	1	0	X	0	1
	1	1	X	1	1

6.93 ePWM Trip Zone

This example configures ePWM1 and ePWM2 as follows

- ePWM1 has TZ1 as one shot trip source
- ePWM2 has TZ1 as cycle by cycle trip source

Initially tie TZ1 high. During the test, monitor ePWM1 or ePWM2 outputs on a scope. Pull TZ1 low to see the effect.

External Connections

- ePWM1A is on GPIO0
- ePWM2A is on GPIO2
- TZ1 is on GPIO12

This example also makes use of the Input X-BAR. GPIO12 (the external trigger) is routed to the input X-BAR, from which it is routed to TZ1.

The TZ-Event is defined such that ePWM1A will undergo a One-Shot Trip and ePWM2A will undergo a Cycle-By-Cycle Trip.

6.94 ePWM Up Down Count Action Qualifier

This example configures ePWM1, ePWM2, ePWM3 to produce a waveform with independent modulation on ePWMxA and ePWMxB.

The compare values CMPA and CMPB are modified within the ePWM's ISR.

The TB counter is in up/down count mode for this example.

View the ePWM1A/B(GPIO0 & GPIO1), ePWM2A/B(GPIO2 &GPIO3) and ePWM3A/B(GPIO4 & GPIO5) waveforms on oscilloscope.

6.95 ePWM Synchronization

This example configures ePWM1, ePWM2, ePWM3 and ePWM4 as follows

- ePWM1 without phase shift as sync source
- ePWM2 with phase shift of 300 TBCLKs
- ePWM3 with phase shift of 600 TBCLKs
- ePWM4 with phase shift of 900 TBCLKs

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO2 EPWM2A
- GPIO3 EPWM2B
- GPIO4 EPWM3A
- GPIO5 EPWM3B
- GPIO6 EPWM4A
- GPIO7 EPWM4B

Watch Variables

- None.

6.96 ePWM Digital Compare

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW
- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25's PULL-UP resistor is enabled, in order to test the trip, PULL this pin to GND

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO25 TZ1, pull this pin low to trip the ePWM

Watch Variables

- None.

6.97 ePWM Digital Compare Event Filter Blanking Window

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW

- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25's PULL-UP resistor is enabled, in order to test the trip, PULL this pin to GND

- ePWM1 with DCBEVT1 forcing the ePWM output LOW
- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25's PULL-UP resistor is enabled, in order to test the trip, PULL this pin to GND
- DCBEVT1 uses the filtered version of DCBEVT1
- The DCFILT signal uses the blanking window to ignore the DCBEVT1 for the duration of DC Blanking window

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO25 TRIPIN1, pull this pin low to trip the ePWM

Watch Variables

- None.

6.98 ePWM Valley Switching

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW
- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25 is set to output and toggled in the main loop to trip the PWM

- ePWM1 with DCBEVT1 forcing the ePWM output LOW
- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO25 is set to output and toggled in the main loop to trip the PWM
- DCBEVT1 uses the filtered version of DCBEVT1
- The DCFILT signal uses the valley switching module to delay the
- DCFILT signal by a software defined DELAY value.

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO25 TRIPIN1 (Output Pin, toggled through software)

Watch Variables

- None.

6.99 ePWM Digital Compare Edge Filter

This example configures ePWM1 as follows

- ePWM1 with DCBEVT2 forcing the ePWM output LOW as a CBC source
- GPIO25 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCBEVT2
- GPIO25 is set to output and toggled in the main loop to trip the PWM
- The DCBEVT2 is the source for DCFILT
- The DCFILT will count edges of the DCBEVT2 and generate a signal to to trip the ePWM on the 4th edge of DCBEVT2

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO25 TRIPIN1 (Output Pin, toggled through software)

Watch Variables

- None.

6.100 ePWM Deadband

This example configures ePWM1 through ePWM6 as follows

- ePWM1 with Deadband disabled (Reference)
- ePWM2 with Deadband Active High
- ePWM3 with Deadband Active Low
- ePWM4 with Deadband Active High Complimentary
- ePWM5 with Deadband Active Low Complimentary
- ePWM6 with Deadband Output Swap (switch A and B outputs)

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO2 EPWM2A
- GPIO3 EPWM2B
- GPIO4 EPWM3A
- GPIO5 EPWM3B
- GPIO6 EPWM4A
- GPIO7 EPWM4B
- GPIO8 EPWM5A

- GPIO9 EPWM5B
- GPIO10 EPWM6A
- GPIO11 EPWM6B

Watch Variables

- None.

6.101 ePWM DMA

This example configures ePWM1 and DMA as follows:

- ePWM1 is set up to generate PWM waveforms
- DMA5 is set up to update the CMPAHR, CMPA, CMPBHR and CMPB every period with the next value in the configuration array. This allows the user to create a DMA enabled fifo for all the CMPx and CMPxHR registers to generate unconventional PWM waveforms.
- DMA6 is set up to update the TBPHSHR, TBPHS, TBPRDHR and TBPRD every period with the next value in the configuration array.
- Other registers such as AQCTL can be controlled through the DMA as well by following the same procedure. (Not used in this example)

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B

Watch Variables

- None.

6.102 Frequency Measurement Using eQEP

This example will calculate the frequency of an input signal using the eQEP module. ePWM1A is configured to generate this input signal with a frequency of 5 kHz. It will interrupt once every period and call the frequency calculation function. This example uses the IQMath library to simplify high-precision calculations.

In addition to the main example file, the following files must be included in this project:

- **eqep_ex1_calculation.c** - contains frequency calculation function
- **eqep_ex1_calculation.h** - includes initialization values for frequency structure

The configuration for this example is as follows

- Maximum frequency is configured to 10KHz (baseFreq)
- Minimum frequency is assumed at 50Hz for capture pre-scalar selection

SPEED_FR: High Frequency Measurement is obtained by counting the external input pulses for 10ms (unit timer set to 100Hz).

$$SPEED\_FR = \frac{Count\ Delta}{10ms}$$

SPEED_PR: Low Frequency Measurement is obtained by measuring time period of input edges. Time measurement is averaged over 64 edges for better results and the capture unit performs the time measurement using pre-scaled SYSCLK.

Note that the pre-scaler for capture unit clock is selected such that the capture timer does not overflow at the required minimum frequency. This example runs indefinitely until the user stops it.

For more information about the frequency calculation see the comments at the beginning of eqep_ex1_calculation.c and the XLS file provided with the project, eqep_ex1_calculation.xls.

External Connections

- Connect GPIO20/eQEP1A to GPIO0/ePWM1A

Watch Variables

- **freq.freqHzFR** - Frequency measurement using position counter/unit time out
- **freq.freqHzPR** - Frequency measurement using capture unit

6.103 Position and Speed Measurement Using eQEP

This example provides position and speed measurement using the capture unit and speed measurement using unit time out of the eQEP module. ePWM1 and a GPIO are configured to generate simulated eQEP signals. The ePWM module will interrupt once every period and call the position/speed calculation function. This example uses the IQMath library to simplify high-precision calculations.

In addition to the main example file, the following files must be included in this project:

- **eqep_ex2_calculation.c** - contains position/speed calculation function
- **eqep_ex2_calculation.h** - includes initialization values for position/speed structure

The configuration for this example is as follows

- Maximum speed is configured to 6000rpm (baseRPM)
- Minimum speed is assumed at 10rpm for capture pre-scalar selection
- Pole pair is configured to 2 (polePairs)
- Encoder resolution is configured to 4000 counts/revolution (mechScaler)
- Which means: 4000 / 4 = 1000 line/revolution quadrature encoder (simulated by ePWM1)
- ePWM1 (simulating QEP encoder signals) is configured for a 5kHz frequency or 300 rpm (= 4 $*$ 5000 cnts/sec $*$ 60 sec/min) / 4000 cnts/rev)

SPEEDRPM_FR: High Speed Measurement is obtained by counting the QEP input pulses for 10ms (unit timer set to 100Hz).

SPEEDRPM_FR = (Position Delta / 10ms) $*$ 60 rpm

SPEEDRPM_PR: Low Speed Measurement is obtained by measuring time period of QEP edges. Time measurement is averaged over 64 edges for better results and the capture unit performs the time measurement using pre-scaled SYSCLK.

Note that the pre-scaler for capture unit clock is selected such that the capture timer does not overflow at the required minimum frequency. This example runs indefinitely until the user stops it.

For more information about the position/speed calculation see the comments at the beginning of eqep_ex2_calculation.c and the XLS file provided with the project, eqep_ex2_calculation.xls.

External Connections

- Connect GPIO20/eQEP1A to GPIO0/ePWM1A (simulates eQEP Phase A signal)
- Connect GPIO21/eQEP1B to GPIO1/ePWM1B (simulates eQEP Phase B signal)
- Connect GPIO23/eQEP1I to GPIO2 (simulates eQEP Index Signal)

Watch Variables

- **posSpeed.speedRPMFR** - Speed meas. in rpm using QEP position counter
- **posSpeed.speedRPMPR** - Speed meas. in rpm using capture unit
- **posSpeed.thetaMech** - Motor mechanical angle (Q15)
- **posSpeed.thetaElec** - Motor electrical angle (Q15)

6.104 Frequency Measurement Using eQEP via unit timeout interrupt

This example will calculate the frequency of an input signal using the eQEP module. ePWM1A is configured to generate this input signal with a frequency of 5 kHz. EQEP unit timeout is set which will generate an interrupt every **UNIT_PERIOD** microseconds and frequency calculation occurs continuously

The configuration for this example is as follows

- PWM frequency is specified as 5000Hz
- UNIT_PERIOD is specified as 10000 us
- Min frequency is (1/(2∗10ms)) i.e 50Hz
- Highest frequency can be (2^32)/ ((2∗10ms))
- Resolution of frequency measurement is 50hz

freq : Frequency Measurement is obtained by counting the external input pulses for UNIT_PERIOD (unit timer set to 10 ms).

External Connections

- Connect GPIO20/eQEP1A to GPIO0/ePWM1A

Watch Variables

- **freq** - Frequency measurement using position counter/unit time out
- **pass** - If measured frequency matches with PWM frequency then pass = 1 else 0

6.105 Motor speed and direction measurement using eQEP via unit timeout interrupt

This example can be used to sense the speed and direction of motor using eQEP in quadrature encoder mode. ePWM1A is configured to simulate motor encoder signals with frequency of 5 kHz on both A and B pins with 90 degree phase shift (so as to run this example without motor). EQEP unit timeout is set which will generate an interrupt every **UNIT_PERIOD** microseconds and speed calculation occurs continuously based on the direction of motor

The configuration for this example is as follows

- PWM frequency is specified as 5000Hz
- UNIT_PERIOD is specified as 10000 us
- Simulated quadrature signal frequency is 20000Hz (4 $*$ 5000)
- Encoder holes assumed as 1000
- Thus Simulated motor speed is 300rpm (5000 $*$ (60 / 1000))

freq : Simulated quadrature signal frequency measured by counting the external input pulses for UNIT_PERIOD (unit timer set to 10 ms). **speed** : Measure motor speed in rpm **dir** : Indicates clockwise (1) or anticlockwise (-1)

External Connections (if motor encoder signals are simulated by ePWM)

- Connect GPIO20/eQEP1A to GPIO0/ePWM1A
- Connect GPIO21/eQEP1B to GPIO1/ePWM1B With motor
- Comment in "MOTOR" in includes
- Connect GPIO20/eQEP1A to encoder A output
- Connect GPIO21/eQEP1B to encoder B output

Watch Variables

- **freq** : Simulated motor frequency measurement is obtained by counting the external input pulses for UNIT_PERIOD (unit timer set to 10 ms).
- **speed** : Measure motor speed in rpm
- **dir** : Indicates clockwise (1) or anticlockwise (-1)
- **pass** - If measured qudrature frequency matches with i.e. input quadrature frequency (4 $*$ PWM frequency) then pass = 1 else fail = 1 ($**$ only when "MOTOR" is commented out)

6.106 ERAD Profile Function

This example uses BUSCOMP1, BUSCOMP2 and COUNTER1 of the ERAD module to profile a function (delay-Function). It calculates the CPU cycles taken between the the start address of the function to the end address of the function

Two dummy variable are written to inside the function - startCount and endCount. BUSCOMP3, BUSCOMP4 and COUNTER2 are used to profile the time taken between the access to startCount variable till the access to endCount variable.

Both the counters are setup to operate in START-STOP mode and count the number of CPU cycles spend between the respective bus comparator events.

Watch Variables

- cycles_Functio - the maximum number of cycles between the start of function to the end of function
- cycles_Data - the maximum number of cycles taken between accessing startCount variable to endCount variable

External Connections

None

6.107 ERAD Profile Function

This example uses BUSCOMP1, BUSCOMP2 and COUNTER1 of the ERAD module to profile a function (delayFunction). It calculates the CPU cycles taken between the the start address of the function to the end address of the function

Two dummy variable are written to inside the function - startCount and endCount. BUSCOMP3, BUSCOMP4 and COUNTER2 are used to profile the time taken between the access to startCount variable till the access to endCount variable.

Both the counters are setup to operate in START-STOP mode and count the number of CPU cycles spend between the respective bus comparator events.

Watch Variables

- cycles_Function - the maximum number of cycles between the start of function to the end of function
- cycles_Data - the maximum number of cycles taken between accessing startCount variable to endCount variable

External Connections

None

6.108 ERAD HWBP Monitor Program Counter

In this example, the function delayFunction is called multiple times. The function does read and writes to the global variables startCount and endCount.

The BUSCOMP1 and COUNTER1 is used to count the number of times the function delayFunction was invoked. BUSCOMP2 is used to generate an interrupt when there is read access to the startCount variable and BUSCOMP3 is used to generate an interrupt when there is a write access to the endCount variable

Watch Variables

- funcCount - number of times the function delayFunction was invoked
- isrCount - number of times the ISR was invoked

External Connections

- None

6.109 ERAD HWBP Monitor Program Counter

In this example, the function delayFunction is called multiple times. The function does read and writes to the global variables startCount and endCount.

The BUSCOMP1 and COUNTER1 is used to count the number of times the function delayFunction was invoked. BUSCOMP2 is used to generate an interrupt when there is read access to the startCount variable and BUSCOMP3 is used to generate an interrupt when there is a write access to the endCount variable

Watch Variables

- funcCount - number of times the function delayFunction was invoked
- isrCount - number of times the ISR was invoked

External Connections

- None

6.110 ERAD HWBP Stack Overflow Detection

This example uses BUSCOMP1 to monitor the stack. The Bus comparator is set to monitor the data write access bus and generate an RTOS interrupt CPU when a write is detected to end of the STACK within a threshold.

Watch Variables

- functionCallCount - the number of times the recursive function overflowing the STACK is called.
- x indicates that the ISR has been entered

External Connections

None

6.111 ERAD HWBP Stack Overflow Detection

This example uses BUSCOMP1 to monitor the stack. The Bus comparator is set to monitor the data write access bus and generate an RTOS interrupt CPU when a write is detected to end of the STACK within a threshold.

Watch Variables

- functionCallCount - the number of times the recursive function overflowing the STACK is called.
- x indicates that the ISR has been entered

External Connections

None

6.112 ERAD Profiling Interrupts

This example shows how an ISR can be profiled by ERAD. The CPU timer generates interrupts periodically. We set up the counters to count the CPU cycles elapsed while executing the ISR, to count the number of interrupts, the number of ISR executions and the CPU cycles elapsed between the interrupt and the execution of the ISR.

This example uses 2 bus comparators and 4 counters:

- BUSCOMP_1 : PC = start address of cpuTimer1ISR
- BUSCOMP_2 : PC = address of cpuTimer1IntCount variable access. This specifies the end address of the code of interest.
- COUNTER_1 : Used to count the cpuTimer1ISR execution cycles. Configured in start-stop mode with start event as BUSCOMP_1 and stop event as BUSCOMP_2
- COUNTER_2 : Used to count the number of times the system event TIMER1_TINT1 has occurred. Configured in rising-edge count mode with counting input as system event TIMER1_TINT1
- COUNTER_3 : Used to count the number of times cputTimer2ISR executes. Configured in rising-edge count mode with counting input as BUSCOMP_1
- COUNTER_4 : Used to count the latency from the system event TIMER1_TINT1 to cpuTimer1ISR entry. Configured in start-stop mode with start event as TIMER1_TINT1 and stop event as BUSCOMP_1

We configure the COUNTER1 to generate an interrupt once it reaches a threshold value.

External Connections

- None

Profiling Output

- Current ISR cycle count (COUNTER_1)
- Interrupt occurrence count (COUNTER_2)
- ISR execution count (COUNTER_3)
- ISR entry delay cycle count (maximum value of COUNTER_4)
- x - To show that the ISR executed

6.113 ERAD Profiling Interrupts

This example shows how an ISR can be profiled by ERAD. The CPU timer generates interrupts periodically. We set up the counters to count the CPU cycles elapsed while executing the ISR, to count the number of interrupts, the number of ISR executions and the CPU cycles elapsed between the interrupt and the execution of the ISR.

This example uses 2 bus comparators and 4 counters:

- BUSCOMP_1 : PC = start address of cpuTimer1ISR
- BUSCOMP_2 : PC = address of cpuTimer1IntCount variable access. This specifies the end address of the code of interest.
- COUNTER_1 : Used to count the cpuTimer1ISR execution cycles. Configured in start-stop mode with start event as BUSCOMP_1 and stop event as BUSCOMP_2

- COUNTER_2 : Used to count the number of times the system event TIMER1_TINT1 has occurred. Configured in rising-edge count mode with counting input as system event TIMER1_TINT1
- COUNTER_3 : Used to count the number of times cputTimer2ISR executes. Configured in rising-edge count mode with counting input as BUSCOMP_1
- COUNTER_4 : Used to count the latency from the system event TIMER1_TINT1 to cpuTimer1ISR entry. Configured in start-stop mode with start event as TIMER1_TINT1 and stop event as BUSCOMP_1

We configure the COUNTER1 to generate an interrupt once it reaches a threshold value.

External Connections

- None

Profiling Output

- Current ISR cycle count (COUNTER_1)
- Interrupt occurrence count (COUNTER_2)
- ISR execution count (COUNTER_3)
- ISR entry delay cycle count (maximum value of COUNTER_4)
- x - To show that the ISR executed

FILE: erad_ex5_restricted_write_detect.c

TITLE: erad_ex5_restrictedwrite_detect

6.114 ERAD MEMORY ACCESS RESTRICT

This example uses BUSCOMP1 to monitor the Data Write Address Bus. It monitors the bus and generates an RTOS interrupt if a certain region of memory is accessed by the PC. The user may disable the Bus Comparator to access that region.

Use the COM port (Baud=9600) to try to write to the restricted area.

Watch Variables

- x : stores the number of times the region of memory is accessed

External Connections

- None

##

FILE: erad_ex6_interrupt_order.c

TITLE: ERAD INTERRUPT ORDER

6.115 ERAD INTERRUPT ORDER

This example uses a COUNTER to monitor the sequence of ISRs executed. An interrupt is generated if the ISRs executed are not in the expected order. The expected order is CPUTimer0 ,then CPUTimer1 and then CPUTimer2

The counter is configured in Start-Stop Mode to count the number of times CPUTimer interrupt occurs between the CPUTimer1 interrupt and CPUTimer2 ISRs. Ideally, this count should be zero if the interrupts are occurring in the expected order. we configure a threshold value of 1 to genarete an RTOS interrupt. This indicates that the CPUTimer2 interrupt has come out of order.

For demonstaration puproses, this example disables CPUTimer1 to simulate this error.

Watch Variables

- cpuTimer0IntCount: Number of executions of ISR0
- cpuTimer1IntCount: Number of executions of ISR1
- cpuTimer2IntCount: Number of executions of ISR2

External Connections

- None

6.116 ERAD AND CLB

This example uses 4 BUS COMPARATORS of ERAD along with the CLB. One bus comparator monitors a write to x, another one monitors a write to y. The other two monitor a write of 0x1 and 0x0. By using the LUTs in the CLB1 tile, we can monitor a write of 0x1 to x or 0x0 to x. These are used to change the state of FSM2 in the CLB1 tile. If y is accessed before writing a 0x1 to x, an interrupt is generated and y is changed to 0x0 again. The LED2 indicates when access to y is allowed(it is off at this point) The LED1 indicates if an invalid access is attempted. A COUNTER in ERAD is used to count the number of access attempts to y.

Watch Variables

- y
- x
- a - counts the number of access attempts to y

External Connections

None

6.117 ERAD PWM PROTECTION

This example uses a BUS COMPARATOR and the CLB to detect the event when the delay between the interrupt and the ISR execution is longer than expected. The PWM output is also tripped in this case.

Watch Variables

- adcAResults stores the results of the conversions from the ADC

External Connections

- Monitor the PWM output (GPIO0)

6.118 ERAD Profiling Interrupts

This example configures CPU Timer0, 1, and 2 to be profiled using the ERAD module. Included is a JavaScript file, profile_interrupts.js, which is used with the scripting console to program ERAD registers and view profiling data.

To properly use the provided ERAD script, the following variables must be set in the scripting environment prior to launching the ERAD script:

- var PROJ_NAME = "erad_debugger_ex1_profileinterrupts"
- var PROJ_WKSPC_LOC = "<proj_workspace_path>"
- var PROJ_CONFIG = "<name of active configuration [CPU1_FLASH|CPU1_RAM]>"

To run the ERAD script, use the following command in the scripting console:

- loadJSFile("<proj_workspace_path>\\erad_debugger_ex1_profileinterrupts\\erad_ex1_profile_interrupts.j
  0);

The included JavaScript file, erad_ex1_profile_interrupts.js, uses Debug Server Scripting (DSS) features. For information on using the DSS, please visit: http://software-dl.ti.com/ccs/esd/documents/users_guide/sdto_dss_handbook.html

Note that the script must be run after loading and running the .out on the C28x core. Only CPU timer 2 ISR is profiled in this example.

This example uses 2 HW breakpoints and 4 counters:

- HWBP_1 : PC = start address of cpuTimer2ISR
- HWBP_2 : PC = end address of cpuTimer2ISR
- CTM_1 : Used to count the cpuTimer2ISR execution cycles. Configured in start-stop mode with start event as HWBP_1 and stop event as HWBP_2
- CTM_2 : Used to count the number of times the system event TIMER2_TINT2 has occurred. Configured in rising-edge count mode with counting input as system event TIMER2_TINT2 (INP_SEL[25])
- CTM_3 : Used to count the number of times cputTimer2ISR executes. Configured in rising-edge count mode with counting input as HWBP_1 (INP_SEL[0])
- CTM_4 : Used to count the latency from the system event TIMER2_TINT2 to cpuTimer2ISR entry. Configured in start-stop mode with start event as TIMER2_TINT2 and stop event as HWBP_1

External Connections

- None

Watch Variables

- cpuTimer0IntCount
- cpuTimer1IntCount
- cpuTimer2IntCount

Profiling Script Output

- Current ISR cycle count (CTM_1)
- Max ISR cycle count (maximum value of CTM_1)
- Interrupt occurrence count (CTM_2)
- ISR execution count (CTM_3)
- ISR entry delay cycle count (maximum value of CTM_4)

Note that the large difference between Interrupt occurrence count (CTM_2) and ISR execution count (CTM_3) is because the ISR takes more number of cycles than the actual interrupt period. ISR entry delay cycle count will also be higher due to the same reason.

6.119 ERAD Profile Function

This example contains a basic FIR calculation and sorting algorithm to help demonstrate the function profiling capability of the ERAD peripheral. A number of FIR sums are calculated within a loop and are then sorted using the insertion sort algorithm. Cycle counts of both the FIR calculations and the sorting algorithm are output to the screen through the scripting console. In this example, it can be seen that sorting the data takes up a majority of the CPU cycles executed in this program.

To properly use the provided ERAD script, the following variables must be set in the scripting environment prior to launching the ERAD script:

- var PROJ_NAME = "erad_debugger_ex2_profilefunction"
- var PROJ_WKSPC_LOC = "<proj_workspace_path>"
- var PROJ_CONFIG = "<name of active configuration [CPU1_FLASH|CPU1_RAM]>"

To run the ERAD script, use the following command in the scripting console:

- loadJSFile("<proj_workspace_path>\\erad_debugger_ex2_profilefunction\\erad_ex2_profile_function.js", 0);

Note that the script must be run after loading and running the .out on the C28x core.

The included JavaScript file, erad_ex2_profile_function.js, uses Debug Server Scripting (DSS) features. For information on using the DSS, please visit: http://software-dl.ti.com/ccs/esd/documents/users_guide/sdto_dss_handbook.html

This example uses 4 HW breakpoints and 2 counters:

- HWBP_1 : PC = start address of performFIR
- HWBP_2 : PC = end address of performFIR
- HWBP_3 : PC = start address of sortMax

- HWBP_4 : PC = end address of sortMax
- CTM_1 : Used to count the performFIR execution cycles. Configured in start-stop mode with start event as HWBP_1 and stop event as HWBP_2
- CTM_2 : Used to count the sortMax execution cycles. Configured in start-stop mode with start event as HWBP_3 and stop event as HWBP_4

External Connections

- None.

Watch Variables

- FIR_iterationCounter - A counter for the number of times FIR calculation and sorting was performed

Profiling Script Output

- Current FIR cycle count (CTM_1)
- Max FIR cycle count (maximum value of CTM_1)
- Current sorting function cycle count (CTM_2)
- Max sorting function cycle count (maximum value of CTM_2)

Note that the the counters are reset after the stop event. The counter value remains 0 till the next start event occurs. The javascript continuously reads the counter value in a while(1) and hence the current counter may return 0.

6.120 ERAD Stack Overflow

This example shows the basic setup of CAN in order to transmit and receive messages on the CAN bus. The CAN peripheral is configured to transmit messages with a specific CAN ID. A message is then transmitted once per second, using a simple delay loop for timing. The message that is sent is a 2 byte message that contains an incrementing pattern.

This example sets up the CAN controller in External Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core.

A buffer is created to store message history up to 50 messages for the duration of the program. A logic error is intentionally made to allow the buffer to overflow, eventually causing a stack overflow. The included JavaScript file, stack_overflow.js, programs ERAD registers in order to detect the stack overflow and halt the CPU once the illegal write is made. The illegal write is made after 507 messages are received.

To properly use the provided ERAD script, the following variables must be set in the scripting environment prior to launching the ERAD script:

- var PROJ_NAME = "erad_debugger_ex3_stackoverflow"
- var PROJ_WKSPC_LOC = <proj_workspace_path>

To run the ERAD script, use the following command in the scripting console:

- loadJSFile("<proj_workspace_path>\\erad_debugger_ex3_stackoverflow\\erad_ex3_stack_overflow.js", 0);

Note that the script must be run after loading and running the .out on the C28x core.

The included JavaScript file, erad_ex3_stack_overflow.js, uses Debug Server Scripting (DSS) features. For information on using the DSS, please visit: http://software-dl.ti.com/ccs/esd/documents/users_guide/sdto_dss_handbook.html

This example uses 1 HW watchpoint :

- HWBP_1 : Data Write Address Bus = Stack end address + 1

External Connections

- None.

Watch Variables

- msgCount - A counter for the number of successful messages received
- txMsgData - An array with the data being sent
- rxMsgData - An array with the data that was received
- msgHistoryBuff - An array meant to store the last 50 messages received

Profiling Script Output

- "STACK OVERFLOW detected. Halting CPU." will be printed in the scripting console when a stack overflow occurs (that is, when the watchpoint is hit)

6.121 ERAD Profile Interrupts CLA

This example configures EPWM1A to run at 1 KHz (period = 1 ms) to trigger a start-of-conversion on ADC channel A0. This channel will, in turn, sample EPWM4A which is set to run at 100Hz. At the end-of-conversion the ADC interrupt is fired. The interrupt signal will be used to trigger a CLA task that runs an FIR filter. The filter is designed to be low pass with a cutoff frequency of 100Hz; it will remove the odd harmonics in the input signal smoothing the square wave to a sinusoidal shape. The CLA background task will continuously buffer the filtered output in a circular buffer.

This example also utilizes the ERAD peripheral to profile the Interrupt Service Routine (ISR) cla1ISR1 (on the C28x core). The ISR contains a loop that simulates storing a random amount of data to a location in order to introduce variability into the cycle measurements. The ERAD peripheral is also configured to count the number of times the system event CLA_INTERRUPT1 occurs.

To properly use the provided ERAD script, the following variables must be set in the scripting environment prior to launching the ERAD script:

- var PROJ_NAME = "erad_debugger_ex4_profileinterrupts_cla"
- var PROJ_WKSPC_LOC = "<proj_workspace_path>"
- var PROJ_CONFIG = "<name of active configuration [CPU1_FLASH|CPU1_RAM]>"

To run the ERAD script, use the following command in the scripting console:

- loadJSFile("<proj_workspace_path>\\erad_debugger_ex4_profileinterrupts_cla\\erad_ex4_profile_interru
  0);

Note that the script must be run after loading and running the .out on the C28x core.

The included JavaScript file, erad_ex4_profile_interrupts_cla.js, uses Debug Server Scripting (DSS) features. For information on using the DSS, please visit: http://software-dl.ti.com/ccs/esd/documents/users_guide/sdto_dss_handbook.html

This example uses 4 HW breakpoints and 2 counters:

- HWBP_1 : PC = start address of cla1Isr1
- HWBP_2 : PC = end address of cla1Isr1
- CTM_1 : Used to count the cla1Isr1 execution cycles. Configured in start-stop mode with start event as HWBP_1 and stop event as HWBP_2
- CTM_2 : Used to count the number of times the system event CLA_INTERRUPT1 event has occurred. Configured in rising-edge count mode with counting input as system event CLA_INTERRUPT1 (INP_SEL[26])

External Connections

- connect A0 to EPWM4A

Watch Variables

- ISR_count - A counter that signifies how many times cla1ISR1 executes

Profiling Script Output

- Current ISR cycle count (CTM_1)
- Max ISR cycle count (maximum value of CTM_1)
- Interrupt occurrence count (CTM_2)

6.122 FSI daisy chain topology, lead device example

fsi_ex16_daisy_handshake_lead is for the lead device in the daisy-chain loop, fsi_ex16_daisy_handshake_node for the other N-1 devices(N>=2).

In the code, there are different settings provided: [define FSI_DMA_ENABLE 0] represents FSI communication using CPU control. [define FSI_DMA_ENABLE 1] represents FSI communication using DMA control, enabling FSIRX to trigger a DMA event and move the RX FSI data to the TX FSI buffer

In a real scenario two separate devices may power up in arbitrary order and there is a need to establish a clean communication link which ensures that receiver side is flushed to properly interpret the start of a new valid frame.

The node devices in the daisy chain topology respond to the handshake sequence and forwards the information to the next device in the chain.

After above synchronization steps, FSI Rx can be configured as per use case i.e. nWords, lane width, enabling events, etc and start the infinite transfers. More details on establishing the communication link can be found in the device TRM.

User can edit some of configuration parameters as per use case, similar to other examples.

nWords - Number of words per transfer may be from 1 -16 **nLanes** - Choice to select single or double lane for frame transfers **txUserData** - User data to be sent with Data frame **txDataFrameTag** - Frame tag used for Data

transfers **txPingFrameTag** - Frame tag used for Ping transfers **txPingTimeRefCntr** - Tx Ping timer reference counter **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

External Connections

For the FSI daisy-chain topology external connections are required to be made between the devices in the chain. Each devices FSI TX pins need to be connected to the FSI RX pins of the next device in the chain (or ring). See below for external connections to include and GPIOs used:

External Connections Required:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frames received back
- **error** Non zero for transmit/receive data mismatch

6.123 FSI daisy chain topology, node device example

fsi_ex16_daisy_handshake_lead is for the lead device in the daisy-chain loop, fsi_ex16_daisy_handshake_node for the other N-1 devices(N>=2).

In the code, there are different settings provided: [define FSI_DMA_ENABLE 0] represents FSI communication using CPU control. [define FSI_DMA_ENABLE 1] represents FSI communication using DMA control, enabling FSIRX to trigger a DMA event and move the RX FSI data to the TX FSI buffer

In a real scenario two separate devices may power up in arbitrary order and there is a need to establish a clean communication link which ensures that receiver side is flushed to properly interpret the start of a new valid frame.

The node devices in the daisy chain topology respond to the handshake sequence and forwards the information to the next device in the chain.

After above synchronization steps, FSI Rx can be configured as per use case i.e. nWords, lane width, enabling events, etc and start the infinite transfers. More details on establishing the communication link can be found in the device TRM.

User can edit some of configuration parameters as per use case, similar to other examples.

nWords - Number of words per transfer may be from 1 -16 **nLanes** - Choice to select single or double lane for frame transfers **txUserData** - User data to be sent with Data frame **txDataFrameTag** - Frame tag used for Data

transfers **txPingFrameTag** - Frame tag used for Ping transfers **txPingTimeRefCntr** - Tx Ping timer reference counter **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

External Connections

For the FSI daisy-chain topology external connections are required to be made between the devices in the chain. Each devices FSI TX pins need to be connected to the FSI RX pins of the next device in the chain (or ring). See below for external connections to include and GPIOs used:

External Connections Required:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frames received back
- **error** Non zero for transmit/receive data mismatch

6.124 FSI Loopback:CPU Control

Example sets up infinite data frame transfers where trigger happens through **CPU**. Automatic(Hw triggered) Ping frame transmission is also setup along with data.

User can edit some of configuration parameters as per usecase. These are as below. Default values can be referred in code where these globals are defined

- **nWords** - Number of words per transfer may be from 1 -16
- **nLanes** - Choice to select single or double lane for frame transfers
- **fsiClock** - FSI Clock used for transfers
- **txUserData** - User data to be sent with Data frame
- **txDataFrameTag** - Frame tag used for Data transfers
- **txPingFrameTag** - Frame tag used for Ping transfers
- **txPingTimeRefCntr** - Tx Ping timer reference counter
- **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

For any errors during transfers i.e. **error** events such as Frame Overrun, Underrun, Watchdog timeout and CRC/EOF/TYPE errors, execution will stop immediately and status variables can be looked into for more details. Execution will also stop for any mismatch between received data and sent ones and also if transfers takes unusually long time(detected through software counters - txTimeOutCntr and rxTimeOutCntr)

External Connections

For FSI internal loopback (EXTERNAL_FSI_ENABLE == 0), no external connections needed

For FSI external loopback (EXTERNAL_FSI_ENABLE == 1), external connections are required. The FSI TX pins should be connected to the FSI RX pins of the same device. See below for external connections to include and GPIOs used:

External Connections Required between FSI TX and RX of the same device:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.125 FSI Loopback CLA control

Example sets up infinite data frame transfers where trigger happens through **CLA**. Automatic(Hw triggered) Ping frame transmission is also setup along with data. This example is similar to fsi_ex1_loopback_cpucontrol and only different in in the sense that data frame transfer are trigged from a CLA task. Using CLA will release some of load from CPU and help it in providing time for other tasks.

User can edit some of configuration parameters as per usecase. These are as below. Default values can be referred in code where these globals are defined

- **nWords** - Number of words per transfer may be from 1 -16
- **nLanes** - Choice to select single or double lane for frame transfers
- **fsiClock** - FSI Clock used for transfers
- **txUserData** - User data to be sent with Data frame
- **txDataFrameTag** - Frame tag used for Data transfers
- **txPingFrameTag** - Frame tag used for Ping transfers

- **txPingTimeRefCntr** - Tx Ping timer reference counter
- **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

For any errors during transfers i.e. **error** events such as Frame Overrun, Underrun, Watchdog timeout and CRC/EOF/TYPE errors, execution will stop immediately and status variables can be looked into for more details. Execution will also stop for any mismatch between received data and sent ones and also if transfers takes unusually long time(detected through software counters - txTimeOutCntr and rxTimeOutCntr)

External Connections

For FSI internal loopback (EXTERNAL_FSI_ENABLE == 0), no external connections needed

For FSI external loopback (EXTERNAL_FSI_ENABLE == 1), external connections are required. The FSI TX pins should be connected to the respective FSI RX pins of the same device. See below for external connections to include and GPIOs used:

External Connections Required between FSI TX and RX of the same device:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.126 FSI DMA frame transfers:DMA Control

Example sets up infinite data frame transfers where DMA trigger happens once through CPU and then DMA takes control to transfer data iteratively. This example demonstrates the FSI feature about triggering DMA events which in turn can copy data and trigger next transfer.

Two DMA channels are setup for FSI Tx operation and two for Rx. Four areas in GSx memories are also setup as source and sink for data and tag values of frame under transmission.

Automatic(Hw triggered) Ping frame transmission is also setup along with data.

If there are any comparison failures during transfers or any of error event occurs, execution will stop.

External Connections

For FSI internal loopback (EXTERNAL_FSI_ENABLE == 0), no external connections needed

For FSI external loopback (EXTERNAL_FSI_ENABLE == 1), external connections are required. The FSI TX pins should be connected to the respective FSI RX pins of the same device. See below for external connections to include and GPIOs used:

External Connections Required between FSI TX and RX of the same device:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **countDMAtransfers** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.127 FSI data transfer by external trigger

FSI frame transfer can be triggered by external sources. It can connect up to 32 trigger sources but as of now, only 16 ePWMx-SOCy(x-1:8, y-A:B) are supported. FSI supports external trigger for both PING and DATA frame transfers and in this example we demonstrate how to setup infinite DATA transfers using selectable ePWM-SOC as a trigger source. The TB counter for ePWM operation is in up/down count mode for this example.

Automatic(Hw triggered) Ping frame transmission is also setup along with data.

If there are any comparison failures during transfers or any of error event occurs, execution will stop.

External Connections

For FSI internal loopback (EXTERNAL_FSI_ENABLE == 0), no external connections needed

For FSI external loopback (EXTERNAL_FSI_ENABLE == 1), external connections are required. The FSI TX pins should be connected to the respective FSI RX pins of the same device. See below for external connections to include and GPIOs used:

External Connections Required between FSI TX and RX of the same device:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK

- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.128 FSI data transfers upon CPU Timer event

Example sets up infinite data frame transfers where trigger comes from ISR handling the periodic CPU Timer event. Automatic(Hw triggered) Ping frame transmission is also setup along with data.

CPU Timer0 is chosen for setting up periodic timer events. User can choose any other Timer-1/Timer-2 as well.

Automatic(Hw triggered) Ping frame transmission is also setup along with data.

If there are any comparison failures during transfers or any of error event occurs, execution will stop.

External Connections

For FSI internal loopback (EXTERNAL_FSI_ENABLE == 0), no external connections needed

For FSI external loopback (EXTERNAL_FSI_ENABLE == 1), external connections are required. The FSI TX pins should be connected to the respective FSI RX pins of the same device. See below for external connections to include and GPIOs used:

External Connections Required between FSI TX and RX of the same device:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.129 FSI and SPI communication(fsi_ex6_spi_main_tx)

FSI supports SPI compatibility mode to talk to the devices not having FSI but SPI module. Example sets up infinite data frame transfers where FSI acts like main Tx and SPI as remote Rx. API to decode FSI frame received at SPI end is implemented and checks are made to ensure received details(frame tag/type, userdata, data) match with transfered frame.

If there are any comparison failures during transfers or any of error event occurs, execution will stop.

External Connections

For FSI <-> SPI communication, make below connections in GPIO settings

- GPIO_2 -> GPIO_18 :: To connect FSITX_CLK with SPICLKA
- GPIO_0 -> GPIO_16 :: To connect FSITX_TX0 with SPIPICOA
- GPIO_1 -> GPIO_19 :: To connect FSITX_TX1 with SPIPTEA

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.130 FSI and SPI communication(fsi_ex7_spi_remote_rx)

FSI supports SPI compatibility mode to talk to the devices not having FSI but SPI module. Example sets up infinite data frame transfers where FSI acts like remote Rx and SPI as main Rx. API to build the FSI frame at SPI end before transfer is implemented in SW and checks are made to ensure received details(frame tag/type, userdata, data) on FSI Rx match with transferred data.

If there are any comparison failures during transfers or any of error event occurs, execution will stop.

External Connections

For FSI(Rx) <-> SPI(Tx) communication, make connections in GPIO settings

There is no requirement for a chip select signal to be used when connected to the FSIRX. This is because the FSIRX will respond to any incoming clock edge.

- GPIO_13 -> GPIO_18 :: To connect FSIRXCLKA with SPICLKA
- GPIO_12 -> GPIO_16 :: To connect FSIRXD0A with SPIPICOA

Watch Variables

- **dataFrameCntr** Number of Data frame transfered
- **error** Non zero for transmit/receive data mismatch

6.131 FSI P2Point Connection:Rx Side

Example sets up FSI receiving device in a point to point connection to the FSI transmitting device. Example code to set up FSI transmit device is implemented in a separate file.

In a real scenario two separate devices may power up in arbitrary order and there is a need to establish a clean communication link which ensures that receiver side is flushed to properly interpret the start of a new valid frame.

There is no true concept of a main or a remote node in the FSI protocol, but to simplify the data flow and connection we can consider transmitting device as main and receiving side as remote. Transmitting side will be driver of initialization sequence.

Handshake mechanism which must take place before actual data transmission can be usecase specific; points described below can be taken as an example on how to implement the handshake from receiving side -

- Setup the receiver interrupts to detect PING type frame reception
- Begin the first PING loop + Wait for receiver interrupt + If the FSI Rx has received a PING frame with **FSI_FRAME_TAG0**, come out of loop. Otherwise iterate the loop again.
- Begin the second PING loop + Send the Flush sequence + Send the PING frame with tag + Wait for receiver interrupt + If the FSI Rx has received a PING frame with **FSI_FRAME_TAG1**, come out of loop. Otherwise iterate the loop again.
  - Now, the receiver side has received the acknowledged PING frame(tag1), so it is ready for normal operation further.

After above synchronization steps, FSI Rx can be configured as per usecase i.e. nWords, lane width, enabling events etc and start the infinite transfers. More details on establishing the communication link can be found in device TRM.

User can edit some of configuration parameters as per usecase, similar to other examples.

nWords - Number of words per transfer may be from 1 -16 **nLanes** - Choice to select single or double lane for frame transfers **fsiClock** - FSI Clock used for transfers **txUserData** - User data to be sent with Data frame **txDataFrameTag** - Frame tag used for Data transfers **txPingFrameTag** - Frame tag used for Ping transfers **txPingTimeRefCntr** - Tx Ping timer reference counter **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

External Connections

For FSI external P2P connection, external connections are required to be made between two devices. Device 1's FSI TX and RX pins need to be connected to device 2's FSI RX and TX pins respectively. See below for external connections to make and GPIOs used:

External connections required between independent RX and TX devices:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK
- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame received
- **error** Non zero for transmit/receive data mismatch

6.132 FSI P2Point Connection:Tx Side

Example sets up FSI transmitting device in a point to point connection to the FSI receiving device. Example code to set up FSI receiving device is implemented in a separate file.

In a real scenario two separate devices may power up in arbitrary order and there is a need to establish a clean communication link which ensures that receiver side is flushed to properly interpret the start of a new valid frame.

There is no true concept of a main or a remote node in the FSI protocol, but to simplify the data flow and connection we can consider transmitting device as main and receiving side as remote. Transmitting side will be driver of initialization sequence.

Handshake mechanism which must take place before actual data transmission can be usecase specific; points described below can be taken as an example on how to implement the handshake from transmitting side -

- Setup the receiver interrupts to detect PING type frame reception
- Begin the PING loop + Send the Flush sequence + Send a PING frame with the frame tag **FSI_FRAME_TAG0** + Wait for some time(determined by application) + If the FSI Rx has received a PING frame with **FSI_FRAME_TAG1**, come out of loop. Otherwise iterate the loop again
  - Send a PING frame with the frame tag FSI_FRAME_TAG1

After above synchronization steps, FSI Tx can be configured as per usecase i.e. nWords, lane width, enabling events etc and start the infinite transfers. More details on establishing the communication link can be found in device TRM.

User can edit some of configuration parameters as per usecase, similar to other examples.

nWords - Number of words per transfer may be from 1 -16 **nLanes** - Choice to select single or double lane for frame transfers **fsiClock** - FSI Clock used for transfers **txUserData** - User data to be sent with Data frame **txDataFrameTag** - Frame tag used for Data transfers **txPingFrameTag** - Frame tag used for Ping transfers **txPingTimeRefCntr** - Tx Ping timer reference counter **rxWdTimeoutRefCntr** - Rx Watchdog timeout reference counter

External Connections

For FSI external P2P connection, external connections are required to be made between two devices. Device 1's FSI TX and RX pins need to be connected to device 2's FSI RX and TX pins respectively. See below for external connections to make and GPIOs used:

External connections required between independent RX and TX devices:

- FSIRX_CLK to FSITX_CLK
- FSIRX_RX0 to FSITX_TX0
- FSIRX_RX1 to FSITX_TX1

ControlCard FSI Header GPIOs:

- GPIO_27 -> FSITX_CLK

- GPIO_26 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_13 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_11 -> FSIRX_RX1

Watch Variables

- **dataFrameCntr** Number of Data frame transmitted
- **error** Non zero for transmit/receive data mismatch

6.133 Device GPIO Setup

Configures the device GPIO into two different configurations This code is verbose to illustrate how the GPIO could be setup. In a real application, lines of code can be combined for improved code size and efficiency.

This example only sets-up the GPIO. Nothing is actually done with the pins after setup.

In general:

- All pullup resistors are enabled. For ePWMs this may not be desired.
- Input qual for communication ports (CAN, SPI, SCI, I2C) is asynchronous
- Input qual for Trip pins (TZ) is asynchronous
- Input qual for eCAP and eQEP signals is synch to SYSCLKOUT
- Input qual for some I/O's and __interrupts may have a sampling window

6.134 Device GPIO Toggle

Configures the device GPIO through the sysconfig file. The GPIO pin is toggled in the infinit loop.

6.135 Device GPIO Interrupt

Configures the device GPIOs through the sysconfig file. One GPIO output pin, and one GPIO input pin is configured. The example then configures the GPIO input pin to be the source of an external interrupt which toggles the GPIO output pin.

6.136 External Interrupt (XINT)

In this example AIO pins are configured as digital inputs. Two other GPIO signals (connected externally to AIO pins) are toggled in software to trigger external interrupt through AIO225 and AIO226 (AIO225 assigned to XINT1 and AIO226 assigned to XINT2). The user is required to externally connect these signals for the program to work properly. Each interrupt is fired in sequence: XINT1 first and then XINT2.

GPIO5 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope. **External Connections**

- Connect GPIO0 to AIO225. AIO225 will be assigned to XINT1
- Connect GPIO1 to AIO226. AIO226 will be assigned to XINT2
- GPIO5 can be monitored on an oscilloscope

Watch Variables

- xint1Count for the number of times through XINT1 interrupt
- xint2Count for the number of times through XINT2 interrupt
- loopCount for the number of times through the idle loop

6.137 HRCAP Capture and Calibration Example

This example configures an ECAP to use HRCAP functionality to capture time between edges on input GPIO2.

External Connections

The user must provide a signal to GPIO2. XCLKOUT has been configured to an output GPIO and can be externally jumped to serve this purpose. See Sysconfig file for XCLKOUT GPIO selected.

Watch Variables

- onTime1, onTime2
- offTime1, offTime2
- period1, period2

6.138 HRPWM Duty Control with SFO

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

External Connections

- Monitor ePWM1/2/3/4 A/B pins on an oscilloscope.

6.139 HRPWM Slider

This example modifies the MEP control registers to show edge displacement due to HRPWM. Control blocks of the respective ePWM module channel A and B will have fine edge movement due to HRPWM logic.

Monitor ePWM1 A/B pins on an oscilloscope.

6.140 HRPWM Period Control

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up-Down count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

External Connections

- Monitor ePWM1/2/3/4 A/B pins on an oscilloscope.

6.141 HRPWM Duty Control with UPDOWN Mode

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

External Connections

- Monitor ePWM1/2/3/4 A/B pins on an oscilloscope.

6.142 HRPWM Slider Test

This example modifies the MEP control registers to show edge displacement due to HRPWM. Control blocks of the respective ePWM module channel A and B will have fine edge movement due to HRPWM logic. Load the hrpwm_slider.gel file. Select the HRPWM_eval from the GEL menu. A FineDuty slider graphics will show up in CCS. Load the program and run. Use the Slider to and observe the EPWM edge displacement for each slider step change. This explains the MEP control on the EPwmxA channels.

Monitor ePWM1 & ePWM2 A/B pins on an oscilloscope.

6.143 HRPWM Duty Up Count

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

To run this example:

1. Run this example at maximum SYSCLKOUT
2. Activate Real time mode
3. Run the code

External Connections

- Monitor ePWM1/2 A/B pins on an oscilloscope.

Watch Variables

- status - Example run status
- updateFine - Set to 1 use HRPWM capabilities and observe in fine MEP steps(default) Set to 0 to disable HRPWM capabilities and observe in coarse SYSCLKOUT cycle steps

6.144 HRPWM Period Up-Down Count

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up-Down count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V8 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value
- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

To run this example:

1. Run this example at maximum SYSCLKOUT
2. Activate Real time mode
3. Run the code

External Connections

- Monitor ePWM1/2 A/B pins on an oscilloscope.

Watch Variables

- updateFine - Set to 1 use HRPWM capabilities and observe in fine MEP steps(default) Set to 0 to disable HRPWM capabilities and observe in coarse SYSCLKOUT cycle steps

6.145 I2C Digital Loopback with FIFO Interrupts

This program uses the internal loopback test mode of the I2C module. Both the TX and RX I2C FIFOs and their interrupts are used. The pinmux and I2C initialization is done through the sysconfig file.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001

0001 0002

0002 0003

....

00FE 00FF

00FF 0000

etc..

This pattern is repeated forever.

External Connections

- None

Watch Variables

- **sData** - Data to send
- **rData** - Received data
- **rDataPoint** - Used to keep track of the last position in the receive stream for error checking

6.146 I2C EEPROM

This program will write 1-14 words to EEPROM and read them back. The data written and the EEPROM address written to are contained in the message structure, i2cMsgOut. The data read back will be contained in the message structure i2cMsgIn.

External Connections

- Connect external I2C EEPROM at address 0x50
- Connect DEVICE_GPIO_PIN_SDAA on to external EEPROM SDA (serial data) pin
- Connect DEVICE_GPIO_PIN_SCLA on to external EEPROM SCL (serial clock) pin

Watch Variables

- **i2cMsgOut** - Message containing data to write to EEPROM
- **i2cMsgIn** - Message containing data read from EEPROM

6.147 I2C Digital External Loopback with FIFO Interrupts

This program uses the I2CA and I2CB modules for achieving external loopback. The I2CA TX FIFO and the I2CB RX FIFO are used along with their interrupts.

A stream of data is sent on I2CA and then compared to the received stream on I2CB. The sent data looks like this:

0000 0001

0001 0002

0002 0003

....

00FE 00FF

00FF 0000

etc..

This pattern is repeated forever.

External Connections

- Connect SCLA(DEVICE_GPIO_PIN_SCLA) to SCLB (DEVICE_GPIO_PIN_SCLB)
- and SDAA(DEVICE_GPIO_PIN_SDAA) to SDAB (DEVICE_GPIO_PIN_SDAB)
- Connect DEVICE_GPIO_PIN_LED1 to an LED used to depict data transfers.

Watch Variables

- **sData** - Data to send
- **rData** - Received data
- **rDataPoint** - Used to keep track of the last position in the receive stream for error checking

6.148 I2C EEPROM

This program will shows how to perform different EEPROM write and read commands using I2C polling method EEPROM used for this example is AT24C256

External Connections

- Connect external I2C EEPROM at address 0x50 ——————————— Signal | I2CA | EEPROM ——————————— SCL | DEVICE_GPIO_PIN_SCLA | SCL SDA | DEVICE_GPIO_PIN_SDAA | SDA Make sure to connect GND pins if EEPROM and C2000 device are in different board. ———————————

6.149 I2C controller target communication using FIFO interrupts

This program shows how to use I2CA and I2CB modules in both controller and target configuration This example uses I2C FIFO interrupts and doesn't using polling

Example1: I2CA as controller Transmitter and I2CB working target Receiver Example2: I2CA as controller Receiver and I2CB working target Transmitter Example3: I2CB as controller Transmitter and I2CA working target Receiver Example4: I2CB as controller Receiver and I2CA working target Transmitter

External Connections on launchpad should be made as shown below

——————————— Signal | I2CA | I2CB ——————————— SCL | DEVICE_GPIO_PIN_SCLA | DEVICE_GPIO_PIN_SCLB SDA | DEVICE_GPIO_PIN_SDAA | DEVICE_GPIO_PIN_SDAB ———————————

Watch Variables in memory window

- **I2CA_TXdata**

- **I2CA_RXdata**
- **I2CB_TXdata**
- **I2CB_RXdata** stream for error checking

##

6.150 I2C EEPROM

This program will shows how to perform different EEPROM write and read commands using I2C interrupts EEPROM used for this example is AT24C256

External Connections

- Connect external I2C EEPROM at address 0x50 ———————— Signal | I2CA | EEPROM ———————— SCL | DEVICE_GPIO_PIN_SCLA | SCL SDA | DEVICE_GPIO_PIN_SDAA | SDA Make sure to connect GND pins if EEPROM and C2000 device are in different board. ———————— Example 1: EEPROM Byte Write Example 2: EEPROM Byte Read Example 3: EEPROM word (16-bit) write Example 4: EEPROM word (16-bit) read Example 5: EEPROM Page write Example 6: EEPROM word Paged read

Watch Variables

- **TX_MsgBuffer** - Message buffer which stores the data to be transmitted
- **RX_MsgBuffer** - Message buffer which stores the data to be received

##

6.151 External Interrupts (ExternalInterrupt)

This program sets up GPIO0 as XINT1 and GPIO1 as XINT2. Two other GPIO signals are used to trigger the interrupt (GPIO10 triggers XINT1 and GPIO11 triggers XINT2). The user is required to externally connect these signals for the program to work properly.

XINT1 input is synced to SYSCLKOUT.

XINT2 has a long qualification - 6 samples at 510∗SYSCLKOUT each.

GPIO16 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope.

Each interrupt is fired in sequence - XINT1 first and then XINT2

External Connections

- Connect GPIO10 to GPIO0. GPIO0 will be assigned to XINT1
- Connect GPIO11 to GPIO1. GPIO1 will be assigned to XINT2

Monitor GPIO16 with an oscilloscope. GPIO16 will be high outside of the ISRs and low within each ISR.

Watch Variables

- xint1Count for the number of times through XINT1 interrupt
- xint2Count for the number of times through XINT2 interrupt
- loopCount for the number of times through the idle loop

6.152 Multiple interrupt handling of I2C, SCI & SPI Digital Loopback

This program is used to demonstrate how to handle multiple interrupts when using multiple communication peripherals like I2C, SCI & SPI Digital Loopback all in a single example. The data transfers would be done with FIFO Interrupts.

It uses the internal loopback test mode of these modules. Both the TX and RX FIFOs and their interrupts are used. Other than boot mode pin configuration, no other hardware configuration is required.

A stream of data is sent and then compared to the received stream. The sent data looks like this for I2C and SCI:

0000 0001

0001 0002

0002 0003

....

00FE 00FF

00FF 0000

etc..

The sent data looks like this for SPI:

0000 0001

0001 0002

0002 0003

....

FFFE FFFF

FFFF 0000

etc..

This pattern is repeated forever.

External Connections

- None

Watch Variables

- **sDatai2cA** - Data to send through I2C
- **rDatai2cA** - Received I2C data
- **rDataPoint** - Used to keep track of the last position in the receive I2C stream for error checking

- **sDataspiA** - Data to send through SPI
- **rDataspiA** - Received SPI data
- **rDataPointspiA** - Used to keep track of the last position in the receive SPI stream for error checking
- **sDatasciA** - SCI Data being sent
- **rDatasciA** - SCI Data received
- **rDataPointA** - Keep track of where we are in the SCI data stream. This is used to check the incoming data

6.153 CPU Timer Interrupt Software Prioritization

This examples demonstrates the software prioritization of interrupts through CPU Timer Interrupts. Software prioritization of interrupts is achieved by enabling interrupt nesting.

In this device, hardware priorities for CPU Timer 0, 1 and 2 are set as timer 0 being highest priority and timer 2 being lowest priority. This example configures CPU Timer0, 1, and 2 priority in software with timer 2 priority being highest and timer 0 being lowest in software and prints a trace for the order of execution.

For most applications, the hardware prioritizing of the interrupts is sufficient. For applications that need custom prioritizing, this example illustrates how this can be done through software.User specific priorities can be configured in sw_prioritized_isr_level.h header file.

To enable interrupt nesting, following sequence needs to followed in ISRs. **Step 1:** Set the global priority: Modify the IER register to allow CPU interrupts with a higher user priority to be serviced. Note: at this time IER has already been saved on the stack. **Step 2:** Set the group priority: (optional) Modify the appropriate PIEIERx register to allow group interrupts with a higher user set priority to be serviced. Do NOT clear PIEIER register bits from another group other than that being serviced by this ISR. Doing so can cause erroneous interrupts to occur. **Step 3:** Enable interrupts: There are three steps to do this: a. Clear the PIEACK bits b. Wait at least one cycle c. Clear the INTM bit. **Step 4:** Run the main part of the ISR **Step 5:** Set INTM to disable interrupts. **Step 6:** Restore PIEIERx (optional depending on step 2) **Step 7:** Return from ISR

Refer to below link on more details on Interrupt nesting in C28x devices: <C2000Ware>.html

External Connections

- None

Watch Variables

- traceISR - shows the order in which ISRs are executed.

6.154 EPWM Real-Time Interrupt

This example configures the ePWM1 Timer and increments a counter each time the ISR is executed. ePWM interrupt can be configured as time critical to demonstrate real-time mode functionality and real-time interrupt capability.

The example uses 2 LEDs - LED1 is toggled in the main loop and LED2 is toggled in the EPWM Timer Interrupt. FREE_SOFT bits and DBGIER.INT3 bit must be set to enable ePWM1 interrupt to be time critical and operational in real time mode after halt command

How to run the example?

- Add the watch variables as mentioned below and enable Continuous Refresh.
- Enable real-time mode (Run->Advanced->Enable Silicon Real-time Mode)
- Initially, the DBGIER register is set to 0 and the EPWM emulation mode is set to EPWM_EMULATION_STOP_AFTER_NEXT_TB (FREE_SOFT = 0)
- When the application is running, you will find both LEDs toggling and the watch variables EPwm1TimerIntCount, EPwm1Regs.TBCTR getting updated.
- When the application is halted, both LEDs stop toggling and the watch variables remain constant. EPWM counter is stopped on debugger halt.
- To enable EPWM counter run during debugger halt, set emulation mode as EPWM_EMULATION_FREE_RUN (FREE_SOFT = 2). You will find EPwm1Regs.TBCTR is running, but EPwm1TimerIntCount remains constant. This means, the EPWM counter is running, but the ISRs are not getting serviced.
- To enable real-time interrupts, set DBGIER.INT3 = 1 (EPWM1 interrupt is part of PIE Group 3). You will find that the EPwm1TimerIntCount is incrementing and the LED starts toggling. The EPWM ISR is getting serviced even during a debugger halt.

For more details, watch this video : [C2000 Real-Time Features](https://training.ti.com/c2000-real-time-features)

External Connections

- None

Watch Variables

- EPwm1TimerIntCount - EPWM1 ISR counter
- EPwm1Regs.TBCTR.TBCTR - EPWM1 Time Base counter
- EPwm1Regs.TBCTL.FREE_SOFT - Set this to 2 to enable free run
- DBGIER.INT3 - Set to 1 to enable real time interrupt

6.155 F28P65X LaunchPad Out of Box Demo Example

This program is the demo program that comes pre-loaded on the F28P65X LaunchPad development kit. The program starts by flashing the two user LEDs. After a few seconds the LEDs stop flashing and the device starts sampling ADCINA4 once a second. If the sample is greater than midscale the red LED on the board is lit, while if it is lower the green LED is lit. Sample data is also displayed in a serial terminal via the board's back channel UART. You may view this data by configuring a serial terminal to the correct COM port at 115200 Baud 8-N-1.

External Connections

- Connect to COM port at 115200 Baud 8-N-1 for serial data
- Connect signal to ADCINA4 to change LED based on value

Watch Variables

- None.

6.156 LED Blinky Example

This example demonstrates how to blink a LED.

External Connections

- None.

Watch Variables

- None.

6.157 LIN Internal Loopback with Interrupts

This example configures the LIN module in commander mode for internal loopback with interrupts. The module is setup to perform 8 data transmissions with different transmit IDs and varying transmit data. Upon reception of an ID header, an interrupt is triggered on line 0 and an interrupt service routine (ISR) is called. The received data is then checked for accuracy.

Note:
The example can be adjusted to use interrupt line 1 instead of line 0 by un-commenting "LIN_setInterruptLevel1()"

External Connections

- None.

Watch Variables

- txData - An array with the data being sent
- rxData - An array with the data that was received
- result - The example completion status (PASS = 0xABCD, FAIL = 0xFFFF)
- level0Count - The number of line 0 interrupts
- level1Count - The number of line 1 interrupts

6.158 LIN SCI Mode Internal Loopback with Interrupts

This example configures the LIN module in SCI mode for internal loopback with interrupts. The LIN module performs as a SCI with a set character and frame length in a non-multi-buffer mode. The module is setup to continuously transmit a character, wait to receive that character, and repeat.

External Connections

- None.

Watch Variables

- rxCount - The number of RX interrupts

- transmitChar - The character being transmitted
- receivedChar - The character received

6.159 LIN SCI MODE Internal Loopback with DMA

This example configures the LIN module in SCI mode for internal loopback with the use of the DMA. The LIN module performs as SCI with a set character and frame length in multi-buffer mode. When the transmit buffers in the LINTD0 and LINTD1 registers have enough space, the DMA will transfer data from global variable sData into those transmit registers. Once the received buffers in the LINRD0 and LINRD1 registers contain data,the DMA will transfer the data into the global variable rdata.

When all data has been placed into rData, a check of the validity of the data will be performed in one of the DMA channels' ISRs.

External Connections

- None

Watch Variables

- **sData** - Data to send
- **rData** - Received data

6.160 LIN Internal Loopback without interrupts(polled mode)

This example configures the LIN module in commander mode for internal loopback without interrupts. The module is setup to perform 8 data transmissions with different transmit IDs and varying transmit data. Waits for reception of an ID header. The received data is then checked for accuracy.

External Connections

- None.

Watch Variables

- txData - An array with the data being sent
- rxData - An array with the data that was received
- result - The example completion status (PASS = 0xABCD, FAIL = 0xFFFF)

6.161 LIN Internal Loopback with Interrupts using Sysconfig

This example is similar to ex1 but using syscfg tool to configure the LIN Module paramters. The file lin_ex5_syscfg.syscfg can be updated using the GUI tool to update the configuration parameters. This example configures the LIN module in commander mode for internal loopback with interrupts. The module is setup to perform 8 data transmissions with different transmit IDs and varying transmit data. Upon reception of

an ID header, an interrupt is triggered on line 0 and an interrupt service routine (ISR) is called. The received data is then checked for accuracy.

External Connections

- None.

Watch Variables

- txData - An array with the data being sent
- rxData - An array with the data that was received
- result - The example completion status (PASS = 0xABCD, FAIL = 0xFFFF)
- level0Count - The number of line 0 interrupts
- level1Count - The number of line 1 interrupts

6.162 LIN Incomplete Header Detection

This example demonstrates how an error in the header field of a LIN frame can be detected. It configures the LIN module in responder mode and waits till new frame is received. It configures an external interrupt to trigger by a falling edge in the LIN Rx pin confirming start of frame. In the ISR of XINT, a timer is configured to trigger after max time consumed by reception of a complete LIN frame. If a LIN frame is successfully received the timer is disabled before getting triggered in the LIN ISR else the timer ISR is called which indicates an error in header frame.

External Connections

- LINATX/RX to host node via transciever.

Watch Variables

- linHeaderError - Number of times header error was detected
- rxData - An array with the data that was received if successful reception
- xint1Count - Number of timer XINT triggered

6.163 LIN External Loopback without interrupts(polled mode)

This example configures the LINA module in commander mode and LINB module in responder mode. Commander transmits 8 byte data to the responder with a matching ID configured in responder Receive ID. The received data is then checked for accuracy.

External Connections

- Connect LINA TX/RX with LINB TX/RX via transciever.

Watch Variables

- txData - An array with the data being sent
- rxData - An array with the data that was received
- result - The example completion status (PASS = 0xABCD, FAIL = 0xFFFF)

6.164 MCAN Loopback with Interrupts Example Using SYSCONFIG Tool

This example illustrates the MCAN Loopback functionality. The internal loopback mode is entered. The message transmitted would be received by the node. The last address of memory is used for the Rx buffer. Peripheral configuration is done through SYSCONFIG

External Connections

- None.

Watch Variables

- error - Checks if there is an error that occurred when the data was sent using internal loopback.

6.165 Correctable & Uncorrectable Memory Error Handling

This example demonstrates error handling in case of various erroneous memory read/write operations. Error handling in case of CPU read/write violations, correctable & uncorrectable memory errors has been demonstrated. Correctable memory errors & violations can generate SYS_INT interrupt to CPU while uncorrectable errors lead to NMI generation.

External Connections

- None

Watch Variables

- **testStatusGlobal** - Equivalent to **TEST_PASS** if test finished correctly, else the value is set to **TEST_FAIL**
- **errCountGlobal** - Error counter

6.166 Tune Baud Rate via UART Example

This example demonstrates the process of tuning the UART/SCI baud rate of a C2000 device based on the UART input from another device. As UART does not have a clock signal, reliable communication requires baud rates to be reasonably matched. This example addresses cases where a clock mismatch between devices is greater than is acceptable for communications, requiring baud compensation between boards. As reliable communication only requires matching the EFFECTIVE baud rate, it does not matter which of the two boards executes the tuning (the board with the less-accurate clock source does not need to be the one to tune; as long as one of the two devices tunes to the other, then proper communication can be established).

To tune the baud rate of this device, SCI data (of the desired baud rate) must be sent to this device. The input SCI baud rate must be within the +/- MARGINPERCENT of the TARGETBAUD chosen below. These two variables are defined below, and should be chosen based on the application requirements. Higher MARGINPERCENT will allow more data to be considered "correct" in noisy conditions, and may decrease accuracy. The TARGETBAUD is what was expected to be the baud rate, but due to clock differences, needs to be tuned for better communication robustness with the other device.

NOTE: Lower baud rates have more granularity in register options, and therefore tuning is more affective at these speeds.

External Connections for Control Card

- SCIA_RX/eCAP1 is on GPIO9, connect to incoming SCI communications
- SCIA_TX is on GPIO8, for observation externally

Watch Variables

- **avgBaud** - Baud rate that was detected and set after tuning

6.167 SCI FIFO Digital Loop Back

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. The pinmux and SCI modules are configured through the sysconfig file.

This test uses the loopback test mode of the SCI module to send characters starting with 0x00 through 0xFF. The test will send a character and then check the receive buffer for a correct match.

Watch Variables

- **loopCount** - Number of characters sent
- **errorCount** - Number of errors detected
- **sendChar** - Character sent
- **receivedChar** - Character received

6.168 SCI Digital Loop Back with Interrupts

This test uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Both interrupts and the SCI FIFOs are used.

A stream of data is sent and then compared to the received stream. The SCI-A sent data looks like this:

00 01

01 02

02 03

....

FE FF

FF 00

etc..

The pattern is repeated forever.

Watch Variables

- **sDataA** - Data being sent

- **rDataA** - Data received
- **rDataPointA** - Keep track of where we are in the data stream. This is used to check the incoming data

6.169 SCI Echoback

This test receives and echo-backs data through the SCI-A port.

A terminal such as 'putty' can be used to view the data from the SCI and to send information to the SCI. Characters received by the SCI port are sent back to the host.

Running the Application Open a COM port with the following settings using a terminal:

- Find correct COM port
- Bits per second = 9600
- Data Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

The program will print out a greeting and then ask you to enter a character which it will echo back to the terminal.

Watch Variables

- loopCounter - the number of characters sent

External Connections

Connect the USB cable from Control card J1:A to PC

6.170 stdout redirect example

This test transmits data through the SCI-A port to a terminal

A terminal such as 'putty' can be used to view the data from the SCI. Characters received by the SCI port are sent back to the host.

Running the Application Open a COM port with the following settings using a terminal:

- Find correct COM port
- Bits per second = 9600
- Data Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

The program will print out three sentences: one to the SCIA, one to CCS, and a final one to SCIA.

External Connections

Connect the SCI-A port to a PC via a transceiver and cable.

- DEVICE_GPIO_PIN_SCIRXDA is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- DEVICE_GPIO_PIN_SCITXDA is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

6.171 SDFM Filter Sync CPU

In this example, SDFM filter data is read by CPU in SDFM ISR routine. The SDFM configuration is shown below:

- SDFM used in this example - SDFM1
- Input control mode selected - MODE0
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - HLT = 0x7FFF (Higher threshold setting)
  - LLT = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 128
  - All the 4 filters are synchronized by using MFE (Master Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 7 bits for Sinc3 filter with OSR = 128
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled
  - All the 4 filter will generate interrupt when a new filter data is available.

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

6.172 SDFM Filter Sync CLA

In this example, SDFM filter data is read by CLA in Cla1Task1. The SDFM configuration is shown below:

- SDFM1 used in this example.For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - hlt = 0x7FFF (Higher threshold setting)
  - llt = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Master Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled
  - All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

6.173 SDFM Filter Sync DMA

In this example, SDFM filter data is read by DMA. The SDFM configuration is shown below:

- SDFM1 used in this example. For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - hlt = 0x7FFF (Higher threshold setting)
  - llt = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Master Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter
  - All the 4 higher threshold comparator interrupts disabled
  - All the 4 lower threshold comparator interrupts disabled
  - All the 4 modulator failure interrupts disabled
  - All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

6.174 SDFM PWM Sync

In this example, SDFM filter data is read by CPU in SDFM ISR routine. The SDFM configuration is shown below:

- SDFM1 is used in this example. For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - hlt = 0x7FFF (Higher threshold setting)
  - llt = 0x0000(Lower threshold setting)

Data filter settings

- All the 4 filter modules enabled
- Sinc3 filter selected
- OSR = 256
- All the 4 filters are synchronized by using PWM (Master Filter enable bit)
- Filter output represented in 16 bit format
- In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256

Interrupt module settings for SDFM filter

- All the 4 higher threshold comparator interrupts disabled
- All the 4 lower threshold comparator interrupts disabled
- All the 4 modulator failure interrupts disabled
- All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

6.175 SDFM Type 1 Filter FIFO

This example configures SDFM1 filter in type 1 to demonstrate data read through CPU in FIFO & non-FIFO mode. Data filter is configured in mode 0 to select SINC3 filter with OSR of 256. Filter output is configured for 16-bit format and data shift of 10 is used.

This example demonstrates the FIFO usage if enabled. FIFO length is set at 16 and data ready interrupt is configured to be triggered when FIFO is full. In this example, SDFM filter data is read by CPU in SDFM Data Ready ISR routine.

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams(SD1-D1, SD1-C1) to (GPIO16, GPIO17)
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams(SD1-D1, SD1-C1) to (GPIO48, GPIO49)
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams(SD1-D1, SD1-C1) to (GPIO122, GPIO123)

Watch Variables

- **filter1Result** - Output of filter 1

6.176 SDFM Filter Sync CLA

In this example, SDFM FIFO will not be filled until a SDSYNC event. On a SDSYNC event, SDFM data filter output will start filling FIFO and stop filling after programmable number 'N' of FIFO is filled.

SDy-C1 (Filter1 channel clock) is internally configured to connected SDy-C2 / SDy-C3 / SDy-C4 SDFM configuration is shown below:

- SDFM1 used in this example.For using SDFM2, few modifications would be needed in the example.
- MODE0 Input control mode selected
- Comparator settings
  - Sinc3 filter selected
  - OSR = 32
  - hlt = 0x7FFF (Higher threshold setting)
  - llt = 0x0000(Lower threshold setting)
- Data filter settings
  - All the 4 filter modules enabled
  - Sinc3 filter selected
  - OSR = 256
  - All the 4 filters are synchronized by using MFE (Master Filter enable bit)
  - Filter output represented in 16 bit format
  - In order to convert 25 bit Data filter into 16 bit format user needs to right shift by 10 bits for Sinc3 filter with OSR = 256
- Interrupt module settings for SDFM filter

- All the 4 higher threshold comparator interrupts disabled
- All the 4 lower threshold comparator interrupts disabled
- All the 4 modulator failure interrupts disabled
- All the 4 filter will generate interrupt when a new filter data is available

External Connections

- SDFM_PIN_MUX_OPTION1 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO16-GPIO31
- SDFM_PIN_MUX_OPTION2 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO48-GPIO63
- SDFM_PIN_MUX_OPTION3 Connect Sigma-Delta streams to (SDx-D1, SDx-C1 to SDx-D4,SDx-C4) on GPIO122-GPIO137

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

6.177 SD FATFS Library Example

This example demonstrates how to use the FATFS library.

External Connections

- Connect the SPI signals identifed in the SysConfig to an SD CARD.

Watch Variables

- None.

6.178 SD FATFS Library Example with exFAT Support

This example demonstrates how to use the FATFS library with exFAT support.

External Connections

- Connect the SPI signals identifed in the SysConfig to an SD CARD.

Watch Variables

- None.

6.179 SPI Digital Loopback

This program uses the internal loopback test mode of the SPI module. This is a very basic loopback that does not use the FIFOs or interrupts. A stream of data is sent and then compared to the received stream. The pinmux and SPI modules are configure through the sysconfig file.

The sent data looks like this:

0000 0001 0002 0003 0004 0005 0006 0007 .... FFFE FFFF 0000

This pattern is repeated forever.

External Connections

- None

Watch Variables

- **sData** - Data to send
- **rData** - Received data

6.180 SPI Digital Loopback with FIFO Interrupts

This program uses the internal loopback test mode of the SPI module. Both the SPI FIFOs and their interrupts are used.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001

0001 0002

0002 0003

....

FFFE FFFF

FFFF 0000

etc..

This pattern is repeated forever.

External Connections

- None

Watch Variables

- **sData** - Data to send
- **rData** - Received data
- **rDataPoint** - Used to keep track of the last position in the receive stream for error checking

6.181 SPI Digital External Loopback without FIFO Interrupts

This program uses the external loopback between two SPI modules. Both the SPI FIFOs and interrupts are not used in this example. SPIA is configured as a peripheral and SPI B is configured as controller. This example demonstrates full duplex communication where both controller and peripheral transmits and receives data simultaneously.

External Connections

-GPIO24 and GPIO16 - SPIPICO -GPIO25 and GPIO17 - SPIPOCI -GPIO26 and GPIO18 - SPICLK -GPIO27 and GPIO19 - SPISTE

Watch Variables

- **TxData_SPIA** - Data send from SPIA (peripheral)
- **TxData_SPIB** - Data send from SPIB (controller)
- **RxData_SPIA** - Data received by SPIA (peripheral)
- **RxData_SPIB** - Data received by SPIB (controller)

6.182 SPI Digital External Loopback with FIFO Interrupts

This program uses the external loopback between two SPI modules. Both the SPI FIFOs and their interrupts are used. SPIA is configured as a peripheral and receives data from SPI B which is configured as a controller.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001

0001 0002

0002 0003

....

FFFE FFFF

FFFF 0000

etc..

This pattern is repeated forever.

External Connections

-GPIO24 and GPIO16 - SPIPICO -GPIO25 and GPIO17 - SPIPOCI -GPIO26 and GPIO18 - SPICLK -GPIO27 and GPIO19 - SPISTE

Watch Variables

- **sData** - Data to send
- **rData** - Received data
- **rDataPoint** - Used to keep track of the last position in the receive stream for error checking

6.183 SPI Digital Loopback with DMA

This program uses the internal loopback test mode of the SPI module. Both DMA interrupts and the SPI FIFOs are used. When the SPI transmit FIFO has enough space (as indicated by its FIFO level interrupt signal), the DMA will transfer data from global variable sData into the FIFO. This will be transmitted to the receive FIFO via the internal loopback.

When enough data has been placed in the receive FIFO (as indicated by its FIFO level interrupt signal), the DMA will transfer the data from the FIFO into global variable rData.

When all data has been placed into rData, a check of the validity of the data will be performed in one of the DMA channels' ISRs.

External Connections

- None

Watch Variables

- **sData** - Data to send
- **rData** - Received data

6.184 SPI EEPROM

This program will write 8 bytes to EEPROM and read them back. The device communicates with the EEPROM via SPI and specific opcodes. This example is written to work with the SPI Serial EEPROM AT25128/256.

External Connections

External Connections

- Connect external SPI EEPROM
- Connect GPIO16 (PICO) to external EEPROM SI pin
- Connect GPIO17 (POCI) to external EEPROM SO pin
- Connect GPIO18 (CLK) to external EEPROM SCK pin
- Connect GPIO11 (CS) to external EEPROM CS pin
- Connect the external EEPROM VCC and GND pins

Watch Variables

- writeBuffer - Data that is written to external EEPROM
- readBuffer - Data that is read back from EEPROM
- error - Error count

6.185 SPI DMA EEPROM

This program will write 8 bytes to EEPROM and read them back. The device communicates with the EEPROM via SPI using DMA and specific opcodes. This example is written to work with the SPI Serial EEPROM AT25128/256.

External Connections

External Connections

- Connect external SPI EEPROM
- Connect GPIO16 (PICO) to external EEPROM SI pin
- Connect GPIO17 (POCI) to external EEPROM SO pin
- Connect GPIO18 (CLK) to external EEPROM SCK pin
- Connect GPIO11 (CS) to external EEPROM CS pin
- Connect the external EEPROM VCC and GND pins

Watch Variables

- writeBuffer - Data that is written to external EEPROM
- SPI_DMA_Handle.RXdata - Data that is read back from EEPROM when number of received bytes is less than 4
- SPI_DMA_Handle.pSPIRXDMA->pbuffer - Start address of received data from EEPROM
- error - Error count

6.186 Missing clock detection (MCD)

This example demonstrates the missing clock detection functionality and the way to handle it. Once the MCD is simulated by disconnecting the OSCCLK to the MCD module an NMI would be generated. This NMI determines that an MCD was generated due to a clock failure which is handled in the ISR.

Before an MCD the clock frequency would be as per device initialization (200Mhz). Post MCD the frequency would move to 10Mhz or INTOSC1.

The example also shows how we can lock the PLL after missing clock, detection, by first explicitly switching the clock source to INTOSC1, resetting the missing clock detect circuit and then re-locking the PLL. Post a re-lock the clock frequency would be 200Mhz but using the INTOSC1 as clock source.

External Connections

- None.

Watch Variables

- **fail** - Indicates that a missing clock was either not detected or was not handled correctly.
- **mcd_clkfail_isr** - Indicates that the missing clock failure caused an NMI to be triggered and called an the ISR to handle it.
- **mcd_detect** - Indicates that a missing clock was detected.
- **result** - Status of a successful handling of missing clock detection

6.187 XCLKOUT (External Clock Output) Configuration

This example demonstrates how to configure the XCLKOUT pin for observing internal clocks through an external pin, for debugging and testing purposes.

In this example, we are using INTOSC1 as the XCLKOUT clock source and configuring the divider as 8. Expected frequency of XCLKOUT = (INTOSC1 freq)/8 = 10/8 = 1.25MHz

View the XCLKOUT on GPIO73 using an oscilloscope.

6.188 CPU Timers

This example configures CPU Timer0, 1, and 2 and increments a counter each time the timer asserts an interrupt.

External Connections

- None

Watch Variables

- cpuTimer0IntCount
- cpuTimer1IntCount
- cpuTimer2IntCount

6.189 UART Echoback

This test receives and echo-backs data through the UARTA port.

A terminal such as 'putty' can be used to view the data from the UART and to send information to the UART. Characters received by the UART port are sent back to the host.

Running the Application Open a COM port with the following settings using a terminal:

- Find correct COM port
- Bits per second = 115200
- Data Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

The program will print out a greeting and then ask you to enter a character which it will echo back to the terminal.

Watch Variables

- None

External Connections

Connect the UARTA port to a PC via a transceiver and cable.

- GPIO85 is UARTARX (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO84 is UARTATX (Connect to Pin2, PC-RX, of serial DB9 cable)

Note:
The pin muxing for the UARTA port needs to be done by the master CPU1. The common configuration example provided in the C28x folder can be used for making GPIO85 as the UART Rx pin and GPIO84 as the UART Tx pin.

6.190 USB HUB Host example

This example application demonstrates how to support a USB keyboard and USB Mouse with a USB Hub. The display will show the connected devices on the USB hub.

To run the example you should connect a USB Hub to the microUSB port on the top of the controlCARD and open up a serial terminal with the above settings to view the characters typed on the keyboard. Allow the example to run with the hub connected and then connect the USB Host Mouse or Keyboard.

When a USB Mouse is connected on the Hub the position of the mouse pointer and the state of the mouse buttons are output to the display. Similarly when a USB Keyboard is connected, any key press on the keyboard will cause them to be sent out the SCI at 115200 baud with no parity, 8 bits and 1 stop bit.

This example is for depicting the usage of Hub.

There are some limitations in this example : 1. The Example fails to recognize the USB Hub and the device if the Mouse/Keyboard is already connected to the USB Hub and the Hub is connected to the Micro USB of the Control Card. 2. The same port should not be used to connect a Keyboard and mouse.

6.191 USB CDC serial example

This example application turns the evaluation kit into a virtual serial port when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect SCIA traffic to and from the USB host system.

Connect USB cables from your PC to both the mini and microUSB connectors on the controlCARD.Figure out what COM ports your controlCARD is enumerating (typically done using Device Manager in Windows) and open a serial terminal to each of with the settings 115200 Baud 8-N-1. Characters typed in one terminal should be echoed in the other and vice versa.

A driver information (INF) file for use with Windows XP, Windows 7 and Windows 10 can be found in the windows_drivers directory.

6.192 USB HID Mouse Device

This example application turns the evaluation board into a USB mouse supporting the Human Interface Device class. After loading and running the example simply connect the PC to the controlCARDs microUSB port using a USB cable, and the mouse pointer will move in a square pattern for the duration of the time it is plugged in.

SCIA, connected to the FTDI virtual COM port and running at 115200, 8-N-1, is used to display messages from this application.

6.193 USB Device Keyboard

This example application turns the evaluation board into a USB keyboard supporting the Human Interface Device class. The global variable ui32Button should be modified to wake up the USB. Care should be taken to ensure that the active window can safely receive the text; enter is not pressed at any point so no actions are attempted by the host if a terminal window is used.

The device implemented by this application also supports USB remote wake up allowing it to request the host to reactivate a suspended bus. If the bus is suspended (as indicated on the application display), updating ui32Button will request a remote wakeup assuming the host has not specifically disabled such requests.

To run the example compile the project, load to the target, and run the example. After the example is running, connect a USB cable from the PC to the microUSB port on the controlCARD.Modify ui32Button value in the expressions window and then focus should be on the window so that we can receive keyboard input (i.e. NotePad).

6.194 USB Generic Bulk Device

This example provides a generic USB device offering simple bulk data transfer to and from the host. The device uses a vendor-specific class ID and supports a single bulk IN endpoint and a single bulk OUT endpoint. Data received from the host is assumed to be ASCII text and it is echoed back with the case of all alphabetic characters swapped.

SCIA, connected to the FTDI virtual COM port and running at 115200, 8-N-1, is used to display messages from this application.

A Windows INF file for the device is provided under the windows drivers directory. This INF contains information required to install the WinUSB subsystem on WindowsXP, Windows 7 and Windows 10. WinUSB is a Windows subsystem allowing user mode applications to access the USB device without the need for a vendor-specific kernel mode driver.

A sample Windows command-line application, usb_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided. Project files are included to allow the examples to be built using Microsoft VisualStudio. Source code for this application can be found in directory ∼/C2000Ware/utilities/tools/{Device}/usb_bulk_example/Release

6.195 USB HID Mouse Host

This application demonstrates the handling of a USB mouse attached to the evaluation kit. Once attached, the position of the mouse pointer and the state of the mouse buttons are output to the display.

SCIA, which is connected to the FTDI virtual serial port on the controlCARD board, is configured for 115200 bits per second, and 8-N-1 mode. When a HID compliant mouse is connected to the microUSB port on the top of the controlCARD, position and button information will be displayed to the console.

6.196 USB HID Keyboard Host

This example application demonstrates how to support a USB keyboard attached to the evaluation kit board. The display will show if a keyboard is currently connected and the current state of the Caps Lock key on the keyboard that is connected on the bottom status area of the screen. Pressing any keys on the keyboard will cause them to be sent out the SCI at 115200 baud with no parity, 8 bits and 1 stop bit. Any keyboard that supports the USB HID BIOS protocol should work with this demo application.

To run the example you should connect a HID compliant keyboard to the microUSB port on the top of the controlCARD and open up a serial terminal with the above settings to view the characters typed on the keyboard.

6.197 USB Mass Storage Class Host

This example application demonstrates reading a file system from a USB mass storage class device. It makes use of FatFs, a FAT file system driver. It provides a simple command console via the SCI for issuing commands to view and navigate the file system on the mass storage device.

The first SCI, which is connected to the FTDI virtual serial port on the controlCARD board, is configured for 115200 bits per second, and 8-N-1 mode. When the program is started a message will be printed to the terminal. Type “help” for command help.

After loading and running the example, open a serial terminal with the above settings to open the command prompt. Then connect a USB MSC device to the microUSB port on the top of the controlCARD.

For additional details about FatFs, see the following site: [FatFs - Generic FAT Filesystem Module](http://elm-chan.org/fsw/ff/00index_e.html)

6.198 USB Dual Detect

This program uses a GPIO to do ID detection. If a host is connected to the device’s USB port, the stack will switch to device mode and enumerate as mouse. If a mouse device is connected to the device’s USB port, the stack will switch to host mode and display the mouses movement and button press information in a serial terminal.

6.199 USB Throughput Bulk Device Example (usb_ex9_throughput_dev_bulk)

This example provides a throughput numbers of bulk data transfer to and from the host. The device uses a vendor-specific class ID and supports a single bulk IN Endpoint and a single bulk OUT Endpoint.

SCIA, connected to the FTDI virtual COM port and running at 115200, 8-N-1, is used to display messages from this application.

A Windows INF file for the device is provided under the windows drivers directory. This INF contains information required to install the WinUSB subsystem on WindowsXP, Windows 7 and Windows 10. This is present in utilities/windows_drivers.

A sample Windows command-line application, usb_throughput_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided. Project files are included to allow the examples to be built using Microsoft VisualStudio. Source code for this application can be found in directory ~/utilities/tools/usb_throughput_bulk_example/Release.

After running the example in CCS Connect the USB Micro to the PC. Then the example will wait to receive data from the application. Run the usb_throughput_bulk example, the throughput and Data Packets Transferred.

6.200 Watchdog

This example shows how to service the watchdog or generate a wakeup interrupt using the watchdog. By default the example will generate a Wake interrupt. To service the watchdog and not generate the interrupt, uncomment the SysCtl_serviceWatchdog() line in the main for loop.

External Connections

- None.

Watch Variables

- wakeCount - The number of times entered into the watchdog ISR
- loopCount - The number of loops performed while not in ISR

6.201 Flash Programming with AutoECC, DataAndECC, DataOnly and EccOnly

This example demonstrates how to program Flash using API's following options 1. AutoEcc generation 2. DataOnly and EccOnly 3. DataAndECC

Note:
CPU1 example must be loaded and executed before CPU2 example. Only CPU1 is able to set the BankMuxSel and GSxMSel registers for CPU2. For both CPU1 and CPU2 load, make sure to set the correct Erase settings for bank selection.

External Connections

- None.

Watch Variables

- None.

6.202 Flash Programming with AutoECC, DataAndECC, DataOnly and EccOnly

This example demonstrates how to program Flash using API's following options 1. AutoEcc generation 2. DataOnly and EccOnly 3. DataAndECC

Note:
CPU1 example must be loaded and executed before CPU2 example. Only CPU1 is able to set the BankMuxSel and GSxMSel registers for CPU2. For both CPU1 and CPU2 load, make sure to set the correct Erase settings for bank selection.

External Connections

- None.

Watch Variables

- None.

7 Dual Core Driver Library Example Applications

These example applications show how to make use of F28P65x device functions which span both the CPU 1 and CPU 2. All of these examples contain two example projects: one for CPU 1 and one for CPU 2.

Like the CPU1 only projects, these projects also contain different build configurations for RAM and Flash builds. All of the CPU1 projects contain RAM and Flash build configurations with debugger support, as well as a standalone flash build configuration which sends an IPC command to boot the second core and begin executing the application in its flash. The CPU2 projects all only contain a flash and RAM build configuration as there are no dependencies in the code regarding whether the application is running with or without a debugger.

All these examples are setup using the Code Composer Studio (CCS) "projectspec" format. For these dual core example applications, the "projectspec" allows for two projects to be defined in one file. Upon importing the "projectspec", the two example projects will be generated in the CCS workspace with copies of the source and header files included for each project. All these examples contain two build configurations which allow you to build each project to run from either RAM or Flash. To change how the project is built simply right click on the project and select "Build Configurations". Then, move over to set the active build configuration, either RAM or Flash.

The examples provided are built for controlCARD compatibility.

To run one of these examples after compiling it, load the appropriate programs on each of the two cores. Then, for more example specific instructions please refer to the documentation regarding the example you wish to run on the following pages or in the comments of the example sources.

All of these examples can be found in the

`driverlib/f28p65x/examples/c28x_dual` subdirectory of the C2000Ware package.

7.1 DMA Transfer Shared Peripheral

This example shows how to initiate a DMA transfer on CPU1 from a shared peripheral which is owned by CPU2. In this specific example, a timer ISR is used on CPU2 to initiate a SPI transfer which will trigger the CPU1 DMA. CPU1's DMA will then in turn update the ePWM1 CMPA value for the PWM which it owns. The PWM output can be observed on the GPIO pins. It is recommended to run the c28x1 core first, followed by the C28x2 core.

Note:

In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

Watch Pins

- GPIO0 and GPIO1 - ePWM output can be viewed with oscilloscope

7.2 IPC basic message passing example with interrupt

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core without message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- pass

7.3 IPC basic message passing example with interrupt

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core without message queues It is recommended to run the C28x1 core first, followed by the C28x2 core.

External Connections

- None.

Watch Variables

- None.

7.4 IPC message passing example with interrupt and message queue

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core with message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- pass

7.5 IPC message passing example with interrupt and message queue

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core with message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

External Connections

- None.

Watch Variables

- None.

7.6 IPC basic message passing example with interrupt

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core without message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2 by using SysConfig. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration.

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- pass

7.7 IPC basic message passing example with interrupt

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core without message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2 by using SysConfig. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration.

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- None.

7.8 IPC message passing example with interrupt and message queue

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core with message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2 by using SysConfig. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration.

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- pass

7.9 IPC message passing example with interrupt and message queue

This example demonstrates how to configure IPC and pass information from C28x1 to C28x2 core with message queues. It is recommended to run the C28x1 core first, followed by the C28x2 core.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2 by using SysConfig. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration.

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- None.

7.10 LED Blinky Example

This example demonstrates how to blink a LED using CPU1 and blink another LED using CPU2 (led_ex1_blinky_cpu2.c).

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

External Connections

- None.

Watch Variables

- None.

7.11 LED Blinky Example

This example demonstrates how to blink a LED using CPU1 and blink another LED using CPU2 (led_ex1_blinky_cpu2.c).

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2 by using SysConfig. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration.

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

Please switch to LAUNCHPAD build configurations if using launchpad.

External Connections

- None.

Watch Variables

- None.

7.12 Shared RAM Management (CPU1)

This example shows how to assign shared RAM for use by both the CPU2 and CPU1 core. Shared RAM regions are defined in both the CPU2 and CPU1 linker files. In this example GS0 and GS4 are assigned to/owned by CPU2. The remaining shared RAM regions are owned by CPU1.

In this example, a pattern is written to cpu1RWArray and then an IPC flag is sent to notify CPU2 that data is ready to be read. CPU2 then reads the data from cpu2RArray and writes a modified pattern to cpu2RWArray. Once CPU2 acknowledges the IPC flag, CPU1 reads the data from cpu1RArray and compares with expected result.

A timer ISR is also serviced in both CPUs. The ISRs are copied into the shared RAM region owned by the respective CPUs. Each ISR toggles a GPIO. Watch the GPIOs on an oscilloscope, or if using the controlCARD, watch LED1 and LED2 blink at different rates.

Following are the memory allocation details of CPU Timer interrupt ISRs & read(R)/read write(RW) arrays in CPU1 & CPU2 as configured in the example.

- cpu1RWArray[] is mapped to shared RAM GS1
- cpu1RArray[] is mapped to shared RAM GS0
- cpu2RArray[] is mapped to shared RAM GS1
- cpu2RWArray[] is mapped to shared RAM GS0
- cpuTimer0ISR in CPU2 is copied to shared RAM GS4, toggles LED1
- cpuTimer0ISR in CPU1 is copied to shared RAM GS3, toggles LED2

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these memory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

Watch Variables

- *error* Indicates that the data written is not correctly received by the other CPU.

7.13 Shared RAM Management (CPU2)

This example shows how to assign shared RAM for use by both the CPU2 and CPU1 core. Shared RAM regions are defined in both the CPU2 and CPU1 linker files. In this example GS0 and GS4 are assigned to/owned by CPU2. The remaining shared RAM regions are owned by CPU1.

In this example, a pattern is written to cpu1RWArray and then an IPC flag is sent to notify CPU2 that data is ready to be read. CPU2 then reads the data from cpu2RArray and writes a modified pattern to cpu2RWArray. Once CPU2 acknowledges the IPC flag, CPU1 reads the data from cpu1RArray and compares with expected result.

A timer ISR is also serviced in both CPUs. The ISRs are copied into the shared RAM region owned by the respective CPUs. Each ISR toggles a GPIO. Watch the GPIOs on an oscilloscope, or if using the controlCARD, watch LED1 and LED2 blink at different rates.

Following are the memory allocation details of CPU Timer interrupt ISRs & read(R)/read write(RW) arrays in CPU1 & CPU2 as configured in the example.

- cpu1RWArray[] is mapped to shared RAM GS1
- cpu1RArray[] is mapped to shared RAM GS0
- cpu2RArray[] is mapped to shared RAM GS1
- cpu2RWArray[] is mapped to shared RAM GS0
- cpuTimer0ISR in CPU2 is copied to shared RAM GS4, toggles LED1
- cpuTimer0ISR in CPU1 is copied to shared RAM GS3, toggles LED2

7.14 NMI handling

This example demonstrates how to handle an NMI.

The watchdog of CPU2 is configured to reset the core once the watchdog overflows and in the CPU1 the NMI is triggered. The NMI status is read and is verified to be due to CPU2 Watchdog reset. The NMI ISR reboots the CPU2 core and the process is repeated.

Note:
In the default CPU2 linker cmd file, GS4, FLASH_BANK3 and FLASH_BANK4 are used for allocating various CPU2 sections. The CPU1 application assigns the ownership of these mem-

ory regions to CPU2. Please note that CPU2 .out file can be loaded only after CPU1 completes this configuration

The erase setting (CPU1/CPU2 On-Chip Flash -> erase setting) needs to be configured as selected banks only (Choose the corresponding BANKS allocated for CPUs) or necessary sectors only before loading CPU1/CPU2.out file (This is applicable only for FLASH configuration)

Watch Variables

- *nmi_isr_count* Indicates the number of times the NMI ISR was hit because of CPU2 watchdog reset.

7.15 Watchdog Reset

This example shows how to configure the watchdog to reset CPU2 which will trigger an NMI in CPU1. LED1 is toggled at the start of main indicating CPU reset.

External Connections

- None.

Watch Variables

- loopCount - The number of loops performed while not in ISR

8 Device APIs for examples

8.1 Introduction

This chapter provides information on the APIs included in device.c file

8.2 API Functions

Functions

- void __error__ (const char *filename, uint32_t line)
- void Device_bootCPU2 (uint32_t bootmode)
- void Device_enableAllPeripherals (void)
- void Device_init (void)
- void Device_initGPIO (void)
- bool Device_verifyXTAL (float freq)

8.2.1 Function Documentation

8.2.1.1 __error__

Error handling function to be called when an ASSERT is violated.

Prototype:

```
void
__error__(const char *filename,
          uint32_t line)
```

Parameters:

filename File name in which the error has occurred

line Line number within the file

Returns:

None

8.2.1.2 void Device_bootCPU2 (uint32_t *bootmode*)

Function to boot CPU2.

Parameters:

bootmode is the mode in which CPU2 should boot.

Description:

Available bootmodes :

- BOOTMODE_BOOT_TO_FLASH_BANK0_SECTOR0
- BOOTMODE_BOOT_TO_FLASH_BANK0_SECTOR127_END
- BOOTMODE_BOOT_TO_FLASH_BANK1_SECTOR0
- BOOTMODE_BOOT_TO_FLASH_BANK2_SECTOR0
- BOOTMODE_BOOT_TO_FLASH_BANK3_SECTOR0
- BOOTMODE_BOOT_TO_FLASH_BANK4_SECTOR0
- BOOTMODE_BOOT_TO_FLASH_BANK4_SECTOR127_END
- BOOTMODE_BOOT_TO_SECURE_FLASH_BANK0_SECTOR0
- BOOTMODE_BOOT_TO_SECURE_FLASH_BANK1_SECTOR0
- BOOTMODE_BOOT_TO_SECURE_FLASH_BANK2_SECTOR0
- BOOTMODE_BOOT_TO_SECURE_FLASH_BANK3_SECTOR0
- BOOTMODE_BOOT_TO_SECURE_FLASH_BANK4_SECTOR0
- BOOTMODE_IPC_MSGRAM_COPY_BOOT_TO_M1RAM
- BOOTMODE_BOOT_TO_M0RAM
- BOOTMODE_BOOT_TO_FWU_FLASH
- BOOTMODE_BOOT_TO_FWU_FLASH_ALT1
- BOOTMODE_BOOT_TO_FWU_FLASH_ALT2
- BOOTMODE_BOOT_TO_FWU_FLASH_ALT3

Note that while using BOOTMODE_IPC_MSGRAM_COPY_BOOT_TO_M1RAM, BOOTMODE_IPC_MSGRAM_COPY_LENGTH_xxxW must be ORed with the bootmode parameter

This function must be called after Device_init function

Returns:
None.

8.2.1.3 void Device_enableAllPeripherals (void)

Function to turn on all peripherals, enabling reads and writes to the peripherals' registers.

Note that to reduce power, unused peripherals should be disabled.

Parameters:
None

Returns:
None

8.2.1.4 Device_init

Function to initialize the device. Primarily initializes system control to aknown state by disabling the watchdog, setting up the SYSCLKOUT frequency, and enabling the clocks to the peripherals.

Prototype:

```
void
Device_init(void)
```

Parameters:
None.

Returns:
None.

8.2.1.5 void Device_initGPIO (void)

Function to disable pin locks on GPIOs.

Parameters:
None

Returns:
None

8.2.1.6 bool Device_verifyXTAL (float *freq*)

Function to verify the XTAL frequency.

Parameters:
freq is the XTAL frequency in MHz

Returns:
The function return true if the the actual XTAL frequency matches with the input value

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI 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 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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265