F280013x 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 18:46:47 IST 2023.

Table of Contents

1 Introduction

The Texas Instruments® F280013x 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 F280013x 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 F280013x family of devices start by reading this section first.

The F280013x 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/f280013x/examples` directory.
- The bit-field example applications can be found in the `~/device_support/f280013x/examples` directory.

F280013x Example Projects

- Driverlib Example projects tested with: C2000 Compiler v22.6.0.LTS

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

v4.01.00.00

- First Release of C2000Ware 4.01.00 with support only for F280013x

2 Getting Started and Troubleshooting

2.1 Introduction

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 F280013x controlCARD, but these same ideas should apply to other boards with minimal translation.

2.2 Project Creation

A typical F280013x 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 "TMS320F2800137". Name your project and choose a location for it to reside. Click Finish and your project will be created.

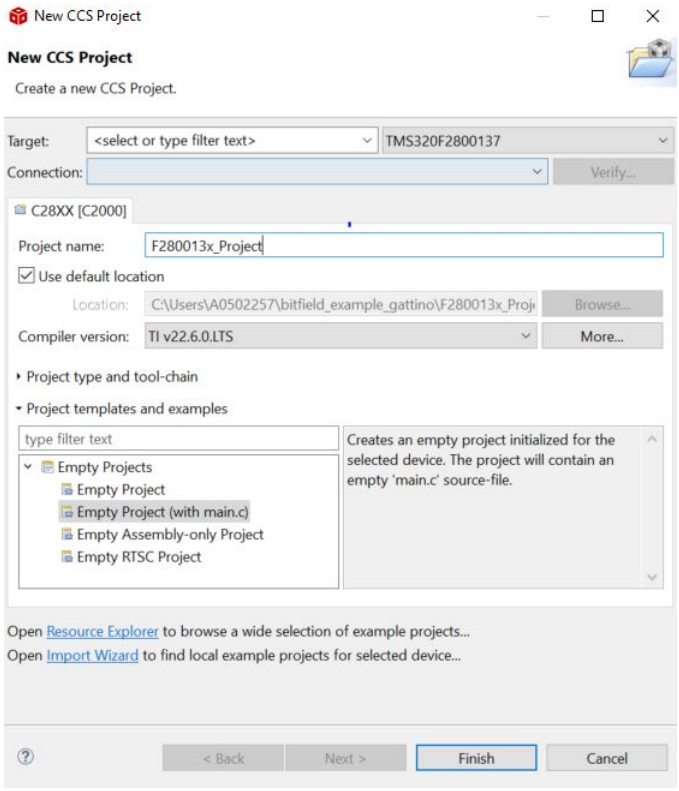

Figure 2.1: Creating a new C28 project

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

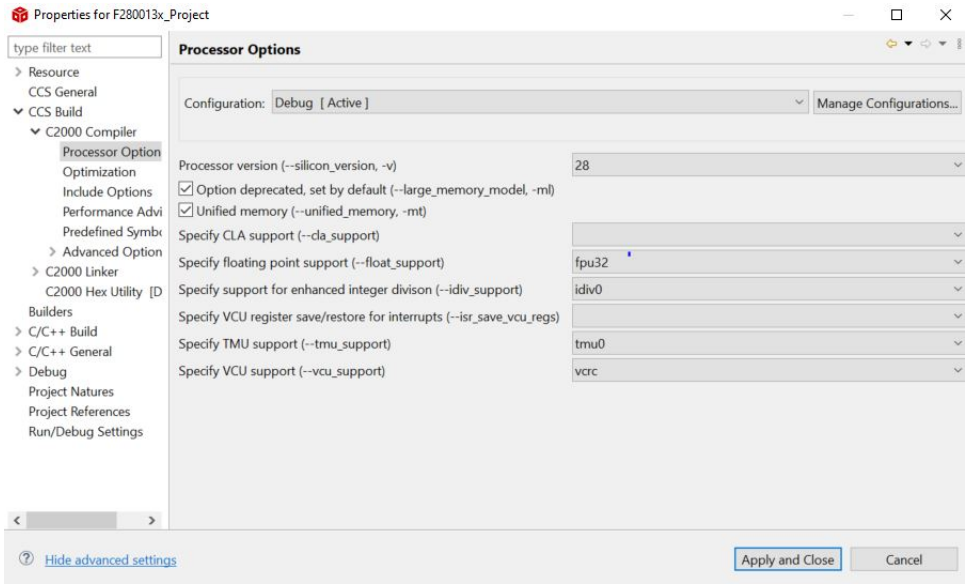

Figure 2.2: Project configuration dialog box

3. In the C2000 Compiler entry look for and select the Include Options. Click on the add directory icon to add a directory to the search path. Click the File System button to browse to the `driverlib\f280013x\driverlib` folder of your C2000Ware installation (typically `C:\ti\c2000\C2000Ware_<version>\driverlib\f280013x\driverlib`). Click ok to add this path, and repeat this same process to add the `device_support\f280013x\common\include` directory.

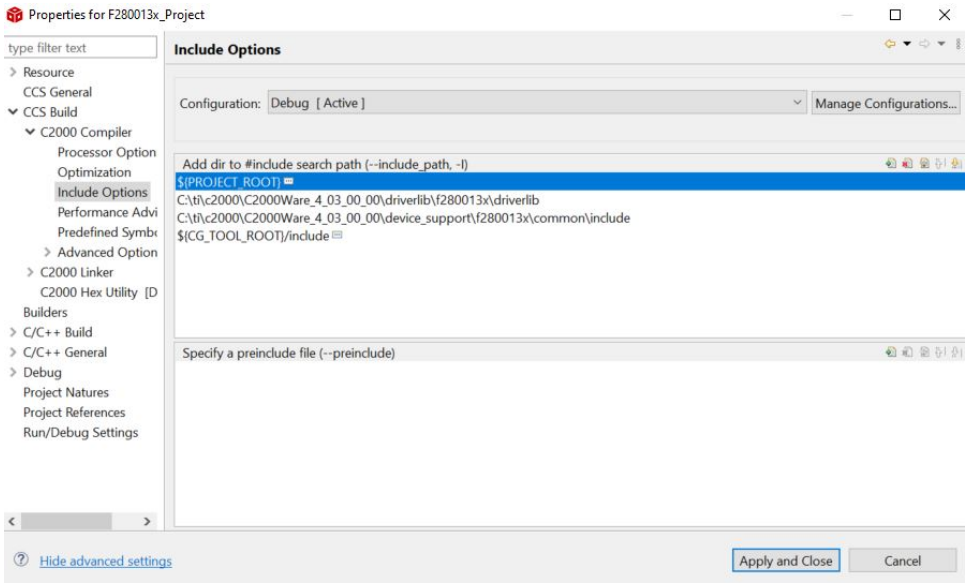

Figure 2.3: Project configuration dialog box

4. Click on the Linker File Search Path. Add the following directory to the search path: `device_support\f280013x\common\cmd`. Then you'll also want to add the following files: `rts2800_fpu32_eabi.lib` and `280013x_generic_ram_lnk.cmd`. Finally, delete `libc.a`, we will use `rts2800_fpu32_eabi.lib` as our run time support library instead. Select ok to close out of the Build Properties.

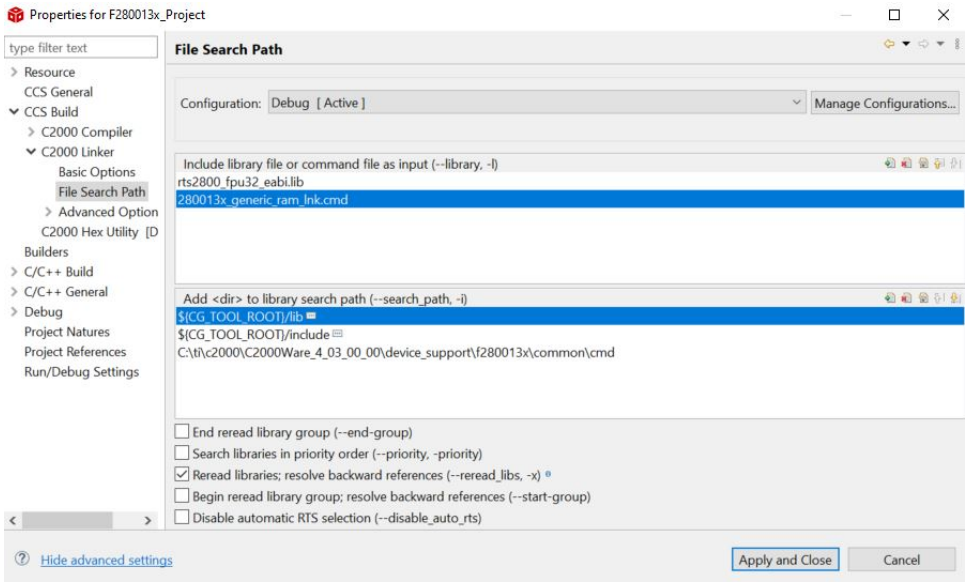

Figure 2.4: Project configuration dialog box

5. 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.
6. 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\f280013x\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 file as well:
   - `driverlib\f280013x\driverlib\ccs\Debug\driverlib.lib`

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

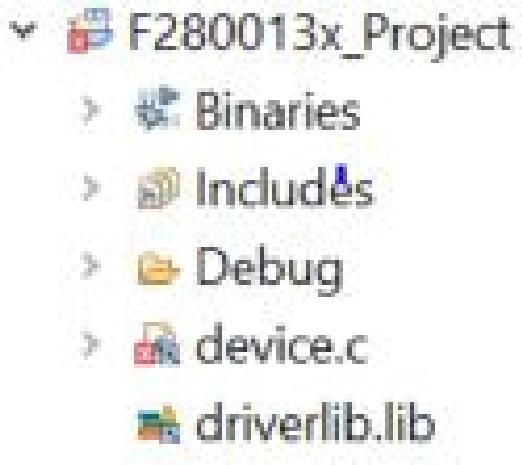

Figure 2.5: Linking files to project

7. 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);
        }
}
```

8. 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 F280013x device. When the code runs, you should see the LED blink.

2.3 Project: Adding Bitfield or Driverlib Support

F280013x 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\f280013x\driverlib`
2. Include the following header file in the project main source file: `device_support\f280013x\common\include\driverlib.h`
3. Add or link the `driverlib.lib` library to the project. Location of file: `driverlib\f280013x\driverlib\ccs\Debug`

Adding Bitfield Support

1. Add the following include directory path to the project: `device_support\f280013x\headers\include`
2. Include the following header file in the project main source file: `device_support\f280013x\headers\include\f280013x_device.h`
3. Add or link the `f280013x_globalvariabledefs.c` file to the project. Location of file: `device_support\f280013x\headers\source`
4. Add or link the `f280013x_headers_nonbios.cmd` file to the project. Location of file: `device_support\f280013x\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 F280013x 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"

F280013x 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.

"\"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	4
EMUINT	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 16 interrupts. Thus the total possible number of available interrupts in the PIE is 192.

Each of the PIE groups has its own interrupt enable register (PIEIERx) to control which of the 16 interrupts (INTx.1 - INTx.16) 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 F280013x 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 F280013x 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 F280013x 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 F280013x 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 16-interrupts multiplexed within that group.

The 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 - 16)

   These values are used to assign a priority level to each of the 16 interrupts within a PIE group. 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.

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 - 16)

  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.

```
//
// 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 F280013x 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/f280013x/examples` subdirectory of the C2000Ware package.

Example Projects require CCS v11.0.0 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 OSDETECT 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 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

4.13 ADC PPB Limits (adc_ppb_limits)

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

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

External Connections

- A0 should be connected to a signal to convert

Watch Variables

- None

4.14 ADC PPB Delay Capture (adc_ppb_delay)

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

Two asynchronous ADC triggers are setup:

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

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

After the program runs, the memory will contain:

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

4.15 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

4.16 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

4.17 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.

4.18 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

4.19 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

4.20 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

4.21 CMPSS Asynchronous Trip

This example enables the CMPSS1 COMPH comparator and feeds the asynchronous CTRIPOUTH signal to the GPIO4/OUTPUTXBAR3 pin and CTRIPH to GPIO13/EPWM7B.

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 GPIO13 and is configured to be tripped by CTRIPOUTH.

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

- Trip signal(GPIO4) output is low
- PWM7B(GPIO13) gives a PWM signal

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

- Trip signal(GPIO4) output turns high
- PWM7B(GPIO13) gets tripped and outputs as high

External Connections

- Give input on CMPIN1P (The pin is shared with ADCINA2)
- Outputs can be observed on GPIO4 and GPIO13 using an oscilloscope

Watch Variables

- None

4.22 CMPSS Digital Filter Configuration

This example enables the CMPSS1 COMPH comparator and feeds the output through the digital filter to the GPIO4/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,

- GPIO4 output is low

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

- GPIO4 output turns high

4.23 CMPSSLITE Asynchronous Trip

This example enables the CMPSSLITE2 COMPH comparator and feeds the asynchronous CTRIPOUTH signal to the GPIO4/OUTPUTXBAR3 pin and CTRIPH to GPIO29/EPWM7B.

CMPSS is configured to generate trip signals to trip the EPWM signals. CMPIN2P 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 GPIO29 and is configured to be tripped by CTRIPOUTH.

When a low input(VSS) is provided to CMPIN2P,

- Trip signal(GPIO4) output is low
- PWM7B(GPIO29) gives a PWM signal

When a high input(higher than VDD/2) is provided to CMPIN2P,

- Trip signal(GPIO4) output turns high
- PWM7B(GPIO29) gets tripped and outputs as high

External Connections

- Give input on CMPIN2P. The pin is shared with ADCINA9 for CMPHPMXSEL = 2
- Outputs can be observed on GPIO4 and GPIO29 using an oscilloscope

Watch Variables

- None

4.24 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 = 120Mhz). Seed is the value that gets loaded into the Counter.

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

Note:

In this device, by default, the XTAL is disabled and INTOSC2 is configured as the PLL source. To use XTAL (of frequency 20MHz) as the PLL source, update the device.h file with following changes :

- Comment the line define USE_PLL_SRC_INTOSC

- Uncomment the line define USE_PLL_SRC_XTAL If you are using a XTAL with different frequency, update the macros DEVICE_OSCSRC_FREQ, DEVICE_SETCLOCK_CFG, DEVICE_SYSCLK_FREQ accordingly.

To run this example, XTAL needs to be enabled. XTAL is enabled if you use it as the PLL source. You can also enable it by using the function *SysCtl_turnOnOsc(SYSCTL_OSCSRC_XTAL)*

External Connections

- None

Watch Variables

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

4.25 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.

Note:

In this device, by default, the XTAL is disabled and INTOSC2 is configured as the PLL source. To use XTAL (of frequency 20MHz) as the PLL source, update the device.h file with following changes :

- Comment the line define USE_PLL_SRC_INTOSC
- Uncomment the line define USE_PLL_SRC_XTAL If you are using a XTAL with different frequency, update the macros DEVICE_OSCSRC_FREQ, DEVICE_SETCLOCK_CFG, DEVICE_SYSCLK_FREQ accordingly.

To run this example, XTAL needs to be enabled. XTAL is enabled if you use it as the PLL source. You can also enable it by using the function *SysCtl_turnOnOsc(SYSCTL_OSCSRC_XTAL)*

External Connections

- None

Watch Variables

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

4.26 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 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

4.27 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

4.28 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 = 100Mhz). 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.

Note:

In this device, by default, the XTAL is disabled and INTOSC2 is configured as the PLL source. To use XTAL (of frequency 20MHz) as the PLL source, update the device.h file with following changes :

- Comment the line define USE_PLL_SRC_INTOSC
- Uncomment the line define USE_PLL_SRC_XTAL If you are using a XTAL with different frequency, update the macros DEVICE_OSCSRC_FREQ, DEVICE_SETCLOCK_CFG, DEVICE_SYSCLK_FREQ accordingly.

To run this example, XTAL needs to be configured as PLL source.

External Connections

- None

Watch Variables

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

4.29 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)

4.30 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.

4.31 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.

4.32 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.

4.33 Empty Project Example

This example is an empty project setup for Driverlib development.

4.34 EPG Generate Serial Data Shift Mode

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

External Connections

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

4.35 EPG Generating Synchronous Clocks

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

External Connections

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

Watch Variables

- sigGenActiveData - Active Data of signal generator transform output

4.36 EPG Generating Two Offset Clocks

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

External Connections

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

4.37 EPG Generating Two Offset Clocks With SIGGEN

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

External Connections

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

4.38 EPG Generate Serial Data

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

External Connections

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

4.39 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.

4.40 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

4.41 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

4.42 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.

4.43 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.

4.44 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.

4.45 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.

4.46 ePWM Digital Compare

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO24's PULL-UP resistor is enabled, in order to test the trip, PULL this pin to GND

External Connections

- GPIO0 EPWM1A
- GPIO1 EPWM1B
- GPIO24 TZ1, pull this pin low to trip the ePWM

Watch Variables

- None.

4.47 ePWM Digital Compare Event Filter Blanking Window

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO24'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
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO24'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
- GPIO24 TRIPIN1, pull this pin low to trip the ePWM

Watch Variables

- None.

4.48 ePWM Valley Switching

This example configures ePWM1 as follows

- ePWM1 with DCAEVT1 forcing the ePWM output LOW
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO24 is set to output and toggled in the main loop to trip the PWM

- ePWM1 with DCBEVT1 forcing the ePWM output LOW
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCAEVT1
- GPIO24 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
- GPIO24 TRIPIN1 (Output Pin, toggled through software)

Watch Variables

- None.

4.49 ePWM Digital Compare Edge Filter

This example configures ePWM1 as follows

- ePWM1 with DCBEVT2 forcing the ePWM output LOW as a CBC source
- GPIO24 is used as the input to the INPUT XBAR INPUT1
- INPUT1 (from INPUT XBAR) is used as the source for DCBEVT2
- GPIO24 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
- GPIO24 TRIPIN1 (Output Pin, toggled through software)

Watch Variables

- None.

4.50 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.

4.51 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

4.52 Position and Speed Measurement Using eQEP

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

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

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

The configuration for this example is as follows

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

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

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

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

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

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

External Connections

- Connect GPIO6/eQEP1A to GPIO0/ePWM1A (simulates eQEP Phase A signal)
- Connect GPIO7/eQEP1B to GPIO1/ePWM1B (simulates eQEP Phase B signal)
- Connect GPIO9/eQEP1I to GPIO4 (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)

4.53 ePWM frequency Measurement Using eQEP via xbar connection

This example will calculate the frequency of an PWM signal using the eQEP module. ePWM1A is configured to generate this input signal with a frequency of 5 kHz. This ePWM signal is connected to input of eQEP using Input CrossBar and EPWM XBAR. ePWM module 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
- GPIO0 is connected to output of INPUT_XBAR1
- INPUT_XBAR1 is connected to output of PWMXBAR at TRIP4
- eQEPA source is configured as PWMXBAR.1 output (TRIP4)

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.

Watch Variables

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

4.54 Frequency Measurement Using eQEP via unit timeout interrupt

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

The configuration for this example is as follows

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

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

External Connections for Control Card

- Connect GPIO6/eQEP1A to GPIO0/ePWM1A

Watch Variables

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

4.55 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
- Connect GPIO6/eQEP1A to GPIO0/ePWM1A (simulates eQEP Phase A signal)
- Connect GPIO7/eQEP1B to GPIO1/ePWM1B (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)

4.56 Boot Source Code

Functions:

void copyData(void) uint32_t getLongData(void) void readReservedFn(void)

4.57 Erase Source Code

Functions:

4.58 Live DFU Command Functionality

4.59 SCI Boot Mode Routines

Functions:

uint32_t sciBoot(void) void sciaInit(void) uint32_t sciaGetWordData(void)

4.60 Flash Programming Solution using SCI

In this example, we set up a UART connection with a host using SCI, receive commands for CPU1 to perform which then sends ACK, NAK, and status packets back to the host after receiving and completing the tasks. This kernel has the ability to program, verify, unlock, reset, and run an application. Each command either expects no data from the command packet or specific data relative to the command.

In this example, we set up a UART connection with a host using SCI, receive an application for CPU01 in -sci8 ascii format to run on the device and program it into Flash.

Functions:

uint32_t sciBoot(void) void sciaInit(void) uint32_t sciaGetWordData(void)

4.61 Verify Source Code

##

4.62 Flash Programming with AutoECC, DataAndECC, DataOnly and EccOnly

This example demonstrates how to program Flash using API's following options 1. AutoEcc generation 2. DataOnly and EccOnly 3. DataAndECC

External Connections

- None.

Watch Variables

- None.

4.63 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 (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

4.64 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.

4.65 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.

4.66 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.

GPIO5 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope. **External Connections**

- Connect GPIO0 to AIO224. AIO224 will be assigned to XINT1
- Connect GPIO1 to AIO225. AIO225 will be assigned to XINT2
- GPIO5 can be monitored on an oscilloscope

Watch Variables

- xint1Count for the number of times through XINT1 interrupt
- xint2Count for the number of times through XINT2 interrupt
- loopCount for the number of times through the idle loop

4.67 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 A/B pins on an oscilloscope.

4.68 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.

4.69 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 A/B pins on an oscilloscope.

4.70 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 A/B pins on an oscilloscope.

4.71 HRPWM Slider Test

This example modifies the MEP control registers to show edge displacement due to HRPWM. Control blocks of the respective ePWM module channel A and B will have fine edge movement due to HRPWM logic. Load the hrpwm_slider.gel file. Select the HRPWM_eval from the GEL menu. A FineDuty slider graphics will show up in CCS. Load the program and run. Use the Slider to and

observe the EPWM edge displacement for each slider step change. This explains the MEP control on the EPwmxA channels.

Monitor ePWM1 A/B pins on an oscilloscope.

4.72 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

4.73 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

4.74 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

4.75 I2C EEPROM

This program will write 1-14 words to EEPROM and read them back. The data written and the EEPROM address written to are contained in the message structure, i2cMsgOut. The data read back will be contained in the message structure i2cMsgIn.

External Connections

- Connect external I2C EEPROM at address 0x50
- Connect DEVICE_GPIO_PIN_SDAA on to external EEPROM SDA (serial data) pin
- Connect DEVICE_GPIO_PIN_SCLA on to external EEPROM SCL (serial clock) pin

Watch Variables

- **i2cMsgOut** - Message containing data to write to EEPROM
- **i2cMsgIn** - Message containing data read from EEPROM

4.76 I2C Digital External Loopback with FIFO Interrupts

This program uses the I2CA and I2CB modules for achieving external loopback. The I2CA TX FIFO and the I2CB RX FIFO are used along with their interrupts.

A stream of data is sent on I2CA and then compared to the received stream on I2CB. The sent data looks like this:

0000 0001

0001 0002

0002 0003

....

00FE 00FF

00FF 0000

etc..

This pattern is repeated forever.

External Connections

- Connect SCLA(DEVICE_GPIO_PIN_SCLA) to SCLB (DEVICE_GPIO_PIN_SCLB)
- and SDAA(DEVICE_GPIO_PIN_SDAA) to SDAB (DEVICE_GPIO_PIN_SDAB)
- Connect DEVICE_GPIO_PIN_LED1 to an LED used to depict data transfers.

Watch Variables

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

4.77 I2C EEPROM

This program will shows how to perform different EEPROM write and read commands using I2C polling method EEPROM used for this example is AT24C256

External Connections

- Connect external I2C EEPROM at address 0x50 ——————————— Signal | I2CA | EEPROM ——————————— SCL | DEVICE_GPIO_PIN_SCLA | SCL SDA | DEVICE_GPIO_PIN_SDAA | SDA Make sure to connect GND pins if EEPROM and C2000 device are in different board. ———————————

4.78 I2C controller target communication using FIFO interrupts

This program shows how to use I2CA and I2CB modules in both controller and target configuration This example uses I2C FIFO interrupts and doesn't using polling

Example1: I2CA as controller Transmitter and I2CB working target Receiver Example2: I2CA as controller Receiver and I2CB working target Transmitter Example3: I2CB as controller Transmitter and I2CA working target Receiver Example4: I2CB as controller Receiver and I2CA working target Transmitter

External Connections on launchpad should be made as shown below

——————————— Signal | I2CA | I2CB ——————————— SCL | DEVICE_GPIO_PIN_SCLA | DEVICE_GPIO_PIN_SCLB SDA | DEVICE_GPIO_PIN_SDAA | DEVICE_GPIO_PIN_SDAB ———————————

Watch Variables in memory window

- **I2CA_TXdata**
- **I2CA_RXdata**
- **I2CB_TXdata**
- **I2CB_RXdata** stream for error checking

##

4.79 I2C EEPROM

This program will shows how to perform different EEPROM write and read commands using I2C interrupts EEPROM used for this example is AT24C256

External Connections

- Connect external I2C EEPROM at address 0x50 ———————————— Signal | I2CA | EEPROM ———————————— SCL | DEVICE_GPIO_PIN_SCLA | SCL SDA | DEVICE_GPIO_PIN_SDAA | SDA Make sure to connect GND pins if EEPROM and C2000 device are in different board. ———————————— Example 1: EEPROM Byte Write Example 2: EEPROM Byte Read Example 3: EEPROM word (16-bit) write Example 4: EEPROM word (16-bit) read Example 5: EEPROM Page write Example 6: EEPROM word Paged read

Watch Variables

- **TX_MsgBuffer** - Message buffer which stores the data to be transmitted
- **RX_MsgBuffer** - Message buffer which stores the data to be received

##

4.80 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

4.81 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

4.82 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.

4.83 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

4.84 F2800137 LaunchPad Out of Box Demo Example

This program is the demo program that comes pre-loaded on the F2800137 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 ADCINA6 once a second. If the sample is greater than midscale the red LED on the board is lit, while if it is lower the green LED is lit. Sample data is also displayed in a serial terminal via the board's back channel UART. You may view this data by configuring a serial terminal to the correct COM port at 115200 Baud 8-N-1.

External Connections

- Connect to COM port at 115200 Baud 8-N-1 for serial data
- Connect signal to ADCINA6 to change LED based on value

Watch Variables

- None.

4.85 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

4.86 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

4.87 Low Power Modes: Device Standby Mode and Wakeup using GPIO

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

This example puts the device into STANDBY mode and then wakes up the device from STANDBY 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. Initially, pull GPIO0 high externally. To wake device from STANDBY mode, pull GPIO0 low for at least (2+QUALSTDBY), OSCLKS, then pull it high again.

The example then wakes up the device from STANDBY using GPIO0. GPIO0 wakes the device from STANDBY mode when a low pulse (signal goes high->low->high)is detected on the pin. This pin must be pulsed 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

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

4.88 Low Power Modes: Device Standby Mode and Wakeup using Watchdog

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

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

The device wakes up from the STANDBY mode when the watchdog timer overflows triggering an interrupt. In the ISR, the GPIO1 is pulled low. the GPIO1 is toggled to indicate the device is out of STANDBY mode. A pre scalar is set for the watchdog timer to change the counter overflow time.

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

4.89 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

4.90 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

4.91 Correctable & Uncorrectable Memory Error Handling

This example demonstrates error handling in case of various erroneous memory read/write operations. Error handling in case of CPU read/write violations, correctable & uncorrectable memory errors has been demonstrated. Correctable memory errors & violations can generate SYS_INT interrupt to CPU while uncorrectable errors lead to NMI generation.

External Connections

- None

Watch Variables

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

4.92 Empty SysCfg & Driverlib Example

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

4.93 Tune Baud Rate via UART Example

This example demonstrates the process of tuning the UART/SCI baud rate of a C2000 device based on the UART input from another device. As UART does not have a clock signal, reliable communication requires baud rates to be reasonably matched. This example addresses cases where a clock mismatch between devices is greater than is acceptable for communications, requiring baud compensation between boards. As reliable communication only requires matching the EFFECTIVE baud rate, it does not matter which of the two boards executes the tuning (the board with the less-accurate clock source does not need to be the one to tune; as long as one of the two devices tunes to the other, then proper communication can be established).

To tune the baud rate of this device, SCI data (of the desired baud rate) must be sent to this device. The input SCI baud rate must be within the +/- MARGINPERCENT of the TARGETBAUD chosen below. These two variables are defined below, and should be chosen based on the application requirements. Higher MARGINPERCENT will allow more data to be considered "correct" in noisy conditions, and may decrease accuracy. The TARGETBAUD is what was expected to be the baud rate, but due to clock differences, needs to be tuned for better communication robustness with the other device.

NOTE: Lower baud rates have more granularity in register options, and therefore tuning is more affective at these speeds.

External Connections for Control Card

- SCIA_RX/eCAP1 is on GPIO9, connect to incoming SCI communications
- SCIA_TX is on GPIO8, for observation externally

Watch Variables

- **avgBaud** - Baud rate that was detected and set after tuning

4.94 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

4.95 SCI Digital Loop Back with Interrupts

This test uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Both interrupts and the SCI FIFOs are used.

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

00 01

01 02

02 03

....

FE FF

FF 00

etc..

The pattern is repeated forever.

Watch Variables

- **sDataA** - Data being sent
- **rDataA** - Data received
- **rDataPointA** - Keep track of where we are in the data stream. This is used to check the incoming data

4.96 SCI Echoback

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

A terminal such as 'putty' can be used to view the data from the SCI and to send information to the SCI. Characters received by the SCI port are sent back to the host.

Running the Application Open a COM port with the following settings using a terminal:

- Find correct COM port
- Bits per second = 9600
- Data Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

The program will print out a greeting and then ask you to enter a character which it will echo back to the terminal.

Watch Variables

- loopCounter - the number of characters sent

External Connections

Connect the USB cable from Control card J1:A to PC

4.97 stdout redirect example

This test transmits data through the SCI-A port to a terminal

A terminal such as 'putty' can be used to view the data from the SCI. Characters received by the SCI port are sent back to the host.

Running the Application Open a COM port with the following settings using a terminal:

- Find correct COM port
- Bits per second = 9600
- Data Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

The program will print out three sentences: one to the SCIA, one to CCS, and a final one to SCIA.

External Connections

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

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

4.98 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

4.99 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

4.100 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 GPIO08 (PICO) to external EEPROM SI pin
- Connect GPIO10 (POCI) to external EEPROM SO pin
- Connect GPIO09 (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

4.101 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 (120Mhz). 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

4.102 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.

4.103 External-R based precision Oscillator (XROSC) Example

##

4.104 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

4.105 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

4.106 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 F280013x 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/f280013x/examples` subdirectory of the C2000Ware package.

Example Projects require CCS v11.0.0 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 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 GetTemperatureC function.

After the program runs, the memory will contain:

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

5.3 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.

5.4 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**

5.5 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**

5.6 LED Blinky Example

This example demonstrates how to blink a LED.

External Connections

- None.

Watch Variables

- None.

5.7 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)

5.8 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

5.9 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)

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.

Parameters:

None

Returns:

None

6.2.1.3 void Device_init (void)

Function to initialize the device. Primarily initializes system control to aknown state by disabling the watchdog, setting up the SYSCLKOUT frequency, and enabling the clocks to the peripherals.

Parameters:
None.

Returns:
None.

6.2.1.4 void Device_initGPIO (void)

Function to disable pin locks on GPIOs.

Parameters:
None

Returns:
None

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265