F28004x 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 Fri Nov 17 18:42:36 IST 2023.

Table of Contents

1 Introduction

The Texas Instruments® F28004x Firmware development package includes a device-specific driver library, a group of example applications that demonstrate key device functionality, and other development files such as linker command files that assist in getting started with a F28004x device.

The following chapter provides a step by step guide for creating a new project from scratch as well as debugging. It is highly recommended that users new to the F28004x family of devices start by reading this section first.

The F28004x devices have a set of example applications that users can load and run on their device.

- The driver library example applications can be found in the `~/driverlib/f28004x/examples` directory.
- The bit-field example applications can be found in the `~/device_support/f28004x/examples` directory.

F28004x Example Projects

- Driverlib Example projects tested with: C2000 Compiler v20.2.1.LTS

The examples provided are built for controlCARD compatibility. For LaunchPad use, some minor modifications may be required.

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 hierarchy as follows:

- Main project directory
  - Project folder
    - Project sources (∗.c, ∗.h)
    - CCS folder (ccs)
      - CCS projectspec file

1.1 Detailed Revision History

V1.12.00.00

- Updated Driver Library to v3.04.00.00
- Added ERAD examples.
- Added ROM-Flash API usage example
- Added Flash linker command file with CRC
- Added new I2C, ADC examples
- Added CLB Type 3 Examples

V1.11.00.00

- Updated Driver Library to v3.03.00.00

  - Added projectspec for driverlib.
  - Updated compiler version to 20.2.1.LTS for all driverlib examples
  - Updated the examples to use the new pinmap macro names.
  - Added Sysconfig pinmux support to all examples
  - Added CLB Type2 examples- clb_ex18 to clb_ex23
  - Added eCAP example to demonstrate generation of phase-shifted APWM outputs: ecap$_{e}x3_{a}pwm_{p}hase_{s}hiftAddedEPWMExampletodemonstrateconfiguresignal$: $epwm_{e}x4_{c}onfigure_{s}ignal$
- Added Example for Baud Tune via SCI- baud_tune_via_uart
- Added LIN Loopback example- lin_ex4_loopback_polling
- Added CAN Example- can_ex5_error$_{g}enerationAddedDCSMToolexample - dcsm\_security\_tool$
- Updated examples to disable watchdog by default

<u>V1.10.00.00</u>

  - Updated Driver Library to v3.02.00.00
  - Updated driverlib examples: Examples for GPIO, SPI, I2C, SCI to use sysconfig
  - Updated driverlib examples : All CLB examples updated for EABI
  - Updated DCSM asm files for EABI

<u>V1.09.00.00</u>

  - Updated Driver Library to v3.01.00.00
  - New driverlib examples: Added a new example to demonstrate Live Firmware Update without device reset. Detailed documentation is provided along with the example in the file : LFU_LED_NO_RESET.pdf
  - New driverlib examples: Added pinumx examples with sysconfig support

<u>V1.08.00.00</u>

  - Updated Driver Library to v2.01.00.00
  - Several bug fixes in driverlib examples - details in release notes
  - Added F280049C Launchpad support for examples and removed support for F280049M
  - Updated driverlib examples: CLB, EQEP, ERAD , Flash API, GPIO , SCI
  - New driverlib examples: C28x - Interrupt , CLB , EPWM , GPIO, SDFM
  - Several linker command files updated as part of bug fixes - details in release notes
  - Several bug fixes/ enhancement in bitfield commons - details in release notes

<u>V1.07.00.00</u>

  - Updated Driver Library to v2.00.00.03
  - Several bug fixes in driverlib examples - details in release notes
  - New driverlib examples: CLB, ERAD, LIN and FSI examples
  - Updating default option of driverlib examples to EABI

<u>v1.06.00.00</u>

- Updated Driver Library to v2.00.00.02
- New boostxl_afe031 examples: dacmode, pwmmode, rx
- Several bug fixes in driverlib and bitfield examples - details in release notes
- Updating ERAD examples and device files for Erad Structure name changes
- New CAN flash kernel driverlib example
- New FSI examples: Skew Compensation, Daisy chain examples: master and slave
- Updating projectspec of examples to use indexing of libs
- Several bug fixes in bitfield commons - details in release notes

v1.05.00.00

- Updated Driver Library to v1.04.00.00
- Release-build configuration of driverlib now built and included within /driverlib
- New CLA Background Nesting Task driverlib example (cla_ex3_background_nesting_task)
- Corrected LDFU Linker command files to reserve required RAM for Flash API usage from ROM
- Updated various driverlib examples (where applicable) to support build configurations for F280049C LaunchPad
- Updated FSI driverlib examples to include a flash build configuration
- Added target configuration file for F280049C

v1.04.00.00

- New F280049C Launchpad demo bit-field example (launchxl_ex1_f280049c_demo)
- New bit-field examples: gpio_ex1_setup, ecap_ex1_apwm, spi_ex2_dma_loopback, dac_ex1_enable, cla_ex2_adc_fir32
- New f28004x_erad.h header and updated BIOS/nonBIOS linker command files
- New target configuration file for F280049C LaunchPad
- Driverlib led_ex1_blinky example updated with build configurations for LaunchPad
- Updated all RAM linker command files to define all Flash memories
- adc_ex1_soc_epwm.c - Updated comments
- gpio_ex1_setup.c - Changed XBAR input pin to INPUT8 for ECAP2 on GPIO24
- Corrected comments in f28004x_adc.h, f28004x_cmpss.h, f28004x_sci.h

v1.03.00.00

- IMPORTANT: Removed Low power mode Standby functions, examples, and headers (f28004x_sysctrl.c, f28004x_sysctrl.h, lpm_ex3_standbywake.c)
- Updated Driver Library to v1.03.00.00
- New driverlib examples - Using ERAD (erad_ex1_profileinterrupts, erad_ex2_profilefunction, erad_ex3_stackoverflow, erad_ex4_profileinterrupts_cla)
- f28004x_epwm.h - Marked self clear translator as reserved, added structs for valley and edge modes
- f28004x_flash.h - Marked illegal address detected as reserved in struct
- f28004x_output_xbar.h - Corrected comments

- Updated various files to support code section pragmas for C++
- Empty Driverlib Example - Fixed issue when importing project to CCS
- f28004x_sysctrl.c, device.c - Added memcpy namespace for when building for C++
- CAN External Transmit Example - Fixed comment
- CAN Loopback Interrupts, External Transmit, Loopback DMA Examples - Updated for interrupt numbering changes (1 and 2 to 0 and 1)

v1.02.00.00

- Updated Driver Library to v1.02.00.00
- Corrected CLA flash linker file to fix CLAPROG memory and RAMFUNC overlap
- New driverlib example - Customizing Boot Configurations (boot_ex2_customBootConfig)
- New driverlib example - Performing live firmware update using dual flash banks (flashapi_ex3_liveFirmwareUpdate)
- New driverlib examples - FSI SPI mode TX and RX (fsi_ex9_spi_master_tx_drivers, fsi_ex10_spi_slave_rx_drivers)
- New driverlib example - FSI SPI mode Full Duplex Communication (fsi_ex11_spifsi_full_duplex)
- New driverlib example - ADC temperature sensor conversion (adc_ex3_temp_sensor)
- New driverlib example - SCI Flash Kernel Example (flashapi_ex2_sciKernel)
- Added F021 API symbol library for ROM flash API (revB silicon and newer only)
- Link to F28004x product page now included in documentation directory
- f28004x_sysctrl.c and device.c - InitFlash() now run regardless of build configuration
- can_ex3_external_transmit.c - Updated description that example requires custom board with two CAN transceivers
- spi_ex5_external_loopback_fifo_interrupts.c - Corrected GPIO numbers in description
- timer_ex1_cputimers.c - Fixed GPIO configuration for LED
- f28004x_dcc.h - Renamed DccRegs to Dcc0Regs
- f28004x_adc.c - Clarified comments for SetVREF()
- f28004x_gpio.c - Renamed various function parameters
- f28004x_pievect.c - Added FSI interrupts to PIE vector table

v1.01.00.00

- Updated Flash API Library and Documentation
- Updated Driver Library to v1.01.00.00
- f28004x_can.h - Corrected "name" to "INT0_FLG" in global interrupt flag register and corrected various comments
- f28004x_ecap.h - Added ECAPSYNCINSEL register and corrected comments
- f28004x_fsi.h - Removed bit fields related to SPI mode
- f28004x_epwm.h - Updated reserved sections to align with bit fields
- Added driverlib SPI external loopback with FIFO interrupts example

v1.00.00.00

- Initial release of the F28004x Device Support and Driver Library Package

2 Getting Started and Troubleshooting

2.1 Introduction

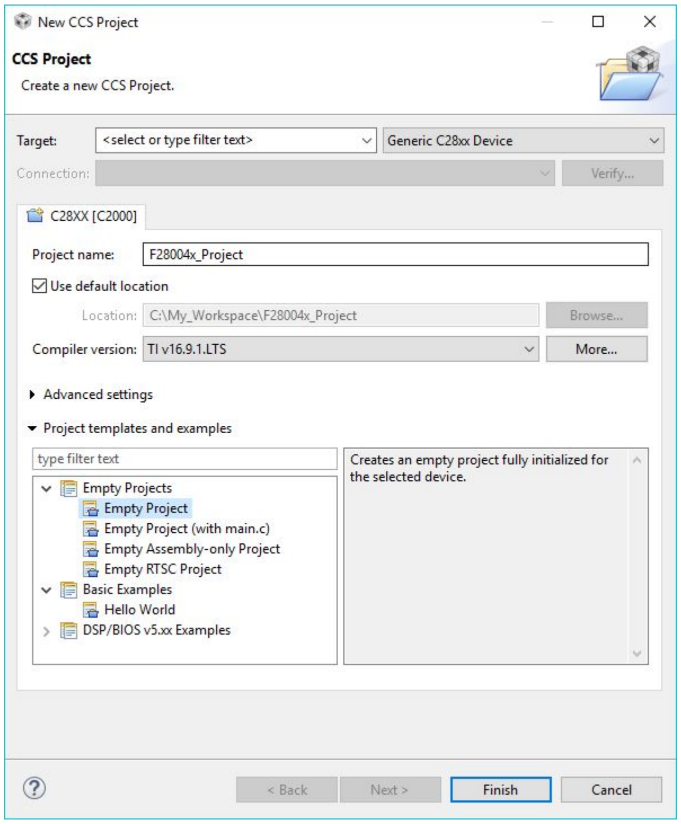
Because of the sheer complexity of the F28004x devices, it is not uncommon for new users to have trouble bringing up the device their first time. This guide aims to give you, the user, a step by step guide for how to create and debug projects from scratch. This guide will focus on the user of a F28004x controlCARD, but these same ideas should apply to other boards with minimal translation.

2.2 Project Creation

A typical F28004x application consists of setting up a CCS project, which involves configuring the build settings, file linking, and adding in any source code.

CCS Project Creation

1. From the main CCS window select File -> New -> CCS Project. Select your Target as "Generic C28xx Device". Name your project and choose a location for it to reside. Click Finish and your project will be created.


New CCS Project

CCS Project

Create a new CCS Project.

Target: <select or type filter text> Generic C28xx Device

Connection:

C28XX [C2000]

Project name: F28004x_Project

☑ Use default location

Location: C:\My_Workspace\F28004x_Project

Compiler version: TI v16.9.1.LTS

Advanced settings

Project templates and examples

Empty Projects
- Empty Project
- Empty Project (with main.c)
- Empty Assembly-only Project
- Empty RTSC Project

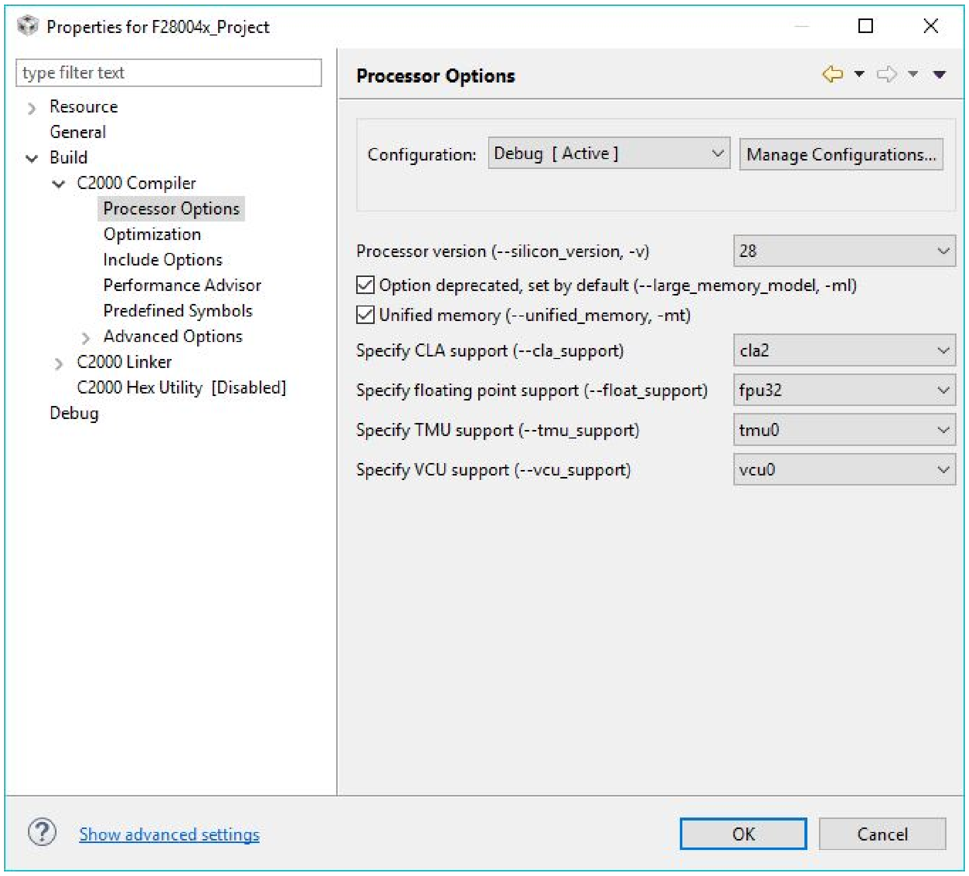
Basic Examples
- Hello World

DSP/BIOS v5.xx Examples

Creates an empty project fully initialized for the selected device.

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:



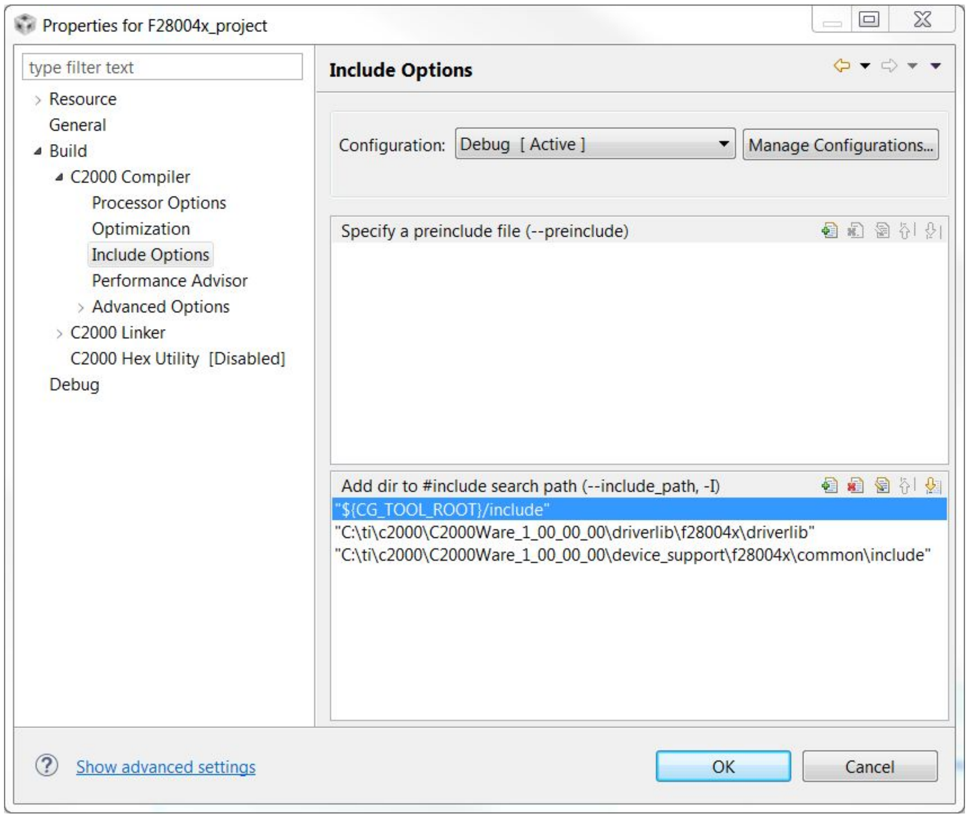
Processor Options

Configuration: Debug [Active]

Processor version (--silicon_version, -v): 28
☑ Option deprecated, set by default (--large_memory_model, -ml)
☑ Unified memory (--unified_memory, -mt)
Specify CLA support (--cla_support): cla2
Specify floating point support (--float_support): fpu32
Specify TMU support (--tmu_support): tmu0
Specify VCU support (--vcu_support): vcu0

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 `driverlib\f28004x\driverlib` folder of your C2000Ware installation (typically `C:\ti\c2000\C2000Ware_<version>\driverlib\f28004x\driverlib`). Click ok to add this path, and repeat this same process to add the `device_support\f28004x\common\include` directory.



Configuration: Debug [Active]

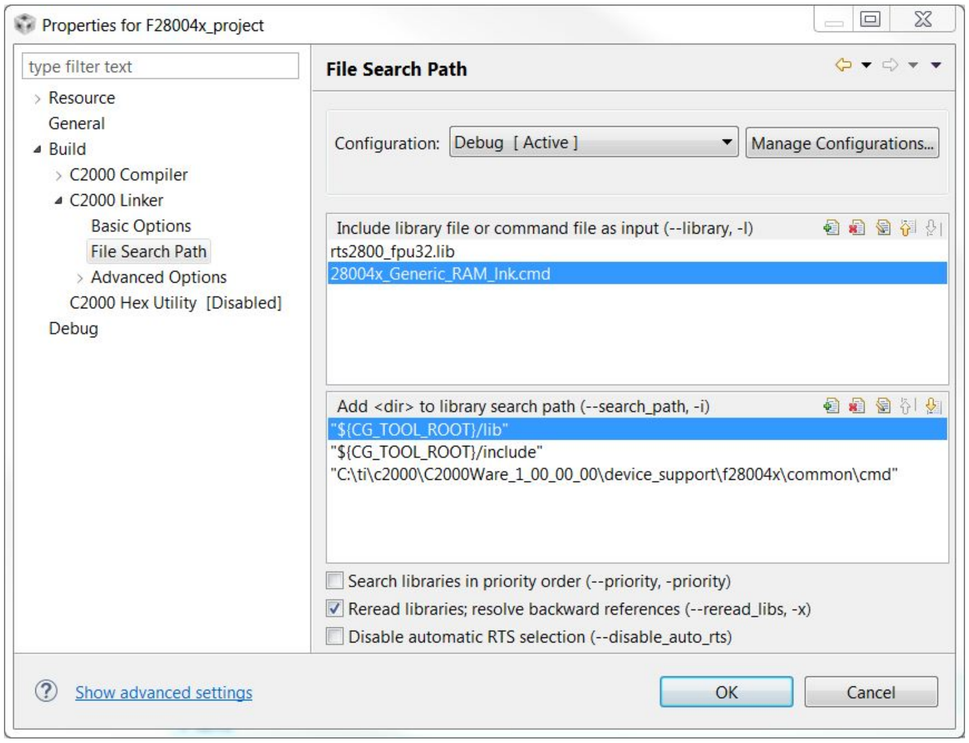
Specify a preinclude file (--preinclude)

Add dir to #include search path (--include_path, -I)

```
"${CG_TOOL_ROOT}/include"
"C:\ti\c2000\C2000Ware_1_00_00_00\driverlib\f28004x\driverlib"
"C:\ti\c2000\C2000Ware_1_00_00_00\device_support\f28004x\common\include"
```

Figure 2.3: Project configuration dialog box

4. Click on the Linker File Search Path. Add the following directory to the search path: `device_support\f28004x\common\cmd`. Then you'll also want to add the following files: `rts2800_fpu32.lib` and `28004x_Generic_RAM_lnk.cmd`. Finally, delete `libc.a`, we will use `rts2800_fpu32.lib` as our run time support library instead. Select ok to close out of the Build Properties.



Properties for F28004x_project

File Search Path

Configuration: Debug [Active] Manage Configurations...

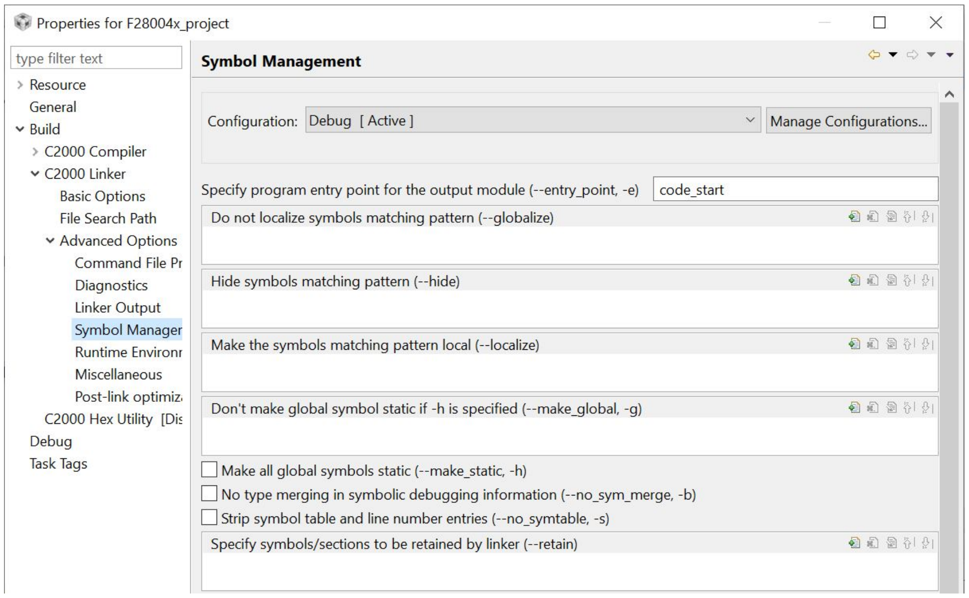
Include library file or command file as input (--library, -l)
rts2800_fpu32.lib
28004x_Generic_RAM_lnk.cmd

Add <dir> to library search path (--search_path, -i)
"${CG_TOOL_ROOT}/lib"
"${CG_TOOL_ROOT}/include"
"C:\ti\c2000\C2000Ware_1_00_00_00\device_support\f28004x\common\cmd"

☐ Search libraries in priority order (--priority, -priority)
☑ Reread libraries; resolve backward references (--reread_libs, -x)
☐ Disable automatic RTS selection (--disable_auto_rts)

Figure 2.4: Project configuration dialog box

5. 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. For the examples, code_start is the first code that is executed after exiting the boot ROM code.



Symbol Management

Configuration: Debug [Active]

Specify program entry point for the output module (--entry_point, -e) code_start

Do not localize symbols matching pattern (--globalize)

Hide symbols matching pattern (--hide)

Make the symbols matching pattern local (--localize)

Don't make global symbol static if -h is specified (--make_global, -g)

☐ Make all global symbols static (--make_static, -h)

☐ No type merging in symbolic debugging information (--no_sym_merge, -b)

☐ Strip symbol table and line number entries (--no_symtable, -s)

Specify symbols/sections to be retained by linker (--retain)

Figure 2.5: Entry point setup

6. If LaunchPad is being used, then make sure to add the pre-define NAME "_LAUNCHXL_F280049C" within the project's properties->Advanced Options->Predefined Symbols.
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. Navigate to the `device_support\f28004x\common\source` directory, and select `device.c`. After you select the file, you'll have the option to copy the file into the project or link it. We recommend you link files like this to the project as you will probably not modify these files. Link in the following files as well:
   - `driverlib\f28004x\driverlib\ccs\Debug\driverlib.lib`
   - `device_support\f28004x\common\source\f28004x_codestartbranch.asm` (code_start is the first code that is executed after exiting the boot ROM code for the example f28004x projects. The projects are setup such that the codegen entry point is also set to the codestart label using linker options. The codestart code will automatically re-direct the execution to _c_init00.)

   The codestart section in the linker cmd file is used to physically place this code at the correct memory location. This section should be placed at the location the BOOT ROM will re-direct the code to. For example,for boot to FLASH this code will be located at 0x80000.

   In addition, the example projects are setup such that the codegen entry point is also set to the code_start label. This is done by linker option -e in the project build options. When the debugger loads the code, it will automatically set the PC to the "entry point" address indicated by the -e linker option. In this case the debugger is simply assigning the PC, it is not the same as a full reset of the device.

   The compiler may warn that the entry point for the project is other than _c_init00. _c_init00 is the C environment setup and is run before main() is entered. The code_start code will re-direct the execution to _c_init00 and thus there is no worry and this warning can be ignored. Code for redirecting code execution after boot is added to f28004x_codestartbranch.asm file and hence this file needs to be linked to the CCS project.

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

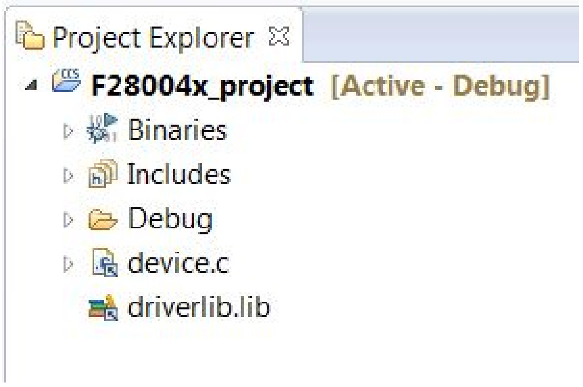


Figure 2.6: Linking files to project

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:

```
#include "driverlib.h"
#include "device.h"

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);
    }
}
```

10. Save main.c and then attempt to build the project by right clicking on it and selecting Build Project. Assuming the project builds, setup a target configuration file for your device (View -> Target Configurations), and try debugging this project on a F28004x device. When the code runs, you should see the LED blink.

2.3 Project: Adding Bitfield or Driverlib Support

F28004x devices support two types of development software, driver library APIs and bitfield 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 bitfield project as well as how to add bitfield 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\f28004x\driverlib`
2. Include the following header file in the project main source file: `device_support\f28004x\common\include\driverlib.h`
3. Add or link the `driverlib.lib` library to the project. Location of file: `driverlib\f28004x\driverlib\ccs\Debug`

Adding Bitfield Support

1. Add the following include directory path to the project: `device_support\f28004x\headers\include`
2. Include the following header file in the project main source file: `device_support\f28004x\headers\include\f28004x_device.h`
3. Add or link the `f28004x_globalvariabledefs.c` file to the project. Location of file: `device_support\f28004x\headers\source`
4. Add or link the `f28004x_headers_nonbios.cmd` file to the project. Location of file: `device_support\f28004x\headers\cmd`

2.4 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 Piccolo devices.

"I get an error when I try to import the example projects"

This occurs when one imports a project for which he or she doesn't have the code generation tools for or the latest CCS device support update supporting your device. Please ensure that you have at least version 16.9.1.LTS of the C2000 Code Generation Tools and have updated your CCS device support through the CCS "Install New Software" menu under "Help".

"My F28004x 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. Advanced users may potentially alter GEL files depending on their overall system configuration. 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 the core, 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 CPU comes up, it is not fresh out of reset"

F28004x 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 accidently 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.

"I'm using a Launchpad and my device clocking is incorrect"

Few of the examples have a "CPU1_LAUNCHXL_RAM" and "CPU1_LAUNCHXL_FLASH" build configuration. Make sure to set the appropriate build configuration as Active within CCS. In your project, add the pre-define NAME "_LAUNCHXL_F280049C" within the project's properties->Advanced Options->Predefined Symbols.

"\"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

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.

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 F28004x 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 F28004x 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 F28004x 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 F28004x 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 is 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 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 example of an Interrupt service routine for a SPI transmit FIFO. This interrupt is connected to PIE group 6.

```c
//
// SPI A Transmit FIFO ISR
//
__interrupt void spiTxFIFOISR(void)
{
    uint16_t i;

    //
    // Send data
    //
    for(i = 0; i < 2; i++)
    {
       SPI_writeDataNonBlocking(SPIA_BASE, sData[i]);
    }

    //
    // Increment data for next cycle
    //
    for(i = 0; i < 2; i++)
    {
       sData[i] = sData[i] + 1;
    }

    //
    // Clear interrupt flag and issue ACK
    //
    SPI_clearInterruptStatus(SPIA_BASE, SPI_INT_TXFF);
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP6);
}
```

/*!

4 Driver Library Example Applications

These example applications show how to make use of various peripherals of a F28004x 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 of these examples reside in the `driverlib/f28004x/examples` subdirectory of the C2000Ware package.

The examples provided are built for controlCARD compatibility. For LaunchPad use, some minor modifications may be required.

If using a Launchpad, add a pre-defined symbol within the project properties called "_LAUNCHXL_F280049C".

Example Projects require CCS v6.2.0.00050 or newer

4.1 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

4.2 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

4.3 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)

4.4 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

4.5 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

4.6 ADC Open Shorts Detection (adc_open_shorts_detection)

This example demonstrates the ADC open/shorts detection(ADCOSDETECT) circuit configuration for detecting pin faults in the system. The example enables the open/shorts detection circuit along with mandatory ADC configurations and diagnoses ADCA A0 input pin state before starting normal ADC conversions.

To enable the ADC OSDetect circuit: 1. Configure the ADC for conversion (E.g. channel, SOC, ACQPS, prescalar, trigger etc). The OSDetect functionality is available in 12-bit only. 2. Set up the ADCOSDETECT register for the desired voltage divider connection. Refer device TRM for details on available OSDetect configurations. 3. Initiate a conversion and inspect the conversion result.

Note:

Note: The results must be interpreted based on what is driving on the input side and what are the values of Rs and Cp. If the Vs signal can be disconnected from the input pin, the circuit can be used to detect open and shorted input pins.

In the example, ADCA A0 channel is configured and following algorithm is used to check the A0 pin status: Step 1: Configure full scale OSDETECT mode & capture ADC results(resultHi) Step 2: Configure zero scale OSDETECT mode & capture ADC results(resultLo) Step 3: Disable OS-DETECT mode and capture ADC results(resultNormal) Step 4: Determine the state of the ADC

pin a. If the pin is open, resultLo would be equal to Vreflo and resultHi would be equal to Vrefhi b. If the pin is shorted to Vrefhi, resultLo should be approximately equal to Vrefhi and resultHi should be equal to Vrefhi c. If the pin is shorted to Vreflo, resultLo should be equal to Vreflo and resultHi should be approximately equal to Vreflo d. If the pin is connected to a valid signal, resultLo should be greater than osdLoLimit but less than resultNormal while resultHi should be less than osdHiLimit but greater than resultNormal ——————————————
Input | Full-Scale output | Zero-scale Output | Pin Status ——————————————
—————————— Unknown| VREFHI | VREFLO | Open VREFHI | VREFHI | approx. VREFHI | Shorted to VREFHI VREFLO | approx. VREFLO | VREFLO | Shorted to VREFLO Vn | Vn < resultHi < VREFHI | VREFLO < resultLo < Vn | Good ——————————————
—————————— Step 5: osDetectStatusVal of value greater than 4 would mean that there is no pin fault. a. If osDetectStatusVal == 1, means pin A0 is OPEN b. If osDetectStatusVal == 2, means pin A0 is shorted to VREFLO c. If osDetectStatusVal == 4, means pin A0 is shorted to VREFHI d. If osDetectStatusVal == 8, means pin A0 is in GOOD/VALID state e. Any value of osDetectStatusVal > 4, means pin A0 is in VALID state

Following points should be noted while configuring the ADC in OSDETECT mode. 1. The divider resistance tolerances can vary widely, hence this feature should not be used to check for conversion accuracy. 2. Consult the device data manual for implementation and availability of analog input channels. 3. Due to high drive impedance, a S+H duration much longer than the ADC minimum will be needed.

External Connections

- A0 pin should be connected to signals to convert

Watch Variables

- **osDetectStatusVal** : OS detection status of voltage on pin A0
- **adcAResult0** : a digital representation of the voltage on pin A0

4.7 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

4.8 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.

4.9 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.

4.10 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

4.11 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.

4.12 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

4.13 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

4.14 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

4.15 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

4.16 Boot Error Status Pin Example with DCSM OTP

This example demonstrates how to configure the boot modes, boot mode select pins, and error status pin.

NOTE: DCSM OTP (one time programmable) memory is used to configure the boot modes, boot mode select pins, and error status pin. Once the DCSM OTP sections are programmed, they cannot be erased and programmed again.

This example is designed to show how to configure boot control as well as the function of the error status pin. Once the error status pin is enabled, an NMI will be triggered by software and the error status pin will go high. Then the NMI ISR will clear error status and drive the error status pin low and return to the main loop where the NMI will be triggered again. More details are available in the "ROM Code and Peripheral Booting" chapter of the Technical Reference Manual.

External Connections

- Scope the error status pin used according to what is programmed into GPREG2. The error status pin may be either GPIO 24, GPIO 28, or GPIO 29.

Watch Variables

- None.

4.17 Customized Boot Configuration Example

The example implements a custom boot configuration according to the CCS build configuration selected: 0 boot mode select pins (ZERO_BMSPS), 1 boot mode select pin (ONE_BMSP), or 3 boot mode select pins (THREE_BMSPS).

Select the build configuration and run the program. To test a specific boot mode: pause the program, apply appropriate voltages to BMSPs as necessary, perform a reset through Scripts -> Realtime Emulation Control -> Run_Realtime_with_Reset. If more than one specific boot mode is to be tested, reset the debugger and run the program before following the steps above; this ensures that the device is not executing 'non-debuggable' code from the previous boot mode and the emulation-equivalent Boot ROM registers contain the appropriate values. Do not power-cycle the device in order to perform a reset; values in the emulation-equivalent Boot ROM registers may change if the device is power-cycled.

By default, the example will configure the boot mode select pins and boot mode options for the emulation boot process to be executed.

The following constants can be changed in the header:

- STANDALONE_BOOT - Determines if the standalone boot or emulation boot process is to be emulated when the device is reset (while the emulator is connected). WARNING: the standalone boot process requires the user-configurable DCSM OTP (one-time programmable) registers to be programmed. Please ensure the DCSM OTP registers are programmed before choosing to emulate the standalone boot process. OTP registers can only be programmed once and cannot be erased.

- BOOTPIN_CONFIG_BMSP2 - Boot mode select pin 2, default is GPIO2
- BOOTPIN_CONFIG_BMSP1 - Boot mode select pin 1, default is GPIO1
- BOOTPIN_CONFIG_BMSP0 - Boot mode select pin 0, default is GPIO0

The BOOTDEF options can also be changed as necessary. Default options for the example are in accordance with the build configurations described below.

ZERO_BMSPS: Boot Mode Number BMSP2 BMSP1 BMSP0 Boot Mode 0 N/A N/A N/A Flash Boot

ONE_BMSP: 0 N/A N/A 0 Flash Boot 1 N/A N/A 1 SCI Boot

THREE_BMSPS 0 0 0 0 Flash Boot 1 0 0 1 SCI Boot 2 0 1 0 Flash Boot Alt.1 3 0 1 1 SCI Boot Alt.1 4 1 0 0 CAN Boot 5 1 0 1 SPI Boot 6 1 1 0 RAM Boot 7 1 1 1 I2C Boot

Watch the addresses below in the memory browser when running the emulation boot configuration:

- 0xD00 and 0xD01 for EMU BOOTPIN CONFIG
- 0xD04 and 0xD05 for EMU BOOTDEF LOW
- 0xD06 and 0xD07 for EMU BOOTDEF HIGH

Watch the address below in the memory browser when STANDALONE_BOOT constant is set to a non-zero value:

- 0xD00 for EMU BOOTPIN CONFIG

External Connections

- Connect 3V or GND to BMSP2 as appropriate
- Connect 3V or GND to BMSP1 as appropriate
- Connect 3V or GND to BMSP0 as appropriate

Watch Variables

- None

4.18 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

4.19 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

4.20 CAN-A to CAN-B External Transmit

This example initializes CAN module A and CAN module B for external communication. CAN-A module is setup to transmit incrementing data for "n" number of times to the CAN-B module, where "n" is the value of TXCOUNT. CAN-B module is setup to trigger an interrupt service routine (ISR) when data is received. An error flag will be set if the transmitted data doesn't match the received data.

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

Hardware Required

- A C2000 board with two CAN transceivers

External Connections

- ControlCARD CANA is on DEVICE_GPIO_PIN_CANTXA (CANTXA)
- and DEVICE_GPIO_PIN_CANRXA (CANRXA)
- ControlCARD CANB is on DEVICE_GPIO_PIN_CANTXB (CANTXB)
- and DEVICE_GPIO_PIN_CANRXB (CANRXB)

Watch Variables

- TXCOUNT - Adjust to set the number of messages to be transmitted
- 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

4.21 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

4.22 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.

4.23 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

4.24 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

4.25 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

4.26 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 $arcsin(fVal)$

4.27 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 $arctan(fVal)$

4.28 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

##

4.29 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

4.30 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 (440 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

4.31 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.

4.32 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 ∼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.

4.33 CLAPROMCRC CPU Interrupt Example

This example demonstrates how to configure and run the CLAPROMCRC from the CPU. This uses the golden CRC values in 'clapromcrc_ex1_crctable.h' The CRC calculation uses the 32-bit polynomial 0x04C11DB7.

External Connections

- None.

Watch Variables

- None.

4.34 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.

4.35 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.

4.36 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.

4.37 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.

4.38 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.

4.39 CLB Glue Logic

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

In this example the user is walked through how to migrate custom logic from an FPGA/CPLD to C2000ÃćÂĎÂć microcontrollers.

4.40 CLB based One-shot PWM

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

4.41 CLB AOC Control

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

In this example the Asynchronous Output Conditioning block is used to asynchronously AND gate the input signals to the CLB. This module is only available for CLB types 2 and up.

4.42 CLB AOC Release Control

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

In this example the Asynchronous Output Conditioning block is used to asynchronously set/release the input signals to the CLB. This module is only available for CLB types 2 and up.

4.43 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.

4.44 CLB AOC Control

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

In this example the clock prescalar of the CLB module is used to divide down the CLB clock and use it as an input to the TILE logic. Also the HLC module is used to generate NMI interrupts. This module is only available for CLB types 2 and up.

4.45 CLB Serializer

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

In this example the CLB COUNTER is used in serializer mode to act as a shift register. This module is only available for CLB types 2 and up.

4.46 CLB LFSR

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

In this example the CLB COUNTER module is used in Linear Feedback Shift Register (LFSR) mode. This module is only available for CLB types 2 and up.

4.47 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.

4.48 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.

4.49 CLB CRC

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

In this example, the CLB module is used to perform the cyclic redundancy check (C.R.C.) with twelve messages in bits checked with ten different CRC polynomials.

First element passed in is message length, second is the message stored in input_data

This example is only available for CLB types 2 and up.

The known values in the output_data are compared with expected values from the CLB-based CRC calculation. A total of 120 messages are verified, and the number of matching messages are displayed in passCount

Variables to add to Watch Expressions in debug view: passCount - number of messages that match between generated and known CRC values failCount - number of messages that fail the CRC value verification

###

4.50 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.

4.51 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.

4.52 CLB Event Window

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

This example uses the counter, FSM, and HLC sub-modules of the CLB to implement an event timing feature which detects whether an interrupt service routine takes too long to respond to an interrupt. The example configures four PWM modules to operate in up-count mode and generate a low-to-high edge on a timer zero match event. The zero match event also triggers a PWM ISR which, for the purposes of this example, contains a dummy payload of variable length. At the end of the ISR, a write operation takes place to a CLB GP register to indicate the ISR has ended.

4.53 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.

4.54 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.

4.55 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.

4.56 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.

4.57 CLB Empty Project

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

4.58 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 (The pin is shared with ADCINB6)
- Outputs can be observed on GPIO14 and GPIO15 using an oscilloscope

Watch Variables

- None

4.59 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

4.60 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 ADCINB3.

Watch Variables

- None.

4.61 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 ADCINB3.

Watch Variables

- None.

4.62 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 ADCINB3.

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.

4.63 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

4.64 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.

4.65 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 300us. 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

4.66 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 = 100Mhz). 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

4.67 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

4.68 DCDC Enable

This example enables the DC-DC regulator. The procedure to enable the internal DC-DC regulator must be completed prior to all other initialization functions and application code.

This example later disables the DC-DC and asks the user to measure the I/O current before and after. Breakpoints are provided for this purpose to test the functionality, this is an optional feature and can be removed.

External Connections

- Refer to the device Datasheet for the External Connections for the DC-DC module.

4.69 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)

4.70 DMA GSRAM Transfer (dma_ex1_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

4.71 DMA GSRAM Transfer (dma_ex2_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

4.72 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 5Hz and 10Hz using the shadow registers to load the next period/compare values.

4.73 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.

4.74 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.

4.75 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.

4.76 Empty Project Example

This example is an empty project setup for Driverlib development.

4.77 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.

4.78 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

4.79 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

4.80 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.

4.81 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.

4.82 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.

4.83 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.

4.84 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.

4.85 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.

4.86 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.

4.87 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.

4.88 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.

4.89 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.

4.90 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.

For External connections, Control Card settings are used by default. To use launchpad pins for eQEP1A select them in SysConfig.

External Connections for Control Card

- Connect GPIO6/eQEP1A to GPIO0/ePWM1A

4.91 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

For External connections, Control Card settings are used by default. To use launchpad pins for output select them in SysConfig.

- On controlCARD, Connect GPIO6/eQEP1A to GPIO0/ePWM1A (simulates eQEP Phase A signal)
- On controlCARD, Connect GPIO7/eQEP1B to GPIO1/ePWM1B (simulates eQEP Phase B signal)
- On controlCARD, Connect GPIO9/eQEP1I to GPIO4 (simulates eQEP Index Signal)
- On LaunchPad, Connect GPIO35/eQEP1A to GPIO10/ePWM6A (simulates eQEP Phase A signal)
- On LaunchPad, Connect GPIO37/eQEP1B to GPIO11/ePWM6B (simulates eQEP Phase B signal)
- On LaunchPad, Connect GPIO59/eQEP1I to GPIO8 (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)

4.92 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).

For External connections, Control Card settings are used by default. To use launchpad pins for eQEP1A select them in SysConfig.

External Connections for Control Card

- Connect GPIO6/eQEP1A to GPIO0/ePWM1A

External Connections for LaunchPad

- Connect GPIO35/eQEP1A to GPIO10/ePWM6A

Watch Variables

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

4.93 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)

With motor

- Comment in "MOTOR" in includes For External connections, Control Card settings are used by default. To use launchpad pins for output select them in SysConfig.

- On controlCARD, Connect GPIO6/eQEP1A to GPIO0/ePWM1A (simulates eQEP Phase A signal)
- On controlCARD, Connect GPIO7/eQEP1B to GPIO1/ePWM1B (simulates eQEP Phase B signal)
- On LaunchPad, Connect GPIO35/eQEP1A to GPIO10/ePWM6A (simulates eQEP Phase A signal)

- On LaunchPad, Connect GPIO37/eQEP1B to GPIO11/ePWM6B (simulates eQEP Phase B signal)

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)

4.94 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_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

4.95 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

4.96 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 start-Count 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

4.97 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 start-Count 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

4.98 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

4.99 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

4.100 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

4.101 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

4.102 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

4.103 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

4.104 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

4.105 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)

4.106 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.

4.107 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.

4.108 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)

4.109 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)

4.110 Flash Programming for AutoECC

This example demonstrates how to program Flash using API's AutoEcc generation option.

External Connections

- None.

Watch Variables

- None.

4.111 Live Firmware Update (LFU) example with device reset

For more details on LFU with device reset, refer to the Application Note SPRUIU8

This example demonstrates how to perform a live firmware update with use of the Live Device Firmware Update (Live DFU or LDFU) command; it is to be used with the Serial Flash Programmer as well as the SCI Flash Kernel.

In the example, an SCI autobaud lock is performed and the byte used for autobaud lock is echoed back. Two interrupts are initialized and enabled: SCI Rx FIFO interrupt and CPU Timer 0 interrupt. The CPU Timer 0 interrupt occurs every 1 second; the interrupt service routine (ISR) for CPU Timer 0 toggles an LED based on the build configuration that is running. LED1 is toggled for the BANK0_FLASH and BANK0_ROM build configurations and LED2 is toggled for the BANK1_FLASH and BANK1_ROM build configuration. The SCI Rx FIFO interrupt is set for a FIFO interrupt level of 10 bytes. The number of bytes in a packet from the Serial Flash Programmer (when using the LDFU command) is 10. When a command is sent to the device from the Serial Flash Programmer, the SCI Rx FIFO ISR receives a command from the 10 byte packet in the FIFO. If the command matches the Live Device Firmware Update (Live DFU) command, then the code branches to the Live DFU function located inside of the SCI Flash Kernel for the corresponding bank.

The project contains 4 build configurations:

- BANK0_FLASH: Links the program sections to the appropriate locations in Bank 0 of flash and uses the Flash API library that links the Flash API functions to flash. The 'codestart' section is linked to the alternative flash entry point for Bank 0 (0x8EFF0) and the rest of the sections are linked to 0x082008 or above. Bank 0 configurations of the flash kernel reserve sector 0, sector 1, and the first 128 bits of sector 2 of Bank 0; therefore, sections must be linked to 0x082008 or higher. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK0FLASH.txt'.

- BANK0_ROM: Links the program sections to the appropriate locations in Bank 0 of flash. The sections are linked identical to that of the BANK0_FLASH configuration.This build configuration uses the Flash API library that uses the Flash API functions from the ROM of the device. Revision A of F28004x does not have the Flash API functions in ROM. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK0ROM.txt'.

- BANK1_FLASH: Links the program sections to the appropriate locations in Bank 1 of flash and uses the Flash API library that links the Flash API functions to flash. The 'codestart' section is linked to the alternative flash entry point for Bank 1 (0x9EFF0) and the rest of the sections are linked to 0x092008 or above. Bank 1 configurations of the flash kernel reserve sector 0, sector 1, and the first 128 bits of sector 2 of Bank 1; therefore, sections must be linked to 0x092008 or higher. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK1FLASH.txt'.

- BANK1_ROM: Links the program sections to the appropriate locations in Bank 1 of flash. The sections are linked identical to that of the BANK1_FLASH configuration.This build configuration uses the Flash API library that uses the Flash API functions from the ROM of the device. Revision A of F28004x does not have the Flash API functions in ROM. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK1ROM.txt'.

To use the example, make sure line 867 is commented in the Serial Flash Programmer project in order to be able to load the kernel through CCS. Configure the device to boot to flash at 0x80000 and follow the steps below: 1. Place the output files of the C2000 Hex Utility tool for the build configurations to be used (one for each bank) in the 'hex' directory of the Serial Flash Programmer project. 2. Load LDFU build configurations of the SCI Flash Kernel (one for each bank) to flash through CCS. If a debug session is launched in order to load a build configuration of the flash kernel to flash, go to Tools -> On-Chip Flash -> Erase Settings in order to ensure that 'Necessary Sectors Only (for Program Load)' is selected. This allows only the sectors where the flash kernel is linked to be erased when it is loaded. If 'Entire Flash' was selected, ensure that both build configurations are in flash. 3. Run the bank 0 build configuration of the flash kernel in CCS, ensuring that the bank 1 build configuration is not erased. 4. In the Serial Flash Programmer project, make sure the command arguments of the debug properties (Debug -> serial_flash_programmer Properties -> Debugging -> Command Arguments) contains the path to the hex formatted file of the bank 1 configuration of this example. Refer to the 'README.txt' file in the Serial Flash Programmer project to understand the command arguments of the debug properties. Verify that the correct COM port is selected. 5. Run the Serial Flash Programmer project. A menu should appear inside a terminal. 6. In the terminal, enter '8' to send the Live DFU command. The bank 1 build configuration of the example will start loading to the device and each byte that is sent will show in the terminal. Keep the terminal open until the file is done being loaded to the device, indicated by an "Application load successful!" message. Once the program is done loading, the bank 1 build configuration of the example will be running on the device. 7. Exit the terminal and edit the command line arguments of the Serial Flash Programmer's debug properties so that it contains the path to the hex formatted file for the bank 0 build configuration of this example. 8. Run the Serial Flash Programmer again in order to perform an autobaud lock and view the menu in the terminal. LED2 should be blinking to indicate that code is running on bank 1. 8. Enter '8' in the terminal to send the Live DFU command to the device. The bank 0 build configuration of the example will start loading to the device. Again, each byte that is sent will be shown in the terminal; the end of the loading is marked by an "Application load successful!" message. When the program is done loading, the bank 0 build configuration of this example should be running on the device. 9. Exit the terminal and edit the command arguments of Serial Flash Programmer's debug properties so that it includes the path to the hex formatted file of the bank 1 configuration of this example. 10. Run the Serial Flash Programmer to perform an autobaud lock and view the menu. LED1 should be blinking to indicate that code is running on bank 0. 11. Restart from step 6.

If BANK1_FLASH and BANK0_FLASH build configurations are used for the example, then the corresponding build configurations of the flash kernel must be loaded to flash: BANK1_LDFU and BANK0_LDFU respectively. If BANK1_ROM and BANK0_ROM build configurations are used for the example, then the corresponding build configurations of the flash kernel must be loaded to flash: BANK1_LDFU_ROM and BANK0_LDFU_ROM respectively.

External Connections

- Connect GPIO28 (SCI Rx) and GPIO29 (SCI Tx) to a COM port of the computer running the Serial Flash Programmer project

Watch Variables

- None

4.112 Live Firmware Update (LFU) example without device reset

For more details on LFU without device reset, refer to the Application Note SPRUIU9

This example demonstrates how to perform a live firmware update with use of the Live Device Firmware Update (Live DFU or LDFU) command. The LDFU command is supported in Serial Flash Programmer which communicates with the SCI Flash Kernel.

In the example, an SCI autobaud lock is performed and the byte used for autobaud lock is echoed back. Two interrupts are initialized and enabled: SCI Rx FIFO interrupt and CPU Timer 0 interrupt. The CPU Timer 0 interrupt occurs every 1 second; the interrupt service routine (ISR) for CPU Timer 0 toggles an LED based on the build configuration that is running. LED1 is toggled for the BANK0_FLASH and BANK0_ROM build configurations and LED2 is toggled for the BANK1_FLASH and BANK1_ROM build configuration. The SCI Rx FIFO interrupt is set for a FIFO interrupt level of 10 bytes. The number of bytes in a packet from the Serial Flash Programmer (when using the LDFU command) is 10. When a command is sent to the device from the Serial Flash Programmer, the SCI Rx FIFO ISR receives a command from the 10 byte packet in the FIFO. If the command matches the Live Device Firmware Update (Live DFU) command, then the code branches to the Live DFU function located inside of the SCI Flash Kernel for the corresponding bank.

The project contains 2 build configurations:

- BANK0_FLASH: Links the program sections to the appropriate locations in Bank 0 of flash and uses the Flash API library that links the Flash API functions to flash. The 'codestart' section is linked to the alternative flash entry point for Bank 0 (0x8EFF0) and the rest of the sections are linked to 0x082008 or above. Bank 0 configurations of the flash kernel reserve sector 0, sector 1, and the first 128 bits of sector 2 of Bank 0; therefore, sections must be linked to 0x082008 or higher. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK0FLASH.txt'.

- BANK1_FLASH: Links the program sections to the appropriate locations in Bank 1 of flash and uses the Flash API library that links the Flash API functions to flash. The 'codestart' section is linked to the alternative flash entry point for Bank 1 (0x9EFF0) and the rest of the sections are linked to 0x092008 or above. Bank 1 configurations of the flash kernel reserve sector 0, sector 1, and the first 128 bits of sector 2 of Bank 1; therefore, sections must be linked to 0x092008 or higher. After building the configuration, the C2000 Hex Utility will output the program in the appropriate SCI boot hex format for the flash kernel and serial flash programmer in a file named 'flashapi_ex2_liveFirmwareUpdateBANK1FLASH.txt'.

External Connections

- Connect GPIO28 (SCI Rx) and GPIO29 (SCI Tx) to a COM port of the computer running the Serial Flash Programmer project. In control card this is routed via the USB port. So juct connecting the USB cable to PC will suffice.

Watch LED1 or LED2 blinking and the blink rate.

- None

4.113 FSI and SPI communication (fsi_ex10_spi_slave_rx_driver)

Port of fsi_ex7_spi_slave_rx example using spifsi driver. 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 slave Rx and SPI as master 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 on controlCARD, 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_9 :: To connect FSIRX_CLK with SPICLKA (59 -> 71 on docking station)
- GPIO_12 -> GPIO_8 :: To connect FSIRX_RX0 with SPISIMOA (57 -> 87 on docking station)

Watch Variables

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

4.114 FSI Receive Skew Compensation Block Element Delays

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf) This example uses the HRCAP module to measure the FSI RX delay elements. The measure delays can be graphed using the FSI Skew Compensation Utility.

The FSI receiver module has a programmable delay line on each of the external signal inputs: RXCLK, RXD0, and RXD1. The delay elements introduce delays on the respective lines. This is to facilitate adjustment for signal delays introduced by system level components such as signal buffers, ferrite beads, isolators, and so on, or board delays such as uneven trace lengths, long cable length, and so on. The length of the delay is controlled by setting the RX_DLY_LINE_CTRL register values for each line. There are 32 delay elements available for each of the external signal input. These delay elements must be activated accordingly, in order to ensure that the FSI RX module will meet the requirements for the setup time and hold time. The amount of delay introduced by each delay element can be measure using the high-resolution capture (HRCAP) module. An example project is available with the name of fsi_delay_tap_measurement which measure the delay elements on RXD1 in nano-seconds.

External Connections

- None

Watch Variables

- **delays** Value of delays, in nanoseconds

4.115 FSI Skew Calibration in Single Data Line Mode (RX Device)

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_single_line_delay_select_tx In this example, the FSI module is configured to listen for a ping at single data rate (using RXD0). The software tests whether the ping sent from the TX device is correctly received against all combinations of delay elements activated. RXD0: 0-31 delay elements activated RXCLK: 0-31 delay elements activated The software stores the status of the ping received (fail/pass) for each of the 32x32 combinations of the delay line elements. This result can be graphed using the FSI Skew Compensation Utility.

External Connections

For FSI external connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

- **pingAndDataStatus** The success/failure status for each config

4.116 FSI Skew Calibration in Single Data Line Mode (TX Device)

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_single_line_delay_select_rx This example configures the FSI module to transmit pings at single data rate (using RXD0). Run the C28x device with this application first then run the core with the fsi_single_line_delay_select_rx application. This example must be used with fsi_single_line_delay_select_rx

In fsi_single_line_delay_select_rx (RX Device) example, the FSI module is configured to listen for a ping at single data rate (using RXD0). The software tests whether the ping sent from the TX device is correctly received against all combinations of delay elements activated. RXD0: 0-31 delay elements activated RXCLK: 0-31 delay elements activated The software stores the status of the ping received (fail/pass) for each of the 32x32 combinations of the delay line elements. This result can be graphed using the FSI Skew Compensation Utility.

External Connections

For FSI external connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

4.117 FSI Skew Calibration in Dual Data Line Mode (RX Device)

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_dual_line_delay_select_tx In this example, the FSI module is configured to listen for a ping at dual data rate (using both RXD0 and RXD1). The software tests whether the ping sent from the TX device is correctly received against all combinations of delay elements activated. RXD0: 0-31 delay elements activated RXD1: 0-31 delay elements activated RXCLK: 0-31 delay elements activated The software stores the status of the ping received (fail/pass) for each of the 32x32x32 combinations of the delay line elements. This result can be graphed using the FSI Skew Compensation Utility.

External Connections

For FSI external connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

- **pingAndDataStatus** The success/failure status for each config

4.118 FSI Skew Calibration in Dual Data Line Mode (TX Device)

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_dual_line_delay_select_rx This example configures the FSI module to transmit pings at dual data rate (using RXD0 and RXD1). Run the C28x device with this application first then run the core with the fsi_dual_line_delay_select_rx application. This example must be used with fsi_dual_line_delay_select_rx

In fsi_dual_line_delay_select_rx example, the FSI module is configured to listen for a ping at dual data rate (using both RXD0 and RXD1). The software tests whether the ping sent from the TX device is correctly received against all combinations of delay elements activated. RXD0: 0-31 delay elements activated RXD1: 0-31 delay elements activated RXCLK: 0-31 delay elements activated The software stores the status of the ping received (fail/pass) for each of the 32x32x32 combinations of the delay line elements. This result can be graphed using the FSI Skew Compensation Utility.

External Connections

For FSI external connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

4.119 FSI Find Optimal Number of Delay Elements Activated For FSIRX

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_find_optimal_delay_device2 This example showcases how to find the optimal point for the number of delay elements activated on RXD0, RXD1 and RXCLK for optimal performance.

The optimal number of elements selected for the FSI RX module can be calculated using both single and dual data rate.

External Connections

For FSI external P2P connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

- **exePoint** The RXD0, RXD1 and RXCLK optimal skew compensation mode

4.120 FSI Find Optimal Number of Delay Elements Activated For FSIRX

In order to understand this example better and visualize the results please refer to: [Fast Serial Interface (FSI) Skew Compensation](http://www.ti.com/lit/an/spracj9/spracj9.pdf)

Companion: fsi_find_optimal_delay_device1 This example showcases how to find the optimal point for the number of delay elements activated on RXD0, RXD1 and RXCLK for optimal performance. The optimal number of elements selected for the FSI RX module can be calculated using both single and dual data rate.

External Connections

For FSI external P2P connection, make below GPIO settings in example code.

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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

- **exePoint** The RXD0, RXD1 and RXCLK optimal skew compensation mode

4.121 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.122 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.123 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.124 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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK

- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.125 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.126 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.127 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

LaunchPad FSI Header GPIOs:

- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.128 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_7 -> GPIO_9 :: To connect FSITX_CLK with SPICLKA
- GPIO_6 -> GPIO_8 :: To connect FSITX_TX0 with SPIPICOA
- GPIO_5 -> GPIO_11 :: To connect FSITX_TX1 with SPIPTEA

Watch Variables

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

4.129 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_9 :: To connect FSIRX_CLK with SPICLKA
- GPIO_12 -> GPIO_8 :: To connect FSIRX_RX0 with SPIPICOA

Watch Variables

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

4.130 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.131 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 LaunchPad FSI Header GPIOs:
- GPIO_7 -> FSITX_CLK
- GPIO_6 -> FSITX_TX0
- GPIO_25 -> FSITX_TX1
- GPIO_33 -> FSIRX_CLK
- GPIO_12 -> FSIRX_RX0
- GPIO_2 -> FSIRX_RX1

Watch Variables

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

4.132 FSI and SPI communication (fsi_ex9_spi_master_tx_drivers)

Port of fsi_ex6_spi_mater_tx example using spifsi drivers. 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 master Tx and SPI as slave 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 on controlCard in GPIO settings

- GPIO_7 -> GPIO_9 :: To connect FSITX_CLK with SPICLKA (56 -> 71 on docking station)
- GPIO_6 -> GPIO_8 :: To connect FSITX_TX0 with SPISIMOA (54 -> 87 on docking station)
- GPIO_5 -> GPIO_11 :: To connect FSITX_TX1 with SPISTEA (52 -> 73 on docking station)

Watch Variables

- **dataFrameCntr** Number of Data frame transfered from FSI.
- **spiRxCntr** Number of Data frame received at SPI.
- **error** Non zero for transmit/receive data mismatch

4.133 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

4.134 Device GPIO Toggle

Configures the device GPIO through the sysconfig file. The GPIO pin is toggled in the infinite loop. In order to migrate the project within syscfg to any device, click the swtich button under the device view and select your corresponding device to migrate, saving the project will auto-migrate your project settings.

4.135 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.

4.136 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 AIO224 and AIO225

(AIO224 assigned to XINT1 and AIO225 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.

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

External Connections

- Connect GPIO30 to AIO224. AIO224 will be assigned to XINT1
- Connect GPIO31 to AIO225. AIO225 will be assigned to XINT2
- GPIO34 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

4.137 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

4.138 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.

4.139 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.

4.140 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.

4.141 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.

4.142 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

4.143 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 on Control card

- Connect external I2C EEPROM at address 0x50
- Connect GPIO32/SDAA on controlCARD to external EEPROM SDA (serial data) pin
- Connect GPIO33/SCLA on controlCARD to external EEPROM SCL (serial clock) pin

External Connections on Launchpad

- Connect external I2C EEPROM at address 0x50
- Connect GPIO35/SDAA on Launchpad to external EEPROM SDA (serial data) pin
- Connect GPIO37/SCLA on Launchpad to external EEPROM SCL (serial clock) pin

4.144 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 | GPIO37 | SCL SDA | GPIO35 | SDA Make sure to connect GND pins if EEPROM and C2000 device are in different board. ——————————

4.145 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 | GPIO37 | SCL SDA | GPIO35 | 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

##

4.146 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

4.147 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

4.148 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.

4.149 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

4.150 LED Blinky Example with DCSM

This example demonstrates how to blink a LED and program the DCSM OTP.

External Connections

- None.

Watch Variables

- None.

4.151 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

4.152 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

4.153 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

4.154 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)

4.155 Low Power Modes: Device Idle Mode and Wakeup using GPIO

This example puts the device into IDLE mode and then wakes up the device from IDLE using XINT1 which triggers on a falling edge of GPIO0.

The GPIO0 pin must be pulled from high to low by an external agent for wakeup. GPIO0 is configured as an XINT1 pin to trigger an XINT1 interrupt upon detection of a falling edge.

Initially, pull GPIO0 high externally. To wake device from IDLE mode by triggering an XINT1 interrupt, pull GPIO0 low (falling edge). The wakeup process begins as soon as GPIO0 is held low for the time indicated in the device datasheet.

GPIO1 is pulled high before entering the IDLE mode and is pulled low when in the external interrupt ISR.

External Connections

- GPIO0 needs to be pulled low to wake up the device.
- On device wakeup, the GPIO1 will be low and LED1 will start blinking

4.156 Low Power Modes: Device Idle Mode and Wakeup using Watchdog

This example puts the device into IDLE mode and then wakes up the device from IDLE using watchdog timer.

The device wakes up from the IDLE mode when the watchdog timer overflows, triggering an interrupt. A pre scalar is set for the watchdog timer to change the counter overflow time.

GPIO1 is pulled high before entering the IDLE mode and is pulled low when in the wakeup ISR.

External Connections

- On device wakeup, the GPIO1 will be low and LED1 will start blinking

4.157 Low Power Modes: Halt Mode and Wakeup using GPIO

This example puts the device into HALT mode. If the lowest possible current consumption in HALT mode is desired, the JTAG connector must be removed from the device board while the device is in HALT mode.

For applications that require minimal power consumption during HALT mode, application software should power off the XTAL prior to entering HALT by setting the XTALCR.OSCOFF bit or by using the driverlib function SysCtl_turnOffOsc(SYSCTL_OSCSRC_XTAL);. If the OSCCLK source is configured to be XTAL, the application should first switch the OSSCLK source to INTOSC1 or INTOSC2 prior to setting XTALCR.OSCOFF.

This example puts the device into HALT mode and then wakes up the device from HALT using an LPM wakeup pin.

The pin GPIO0 is configured as the LPM wakeup pin to trigger a WAKEINT interrupt upon detection of a low pulse.The GPIO0 pin must be pulled from high to low by an external agent for wakeup.

GPIO1 is pulled high before entering the STANDBY mode and is pulled low when in the wakeup ISR.

External Connections

- On device wakeup, the GPIO1 will be low and LED1 will start blinking

4.158 Low Power Modes: Halt Mode and Wakeup

This example puts the device into HALT mode. If the lowest possible current consumption in HALT mode is desired, the JTAG connector must be removed from the device board while the device is in HALT mode.

For applications that require minimal power consumption during HALT mode, application software should power off the XTAL prior to entering HALT by setting the XTALCR.OSCOFF bit or by using the driverlib function SysCtl_turnOffOsc(SYSCTL_OSCSRC_XTAL);. If the OSCCLK source is configured to be XTAL, the application should first switch the OSSCLK source to INTOSC1 or INTOSC2 prior to setting XTALCR.OSCOFF.

This example puts the device into HALT mode and then wakes up the device from HALT using an LPM wakeup pin.

The pin GPIO0 is configured as the LPM wakeup pin to trigger a WAKEINT interrupt upon detection of a low pulse.The GPIO0 pin must be pulled from high to low by an external agent for wakeup.

In this example, the watchdog timer is clocked, and is configured to produce watchdog reset as a timeout mechanism.

GPIO1 is pulled high before entering the STANDBY mode and is pulled low when in the wakeup ISR.

External Connections

- On device wakeup, the GPIO1 will be low and LED1 will start blinking

4.159 PGA DAC-ADC External Loopback Example

This example generates 400 mV using the DAC output (it uses an internal voltage reference). The output of the DAC is externally connected to PGA2 for a 3x gain amplification. It uses two ADC channels to sample the DAC output and the amplified voltage output from PGA2. The ADC is connected to these signals internally.

External Connections on Control Card

- Connect DACA_OUT (Analog Pin A0) to PGA2_IN (Pin 21 on HSEC connector of
- ControlCard)
- Connect PGA246NEG to GND

External Connections on Launchpad

- Connect DACA_OUT (Analog Pin A0) to PGA2_IN (Analog Pin A3).
- Connect PGA246NEG to GND

Watch Variables

- **dacResult** - The DAC output voltage.
- **pgaResult** - The amplified DAC voltage.
- **pgaGain** - The ratio of the amplified DAC voltage to the original DAC output. This should always read a value of $\sim$3.0.

4.160 Empty SysCfg & Driverlib Example

This example is an empty project setup for SysConfig and Driverlib development.

4.161 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.

For LaunchPad and custom devices, there may be need to configure different GPIO for the SCI_RX and SCI_TX pins if GPIO9 and GPIO8 are not available for these devices. This can be configured using the included .syscfg file. Open the SCI peripheral, open the "PinMux" / "Peripheral and Pin Configuration" configuration section and choose GPIOs that are available on the given board. Update GPIO_SCIRX_NUMBER below to match the RX choice. Please refer to the LaunchPad user guide for list of available GPIO.

There may also be a need to add a global define to choose the LaunchPad. For example, in device.h, some devices require choosing a LaunchPad configuration, such as writing define _LAUNCHXL_F2####. Please ensure these are defined if used.

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

4.162 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

4.163 SCI Interrupt Echoback

This test receives and echo-backs data through the SCI-A port via interrupts.

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

- counter - the number of characters sent

External Connections

Connect the SCI-A port to a PC via a transceiver and cable.

- GPIO28 is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO29 is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

4.164 SCI Interrupt Echoback with FIFO

This test receives and echo-backs data through the SCI-A port via interrupts two characters at a time. A Rx interrupt is triggered after the FIFO status level is two or greater. Once two characters are in the RXFIFO, the SCI Rx ISR will be triggered and will read two characters from the FIFO and write them to the transmit buffer. The SCI Tx ISR will then be triggered again to request more data from the terminal.

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 two characters which it will echo back to the terminal.

Watch Variables

- counter - the number of character pairs sent

External Connections

Connect the SCI-A port to a PC via a transceiver and cable.

- GPIO28 is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO29 is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

4.165 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 SCI-A port to a PC via a transceiver and cable.

- GPIO28 is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO29 is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

4.166 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.

To view results in graph window, add breakpoint in ISR where counter is reset once the buffer is full and plot the watch variables.

External Connections

- Connect Sigma-Delta streams to (SD-D1, SD-C1 to SD-D4, SD-C4) on GPIO24-GPIO31

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

4.167 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.
- 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

Connect Sigma-Delta streams to (SD-D1, SD-C1 to SD-D4,SD-C4) on GPIO24-GPIO31

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

4.168 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.
- 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

Connect Sigma-Delta streams to (SD-D1, SD-C1 to SD-D4,SD-C4) on GPIO24-GPIO31

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

4.169 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.
- 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

Connect Sigma-Delta streams to (SD-D1, SD-C1 to SD-D4,SD-C4) on GPIO24-GPIO31

Watch Variables

- **filter1Result** - Output of filter 1
- **filter2Result** - Output of filter 2
- **filter3Result** - Output of filter 3
- **filter4Result** - Output of filter 4

4.170 SDFM Type 1 Filter FIFO

This example configures SDFM1 filter 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

Connect Sigma-Delta streams to (SD-D1, SD-C1 to SD-D4,SD-C4) on GPIO24-GPIO31

Watch Variables

- **filter1Result** - Output of filter 1

4.171 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.

4.172 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.

4.173 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

4.174 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

4.175 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

Refer to SysConfig for external connections (GPIO pin numbers) specific to each device

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)

4.176 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

Refer to SysConfig for external connections (GPIO pin numbers) specific to each device

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

4.177 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

4.178 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

- Connect external SPI EEPROM
- Connect GPIO16 (PICO) to external EEPROM SI pin
- Connect GPIO17 (POCI) to external EEPROM SO pin
- Connect GPIO56 (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

4.179 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

- Connect external SPI EEPROM
- Connect GPIO16 (PICO) to external EEPROM SI pin
- Connect GPIO17 (POCI) to external EEPROM SO pin
- Connect GPIO56 (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

4.180 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 (100Mhz). 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 100Mhz 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

4.181 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 GPIO16 using an oscilloscope.

4.182 CPU Timers

This example configures CPU Timer0, 1, and 2 and increments a counter each time the timer asserts an interrupt. In order to migrate the project within syscfg to any device, click the swtich button under the device view and select your corresponding device to migrate, saving the project will auto-migrate your project settings.

External Connections

- None

Watch Variables

- cpuTimer0IntCount
- cpuTimer1IntCount
- cpuTimer2IntCount

4.183 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

4.184 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

5 Bit-Field Example Applications

These example applications show how to make use of various peripherals of a F28004x 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 of these examples reside in the `device_support/f28004x/examples` subdirectory of the C2000Ware package.

Example Projects require CCS v6.2.0.00050 or newer

5.1 ADC ePWM Triggering

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

External Connections

- A1 should be connected to a signal to convert

Watch Variables

- **adcAResults** - A sequence of analog-to-digital conversion samples from pin A1. The time between samples is determined based on the period of the ePWM timer.

5.2 CLA ADC Sampling and Filtering with Buffering in a Background Task

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.

External Connections

- connect A0 to EPWM4A

Watch Variables

- None

5.3 CLA 5 Tap Finite Impulse Response Filter

This example implements a 5 Tap FIR filter. It will setup EPWM1 to trigger ADCA at a frequency of 50KHz. Once the ADC completes sampling, it will trigger task 7 of the CLA which runs the filter on the ADC sample.

EPWM2 is setup to switch at 10KHz. Connect pin EPWM2A to ADCA0 on the board to see the filtering effect.

Memory Allocation

- CPU to CLA1 Message RAM
    - A - Filter Coefficients
- CLA1 to CPU Message RAM
    - voltFilt - Filtered sample
    - X - filter sample delay line

Watch Variables

- voltFilt - Filtered sample
- X - filter sample delay line

External Connections

- EPWM2A (GPIO2) to ADCA0

5.4 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 ADCINB3.

Watch Variables

- None.

5.5 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 5Hz and 10Hz using the shadow registers to load the next period/compare values.

5.6 Device GPIO Setup

Configures the F28004X 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.

5.7 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 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

5.8 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 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

5.9 I2C master slave communication using bit-field and without FIFO

This program shows how to use I2CA in master configuration. This example uses polling and does not use interrupts or FIFO

Requires two Control Cards - one configured as Master and other as Slave

Master will run the binary generated from "i2c_ex1_master.projectspec"

Slave will run the binary generated from "i2c_ex1_slave.projectspec"

External Connections on Control Cards should be made as shown below

——————————————————- Signal | I2CA on Board1 | I2CA on Board 2 — ——————————————————- SCL | GPIO_PIN_SCLA | GPIO_PIN_SCLA SDA | GPIO_PIN_SDAA | GPIO_PIN_SDAA GND | GND | GND ——————————————————————

Example1: Master Transmitter and Slave Receiver Example2: Master Receiver and Slave Transmitter Example3: Master Transmitter and Slave Receiver followed by Master Receiver and Slave Transmitter Example4: Master Receiver and Slave Transmitter followed by Master Transmitter and Slave Receiver

Watch Variables in memory window

- **I2CA_TXdata**
- **I2CA_RXdata**

5.10 I2C master slave communication using bit-field and without FIFO

This program shows how to use I2CA in slave configuration. This example uses I2C interrupts and doesn't use FIFO.

Requires two Control Cards - one configured as Master and other as Slave

Master will run the binary generated from "i2c_ex1_master.projectspec"

Slave will run the binary generated from "i2c_ex1_slave.projectspec"

External Connections on Control Cards should be made as shown below

———————————————— Signal | I2CA on Board1 | I2CA on Board 2 —— ———————————————— SCL | GPIO_PIN_SCLA | GPIO_PIN_SCLA SDA | GPIO_PIN_SDAA | GPIO_PIN_SDAA GND | GND | GND ————————————————

Watch Variables in memory window

- **I2CA_TXdata**
- **I2CA_RXdata**

5.11 Software Prioritized Interrupts (interrupts_ex1_sw_prioritization)

For most applications, the hardware prioritizing of the the PIE module is sufficient. For applications that need custom prioritizing, this example illustrates how this can be done through software.

This program simulates interrupt conflicts by writing to the PIEIFR registers. This will cause multiple interrupt requests to come into the PIE block at the same time.

The interrupt service routines are software prioritized as per the table found in the f28004x_sw_prioritized_isr_levels.h file.

Running the Application

1. Before compiling you must set the Global and Group interrupt priorities in the f28004x_sw_prioritized_isr_levels.h file.
2. Select which test case you'd like to run with the #define CASE directive (1-9, default 1).

3. Compile the code, load, and run
4. At the end of each test there is a hard coded breakpoint (ESTOP0). When code stops at the breakpoint, examine the ISRTrace buffer to see the order in which the ISRs completed. All PIE interrupts will be added to the ISRTrace. The ISRTrace will consist of a list of hex values as shown:

   0x00wx <- PIE Group w interrupt x finished first

   0x00yz <- PIE Group y interrupt z finished next
5. If desired, set a new set of Global and Group interrupt priorities and repeat the test to see the change.

Watch Variables

- **ISRTrace** - Trace of ISRs in the order they complete. After each test, examine this buffer to determine if the ISRs completed in the order desired.

5.12 F280049C LaunchPad Out of Box Demo Example

This program is the demo program that comes pre-loaded on the F280049C 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 ADCINA5 once a second. If the sample is greater than midescale the red LED on the board is lit, while if it is lower a 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 to ADCINA5 to change LED based on value

Watch Variables

- None.

5.13 LED Blinky Example

This example demonstrates how to blink a LED.

External Connections

- None.

Watch Variables

- None.

5.14 PGA DAC-ADC External Loopback Example

This example generates 400 mV using the DAC output (it uses an internal voltage reference). The output of the DAC is externally connected to PGA1 for a 3x gain amplification. It uses two ADC channels to sample the DAC output and the amplified voltage output from PGA1. The ADC is connected to these signals internally.

External Connections

- Connect DACA_OUT (Analog Pin A0) to PGA1_IN (Pin 15 on HSEC connector of
- CCard)
- Connect PGA1NEG to GND

Watch Variables

- **dacResult** - The DAC output voltage.
- **pgaResult** - The amplified DAC voltage.
- **pgaGain** - The ratio of the amplified DAC voltage to the original DAC output. This should always read a value of $\sim$3.0.

5.15 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 SCI-A port to a PC via a transceiver and cable.

- GPIO28 is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO29 is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

5.16 SPI Digital Loop Back

This program uses the internal loop back test mode of the peripheral. Other than boot mode pin configuration, no other hardware configuration is required. Interrupts are not used.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001 0002 0003 0004 0005 0006 0007 .... FFFE FFFF

This pattern is repeated forever.

Watch Variables

- **sdata** - sent data
- **rdata** - received data

5.17 SPI Digital Loop Back with DMA (spi_loopback_dma)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Both DMA Interrupts and the SPI FIFOs 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

....

007E 007F

Watch Variables

- **sData** - Data to send
- **rData** - Received data
- **rData_point** - Used to keep track of the last position in the receive stream for error checking

5.18 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

- CpuTimer0.InterruptCount

- CpuTimer1.InterruptCount
- CpuTimer2.InterruptCount

6 Device APIs for examples

6.1 Introduction

This chapter provides information on the APIs included in device.c file

6.2 API Functions

Functions

- void __error__ (const char *filename, uint32_t line)
- void Device_enableAllPeripherals (void)
- void Device_init (void)
- void Device_initGPIO (void)
- bool Device_verifyXTAL (float freq)

6.2.1 Function Documentation

6.2.1.1 __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

6.2.1.2 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.

Returns:

None

6.2.1.3 void Device_init (void)

Function to initialize the device. Primarily initializes system control to a known state by disabling the watchdog, setting up the SYSCLKOUT frequency, and enabling the clocks to the peripherals. The function also configures the GPIO pins 22 and 23 in digital mode. To configure these pins as analog pins, use the function GPIO_setAnalogMode.

Returns:
None

6.2.1.4 void Device_initGPIO (void)

Function to disable pin locks and enable pullups on GPIOs.

Returns:
None

6.2.1.5 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