MSP430® Peripheral Driver Library

USER'S GUIDE

Copyright

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
http://www.ti.com/msp430

Revision Information

This is version 1.20.01.00 of this document, last updated on 2012-06-29$_{1}$6 : 06 : 00$_$0500.

Table of Contents

1 Introduction

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430 family of microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C except where absolutely not possible.
- They demonstrate how to use the peripheral in its common mode of operation.
- They are easy to understand.
- They are reasonably efficient in terms of memory and processor usage.
- They are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They can be built with more than one tool chain.

Some consequences of these design goals are:

- The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.
- The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which cannot be utilized by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.
- The APIs have a means of removing all error checking code. Because the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

Each MSP430ware driverlib API takes in the base address of the corresponding peripheral as the first parameter. This base address is obtained from the msp430 device specific header files (or from the device datasheet). The example code for the various peripherals show how base address is used. When using CCS, the eclipse shortcut "Ctrl + Space" helps. Type __MSP430 and "Ctrl + Space", and the list of base addresses from the included device specific header files is listed.

The following tool chains are supported:

- IAR Embedded Workbench®
- Texas Instruments Code Composer Studio™

2 How to create a new project that uses Driverlib

To create a driverlib project from scratch An emptyProject has been created for the convenience of the user so that he can create a project that uses driverlib. This is available in "C:\ti\msp430\MSP430ware_x_xx_xx_xx\examples\driverlib\5xx_6xx\00_emptyProject\IAR" "C:\ti\msp430\MSP430ware_x_xx_xx_xx\examples\driverlib\5xx_6xx\00_emptyProject\CCS" or the corresponding relative path where MSP430ware is installed. The features of the emptyProject are

- Includes driverlib library file by default
- Includes a main.c by default that has the following statements

"#include "inc/hw_memmap.h"

void main (void) { } "

- Project is build by default for MSP430F5438A and has a large data model since driverlib is built by defualt for large data model.
- The project include path has the following added "C:\ti\msp430\MSP430ware_x_xx_xx_xx" or the corresponding path where MSP430ware is installed.

3 10-Bit Analog-to-Digital Converter (ADC10)

3.1 Introduction

The 10-Bit Analog-to-Digital (ADC10) API provides a set of functions for using the MSP430Ware ADC10 modules. Functions are provided to initializae the ADC10 modules, setup signal sources and reference voltages, and manage interrupts for the ADC10 modules.

The ADC10 module provides the ability to convert analog signals into a digital value in respect to given reference voltages. The ADC10 can generate digital values from 0 to Vcc with an 8- or 10-bit resolution. It operates in 2 different sampling modes, and 4 different conversion modes. The smapling modes are extended sampling and pulse sampling, in extended sampling the sample/hold signal must stay high for the duration of sampling, while in pulse mode a sampling timer is setup to start on a rising edge of the sample/hold signal and sample for a specified amount of clock cycles. The 4 conversion modes are single-channel single conversion, sequence of channels single-conversion, repeated single channel conversions, and repeated sequence of channels conversions.

The ADC10 module can generate multiple interrupts. An interrupt can be asserted when a conversion is complete, when a conversion is about to overwrite the converted data in the memory buffer before it has been read out, and/or when a conversion is about to start before the last conversion is complete. The ADC10 also has a window comparator feature which asserts interrupts when the input signal is above a high threshold, below a low threshold, or between the two at any given moment.

This driver is contained in `driverlib/5xx_6xx/adc10.c`, with `driverlib/5xx_6xx/adc10.h` containing the API definitions for use by applications.

3.2 API Functions

The ADC10 API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxillary features of the ADC10.

The ADC10 initialization and conversion functions are

- ADC10_init
- ADC10_memoryConfigure
- ADC10_setupSamplingTimer
- ADC10_disableSamplingTimer
- ADC10_setWindowComp
- ADC10_startConversion
- ADC10_disableConversions
- ADC10_getResults
- ADC10_isBusy

The ADC10 interrupts are handled by

- ADC10_enableInterrupt
- ADC10_disableInterrupt
- ADC10_clearInterrupt
- ADC10_getInterruptStatus

Auxilary features of the ADC10 are handled by

- ADC10_setResolution

- ADC10_setSampleHoldSignalInversion
- ADC10_setDataReadBackFormat
- ADC10_enableReferenceBurst
- ADC10_disableReferenceBurst
- ADC10_setReferenceBufferSamplingRate
- ADC10_getMemoryAddressForDMA
- ADC10_enable
- ADC10_disable

3.3 Programming Example

The following example shows how to initialize and use the ADC10 API to start a single channel, single conversion.

```
        // Initialize ADC10 with ADC10's built-in oscillator
        ADC10_init (__MSP430_BASEADDRESS_ADC10__,
                                ADC10_SAMPLEHOLDSOURCE_SC,
                                ADC10_CLOCKSOURCE_ADC10OSC,
                                ADC10_CLOCKDIVIDEBY_1);

        //Switch ON ADC10
    ADC10_enable(__MSP430_BASEADDRESS_ADC10__);

        // Setup sampling timer to sample-and-hold for 16 clock cycles
        ADC10_setupSamplingTimer (__MSP430_BASEADDRESS_ADC10__,
                                                        ADC10_CYCLEHOLD_16_CYCLES,
                                                        FALSE);

        // Configure the Input to the Memory Buffer with the specified Reference Voltages
        ADC10_memoryConfigure (__MSP430_BASEADDRESS_ADC10__,
                                                ADC10_INPUT_A0,
                                                ADC10_VREF_AVCC, // Vref+ = AVcc
                                                ADC10_VREF_AVSS  // Vref- = AVss
                                                );
        while (1)
        {
                // Start a single conversion, no repeating or sequences.
                ADC10_startConversion (__MSP430_BASEADDRESS_ADC10__,
                                                        ADC10_SINGLECHANNEL);

                // Wait for the Interrupt Flag to assert
                while( !(ADC10_getInterruptStatus(__MSP430_BASEADDRESS_ADC10__,ADC10IFG0)) );

                // Clear the Interrupt Flag and start another conversion
                ADC10_clearInterrupt(__MSP430_BASEADDRESS_ADC10__,ADC10IFG0);
        }
```

4 10-Bit Analog-to-Digital Converter (ADC10B)

4.1 Introduction

The 10-Bit Analog-to-Digital (ADC10B) API provides a set of functions for using the MSP430Ware ADC10B modules. Functions are provided to initializae the ADC10B modules, setup signal sources and reference voltages, and manage interrupts for the ADC10B modules.

The ADC10B module supports fast 10-bit analog-to-digital conversions. The module implements a 10-bit SAR core together, sample select control and a window comparator.

ADC10B features include:

- Greater than 200-ksps maximum conversion rate
- Monotonic 10-bit converter with no missing codes
- Sample-and-hold with programmable sampling periods controlled by software or timers
- Conversion initiation by software or different timers
- Software-selectable on chip reference using the REF module or external reference
- Twelve individually configurable external input channels
- Conversion channel for temperature sensor of the REF module
- Selectable conversion clock source
- Single-channel, repeat-single-channel, sequence, and repeat-sequence conversion modes
- Window comparator for low-power monitoring of input signals
- Interrupt vector register for fast decoding of six ADC interrupts (ADC10IFG0, ADC10TOVIFG, ADC10OVIFG, ADC10LOIFG, ADC10INIFG, ADC10HIIFG)

This driver is contained in `driverlib/5xx_6xx/adc10b.c`, with `driverlib/5xx_6xx/adc10b.h` containing the API definitions for use by applications.

4.2 API Functions

The ADC10B API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxillary features of the ADC10.

The ADC10B initialization and conversion functions are

- ADC10B_init
- ADC10B_memoryConfigure

- ADC10B_setupSamplingTimer
- ADC10B_disableSamplingTimer
- ADC10B_setWindowComp
- ADC10B_startConversion
- ADC10B_disableConversions
- ADC10B_getResults
- ADC10B_isBusy

The ADC10B interrupts are handled by

- ADC10B_enableInterrupt
- ADC10B_disableInterrupt
- ADC10B_clearInterrupt
- ADC10B_getInterruptStatus

Auxilary features of the ADC10B are handled by

- ADC10B_setResolution
- ADC10B_setSampleHoldSignalInversion
- ADC10B_setDataReadBackFormat
- ADC10B_enableReferenceBurst
- ADC10B_disableReferenceBurst
- ADC10B_setReferenceBufferSamplingRate
- ADC10B_getMemoryAddressForDMA
- ADC10B_enable
- ADC10B_disable

4.3 Programming Example

The following example shows how to initialize and use the ADC10B API to start a single channel, single conversion.

```
    // Initialize ADC10B with ADC10B's built-in oscillator
    ADC10B_init (__MSP430_BASEADDRESS_ADC10_B__,
                          ADC10B_SAMPLEHOLDSOURCE_SC,
                          ADC10B_CLOCKSOURCE_ADC10OSC,
                          ADC10B_CLOCKDIVIDEBY_1);

    //Switch ON ADC10B
ADC10B_enable(__MSP430_BASEADDRESS_ADC10_B__);

    // Setup sampling timer to sample-and-hold for 16 clock cycles
    ADC10B_setupSamplingTimer (__MSP430_BASEADDRESS_ADC10_B__,
                                                ADC10B_CYCLEHOLD_16_CYCLES,
                                                FALSE);

    // Configure the Input to the Memory Buffer with the specified Reference Voltages
    ADC10B_memoryConfigure (__MSP430_BASEADDRESS_ADC10_B__,
                                            ADC10B_INPUT_A0,
```

```
                                                ADC10B_VREFPOS_AVCC, // Vref+ = AVcc
                                                ADC10B_VREFNEG_AVSS  // Vref- = AVss
                                                );
        while (1)
        {
                // Start a single conversion, no repeating or sequences.
                ADC10B_startConversion (__MSP430_BASEADDRESS_ADC10_B__,
                                                        ADC10B_SINGLECHANNEL);

                // Wait for the Interrupt Flag to assert
                while( !(ADC10B_getInterruptStatus(__MSP430_BASEADDRESS_ADC10_B__,ADC10IFG0)) );

                // Clear the Interrupt Flag and start another conversion
                ADC10B_clearInterrupt(__MSP430_BASEADDRESS_ADC10_B__,ADC10IFG0);
        }
```

5 12-Bit Analog-to-Digital Converter (ADC12)

5.1 Introduction

The 12-Bit Analog-to-Digital (ADC12) API provides a set of functions for using the MSP430Ware ADC12 modules. Functions are provided to initializae the ADC12 modules, setup signal sources and reference voltages for each memory buffer, and manage interrupts for the ADC12 modules.

The ADC12 module provides the ability to convert analog signals into a digital value in respect to given reference voltages. The ADC12 can generate digital values from 0 to Vcc with an 8-, 10- or 12-bit resolution, with 16 different memory buffers to store conversion results. It operates in 2 different sampling modes, and 4 different conversion modes. The sampling modes are extended sampling and pulse sampling, in extended sampling the sample/hold signal must stay high for the duration of sampling, while in pulse mode a sampling timer is setup to start on a rising edge of the sample/hold signal and sample for a specified amount of clock cycles. The 4 conversion modes are single-channel single conversion, sequence of channels single-conversion, repeated single channel conversions, and repeated sequence of channels conversions.

The ADC12 module can generate multiple interrupts. An interrupt can be asserted for each memory buffer when a conversion is complete, or when a conversion is about to overwrite the converted data in any of the memory buffers before it has been read out, and/or when a conversion is about to start before the last conversion is complete.

This driver is contained in `driverlib/5xx_6xx/adc12.c`, with `driverlib/5xx_6xx/adc12.h` containing the API definitions for use by applications.

5.2 API Functions

The ADC12 API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxillary features of the ADC12.

The ADC12 initialization and conversion functions are

- ADC12_init
- ADC12_memoryConfigure
- ADC12_setupSamplingTimer
- ADC12_disableSamplingTimer
- ADC12_startConversion
- ADC12_disableConversions
- ADC12_readResults
- ADC12_isBusy

The ADC12 interrupts are handled by

- ADC12_enableInterrupt
- ADC12_disableInterrupt
- ADC12_clearInterrupt
- ADC12_getInterruptStatus

Auxilary features of the ADC12 are handled by

- ADC12_setResolution
- ADC12_setSampleHoldSignalInversion
- ADC12_setDataReadBackFormat
- ADC12_enableReferenceBurst
- ADC12_disableReferenceBurst
- ADC12_setReferenceBufferSamplingRate
- ADC12_getMemoryAddressForDMA
- ADC12_enable
- ADC12_disable

5.3 Programming Example

The following example shows how to initialize and use the ADC12 API to start a single channel, single conversion.

```
    // Initialize ADC12 with ADC12's built-in oscillator
    ADC12_init (__MSP430_BASEADDRESS_ADC12__,
                          ADC12_SAMPLEHOLDSOURCE_SC,
                          ADC12_CLOCKSOURCE_ADC12OSC,
                          ADC12_CLOCKDIVIDEBY_1);

//Switch ON ADC12
ADC12_enable(__MSP430_BASEADDRESS_ADC12__);

    // Setup sampling timer to sample-and-hold for 16 clock cycles
    ADC12_setupSamplingTimer (__MSP430_BASEADDRESS_ADC12__,
                                                    ADC12_CYCLEHOLD_64_CYCLES,
                                                    ADC12_CYCLEHOLD_4_CYCLES,
                                                    FALSE);

    // Configure the Input to the Memory Buffer with the specified Reference Voltages
    ADC12_memoryConfigure (__MSP430_BASEADDRESS_ADC12__,
                           ADC12_MEMORY_0,
                                                ADC12_INPUT_A0,
                                                ADC12_VREF_AVCC, // Vref+ = AVcc
                                                ADC12_VREF_AVSS, // Vref- = AVss
                                                FALSE
                                                );
    while (1)
    {
            // Start a single conversion, no repeating or sequences.
            ADC12_startConversion (__MSP430_BASEADDRESS_ADC12__,
                                   ADC12_MEMORY_0,
                                                        ADC12_SINGLECHANNEL);

            // Wait for the Interrupt Flag to assert
            while( !(ADC12_getInterruptStatus(__MSP430_BASEADDRESS_ADC12__,ADC12IFG0)) );
```

```
        // Clear the Interrupt Flag and start another conversion
        ADC12_clearInterrupt(__MSP430_BASEADDRESS_ADC12__,ADC12IFG0);
}
```

6 12-Bit Analog-to-Digital Converter (ADC12B)

6.1 Introduction

The 12-Bit Analog-to-Digital (ADC12B) API provides a set of functions for using the MSP430Ware ADC12B modules. Functions are provided to initializae the ADC12B modules, setup signal sources and reference voltages for each memory buffer, and manage interrupts for the ADC12B modules.

The ADC12B module provides the ability to convert analog signals into a digital value in respect to given reference voltages. The module implements a 12-bit SAR core, sample select control, and up to 32 independent conversion-and-control buffers. The conversion-and-control buffer allows up to 32 independent analog-to-digital converter (ADC) samples to be converted and stored without any CPU intervention. The ADC12B can also generate digital values from 0 to Vcc with an 8-, 10- or 12-bit resolution and it can operate in 2 different sampling modes, and 4 different conversion modes. The sampling modes are extended sampling and pulse sampling, in extended sampling the sample/hold signal must stay high for the duration of sampling, while in pulse mode a sampling timer is setup to start on a rising edge of the sample/hold signal and sample for a specified amount of clock cycles. The 4 conversion modes are single-channel single conversion, sequence of channels single-conversion, repeated single channel conversions, and repeated sequence of channels conversions.

The ADC12B module can generate multiple interrupts. An interrupt can be asserted for each memory buffer when a conversion is complete, or when a conversion is about to overwrite the converted data in any of the memory buffers before it has been read out, and/or when a conversion is about to start before the last conversion is complete.

ADC12_B features include:

- 200 ksps maximum conversion rate at maximum resolution of 12-bits
- Monotonic 12-bit converter with no missing codes
- Sample-and-hold with programmable sampling periods controlled by software or timers.
- Conversion initiation by software or timers.
- Software-selectable on-chip reference voltage generation (1.2 V, 2.0 V, or 2.5 V) with option to make available externally
- Software-selectable internal or external reference
- Up to 32 individually configurable external input channels, single-ended or differential input selection available
- Internal conversion channels for internal temperature sensor and 2/3 $\times$ AVCC and four more internal channels available on select devices see device data sheet for availability as well as function
- Independent channel-selectable reference sources for both positive and negative references
- Selectable conversion clock source

- Single-channel, repeat-single-channel, sequence (autoscan), and repeat-sequence (repeated autoscan) conversion modes
- Interrupt vector register for fast decoding of 38 ADC interrupts
- 32 conversion-result storage registers
- Window comparator for low power monitoring of input signals of conversion-result registers

This driver is contained in `driverlib/5xx_6xx/ADC12B.c`, with `driverlib/5xx_6xx/ADC12B.h` containing the API definitions for use by applications.

6.2 API Functions

The ADC12B API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxillary features of the ADC12B.

The ADC12B initialization and conversion functions are

- ADC12B_init
- ADC12B_memoryConfigure
- ADC12B_setupSamplingTimer
- ADC12B_disableSamplingTimer
- ADC12B_startConversion
- ADC12B_disableConversions
- ADC12B_getResults
- ADC12B_isBusy

The ADC12B interrupts are handled by

- ADC12B_enableInterrupt
- ADC12B_disableInterrupt
- ADC12B_clearInterrupt
- ADC12B_getInterruptStatus

Auxilary features of the ADC12B are handled by

- ADC12B_setResolution
- ADC12B_setSampleHoldSignalInversion
- ADC12B_setDataReadBackFormat
- ADC12B_enableReferenceBurst
- ADC12B_disableReferenceBurst
- ADC12B_setAdcPowerMode
- ADC12B_getMemoryAddressForDMA
- ADC12B_enable
- ADC12B_disable

6.3 Programming Example

The following example shows how to initialize and use the ADC12B API to start a single channel with single conversion using an external positive reference for the ADC12B.

```
 //Initialize the ADC12 Module
 /*
Base address of ADC12 Module
Use internal ADC12 bit as sample/hold signal to start conversion
USE MODOSC 5MHZ Digital Oscillator as clock source
Use default clock divider/pre-divider of 1
Map to internal channel 0
  */
 ADC12B_init(__MSP430_BASEADDRESS_ADC12_B__,
               ADC12B_SAMPLEHOLDSOURCE_SC,
               ADC12B_CLOCKSOURCE_ADC12OSC,
               ADC12B_CLOCKDIVIDER_1,
               ADC12B_CLOCKPREDIVIDER__1,
               ADC12B_MAPINTCH0);

 //Enable the ADC12B module
 ADC12B_enable(__MSP430_BASEADDRESS_ADC12_B__);

 /*
Base address of ADC12 Module
For memory buffers 0-7 sample/hold for 16 clock cycles
For memory buffers 8-15 sample/hold for 4 clock cycles (default)
Disable Multiple Sampling
  */
 ADC12B_setupSamplingTimer(__MSP430_BASEADDRESS_ADC12_B__,
     ADC12B_CYCLEHOLD_16_CYCLES,
     ADC12B_CYCLEHOLD_4_CYCLES,
     ADC12B_MULTIPLESAMPLESDISABLE);

 //Configure Memory Buffer
 /*
Base address of the ADC12 Module
Configure memory buffer 0
Map input A0 to memory buffer 0
Vref+ = AVcc
Vref- = EXT Positive
Memory buffer 0 is not the end of a sequence
  */
 ADC12B_memoryConfigure(__MSP430_BASEADDRESS_ADC12_B__,
     ADC12B_MEMORY_0,
     ADC12B_INPUT_A0,
     ADC12B_VREFPOS_EXTPOS_VREFNEG_VSS,
     ADC12B_NOTENDOFSEQUENCE);

 while (1)
 {
   //Enable/Start first sampling and conversion cycle
   /*
Base address of ADC12 Module
Start the conversion into memory buffer 0
Use the single-channel, single-conversion mode
    */
       ADC12B_startConversion(__MSP430_BASEADDRESS_ADC12_B__,
               ADC12B_MEMORY_0,
               ADC12B_SINGLECHANNEL);

   //Poll for interrupt on memory buffer 0
   while (!ADC12B_getInterruptStatus(__MSP430_BASEADDRESS_ADC12_B__,
              0,
```

```
                    ADC12B_IFG0));

  __no_operation();                         // SET BREAKPOINT HERE
}
```

7 Advanced Encryption Standard (AES)

7.1 Introduction

The AES accelerator module performs encryption and decryption of 128-bit data with 128-bit keys according to the advanced encryption standard (AES) (FIPS PUB 197) in hardware. The AES accelerator features are:

- Encryption and decryption according to AES FIPS PUB 197 with 128-bit key
- On-the-fly key expansion for encryption and decryption
- Off-line key generation for decryption
- Byte and word access to key, input, and output data
- AES ready interrupt flag The AES256 accelerator module performs encryption and decryption of 128-bit data with 128-/192-/256-bit keys according to the advanced encryption standard (AES) (FIPS PUB 197) in hardware. The AES accelerator features are: AES encryption Ű 128 bit - 168 cycles Ű 192 bit - 204 cycles Ű 256 bit - 234 cycles AES decryption Ű 128 bit - 168 cycles Ű 192 bit - 206 cycles Ű 256 bit - 234 cycles
- On-the-fly key expansion for encryption and decryption
- Offline key generation for decryption
- Shadow register storing the initial key for all key lengths
- Byte and word access to key, input data, and output data
- AES ready interrupt flag

This driver is contained in `driverlib/5xx_6xx/aes.c`, with `driverlib/5xx_6xx/aes.h` containing the API definitions for use by applications.

7.2 API Functions

The AES module APIs are

- AES_setCipherKey(),
- AES256_setCipherKey(),
- AES_encryptData(),
- AES_decryptDataUsingEncryptionKey(),
- AES_generateFirstRoundKey(),
- AES_decryptData(),
- AES_reset(),
- AES_startEncryptData(),

- AES_startDecryptDataUsingEncryptionKey(),
- AES_startDecryptData(),
- AES_startGenerateFirstRoundKey(),
- AES_getDataOut()

The AES interrupt handler functions

- AES_enableInterrupt(),
- AES_disableInterrupt(),
- AES_clearInterruptFlag(),

7.3 Programming Example

The following example shows some AES operations using the APIs

```
unsigned char Data[16] =                {       0x30, 0x31, 0x32, 0x33,
                                                                        0x34, 0x35, 0x36, 0x37,
                                                                        0x38, 0x39, 0x0A, 0x0B,
                                                                        0x0C, 0x0D, 0x0E, 0x0F
unsigned char CipherKey[16] =   {       0xAA, 0xBB, 0x02, 0x03,
                                                                        0x04, 0x05, 0x06, 0x07,
                                                                        0x08, 0x09, 0x0A, 0x0B,
                                                                        0x0C, 0x0D, 0x0E, 0x0F
unsigned char DataAES[16];              // Encrypted data
unsigned char DataunAES[16];    // Decrypted data

// Load a cipher key to module
AES_setCipherKey(__MSP430_BASEADDRESS_AES__, CipherKey);

// Encrypt data with preloaded cipher key
AES_encryptData(__MSP430_BASEADDRESS_AES__, Data, DataAES);

// Decrypt data with keys that were generated during encryption - takes 214 MCLK
// This function will generate all round keys needed for decryption first and then
// the encryption process starts
AES_decryptDataUsingEncryptionKey(__MSP430_BASEADDRESS_AES__, DataAES, DataunAES);
```

8 Comparator (COMPB)

8.1 Introduction

The Comparator B (COMPB) API provides a set of functions for using the MSP430Ware COMPB modules. Functions are provided to initialize the COMPB modules, setup reference voltages for input, and manage interrupts for the COMPB modules.

The COMPB module provides the ability to compare two analog signals and use the output in software and on an output pin. The output represents whether the signal on the positive terminal is higher than the signal on the negative terminal. The COMPB may be used to generate a hysteresis. There are 16 different inputs that can be used, as well as the ability to short 2 input together. The COMPB module also has control over the REF module to generate a reference voltage as an input.

The COMPB module can generate multiple interrupts. An interrupt may be asserted for the output, with seperate interrupts on whether the output rises, or falls.

This driver is contained in `driverlib/5xx_6xx/compb.c`, with `driverlib/5xx_6xx/compb.h` containing the API definitions for use by applications.

8.2 API Functions

The COMPB API is broken into three groups of functions: those that deal with initialization and output, those that handle interrupts, and those that handle auxillary features of the COMPB.

The COMPB initialization and output functions are

- COMPB_init
- COMPB_setReferenceVoltage
- COMPB_enable
- COMPB_disable
- COMPB_outputValue

The COMPB interrupts are handled by

- COMPB_enableInterrupt
- COMPB_disableInterrupt
- COMPB_clearInterrupt
- COMPB_getInterruptStatus
- COMPB_interruptSetEdgeDirection
- COMPB_interruptToggleEdgeDirection

Auxilary features of the COMPB are handled by

- COMPB_enableShortOfInputs
- COMPB_disableShortOfInputs
- COMPB_disableInputBuffer
- COMPB_enableInputBuffer
- COMPB_IOSwap

8.3 Programming Example

The following example shows how to initialize and use the COMPB API to turn on an LED when the input to the positive terminal is highed than the input to the negative terminal.

```
// Initialize the Comparator B module
/* Base Address of Comparator B,
  Pin CB0 to Positive(+) Terminal,
  Reference Voltage to Negative(-) Terminal,
  Normal Power Mode,
  Output Filter On with minimal delay,
  Non-Inverted Output Polarity
*/
COMPB_init(__MSP430_BASEADDRESS_COMPB__,
          COMPB_INPUT0,
          COMPB_VREF,
          COMPB_POWERMODE_NORMALMODE,
          COMPB_FILTEROUTPUT_DLYLVL1,
          COMPB_NORMALOUTPUTPOLARITY
          );

// Set the reference voltage that is being supplied to the (-) terminal
/* Base Address of Comparator B,
 Reference Voltage of 2.0 V,
 Upper Limit of 2.0*(32/32) = 2.0V,
 Lower Limit of 2.0*(32/32) = 2.0V
*/
COMPB_setReferenceVoltage(__MSP430_BASEADDRESS_COMPB__,
                          COMPB_VREFBASE2_5V,
                          32,
                          32
                          );

// Allow power to Comparator module
COMPB_enable(__MSP430_BASEADDRESS_COMPB__);

// delay for the reference to settle
__delay_cycles(75);
```

9 Comparator (COMPD)

9.1 Introduction

The Comparator D (COMPD) API provides a set of functions for using the MSP430Ware COMPD modules. Functions are provided to initialize the COMPD modules, setup reference voltages for input, and manage interrupts for the COMPD modules.

The COMPD module provides the ability to compare two analog signals and use the output in software and on an output pin. The output represents whether the signal on the positive terminal is higher than the signal on the negative terminal. The COMPD may be used to generate a hysteresis. There are 16 different inputs that can be used, as well as the ability to short 2 input together. The COMPD module also has control over the REF module to generate a reference voltage as an input.

The COMPD module can generate multiple interrupts. An interrupt may be asserted for the output, with seperate interrupts on whether the output rises, or falls.

This driver is contained in `driverlib/5xx_6xx/compd.c`, with `driverlib/5xx_6xx/compd.h` containing the API definitions for use by applications.

9.2 API Functions

The COMPD API is broken into three groups of functions: those that deal with initialization and output, those that handle interrupts, and those that handle auxillary features of the COMPD.

The COMPD initialization and output functions are

- COMPD_init
- COMPD_setReferenceVoltage
- COMPD_enable
- COMPD_disable
- COMPD_outputValue

The COMPD interrupts are handled by

- COMPD_enableInterrupt
- COMPD_disableInterrupt
- COMPD_clearInterrupt
- COMPD_getInterruptStatus
- COMPD_interruptSetEdgeDirection
- COMPD_interruptToggleEdgeDirection

Auxilary features of the COMPD are handled by

- COMPD_enableShortOfInputs
- COMPD_disableShortOfInputs
- COMPD_disableInputBuffer
- COMPD_enableInputBuffer
- COMPD_IOSwap

9.3 Programming Example

The following example shows how to initialize and use the COMPD API to turn on an LED when the input to the positive terminal is highed than the input to the negative terminal.

```
// Initialize the Comparator D module
/* Base Address of Comparator D,
  Pin CD2 to Positive(+) Terminal,
  Reference Voltage to Negative(-) Terminal,
  Normal Power Mode,
  Output Filter On with minimal delay,
  Non-Inverted Output Polarity
*/
COMPD_init(__MSP430_BASEADDRESS_COMPD__,
    COMPD_INPUT2,
    COMPD_VREF,
    COMPD_FILTEROUTPUT_OFF,
    COMPD_NORMALOUTPUTPOLARITY
    );

// Set the reference voltage that is being supplied to the (-) terminal
/* Base Address of Comparator D,
 Reference Voltage of 2.0 V,
 Upper Limit of 2.0*(32/32) = 2.0V,
 Lower Limit of 2.0*(32/32) = 2.0V
*/
COMPD_setReferenceVoltage(__MSP430_BASEADDRESS_COMPD__,
    COMPD_VREFBASE2_0V,
                          32,
                          32
                          );

  //Disable Input Buffer on P1.2/CD2
      /* Base Address of Comparator D,
Input Buffer port
Selecting the CDx input pin to the comparator
multiplexer with the CDx bits automatically
disables output driver and input buffer for
that pin, regardless of the state of the
associated CDPD.x bit
       */
        COMPD_disableInputBuffer(__MSP430_BASEADDRESS_COMPD__,
                COMPD_INPUT2);
// Allow power to Comparator module
COMPD_enable(__MSP430_BASEADDRESS_COMPD__);

__delay_cycles(400);           // delay for the reference to settle
```

10 Clock System (CS)

10.1 Introduction

The clock system module supports low system cost and low power consumption. Using three internal clock signals, the user can select the best balance of performance and low power consumption. The clock module can be configured to operate without any external components, with one or two external crystals,or with resonators, under full software control.

The clock system module includes up to five clock sources:

- XT1CLK - Low-frequency/high-frequency oscillator that can be used either with low-frequency 32768-Hz watch crystals, standard crystals, resonators, or external clock sources in the 4 MHz to 24 MHz range. When optional XT2 is present, the XT1 high-frequency mode may or may not be available, depending on the device configuration. See the device-specific data sheet for supported functions.
- VLOCLK - Internal very-low-power low-frequency oscillator with 10-kHz typical frequency
- DCOCLK - Internal digitally controlled oscillator (DCO) with three selectable fixed frequencies
- XT2CLK - Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 4 MHz to 24 MHz range. See device-specific data sheet for availability.

Four system clock signals are available from the clock module:

- ACLK - Auxiliary clock. The ACLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. ACLK can be divided by 1, 2, 4, 8, 16, or 32. ACLK is software selectable by individual peripheral modules.
- MCLK - Master clock. MCLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. MCLK can be divided by 1, 2, 4, 8, 16, or 32. MCLK is used by the CPU and system.
- SMCLK - Subsystem master clock. SMCLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. SMCLK is software selectable by individual peripheral modules.
- MODCLK - Module clock. MODCLK is used by various peripheral modules and is sourced by MODOSC.

Fail-Safe logic The crystal oscillator faults are set if the corresponding crystal oscillator is turned on and not operating properly. Once set, the fault bits remain set until reset in software, regardless if the fault condition no longer exists. If the user clears the fault bits and the fault condition still exists, the fault bits are automatically set, otherwise they remain cleared.

The OFIFG oscillator-fault interrupt flag is set and latched at POR or when any oscillator fault is detected. When OFIFG is set and OFIE is set, the OFIFG requests a user NMI. When the interrupt is granted, the OFIE is not reset automatically as it is in previous MSP430 families. It is no longer required to reset the OFIE. NMI entry/exit circuitry removes this requirement. The OFIFG flag must

be cleared by software. The source of the fault can be identified by checking the individual fault bits.

If XT1 in LF mode is sourcing any system clock (ACLK, MCLK, or SMCLK), and a fault is detected, the system clock is automatically switched to the VLO for its clock source (VLOCLK). Similarly, if XT1 in HF mode is sourcing any system clock and a fault is detected, the system clock is automatically switched to MODOSC for its clock source (MODCLK).

When XT2 (if available) is sourcing any system clock and a fault is detected, the system clock is automatically switched to MODOSC for its clock source (MODCLK).

The fail-safe logic does not change the respective SELA, SELM, and SELS bit settings. The fail-safe mechanism behaves the same in normal and bypass modes.

This driver is contained in `driverlib/5xx_6xx/cs.c`, with `driverlib/5xx_6xx/cs.h` containing the API definitions for use by applications.

10.2 API Functions

The CS API is broken into four groups of functions: an API that initializes the clock module, those that deal with clock configuration and control, and external crystal and bypass specific configuration and initialization, and those that handle interrupts.

General CS configuration and initialization are handled by the following API

- CS_clockSignalInit
- CS_enableClockRequest
- CS_disableClockRequest
- CS_getACLK
- CS_getSMCLK
- CS_getMCLK
- CS_setDCOFreq

The following external crystal and bypass specific configuration and initialization functions are available for FR57xx devices:

- CS_XT1Start
- CS_bypassXT1
- CS_bypassXT1WithTimeout
- CS_XT1StartWithTimeout
- CS_XT1Off
- CS_XT2Start
- CS_bypassXT2
- CS_XT2StartWithTimeout
- CS_bypassXT2WithTimeout
- CS_XT2Off

The CS interrupts are handled by

- CS_enableClockRequest
- CS_disableClockRequest
- CS_faultFlagStatus
- CS_clearFaultFlag
- CS_clearAllOscFlagsWithTimeout

CS_setExternalClockSource must be called if an external crystal XT1 or XT2 is used and the user intends to call CS_getMCLK, CS_getSMCLK or CS_getACLK APIs and XT1Start, XT1ByPass, XT1StartWithTimeout, XT1ByPassWithTimeout. If not any of the previous API are going to be called, it is not necessary to invoke this API.

10.3 Programming Example

The following example shows the configuration of the CS module that sets ACLK=SMCLK=MCLK=DCOCLK

```
//Set DCO Frequency to 8MHz
CS_setDCOFreq(__MSP430_BASEADDRESS_CS__,CS_DCORSEL_0,CS_DCOFSEL_3);

//configure MCLK, SMCLK and ACLK to be source by DCOCLK
CS_clockSignalInit(__MSP430_BASEADDRESS_CS__,CS_ACLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
CS_clockSignalInit(__MSP430_BASEADDRESS_CS__,CS_SMCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
CS_clockSignalInit(__MSP430_BASEADDRESS_CS__,CS_MCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
```

11 Clock System (CSA)

11.1 Introduction

The clock system module supports low system cost and low power consumption. Using three internal clock signals, the user can select the best balance of performance and low power consumption. The clock module can be configured to operate without any external components, with one or two external crystals,or with resonators, under full software control.

The clock system module includes the following clock sources:

- LFXTCLK - Low-frequency oscillator that can be used either with low-frequency 32768-Hz watch crystals, standard crystals, resonators, or external clock sources in the 50 kHz or below range. When in bypass mode, LFXTCLK can be driven with an external square wave signal.
- VLOCLK - Internal very-low-power low-frequency oscillator with 10-kHz typical frequency
- DCOCLK - Internal digitally controlled oscillator (DCO) with selectable frequencies
- MODCLK - Internal low-power oscillator with 5-MHz typical frequency. LFMODCLK is MODCLK divided by 128.
- HFXTCLK - High-frequency oscillator that can be used with standard crystals or resonators in the 4-MHz to 24-MHz range. When in bypass mode, HFXTCLK can be driven with an external square wave signal.

Four system clock signals are available from the clock module:

- ACLK - Auxiliary clock. The ACLK is software selectable as LFXTCLK, VLOCLK, or LFMODCLK. ACLK can be divided by 1, 2, 4, 8, 16, or 32. ACLK is software selectable by individual peripheral modules.
- MCLK - Master clock. MCLK is software selectable as LFXTCLK, VLOCLK, LFMODCLK, DCOCLK, MODCLK, or HFXTCLK. MCLK can be divided by 1, 2, 4, 8, 16, or 32. MCLK is used by the CPU and system.
- SMCLK - Sub-system master clock. SMCLK is software selectable as LFXTCLK, VLOCLK, LFMODCLK, DCOCLK, MODCLK, or HFXTCLK. SMCLK is software selectable by individual peripheral modules.
- MODCLK - Module clock. MODCLK may also be used by various peripheral modules and is sourced by MODOSC.
- VLOCLK - VLO clock. VLOCLK may also be used directly by various peripheral modules and is sourced by VLO.

Fail-Safe logic The crystal oscillator faults are set if the corresponding crystal oscillator is turned on and not operating properly. Once set, the fault bits remain set until reset in software, regardless if the fault condition no longer exists. If the user clears the fault bits and the fault condition still exists, the fault bits are automatically set, otherwise they remain cleared.

The OFIFG oscillator-fault interrupt flag is set and latched at POR or when any oscillator fault is detected. When OFIFG is set and OFIE is set, the OFIFG requests a user NMI. When the interrupt

is granted, the OFIE is not reset automatically as it is in previous MSP430 families. It is no longer required to reset the OFIE. NMI entry/exit circuitry removes this requirement. The OFIFG flag must be cleared by software. The source of the fault can be identified by checking the individual fault bits.

If LFXT is sourcing any system clock (ACLK, MCLK, or SMCLK) and a fault is detected, the system clock is automatically switched to LFMODCLK for its clock source. The LFXT fault logic works in all power modes, including LPM3.5.

If HFXT is sourcing MCLK or SMCLK, and a fault is detected, the system clock is automatically switched to MODCLK for its clock source. By default, the HFXT fault logic works in all power modes, except LPM3.5 or LPM4.5, because high-frequency operation in these modes is not supported.

The fail-safe logic does not change the respective SELA, SELM, and SELS bit settings. The fail-safe mechanism behaves the same in normal and bypass modes.

This driver is contained in `driverlib/5xx_6xx/cs_a.c`, with `driverlib/5xx_6xx/cs_a.h` containing the API definitions for use by applications.

11.2 API Functions

The CSA API is broken into four groups of functions: an API that initializes the clock module, those that deal with clock configuration and control, and external crystal and bypass specific configuration and initialization, and those that handle interrupts.

General CSA configuration and initialization are handled by the following API

- CSA_clockSignalInit
- CSA_enableClockRequest
- CSA_disableClockRequest
- CSA_getACLK
- CSA_getSMCLK
- CSA_getMCLK
- CSA_setDCOFreq

The following external crystal and bypass specific configuration and initialization functions are available

- CSA_LFXTStart
- CSA_bypassLFXT
- CSA_bypassLFXTWithTimeout
- CSA_LFXTStartWithTimeout
- CSA_LFXTOff
- CSA_HFXTStart
- CSA_bypassHFXT
- CSA_HFXTStartWithTimeout
- CSA_bypassHFXTWithTimeout
- CSA_HFXTOff

- CSA_VLOoff

The CSA interrupts are handled by

- CSA_enableClockRequest
- CSA_disableClockRequest
- CSA_faultFlagStatus
- CSA_clearFaultFlag
- CSA_clearAllOscFlagsWithTimeout

CSA_setExternalClockSource must be called if an external crystal LFXT or HFXT is used and the user intends to call CSA_getMCLK, CSA_getSMCLK or CSA_getACLK APIs and HFXTStart, HFXTByPass, HFXTStartWithTimeout, HFXTByPassWithTimeout. If not any of the previous API are going to be called, it is not necessary to invoke this API.

11.3 Programming Example

The following example shows the configuration of the CS module that sets SMCLK = MCLK = 8MHz

```
      //Set DCO Frequency to 8MHz
CSA_setDCOFreq(__MSP430_BASEADDRESS_CS_A__,CSA_DCORSEL_0,CSA_DCOFSEL_6);

//configure MCLK, SMCLK to be source by DCOCLK
CSA_clockSignalInit(__MSP430_BASEADDRESS_CS_A__,CSA_SMCLK,CSA_DCOCLK_SELECT,CSA_CLOCK_DIVIDER_1);
CSA_clockSignalInit(__MSP430_BASEADDRESS_CS_A__,CSA_MCLK,CSA_DCOCLK_SELECT,CSA_CLOCK_DIVIDER_1);
```

12 Cyclical Redundancy Check (CRC)

12.1 Introduction

The Cyclic Redundancy Check (CRC) API provides a set of functions for using the MSP430Ware CRC module. Functions are provided to initialize the CRC and create a CRC signature to check the validity of data. This is mostly useful in the communication of data, or as a startup procedure to as a more complex and accurate check of data.

The CRC module offers no interrupts and is used only to generate CRC signatures to verify against pre-made CRC signatures (Checksums).

This driver is contained in `driverlib/5xx_6xx/crc.c`, with `driverlib/5xx_6xx/crc.h` containing the API definitions for use by applications.

12.2 API Functions

The CRC API is one group that controls the CRC module.

- CRC_setSeed
- CRC_setData
- CRC_setSignatureByteReversed
- CRC_getSignature
- CRC_getResult
- CRC_getResultBitReversed

12.3 Programming Example

The following example shows how to initialize and use the CRC API to generate a CRC signature on an array of data that can be included in a UART message with the data to check for validity.

```
unsigned int crcSeed = 0xBEEF;
unsigned int data[] = {0x0123,
                       0x4567,
                       0x8910,
                       0x1112,
                       0x1314};
unsigned int crcResult;
int i;

// Stop WDT
WDT_hold(__MSP430_BASEADDRESS_WDT_A__);
```

```
// Set P1.0 as an output
GPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
                    GPIO_PORT_P1,
                    GPIO_PIN0);

// Set the CRC seed
CRC_setSeed(__MSP430_BASEADDRESS_CRC__,
            crcSeed);

for(i=0; i<5; i++)
{
  // Add all of the values into the CRC signature
  CRC_setData(__MSP430_BASEADDRESS_CRC__,
                   data[i]);
}

// Save the current CRC signature checksum to be compared for later
crcResult = CRC_getResult(__MSP430_BASEADDRESS_CRC__);
```

13 12-bit Digital-to-Analog Converter (DAC12)

13.1 Introduction

The 12-Bit Digital-to-Analog (DAC12) API provides a set of functions for using the MSP430Ware DAC12 modules. Functions are [rpvided to initialize setup the DAC12 modules, calibrate the output signal, and manage the interrupts for the DAC12 modules.

The DAC12 module provides the ability to convert digital values into an analog signal for output to a pin. The DAC12 can generate signals from 0 to Vcc from an 8- or 12-bit value. There can be one or two DAC12 modules in a device, and if there are two they can be grouped together to create two analog signals in simultaneously. There are 3 ways to latch data in to the DAC module, and those are by software with the startConversion API function call, as well as by the Timer A output of CCR1 or Timer B output of CCR2.

The calibration API will unlock and start calibration, then wait for the calibration to end before locking it back up, all in one API. There are also functions to read out the calibration data, as well as be able to set it manually.

The DAC12 module can generate one interrupt for each DAC module. It will generate the interrupt when the data has been latched into the DAC module to be output into an analog signal.

This driver is contained in `driverlib/5xx_6xx/dac12.c`, with `driverlib/5xx_6xx/dac12.h` containing the API definitions for use by applications.

13.2 API Functions

The DAC12 API is broken into three groups of functions: those that deal with initialization and conversions, those that deal with calibration of the output, and those that handle interrupts.

The DAC12 initialization and conversion functions are

- DAC12_init
- DAC12_setAmplifierSetting
- DAC12_disable
- DAC12_enableGrouping
- DAC12_disableGrouping
- DAC12_enableConversions
- DAC12_setData
- DAC12_disableConversions
- DAC12_setResolution
- DAC12_setInputDataFormat
- DAC12_getDataBufferMemoryAddressForDMA

Calibration features of the DAC12 are handled by

- DAC12_calibrateOutput
- DAC12_getCalibrationData
- DAC12_setCalibrationOffset

The DAC12 interrupts are handled by

- DAC12_enableInterrupt
- DAC12_disableInterrupt
- DAC12_getInterruptStatus
- DAC12_clearInterrupt

13.3 Programming Example

The following example shows how to initialize and use the DAC12 API to output a 1.5V analog signal.

```
DAC12_init (__MSP430_BASEADDRESS_DAC12_2__,
            DAC12_SUBMODULE_0,            // Initialize DAC12_0
            DAC12_OUTPUT_1,               // Choose P6.6 as output
            DAC12_VREF_AVCC,              // Use AVcc as Vref+
            DAC12_VREFx1,                 // Multiply Vout by 1
            DAC12_AMP_MEDIN_MEDOUT,       // Use medium settling speed/current
            DAC12_TRIGGER_ENCBYPASS       // Auto trigger as soon as data is set
            );

// Calibrate output buffer for DAC12_0
DAC12_calibrateOutput(__MSP430_BASEADDRESS_DAC12_2__,
                      DAC12_SUBMODULE_0);

DAC12_setData(__MSP430_BASEADDRESS_DAC12_2__,
              DAC12_SUBMODULE_0,          // Set 0x7FF (~1.5V)
              0x7FF                        // into data buffer for DAC12_0
              );
```

14 Direct Memory Access (DMA)

14.1 Introduction

The Direct Memory Access (DMA) API provides a set of functions for using the MSP430Ware DMA modules. Functions are provided to initialize and setup each DMA channel with the source and destination addresses, manage the interrupts for each channel, and set bits that affect all DMA channels.

The DMA module provides the ability to move data from one address in the device to another, and that includes other peripheral addresses to RAM or vice-versa, all without the actual use of the CPU. Please be advised, that the DMA module does halt the CPU for 2 cycles while transfering, but does not have to edit any registers or anything. The DMA can transfer by bytes or words at a time, and will automatically increment or decrement the source or destination address if desired. There are also 6 different modes to transfer by, including single-transfer, block-transfer, and burst-block-transfer, as well as repeated versions of those three different kinds which allows transfers to be repeated without having re-enable transfers.

The DMA settings that affect all DMA channels include prioritization, from a fixed priority to dynamic round-robin priority. Another setting that can be changed is when transfers occur, the CPU may be in a read-modify-write operation which can be disasterous to time sensitive material, so this can be disabled. And Non-Maskable-Interrupts can indeed be maskable to the DMA module if not enabled.

The DMA module can generate one interrupt per channel. The interrupt is only asserted when the specified amount of transfers has been completed. With single-transfer, this occurs when that many single transfers have occured, while with block or burst-block transfers, once the block is completely transfered the interrupt is asserted.

14.2 API Functions

The DMA API is broken into three groups of functions: those that deal with initialization and transfers, those that handle interrupts, and those that affect all DMA channels.

The DMA initialization and transfer functions are: DMA_init DMA_setSrcAddress DMA_setDstAddress DMA_enableTransfers DMA_disableTransfers DMA_startTransfer DMA_setTransferSize

The DMA interrupts are handled by: DMA_enableInterrupt DMA_disableInterrupt DMA_getInterruptStatus DMA_clearInterrupt DMA_NMIAbortStatus DMA_clearNMIAbort

Features of the DMA that affect all channels are handled by: DMA_disableTransferDuringReadModifyWrite DMA_enableTransferDuringReadModifyWrite DMA_enableRoundRobinPriority DMA_disableRoundRobinPriority DMA_enableNMIAbort DMA_disableNMIAbort

14.3 Programming Example

The following example shows how to initialize and use the DMA API to transfer words from one spot in RAM to another.

```
 // Initialize and Setup DMA Channel 0
 /*
Base Address of the DMA Module
Configure DMA channel 0
Configure channel for repeated block transfers
DMA interrupt flag will be set after every 16 transfers
Use DMA_startTransfer() function to trigger transfers
Transfer Word-to-Word
Trigger upon Rising Edge of Trigger Source Signal
  */
 DMA_init(__MSP430_BASEADDRESS_DMAX_3__,
          DMA_CHANNEL_0,
          DMA_TRANSFER_REPEATED_BLOCK,
          16,
          DMA_TRIGGERSOURCE_0,
          DMA_SIZE_SRCWORD_DSTWORD,
          DMA_TRIGGER_RISINGEDGE);
 /*
Base Address of the DMA Module
Configure DMA channel 0
Use 0x1C00 as source
Increment source address after every transfer
  */
 DMA_setSrcAddress(__MSP430_BASEADDRESS_DMAX_3__,
                   DMA_CHANNEL_0,
                   0x1C00,
                   DMA_DIRECTION_INCREMENT);
 /*
Base Address of the DMA Module
Configure DMA channel 0
Use 0x1C20 as destination
Increment destination address after every transfer
  */
 DMA_setDstAddress(__MSP430_BASEADDRESS_DMAX_3__,
                   DMA_CHANNEL_0,
                   0x1C20,
                   DMA_DIRECTION_INCREMENT);

 // Enable transfers on DMA channel 0
 DMA_enableTransfers(__MSP430_BASEADDRESS_DMAX_3__,
                     DMA_CHANNEL_0);

 while(1)
 {
   // Start block tranfer on DMA channel 0
   DMA_startTransfer(__MSP430_BASEADDRESS_DMAX_3__,
                     DMA_CHANNEL_0);
 }
```

15 EUSCI Inter-Integrated Circuit (I2C)

15.1 Introduction

In I2C mode, the eUSCI_B module provides an interface between the device and I2C-compatible devices connected by the two-wire I2C serial bus. External components attached to the I2C bus serially transmit and/or receive serial data to/from the eUSCI_B module through the 2-wire I2C interface. The Inter-Integrated Circuit (I2C) API provides a set of functions for using the MSP430Ware I2C modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules.

The I2C module provide the ability to communicate to other IC devices over an I2C bus. The I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the I2C bus can be designated as either a master or a slave. The MSP430Ware I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave.

I2C module can generate interrupts. The I2C module configured as a master will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The I2C module configured as a slave will generate interrupts when data has been sent or requested by a master.

15.1.1 Master Operations

To drive the master module, the APIs need to be invoked in the following order

- **eI2C_masterInit**
- **eI2C_setSlaveAddress**
- **eI2C_setMode**
- **eI2C_enable**
- **eI2C_enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first initialize the I2C module and configure it as a master with a call to eI2C_masterInit(). That function will set the clock and data rates. This is followed by a call to set the slave address with which the master intends to communicate with using eI2C_setSlaveAddress. Then the mode of operation (transmit or receieve) is chosen using eI2C_setMode. The I2C module may now be enabled using eI2C_enable. It is recommneded to enable the eI2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Master Single Byte Trasnmission

- eI2C_masterSendSingleByte

Master Mulitple Byte Transmission

- eI2C_masterMultiByteSendStart
- eI2C_masterMultiByteSendNext
- eI2C_masterMultiByteSendStop

Master Single Byte Reception

- eI2C_masterReceiveStart
- eI2C_masterSingleReceive

Master Multiple Byte Reception

- eI2C_masterMultiByteReceiveStart
- eI2C_masterMultiByteReceiveNext
- eI2C_masterMultiByteReceiveFinish
- eI2C_masterMultiByteReceiveStop

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

15.1.2 Slave Operations

To drive the slave module, the APIs need to be invoked in the following order

- **eI2C_slaveInit**
- **eI2C_setMode**
- **eI2C_enable**
- **eI2C_enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first call the eI2C_slaveInit to initialize the slave module in I2C mode and set the slave address. This is followed by a call to set the mode of operation (transmit or receive).The I2C module may now be enabled using eI2C_enable. It is recommneded to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Slave Transmission API

- eI2C_slaveDataPut

Slave Reception API

- eI2C_slaveDataGet

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

This driver is contained in `driverlib/5xx_6xx/ei2c.c`, with `driverlib/5xx_6xx/ei2c.h` containing the API definitions for use by applications.

15.2 API Functions

The eUSCI I2C API is broken into three groups of functions: those that deal with interrupts, those that handle status and initialization, and those that deal with sending and receiving data.

The I2C master and slave interrupts are handled by

- eI2C_enableInterrupt
- eI2C_disableInterrupt
- eI2C_clearInterruptFlag
- eI2C_getInterruptStatus

Status and initialization functions for the I2C modules are

- eI2C_masterInit
- eI2C_enable
- eI2C_disable
- eI2C_isBusBusy
- eI2C_isBusy
- eI2C_slaveInit
- eI2C_interruptStatus
- eI2C_setSlaveAddress
- eI2C_setMode
- eI2C_masterIsSTOPSent
- eI2C_selectMasterEnvironmentSelect

Sending and receiving data from the I2C slave module is handled by

- eI2C_slaveDataPut
- eI2C_slaveDataGet

Sending and receiving data from the I2C slave module is handled by

- eI2C_masterSendSingleByte
- eI2C_masterSendStart
- eI2C_masterMultiByteSendStart
- eI2C_masterMultiByteSendNext
- eI2C_masterMultiByteSendFinish
- eI2C_masterMultiByteSendStop
- eI2C_masterMultiByteReceiveNext

- eI2C_masterMultiByteReceiveFinish
- eI2C_masterMultiByteReceiveStop
- eI2C_masterReceiveStart
- eI2C_masterSingleReceive
- eI2C_getReceiveBufferAddressForDMA
- eI2C_getTransmitBufferAddressForDMA

DMA related

- eI2C_getReceiveBufferAddressForDMA
- eI2C_getTransmitBufferAddressForDMA

15.3 Programming Example

The following example shows how to use the I2C API to send data as a master.

```
//Initialize Master
 eI2C_masterInit(__MSP430_BASEADDRESS_EUSCI_B0__,
                 eI2C_CLOCKSOURCE_SMCLK,
                 //UCS_getSMCLK(__MSP430_BASEADDRESS_UCS__),
                 1000000,
                 eI2C_SET_DATA_RATE_400KBPS,
                 1,
                 eI2C_NO_AUTO_STOP
                 );

 //Specify slave address
 eI2C_setSlaveAddress(__MSP430_BASEADDRESS_EUSCI_B0__,
     SLAVE_ADDRESS
     );

 //Set in transmit mode
 eI2C_setMode(__MSP430_BASEADDRESS_EUSCI_B0__,
     eI2C_TRANSMIT_MODE
     );

 //Enable I2C Module to start operations
 eI2C_enable(__MSP430_BASEADDRESS_EUSCI_B0__);

 while (1)
 {
     //Send single byte data.
     eI2C_masterSendSingleByte(__MSP430_BASEADDRESS_EUSCI_B0__,
         transmitData
         );

     //Delay until transmission completes
     while (eI2C_isBusBusy(__MSP430_BASEADDRESS_EUSCI_B0__)) ;

     //Delay between each transaction
     __delay_cycles(50);

     //Increment transmit data counter
     transmitData++;
 }
```

16 EUSCI Synchronous Peripheral Interface (SPI)

16.1 Introduction

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame.

This library provides the API for handling a SPI communication using EUSCI.

The SPI module can be configured as either a master or a slave device.

The SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the module's input clock.

This driver is contained in `driverlib/5xx_6xx/espi.c`, with `driverlib/5xx_6xx/espi.h` containing the API definitions for use by applications.

16.2 API Functions

To use the module as a master, the user must call eSPI_masterInit() to configure the SPI Master. This is followed by enabling the SPI module using eSPI_enable(). The interrupts are then enabled (if needed). **It** is recommended to enable the SPI module before enabling the interrupts. A data transmit is then initiated using eSPI_transmitData() and then when the receive flag is set, the received data is read using eSPI_receiveData() and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using eSPI_slaveInit() and this is followed by enabling the module using eSPI_enable(). Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using eSPI_transmitData() and this is followed by a data reception by eSPI_receiveData()

The SPI API is broken into 3 groups of functions: those that deal with status and initialization, those that handle data, and those that manage interrupts.

The status and initialization of the SPI module are managed by

- eSPI_masterInit
- eSPI_slaveInit
- eSPI_disable
- eSPI_enable
- eSPI_masterChangeClock
- eSPI_isBusy

- eSPI_select4PinFunctionality
- eSPI_changeClockPhasePolarity

Data handling is done by

- eSPI_transmitData
- eSPI_receiveData

Interrupts from the SPI module are managed using

- eSPI_disableInterrupt
- eSPI_enableInterrupt
- eSPI_getInterruptStatus
- eSPI_clearInterruptFlag

DMA related

- eSPI_getReceiveBufferAddressForDMA
- eSPI_getTransmitBufferAddressForDMA

16.3 Programming Example

The following example shows how to use the SPI API to configure the SPI module as a master device, and how to do a simple send of data.

```
//Initialize Master
returnValue = eSPI_masterInit(__MSP430_BASEADDRESS_EUSCI_A0__,
        eSPI_CLOCKSOURCE_ACLK,
        UCS_getSMCLK(__MSP430_BASEADDRESS_UCS__),
        500000,
        eSPI_MSB_FIRST,
        eSPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,
        eSPI_CLOCKPOLARITY_INACTIVITY_HIGH,
        eSPI_3PIN
        );

    if (STATUS_FAIL == returnValue){
        return;
    }

//Enable SPI module
eSPI_enable(__MSP430_BASEADDRESS_EUSCI_A0__);

// Enable USCI_A0 RX interrupt
eSPI_enableInterrupt(__MSP430_BASEADDRESS_EUSCI_A0__,
                     eSPI_RECEIVE_INTERRUPT);

//Wait for slave to initialize
__delay_cycles(100);

TXData = 0x1;                             // Holds TX data

//USCI_A0 TX buffer ready?
while (!eSPI_getInterruptStatus(__MSP430_BASEADDRESS_EUSCI_A0__,
            eSPI_TRANSMIT_INTERRUPT)) ;
```

```
  //Transmit Data to slave
  eSPI_transmitData(__MSP430_BASEADDRESS_EUSCI_A0__, TXData);


  __bis_SR_register(LPM0_bits + GIE);      // CPU off, enable interrupts
  __no_operation();                       // Remain in LPM0
}

#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR (void)
{
    switch (__even_in_range(UCA0IV,4)){
        //Vector 2 - RXIFG
        case 2:
                        //USCI_A0 TX buffer ready?
                        while (!eSPI_getInterruptStatus(__MSP430_BASEADDRESS_EUSCI_A0__,
                                         eSPI_TRANSMIT_INTERRUPT)) ;

                        RXData = eSPI_receiveData(__MSP430_BASEADDRESS_EUSCI_A0__);

                        //Increment data
                        TXData++;

                        //Send next value
                        eSPI_transmitData(__MSP430_BASEADDRESS_EUSCI_A0__,
                                                    TXData
                                                    );

                        //Delay between transmissions for slave to process information
                        __delay_cycles(40);

            break;
        default: break;
    }
}
```

17 EUSCI UART

17.1 Introduction

The MSP430Ware library for UART mode features include:

- Odd, even, or non-parity
- Independent transmit and receive shift registers
- Separate transmit and receive buffer registers
- LSB-first or MSB-first data transmit and receive
- Built-in idle-line and address-bit communication protocols for multiprocessor systems
- Receiver start-edge detection for auto wake up from LPMx modes
- Status flags for error detection and suppression
- Status flags for address detection
- Independent interrupt capability for receive and transmit

In UART mode, the USCI transmits and receives characters at a bit rate asynchronous to another device. Timing for each character is based on the selected baud rate of the USCI. The transmit and receive functions use the same baud-rate frequency.

This driver is contained in `driverlib/5xx_6xx/euart.c`, with `driverlib/5xx_6xx/euart.h` containing the API definitions for use by applications.

17.2 API Functions

The UART API provides the set of functions required to implement an interrupt driven UART driver. The UART initialization with the various modes and features is done by the eUART_init(). At the end of this fucntion UART is initialized and stays disabled. eUART_enable() enables the UART and the module is now ready for transmit and receive. It is recommended to iniailize the UART via eUART_init(), enable the required interrupts and then enable UART via eUART_enable().

The UART API is broken into three groups of functions: those that deal with configuration and control of the UART modules, those used to send and receive data, and those that deal with interrupt handling and those dealing with DMA.

Configuration and control of the UART are handled by the

- eUART_init()
- eUART_initAdvance()
- eUART_enable()
- eUART_disable()

- eUART_setDormant()
- eUART_resetDormant()
- eUART_selectDeglitchTime()

Sending and receiving data via the UART is handled by the

- eUART_transmitData()
- eUART_receiveData()
- eUART_transmitAddress()
- eUART_transmitBreak()

Managing the UART interrupts and status are handled by the

- eUART_enableInterrupt()
- eUART_disableInterrupt()
- eUART_getInterruptStatus()
- eUART_clearInterruptFlag()
- eUART_queryStatusFlags()

DMA related

- eUART_getReceiveBufferAddressForDMA()
- eUART_getTransmitBufferAddressForDMA()

17.3 Programming Example

The following example shows how to use the UART API to initialize the UART, transmit characters, and receive characters.

```
// Configure UART
  if ( STATUS_FAIL == eUART_init(__MSP430_BASEADDRESS_EUSCI_A0__,
              eUART_CLOCKSOURCE_ACLK,
              32768,
              //eUCS_getACLK(__MSP430_BASEADDRESS_UCS__),
              9600,
              eUART_NO_PARITY,
              eUART_LSB_FIRST,
              eUART_ONE_STOP_BIT,
              eUART_MODE,
              eUART_LOW_FREQUENCY_BAUDRATE_GENERATION )){
         return;
     }

  eUART_enable(__MSP430_BASEADDRESS_EUSCI_A0__);

  // Enable USCI_A0 RX interrupt
  eUART_enableInterrupt(__MSP430_BASEADDRESS_EUSCI_A0__,
                  eUART_RECEIVE_INTERRUPT);                    // Enable interrupt

  __enable_interrupt();
  while (1)
  {
    TXData = TXData+1;                      // Increment TX data
```

```
    // Load data onto buffer
    eUART_transmitData(__MSP430_BASEADDRESS_EUSCI_A0__,
                                TXData);
    while(check != 1);
    check = 0;
  }
}
//******************************************************************************
//
//This is the USCI_A0 interrupt vector service routine.
//
//******************************************************************************
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,USCI_UART_UCTXCPTIFG))
  {
     case USCI_UART_UCRXIFG:
       RXData = eUART_receiveData(__MSP430_BASEADDRESS_EUSCI_A0__);
      if(!(RXData == TXData))                   // Check value
      {
        while(1);
      }
      check =1;
      break;
  }
}
```

18 Flash Memory Controller

18.1 Introduction

The flash memory is byte, word, and long-word addressable and programmable. The flash memory module has an integrated controller that controls programming and erase operations. The flash main memory is partitioned into 512-byte segments. Single bits, bytes, or words can be written to flash memory, but a segment is the smallest size of the flash memory that can be erased. The flash memory is partitioned into main and information memory sections. There is no difference in the operation of the main and information memory sections. Code and data can be located in either section. The difference between the sections is the segment size. There are four information memory segments, A through D. Each information memory segment contains 128 bytes and can be erased individually. The bootstrap loader (BSL) memory consists of four segments, A through D. Each BSL memory segment contains 512 bytes and can be erased individually. The main memory segment size is 512 byte. See the device-specific data sheet for the start and end addresses of each bank, when available, and for the complete memory map of a device. This library provides the API for flash segment erase, flash writes and flash operation status check.

This driver is contained in `driverlib/5xx_6xx/flash.c`, with `driverlib/5xx_6xx/flash.h` containing the API definitions for use by applications.

18.2 API Functions

Flash_segmentErase helps erase a single segment of the flash memory. A pointer to the flash segment being erased is passed on to this function.

Flash_eraseCheck helps check if a specific number of bytes in flash are currently erased. A pointer to the starting location of the erase check and the number of bytes to be checked is passed into this function.

Depending on the kind of writes being performed to the flash, this library provides APIs for flash writes.

Flash_write8 facilitates writing into the flash memory in byte format. Flash_write16 facilitates writing into the flash memory in word format. Flash_write32 facilitates writing into the flash memory in long format, pass by reference. Flash_memoryFill32 facilitates writing into the flash memory in long format, pass by value. Flash_status checks if the flash is currently busy erasing or programming.

The Flash API is broken into 3 groups of functions: those that deal with flash erase, those that write into flash, and those that give status of flash.

The flash erase operations are managed by

- Flash_segmentErase
- Flash_eraseCheck
- Flash_bankErase

Flash writes are managed by

- Flash_write8
- Flash_write16
- Flash_write32
- Flash_memoryFill32

The status is given by

- Flash_status
- Flash_eraseCheck

18.3 Programming Example

The following example shows some flash operations using the APIs

```
do{
    Flash_segmentErase(__MSP430_BASEADDRESS_FLASH__,
                       (unsigned char *)INFOD_START
                       );
    status = Flash_eraseCheck(__MSP430_BASEADDRESS_FLASH__,
                              (unsigned char *)INFOD_START,
                              128
                              );
    }while(status == STATUS_FAIL);

    //Flash write
    Flash_write32(__MSP430_BASEADDRESS_FLASH__,
                  calibration_data,
                  (unsigned long *)(INFOD_START),1);
```

19 FRAM Controller

19.1 Introduction

FRAM memory is a non-volatile memory that reads and writes like standard SRAM. The MSP430 FRAM memory features include:

- Byte or word write access
- Automatic and programmable wait state control with independent wait state settings for access and cycle times
- Error Correction Code with bit error correction, extended bit error detection and flag indicators
- Cache for fast read
- Power control for disabling FRAM on non-usage

This driver is contained in `driverlib/5xx_6xx/fram.c`, with `driverlib/5xx_6xx/fram.h` containing the API definitions for use by applications.

19.2 API Functions

FRAM_enableInterrupt enables selected FRAM interrupt sources.

FRAM_getInterruptStatus returns the status of the selected FRAM interrupt flags.

FRAM_disableInterrupt disables selected FRAM interrupt sources.

Depending on the kind of writes being performed to the FRAM, this library provides APIs for FRAM writes.

FRAM_write8 facilitates writing into the FRAM memory in byte format. FRAM_write16 facilitates writing into the FRAM memory in word format. FRAM_write32 facilitates writing into the FRAM memory in long format, pass by reference. FRAM_memoryFill32 facilitates writing into the FRAM memory in long format, pass by value. FRAM_status checks if the FRAM is currently busy programming.

The FRAM API is broken into 3 groups of functions: those that write into FRAM, those that handle interrupts, and those that give status of FRAM.

FRAM writes are managed by

- FRAM_write8
- FRAM_write16
- FRAM_write32
- FRAM_memoryFill32

The FRAM interrupts are handled by

- FRAM_enableInterrupt
- FRAM_getInterruptStatus
- FRAM_disableInterrupt

The status is given by

- FRAM_status

19.3 Programming Example

The following example shows some FRAM operations using the APIs

```
//Writes the value of "data", 128 times to FRAM
FRAM_memoryFill32(__MSP430_BASEADDRESS_FRAM_FR5XX__,data,
                                        (unsigned long *)FRAM_TEST_START,128);
```

20 FRGPIO

20.1 Introduction

The Digital I/O (FRGPIO) API provides a set of functions for using the MSP430Ware FRGPIO modules. Functions are provided to setup and enable use of input/output pins, setting them up with or without interrupts and those that access the pin value.

The digital I/O features include:

- Independently programmable individual I/Os
- Any combination of input or output
- Individually configurable P1 and P2 interrupts. Some devices may include additional port interrupts.
- Independent input and output data registers
- Individually configurable pullup or pulldown resistors

Devices within the family may have up to twelve digital I/O ports implemented (P1 to P11 and PJ). Most ports contain eight I/O lines; however, some ports may contain less (see the device-specific data sheet for ports available). Each I/O line is individually configurable for input or output direction, and each can be individually read or written. Each I/O line is individually configurable for pullup or pulldown resistors. PJ contains only four I/O lines.

Ports P1 and P2 always have interrupt capability. Each interrupt for the P1 and P2 I/O lines can be individually enabled and configured to provide an interrupt on a rising or falling edge of an input signal. All P1 I/O lines source a single interrupt vector P1IV, and all P2 I/O lines source a different, single interrupt vector P2IV. On some devices, additional ports with interrupt capability may be available (see the device-specific data sheet for details) and contain their own respective interrupt vectors. Individual ports can be accessed as byte-wide ports or can be combined into word-wide ports and accessed via word formats. Port pairs P1/P2, P3/P4, P5/P6, P7/P8, etc., are associated with the names PA, PB, PC, PD, etc., respectively. All port registers are handled in this manner with this naming convention except for the interrupt vector registers, P1IV and P2IV; that is, PAIV does not exist. When writing to port PA with word operations, all 16 bits are written to the port. When writing to the lower byte of the PA port using byte operations, the upper byte remains unchanged. Similarly, writing to the upper byte of the PA port using byte instructions leaves the lower byte unchanged. When writing to a port that contains less than the maximum number of bits possible, the unused bits are a "don't care". Ports PB, PC, PD, PE, and PF behave similarly.

Reading of the PA port using word operations causes all 16 bits to be transferred to the destination. Reading the lower or upper byte of the PA port (P1 or P2) and storing to memory using byte operations causes only the lower or upper byte to be transferred to the destination, respectively. Reading of the PA port and storing to a general-purpose register using byte operations causes the byte transferred to be written to the least significant byte of the register. The upper significant byte of the destination register is cleared automatically. Ports PB, PC, PD, PE, and PF behave similarly. When reading from ports that contain less than the maximum bits possible, unused bits are read as zeros (similarly for port PJ).

The FRGPIO pin may be configured as an I/O pin with FRGPIO_setAsOutputPin(), FRGPIO_setAsInputPin(), FRGPIO_setAsInputPinWithPullDownresistor() or FRGPIO_setAsInputPinWithPullUpresistor(). The FRGPIO pin may instead be configured to operate in the Peripheral Module assigned function by configuring the FRGPIO using FRGPIO_setAsPeripheralModuleFunctionOutputPin() or FRGPIO_setAsPeripheralModuleFunctionInputPin().

This driver is contained in `driverlib/5xx_6xx/frgpio.c`, with `driverlib/5xx_6xx/frgpio.h` containing the API definitions for use by applications.

20.2 API Functions

The FRGPIO API is broken into three groups of functions: those that deal with configuring the FRGPIO pins, those that deal with interrupts, and those that access the pin value.

The FRGPIO pins are configured with

- FRGPIO_setAsOutputPin()
- FRGPIO_setAsInputPin()
- FRGPIO_setAsInputPinWithPullDownresistor()
- FRGPIO_setAsInputPinWithPullUpresistor()
- FRGPIO_setAsPeripheralModuleFunctionOutputPin()
- FRGPIO_setAsPeripheralModuleFunctionInputPin()

The FRGPIO interrupts are handled with

- FRGPIO_enableInterrupt()
- FRGPIO_disbleInterrupt()
- FRGPIO_clearInterruptFlag()
- FRGPIO_getInterruptStatus()
- FRGPIO_interruptEdgeSelect()

The FRGPIO pin state is accessed with

- FRGPIO_setOutputHighOnPin()
- FRGPIO_setOutputLowOnPin()
- FRGPIO_toggleOutputOnPin()
- FRGPIO_getInputPinValue()

20.3 Programming Example

The following example shows how to use the FRGPIO API. A trigger is generated on a hi "TO" low transition on P1.4 (pulled-up input pin), which will generate P1_ISR. In the ISR, we toggle P1.0 (output pin).

```
        //Set P1.0 to output direction
    FRGPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN0
        );



    //Enable P1.4 internal resistance as pull-Up resistance
    FRGPIO_setAsInputPinWithPullUpresistor(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN4
        );

    //P1.4 interrupt enabled
    FRGPIO_enableInterrupt(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN4
        );

    //P1.4 Hi/Lo edge
    FRGPIO_interruptEdgeSelect(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN4,
        FRGPIO_HIGH_TO_LOW_TRANSITION
        );


    //P1.4 IFG cleared
    FRGPIO_clearInterruptFlag(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN4
        );

    //Enter LPM4 w/interrupt
    __bis_SR_register(LPM4_bits + GIE);

    //For debugger
    __no_operation();
}

//------------------------------------------------------------------------------
//
//This is the PORT1_VECTOR interrupt vector service routine
//
//------------------------------------------------------------------------------
#pragma vector=PORT1_VECTOR
__interrupt void Port_1 (void)
{
    //P1.0 = toggle
    FRGPIO_toggleOutputOnPin(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN0
        );


    //P1.4 IFG cleared
    FRGPIO_clearInterruptFlag(
        __MSP430_BASEADDRESS_PORT1_R__,
        FRGPIO_PORT_P1,
        FRGPIO_PIN4
```

```
                );
        }
```

21 Power Management Module (FRPMM)

21.1 Introduction

The PMM manages all functions related to the power supply and its supervision for the device. Its primary functions are first to generate a supply voltage for the core logic, and second, provide several mechanisms for the supervision of the voltage applied to the device (DVCC).

The PMM uses an integrated low-dropout voltage regulator (LDO) to produce a secondary core voltage (VCORE) from the primary one applied to the device (DVCC). In general, VCORE supplies the CPU, memories, and the digital modules, while DVCC supplies the I/Os and analog modules. The VCORE output is maintained using a dedicated voltage reference. The input or primary side of the regulator is referred to as its high side. The output or secondary side is referred to as its low side.

21.2 API Functions

FRPMM_enableLowPowerReset() / **FRPMM_disableLowPowerReset()** If enabled, SVSH does not reset device but triggers a system NMI. If disabled, SVSH resets device. Note: not available on FR57xx devices.

FRPMM_enableSVSH() / **FRPMM_disableSVSH()** If disabled on FR58xx/FR59xx, High-side SVS (SVSH) is disabled in LPM2, LPM3, LPM4, LPM3.5 and LPM4.5. SVSH is always enabled in active mode, LPM0, and LPM1. If disabled on FR57xx, High-side SVS (SVSH) is disabled in LPM4.5. SVSH is always enabled in active mode and LPM0/1/2/3/4 and LPM3.5. If enabled, SVSH is always enabled. Note: this API has different functionality depending on the part.

FRPMM_enableSVSL() / **FRPMM_disableSVSL()** If disabled, Low-side SVS (SVSL) is disabled in low power modes. SVSL is always enabled in active mode and LPM0. If enabled, SVSL is enabled in LPM0/1/2. SVSL is always enabled in AM and always disabled in LPM3/4 and LPM3.5/4.5. Note: not available on FR58xx/59xx devices.

FRPMM_regOff() / **FRPMM_regOn()** If off, Regulator is turned off when going to LPM3/4. System enters LPM3.5 or LPM4.5, respectively. If on, Regulator remains on when going into LPM3/4

FRPMM_clearInterrupt() Clear selected or all interrupt flags for the FRPMM

FRPMM_getInterruptStatus() Returns interrupt status of the selected flag in the FRPMM module

FRPMM_lockLPM5() / **FRPMM_unlockLPM5()** If unlocked, LPMx.5 configuration is not locked and defaults to its reset condition. if locked, LPMx.5 configuration remains locked. Pin state is held during LPMx.5 entry and exit.

This driver is contained in `driverlib/5xx_6xx/frpmm.c`, with `driverlib/5xx_6xx/frpmm.h` containing the API definitions for use by applications.

21.3 Programming Example

The following example shows some pmm operations using the APIs

```
   //Unlock the GPIO pins.
     /*
Base Address of Comparator D,
By default, the pins are unlocked unless waking
up from an LPMx.5 state in which case all GPIO
are previously locked.

      */
   FRPMM_unlockLPM5(__MSP430_BASEADDRESS_PMM_FR5xx__);

         //Get Interrupt Status from the PMMIFG register.
           /* Base Address of Comparator D,
mask:
       FRPMM_PMMBORIFG
       FRPMM_PMMRSTIFG,
       FRPMM_PMMPORIFG,
       FRPMM_SVSLIFG,
       FRPMM_SVSHIFG
       FRPMM_PMMLPM5IFG,
return STATUS_SUCCESS (0x01) or STATUS_FAIL (0x00)
            */
       if (FRPMM_getInterruptStatus(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMLPM5IFG))       // Was t
       {
         //Clear Interrupt Flag from the PMMIFG register.
               /* Base Address of Comparator D,
mask:
       FRPMM_PMMBORIFG
       FRPMM_PMMRSTIFG,
       FRPMM_PMMPORIFG,
       FRPMM_SVSLIFG,
       FRPMM_SVSHIFG
       FRPMM_PMMLPM5IFG,
       FRPMM_ALL
                */
              FRPMM_clearInterrupt(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMLPM5IFG);

   }

       if (FRPMM_getInterruptStatus(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMRSTIFG))        // Was t
       {
              FRPMM_clearInterrupt(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMRSTIFG);

       __delay_cycles(1000000);
         //Lock GPIO output states (before triggering a BOR)
               /*
Base Address of Comparator D,
Forces all GPIO to retain their output
states during a reset.
                */
       FRPMM_lockLPM5(__MSP430_BASEADDRESS_PMM_FR5xx__);
         //Trigger a software Brown Out Reset (BOR)
               /*
Base Address of Comparator D,
Forces the devices to perform a BOR.
                */
       FRPMM_trigBOR(__MSP430_BASEADDRESS_PMM_FR5xx__);
   }

       if (FRPMM_getInterruptStatus(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMBORIFG))        // Was t
       {
              FRPMM_clearInterrupt(__MSP430_BASEADDRESS_PMM_FR5xx__, FRPMM_PMMBORIFG);
```

```
        __delay_cycles(1000000);

        FRPMM_lockLPM5(__MSP430_BASEADDRESS_PMM_FR5xx__);

          //Disable SVSH
                /*
 Base Address of Comparator D,
 High-side SVS (SVSH) is disabled in LPM4.5. SVSH is
 always enabled in active mode and LPM0/1/2/3/4 and LPM3.5.
                 */
        FRPMM_disableSVSH(__MSP430_BASEADDRESS_PMM_FR5xx__);
          //Disable SVSL
                /*
 Base Address of Comparator D,
 Low-side SVS (SVSL) is disabled in low power modes.
 SVSL is always enabled in active mode and LPM0.
                 */
        FRPMM_disableSVSL(__MSP430_BASEADDRESS_PMM_FR5xx__);
          //Disable Regulator
                /*
 Base Address of Comparator D,
 Regulator is turned off when going to LPM3/4.
 System enters LPM3.5 or LPM4.5, respectively.
                 */
        FRPMM_regOff(__MSP430_BASEADDRESS_PMM_FR5xx__);
        __bis_SR_register(LPM4_bits); // Enter LPM4.5, This automatically locks
                                                                    //(if not locked already) all GPIO pir
                                                                    //  and will set the LPM5 flag and set
                                                                    //  in the PM5CTL0 register upon wake
    }
//-------------------
    while (1)
    {
        __no_operation();             // Don't sleep
    }
```

22 GPIO

22.1 Introduction

The digital I/O features include:

- Independently programmable individual I/Os
- Any combination of input or output
- Individually configurable P1 and P2 interrupts. Some devices may include additional port interrupts.
- Independent input and output data registers
- Individually configurable pullup or pulldown resistors

Devices within the family may have up to twelve digital I/O ports implemented (P1 to P11 and PJ). Most ports contain eight I/O lines; however, some ports may contain less (see the device-specific data sheet for ports available). Each I/O line is individually configurable for input or output direction, and each can be individually read or written. Each I/O line is individually configurable for pullup or pulldown resistors, as well as, configurable drive strength, full or reduced. PJ contains only four I/O lines.

Ports P1 and P2 always have interrupt capability. Each interrupt for the P1 and P2 I/O lines can be individually enabled and configured to provide an interrupt on a rising or falling edge of an input signal. All P1 I/O lines source a single interrupt vector P1IV, and all P2 I/O lines source a different, single interrupt vector P2IV. On some devices, additional ports with interrupt capability may be available (see the device-specific data sheet for details) and contain their own respective interrupt vectors. Individual ports can be accessed as byte-wide ports or can be combined into word-wide ports and accessed via word formats. Port pairs P1/P2, P3/P4, P5/P6, P7/P8, etc., are associated with the names PA, PB, PC, PD, etc., respectively. All port registers are handled in this manner with this naming convention except for the interrupt vector registers, P1IV and P2IV; that is, PAIV does not exist. When writing to port PA with word operations, all 16 bits are written to the port. When writing to the lower byte of the PA port using byte operations, the upper byte remains unchanged. Similarly, writing to the upper byte of the PA port using byte instructions leaves the lower byte unchanged. When writing to a port that contains less than the maximum number of bits possible, the unused bits are a "don't care". Ports PB, PC, PD, PE, and PF behave similarly.

Reading of the PA port using word operations causes all 16 bits to be transferred to the destination. Reading the lower or upper byte of the PA port (P1 or P2) and storing to memory using byte operations causes only the lower or upper byte to be transferred to the destination, respectively. Reading of the PA port and storing to a general-purpose register using byte operations causes the byte transferred to be written to the least significant byte of the register. The upper significant byte of the destination register is cleared automatically. Ports PB, PC, PD, PE, and PF behave similarly. When reading from ports that contain less than the maximum bits possible, unused bits are read as zeros (similarly for port PJ).

The GPIO pin may be configured as an I/O pin with GPIO_setAsOutputPin(), GPIO_setAsInputPin(), GPIO_setAsInputPinWithPullDownresistor() or

GPIO_setAsInputPinWithPullUpresistor(). The GPIO pin may instead be configured to operate in the Peripheral Module assigned function by configuring the GPIO using GPIO_setAsPeripheralModuleFunctionOutputPin() or GPIO_setAsPeripheralModuleFunctionInputPin().

This driver is contained in `driverlib/5xx_6xx/gpio.c`, with `driverlib/5xx_6xx/gpio.h` containing the API definitions for use by applications.

22.2 API Functions

The GPIO API is broken into three groups of functions: those that deal with configuring the GPIO pins, those that deal with interrupts, and those that access the pin value.

The GPIO pins are configured with

- GPIO_setAsOutputPin()
- GPIO_setAsInputPin()
- GPIO_setAsInputPinWithPullDownresistor()
- GPIO_setAsInputPinWithPullUpresistor()
- GPIO_setDriveStrength()
- GPIO_setAsPeripheralModuleFunctionOutputPin()
- GPIO_setAsPeripheralModuleFunctionInputPin()

The GPIO interrupts are handled with

- GPIO_enableInterrupt()
- GPIO_disbleInterrupt()
- GPIO_clearInterruptFlag()
- GPIO_getInterruptStatus()
- GPIO_interruptEdgeSelect()

The GPIO pin state is accessed with

- GPIO_setOutputHighOnPin()
- GPIO_setOutputLowOnPin()
- GPIO_toggleOutputOnPin()
- GPIO_getInputPinValue()

22.3 Programming Example

The following example shows how to use the GPIO API.

```
  // Set P1.0 to output direction
GPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
                    GPIO_PORT_P1,
                    GPIO_PIN0
                    );
```

```
// Set P1.4 to input direction
GPIO_setAsInputPin(__MSP430_BASEADDRESS_PORT1_R__,
                   GPIO_PORT_P1,
                   GPIO_PIN4
                   );

while (1)
{
  // Test P1.4
  if(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(
                                    __MSP430_BASEADDRESS_PORT1_R__,
                                    GPIO_PORT_P1,
                                    GPIO_PIN4
                                    ))
  {
    // if P1.4 set, set P1.0
    GPIO_setOutputHighOnPin(
                   __MSP430_BASEADDRESS_PORT1_R__,
                   GPIO_PORT_P1,
                   GPIO_PIN0
                   );
  }
  else
  {
    // else reset
    GPIO_setOutputLowOnPin(
                   __MSP430_BASEADDRESS_PORT1_R__,
                   GPIO_PORT_P1,
                   GPIO_PIN0
                   );
  }
}
```

23 Inter-Integrated Circuit (I2C)

23.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the MSP430Ware I2C modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules.

The I2C module provide the ability to communicate to other IC devices over an I2C bus. The I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the I2C bus can be designated as either a master or a slave. The MSP430Ware I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave. Finally, the MSP430Ware I2C modules can operate at two speeds: Standard (100 kb/s) and Fast (400 kb/s).

I2C module can generate interrupts. The I2C module configured as a master will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The I2C module configured as a slave will generate interrupts when data has been sent or requested by a master.

23.1.1 Master Operations

To drive the master module, the APIs need to be invoked in the following order

- **I2C_masterInit**
- **I2C_setSlaveAddress**
- **I2C_setMode**
- **I2C_enable**
- **I2C_enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first initialize the I2C module and configure it as a master with a call to I2C_masterInit(). That function will set the clock and data rates. This is followed by a call to set the slave address with which the master intends to communicate with using I2C_setSlaveAddress. Then the mode of operation (transmit or receieve) is chosen using I2C_setMode. The I2C module may now be enabled using I2C_enable. It is recommneded to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Master Single Byte Trasnmission

- I2C_masterSendSingleByte

Master Mulitple Byte Trasnmission

- I2C_masterMultiByteSendStart
- I2C_masterMultiByteSendNext
- I2C_masterMultiByteSendFinish
- I2C_masterMultiByteSendStop

Master Single Byte Reception

- I2C_masterSingleReceiveStart
- I2C_masterSingleReceive

Master Multiple Byte Reception

- I2C_masterMultiByteReceiveStart
- I2C_masterMultiByteReceiveNext
- I2C_masterMultiByteReceiveFinish
- I2C_masterMultiByteReceiveStop

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

23.1.2 Slave Operations

To drive the slave module, the APIs need to be invoked in the following order

- **I2C_slaveInit**
- **I2C_setMode**
- **I2C_enable**
- **I2C_enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first call the I2C_slaveInit to initialize the slave module in I2C mode and set the slave address. This is followed by a call to set the mode of operation (transmit or receive).The I2C module may now be enabled using I2C_enable. It is recommneded to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Slave Transmission API

- I2C_slaveDataPut

Slave Reception API

- I2C_slaveDataGet

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

This driver is contained in `driverlib/5xx_6xx/i2c.c`, with `driverlib/5xx_6xx/i2c.h` containing the API definitions for use by applications.

23.2 API Functions

The I2C API is broken into three groups of functions: those that deal with interrupts, those that handle status and initialization, and those that deal with sending and receiving data.

The I2C master and slave interrupts are handled by

- I2C_enableInterrupt
- I2C_disableInterrupt
- UART_clearInterruptFlag
- I2C_getInterruptStatus

Status and initialization functions for the I2C modules are

- I2C_masterInit
- I2C_enable
- I2C_disable
- I2C_isBusBusy
- I2C_isBusy
- I2C_slaveInit
- I2C_interruptStatus
- I2C_setSlaveAddress
- I2C_setMode

Sending and receiving data from the I2C slave module is handled by

- I2C_slaveDataPut
- I2C_slaveDataGet

Sending and receiving data from the I2C slave module is handled by

- I2C_masterSendSingleByte
- I2C_masterMultiByteSendStart
- I2C_masterMultiByteSendNext
- I2C_masterMultiByteSendFinish
- I2C_masterMultiByteSendStop
- I2C_masterMultiByteReceiveStart
- I2C_masterMultiByteReceiveNext
- I2C_masterMultiByteReceiveFinish
- I2C_masterMultiByteReceiveStop
- I2C_masterSingleReceiveStart
- I2C_masterSingleReceive
- I2C_getReceiveBufferAddressForDMA
- I2C_getTransmitBufferAddressForDMA

DMA related

- I2C_getReceiveBufferAddressForDMA
- I2C_getTransmitBufferAddressForDMA

23.3 Programming Example

The following example shows how to use the I2C API to send data as a master.

```
  // Initialize Master
I2C_masterInit(USCI_B0_BASE, SMCLK, CLK_getSMClk(), I2C_SET_DATA_RATE_400KBPS);

// Specify slave address
I2C_setSlaveAddress(USCI_B0_BASE, SLAVE_ADDRESS);

// Set in transmit mode
I2C_setMode(USCI_B0_BASE, I2C_TRANSMIT_MODE);

//Enable I2C Module to start operations
I2C_enable(USCI_B0_BASE);

while (1)
{
  // Send single byte data.
  I2C_masterSendSingleByte(USCI_B0_BASE, transmitData);

  // Delay until transmission completes
  while(I2C_busBusy(USCI_B0_BASE));

  // Increment transmit data counter
  transmitData++;
}
```

24 LDO-PWR

24.1 Introduction

The features of the LDO-PWR module include:

- Integrated 3.3-V LDO regulator with sufficient output to power the entire MSP430Ź microcontroller and system circuitry from 5-V external supply
- Current-limiting capability on 3.3-V LDO output with detection flag and interrupt generation
- LDO input voltage detection flag and interrupt generation

The LDO-PWR power system incorporates an integrated 3.3-V LDO regulator that allows the entire MSP430 microcontroller to be powered from nominal 5-V LDOI when it is made available from the system. Alternatively, the power system can supply power only to other components within the system, or it can be unused altogether.

This driver is contained in `driverlib/5xx_6xx/ldoPwr.c`, with `driverlib/5xx_6xx/ldoPwr.h` containing the API definitions for use by applications.

24.2 API Functions

The ldoPwr configuration is handled by

- LDOPWR_unLockConfiguration()
- LDOPWR_lockConfiguration()
- LDOPWR_enablePort_U_inputs()
- LDOPWR_disablePort_U_inputs()
- LDOPWR_enablePort_U_outputs()
- LDOPWR_disablePort_U_outputs()
- LDOPWR_enable()
- LDOPWR_disable()
- LDOPWR_enableOverloadAutoOff()
- LDOPWR_disableOverloadAutoOff()

Handling the read/write of output data is handled by

- LDOPWR_getPort_U1_inputData()
- LDOPWR_getPort_U0_inputData()
- LDOPWR_getPort_U1_outputData()

- LDOPWR_getPort_U0_outputData()
- LDOPWR_getOverloadAutoOffStatus()
- LDOPWR_setPort_U0_outputData()
- LDOPWR_togglePort_U1_outputData()
- LDOPWR_togglePort_U0_outputData()
- LDOPWR_setPort_U1_outputData()

The interrupt and status operations are handled by

- LDOPWR_enableInterrupt()
- LDOPWR_disableInterrupt()
- LDOPWR_getInterruptStatus()
- LDOPWR_clearInterruptStatus()
- LDOPWR_isLDOInputValid()
- LDOPWR_getOverloadAutoOffStatus()

24.3 Programming Example

The following example shows how to use the LDO-PWR API.

```
{
  // Enable access to config registers
  LDOPWR_unLockConfiguration(__MSP430_BASEADDRESS_PU__);

  // Configure PU.0 as output pins
  LDOPWR_enablePort_U_outputs(__MSP430_BASEADDRESS_PU__);

  //Set PU.1 = high
  LDOPWR_setPort_U1_outputData(__MSP430_BASEADDRESS_PU__,
                               LDOPWR_PORTU_PIN_HIGH
                                                                  );

  //Set PU.0 = low
  LDOPWR_setPort_U0_outputData(__MSP430_BASEADDRESS_PU__,
                               LDOPWR_PORTU_PIN_LOW
                               );

  // Enable LDO overload indication interrupt
  LDOPWR_enableInterrupt(__MSP430_BASEADDRESS_PU__,
                         LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                         );

  // Disbale access to config registers
  LDOPWR_lockConfiguration(__MSP430_BASEADDRESS_PU__);

  // continuous loop
  while(1)
  {
    // Delay
    for(i=50000;i>0;i--);

    // Enable access to config registers
    LDOPWR_unLockConfiguration(__MSP430_BASEADDRESS_PU__);

    // XOR PU.0/1
```

```
    LDOPWR_togglePort_U1_outputData(__MSP430_BASEADDRESS_PU__);
    LDOPWR_togglePort_U0_outputData(__MSP430_BASEADDRESS_PU__);

    // Disbale access to config registers
    LDOPWR_lockConfiguration(__MSP430_BASEADDRESS_PU__);
  }
}


//******************************************************************************
//
// This is the LDO_PWR_VECTOR interrupt vector service routine.
//
//******************************************************************************
__interrupt void LDOInterruptHandler(void)
{
  if(LDOPWR_getInterruptStatus(__MSP430_BASEADDRESS_PU__,
                               LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                               ))

  {
    // Enable access to config registers
    LDOPWR_unLockConfiguration(__MSP430_BASEADDRESS_PU__);

   // Software clear IFG
    LDOPWR_clearInterruptStatus(__MSP430_BASEADDRESS_PU__,
                                LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                                );

   // Disable access to config registers
   LDOPWR_lockConfiguration(__MSP430_BASEADDRESS_PU__);

   // Over load indication; take necessary steps in application firmware
   while(1);

  }
}
```

25 Memory Protection Unit (MPU)

25.1 Introduction

The MPU protects against accidental writes to designated read-only memory segments or execution of code from a constant memory segment memory. Clearing the MPUENA bit disables the MPU, making the complete memory accessible for read, write, and execute operations. After a BOR, the complete memory is accessible without restrictions for read, write, and execute operations.

MPU features include:

- Main memory can be configured up to three segments of variable size
- Access rights for each segment can be set independently
- Information memory can have its access rights set independently
- All MPU registers are protected from access by password

This driver is contained in `driverlib/5xx_6xx/mpu.c`, with `driverlib/5xx_6xx/mpu.h` containing the API definitions for use by applications.

25.2 API Functions

The MPU API is broken into three group of functions: those that handle initialization, those that deal with memory segmentation and access rights definition, and those that handle interrupts.

The MPU initialization function is

- MPU_start

The MPU memory segmentation and access right definition functions are

- MPU_createTwoSegments
- MPU_createThreeSegments

The MPU interrupt handler functions

- MPU_enablePUCOnViolation
- MPU_getInterruptStatus
- MPU_clearInterruptFlag
- MPU_clearAllInterruptFlags

25.3 Programming Example

The following example shows some MPU operations using the APIs

```
//Define memory segment boundaries and set access right for each memory segment
MPU_createThreeSegments(__MSP430_BASEADDRESS_MPU__,0x04,0x08,
                                                MPU_READ|MPU_WRITE|MPU_EXEC,
                                                MPU_READ,
                                                MPU_READ|MPU_WRITE|MPU_EXEC);

// Configures MPU to generate a PUC on access violation on the second segment
MPU_enablePUCOnViolation(__MSP430_BASEADDRESS_MPU__,MPU_SECOND_SEG);

//Enables the MPU module
MPU_start(__MSP430_BASEADDRESS_MPU__);
```

26 32-Bit Hardware Multiplier (MPY32)

26.1 Introduction

The 32-Bit Hardware Multiplier (MPY32) API provides a set of functions for using the MSP430Ware MPY32 modules. Functions are provided to setup the MPY32 modules, set the operand registers, and obtain the results.

The MPY32 Modules does not generate any interrupts.

This driver is contained in `driverlib/5xx_6xx/mpy32.c`, with `driverlib/5xx_6xx/mpy32.h` containing the API definitions for use by applications.

26.2 API Functions

The MPY32 API is broken into three groups of functions: those that control the settings, those that set the operand registers, and those that return the results, sum extension, and carry bit value.

The settings are handled by

- MPY32_setWriteDelay
- MPY32_setSaturationMode
- MPY32_resetSaturationMode
- MPY32_setFractionMode
- MPY32_resetFractionMode

The operand registers are set by

- MPY32_setOperandOne8Bit
- MPY32_setOperandOne16Bit
- MPY32_setOperandOne24Bit
- MPY32_setOperandOne32Bit
- MPY32_setOperandTwo8Bit
- MPY32_setOperandTwo16Bit
- MPY32_setOperandTwo24Bit
- MPY32_setOperandTwo32Bit

The results can be returned by

- MPY32_getResult8Bit
- MPY32_getResult16Bit
- MPY32_getResult24Bit

- MPY32_getResult32Bit
- MPY32_getResult64Bit
- MPY32_getSumExtension
- MPY32_getCarryBitValue

26.3 Programming Example

The following example shows how to initialize and use the MPY32 API to calculate a 16-bit by 16-bit unsigned multiplication operation.

```
WDT_hold(__MSP430_BASEADDRESS_WDT_A__);   // Stop WDT

// Set a 16-bit Operand into the specific Operand 1 register to specify
 // unsigned multiplication
MPY32_setOperandOne16Bit(__MSP430_BASEADDRESS_MPY32__,
                         MPY32_MULTIPLY_UNSIGNED,
                         0x1234);
// Set Operand 2 to begin the multiplication operation
MPY32_setOperandTwo16Bit(__MSP430_BASEADDRESS_MPY32__,
                         0x5678);

__bis_SR_register(LPM4_bits);             // Enter LPM4
__no_operation();                         // BREAKPOINT HERE to verify the
                                           // correct result in the registers
```

27 Power Management Module (PMM)

27.1 Introduction

The PMM manages the following internal circuitry:

- An integrated low-dropout voltage regulator (LDO) that produces a secondary core voltage (VCORE) from the primary voltage that is applied to the device (DVCC)
- Supply voltage supervisors (SVS) and supply voltage monitors (SVM) for the primary voltage (DVCC) and the secondary voltage (VCORE). The SVS and SVM include programmable threshold levels and power-fail indicators. Therefore, the PMM plays a crucial role in defining the maximum performance, valid voltage conditions, and current consumption for an application running on an MSP430x5xx or MSP430x6xx device. The secondary voltage that is generated by the integrated LDO, VCORE, is programmable to one of four core voltage levels, shown as 0, 1, 2, and 3. Each increase in VCORE allows the CPU to operate at a higher maximum frequency. The values of these frequencies are specified in the device-specific data sheet. This feature allows the user the flexibility to trade power consumption in active and low-power modes for different degrees of maximum performance and minimum supply voltage.

NOTE: To align with the nomenclature in the MSP430x5xx/MSP430x6xx Family UserŠs Guide, the primary voltage domain (DVCC) is referred to as the high-side voltage (SvsH/SVMH) and the secondary voltage domain (VCORE) is referred to as the low-side voltage (SvsL/SvmL).

Moving between the different VCORE voltages requires a specific sequence of events and can be done only one level at a time; for example, to change from level 0 to level 3, the application code must step through level 1 and level 2.

VCORE increase: 1. SvmL monitor level is incremented. 2. VCORE level is incremented. 3. The SvmL Level Reached Interrupt Flag (SVSMLVLRIFG) in the PMMIFG register is polled. When asserted, SVSMLVLRIFG indicates that the VCORE voltage has reached its next level. 4. SvsL is increased. SvsL is changed last, because if SVSL were incremented prior to VCORE, it would potentially cause a reset.

VCORE decrease: 5. Decrement SvmL and SVSL levels. 6. Decrement VCORE. The PMM_setVCore() function appropriately handles an increase or decrease of the core voltage. NOTE: The procedure recommended above provides a workaround for the erratum FLASH37. See the device-specific erratasheet to determine if a device is affected by FLASH37. The workaround is also highlighted in the source code for the PMM library

Recommended SVS and SVM Settings The SVS and SVM on both the high side and the low side are enabled in normal performance mode following a brown-out reset condition. The device is held in reset until the SVS and SVM verify that the external and core voltages meet the minimum requirements of the default core voltage, which is level zero. The SVS and SVM remain enabled unless disabled by the firmware. The low-side SVS and SVM are useful for verifying the startup conditions and for verifying any modification to the core voltage. However, in their default mode, they prevent the CPU from executing code on wake-up from low-power modes 2, 3, and 4 for a full 150 μs, not 5 μs. This is because, in their default states, the SVSL and SvmL are powered down in the low-power mode of the PMM and need time for their comparators to wake and stabilize

before they can verify the voltage condition and release the CPU for execution. Note that the high-side SVS and SVM do not influence the wake time from low-power modes. If the wake-up from low-power modes needs to be shortened to 5 μs, the SVSL and SvmL should be disabled after the initialization of the core voltage at the beginning of the application. Disabling SVSL and SvmL prevents them from gating the CPU on wake-up from LPM2, LPM3, and LPM4. The application is still protected on the high side with SvsH and SVMH. The PMM_setVCore() function automatically enables and disables the SVS and SVM as necessary if a non-zero core voltage level is required. If the application does not require a change in the core voltage (that is, when the target MCLK is less than 8 MHz), the PMM_disableSVSLSvmL() and PMM_enableSvsHReset() macros can be used to disable the low-side SVS and SVM circuitry and enable only the high-side SVS POR reset, respectively.

Setting SVS/SVM Threshold Levels The voltage thresholds for the SVS and SVM modules are programmable. On the high side, there are two bit fields that control these threshold levels Ű the SvsHRVL and SVSMHRRL. The SvsHRVL field defines the voltage threshold at which the SvsH triggers a reset (also known as the SvsH ON voltage level). The SVSMHRRL field defines the voltage threshold at which the SvsH releases the device from a reset (also known as SvsH OFF voltage level). The MSP430x5xx/MSP430x6xx Family UserŠs Guide (SLAU208) [1] recommends the settings shown in Table 1 when setting these bits. The PMM_setVCore() function follows these recommendations and ensures that the SVS levels match the core voltage levels that are used.

Advanced SVS Controls and Trade-offs In addition to the default SVS settings that are provided with the PMM_setVCore() function, the SVS/SVM modules can be optimized for wake-up speed, response time (propagation delay), and current consumption, as needed. The following controls can be optimized for the SVS/SVM modules:

- Protection in low power modes - LPM2, LPM3, and LPM4
- Wake-up time from LPM2, LPM3, and LPM4
- Response time to react to an SVS event Selecting the LPM option, wake-up time, and response time that is best suited for the application is left to the user. A few typical examples illustrate the trade-offs: Case A: The most robust protection that stays on in LPMs and has the fastest response and wake-up time consumes the most power. Case B: With SVS high side active only in AM, no protection in LPMs, slow wake-up, and slow response time has SVS protection with the least current consumption. Case C: An optimized case is described - turn off the low-side monitor and supervisor, thereby saving power while keeping response time fast on the high side to help with timing critical applications. The user can call the PMM_setVCore() function, which configures SVS/SVM high side and low side with the recommended or default configurations, or can call the APIs provided to control the parameters as the application demands.

Any writes to the SVSMLCTL and SVSMHCTL registers require a delay time for these registers to settle before the new settings take effect. This delay time is dependent on whether the SVS and SVM modules are configured for normal or full performance. See device-specific data sheet for exact delay times.

27.2 API Functions

PMM_enableSvsL() / **PMM_disableSvsL()** Enables or disables the low-side SVS circuitry

PMM_enableSvmL() / **PMM_disableSvmL()** Enables or disables the low-side SVM circuitry

PMM_enableSvsH() / **PMM_disableSvsH()** Enables or disables the high-side SVS circuitry

PMM_enableSVMH() / **PMM_disableSVMH()** Enables or disables the high-side SVM circuitry

PMM_enableSvsLSvmL() / **PMM_disableSvsLSvmL()** Enables or disables the low-side SVS and SVM circuitry

PMM_enableSvsHSvmH() / **PMM_disableSvsHSvmH()** Enables or disables the high-side SVS and SVM circuitry

PMM_enableSvsLReset() / **PMM_disableSvsLReset()** Enables or disables the POR signal generation when a low-voltage event is registered by the low-side SVS

PMM_enableSvmLInterrupt() / **PMM_disableSvmLInterrupt()** Enables or disables the interrupt generation when a low-voltage event is registered by the low-side SVM

PMM_enableSvsHReset() / **PMM_disableSvsHReset()** Enables or disables the POR signal generation when a low-voltage event is registered by the high-side SVS

PMM_enableSVMHInterrupt() / **PMM_disableSVMHInterrupt()** Enables or disables the interrupt generation when a low-voltage event is registered by the high-side SVM

PMM_clearPMMIFGS() Clear all interrupt flags for the PMM

PMM_SvsLEnabledInLPMFastWake() Enables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

PMM_SvsLEnabledInLPMSlowWake() Enables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_SvsLDisabledInLPMFastWake() Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

PMM_SvsLDisabledInLPMSlowWake() Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_SvsHEnabledInLPMNormPerf() Enables supervisor high side in LPM with tpd = 20 μs(1)

PMM_SvsHEnabledInLPMFullPerf() Enables supervisor high side in LPM with tpd = 2.5 μs(1)

PMM_SvsHDisabledInLPMNormPerf() Disables supervisor high side in LPM with tpd = 20 μs(1)

PMM_SvsHDisabledInLPMFullPerf() Disables supervisor high side in LPM with tpd = 2.5 μs(1)

PMM_SvsLOptimizedInLPMFastWake() Optimized to provide twake-up-fast from LPM2, LPM3, and LPM4 with least power

PMM_SvsHOptimizedinLPMFullPerf() Optimized to provide tpd = 2.5 μs(1) in LPM with least power

PMM_getInterruptStatus() Returns interrupt status of the PMM module

PMM_setVCore() Sets the appropriate VCORE level. Calls the PMM_setVCoreUp() or PMM_setVCoreDown() function the required number of times depending on the current VCORE level, because the levels must be stepped through individually. A status indicator equal to STATUS_SUCCESS or STATUS_FAIL that indicates a valid or invalid VCORE transition, respectively. An invalid VCORE transition exists if DVCC is less than the minimum required voltage for the target VCORE voltage.

This driver is contained in `driverlib/5xx_6xx/pmm.c`, with `driverlib/5xx_6xx/pmm.h` containing the API definitions for use by applications.

27.3 Programming Example

The following example shows some pmm operations using the APIs

```
//Use the line below to bring the level back to 0
    status = PMM_setVCore(__MSP430_BASEADDRESS_PMM__,
        PMMCOREV_0
        );

    //Set P1.0 to output direction
    GPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
        GPIO_PORT_P1,
        GPIO_PIN0
        );

    //continuous loop
    while (1)
    {
        //Toggle P1.0
        GPIO_toggleOutputOnPin(
            __MSP430_BASEADDRESS_PORT1_R__,
            GPIO_PORT_P1,
            GPIO_PIN0
            );
        //Delay
        __delay_cycles(20000);
    }
```

28 Port Mapping Controller

28.1 Introduction

The port mapping controller allows the flexible and reconfigurable mapping of digital functions to port pins. The port mapping controller features are:

- Configuration protected by write access key.
- Default mapping provided for each port pin (device-dependent, the device pinout in the device-specific data sheet).
- Mapping can be reconfigured during runtime.
- Each output signal can be mapped to several output pins.

This driver is contained in `driverlib/5xx_6xx/pmap.c`, with `driverlib/5xx_6xx/pmap.h` containing the API definitions for use by applications.

28.2 API Functions

The MSP430ware API that configures Port Mapping is PMAP_configurePorts()

It needs the following data to configure port mapping. portMapping - pointer to init Data PxMAPy - pointer start of first Port Mapper to initialize numberOfPorts - number of Ports to initialize portMapReconfigure - to enable/disable reconfiguration

28.3 Programming Example

The following example shows some Port Mapping Controller operations using the APIs

```
    const unsigned char port_mapping[] = {
            //Port P4:
            PM_TB0CCR0A,
            PM_TB0CCR1A,
            PM_TB0CCR2A,
            PM_TB0CCR3A,
            PM_TB0CCR4A,
            PM_TB0CCR5A,
            PM_TB0CCR6A,
            PM_NONE
    };

    //CONFIGURE PORTS- pass the port_mapping array, start @ P4MAP01, initialize
//a single port, do not allow run-time reconfiguration of port mapping

    PMAP_configurePorts(__MSP430_BASEADDRESS_PORT_MAPPING__,
```

```
(const unsigned char  *)port_mapping,
(unsigned char  *)&P4MAP01,
1,
PMAP_DISABLE_RECONFIGURATION
);
```

29 RAM Controller

29.1 Introduction

The RAMCTL provides access to the different power modes of the RAM. The RAMCTL allows the ability to reduce the leakage current while the CPU is off. The RAM can also be switched off. In retention mode, the RAM content is saved while the RAM content is lost in off mode. The RAM is partitioned in sectors, typically of 4KB (sector) size. See the device-specific data sheet for actual block allocation and size. Each sector is controlled by the RAM controller RAM Sector Off control bit (RCRSyOFF) of the RAMCTL Control 0 register (RCCTL0). The RCCTL0 register is protected with a key. Only if the correct key is written during a word write, the RCCTL0 register content can be modified. Byte write accesses or write accesses with a wrong key are ignored.

This driver is contained in `driverlib/5xx_6xx/ramcontroller.c`, with `driverlib/5xx_6xx/ramcontroller.h` containing the API definitions for use by applications.

29.2 API Functions

The MSP430ware API that configure the RAM controller are:

ramController_setSectorOff() - Set specified RAM sector off ramController_getSectorState() - Get RAM sector ON/OFF status

29.3 Programming Example

The following example shows some RAM Controller operations using the APIs

```
//Start timer
Timer_startUpMode(   __MSP430_BASEADDRESS_T0B7__,
    TIMER_CLOCKSOURCE_ACLK,
    TIMER_CLOCKSOURCE_DIVIDER_1,
    25000,
    TIMER_TAIE_INTERRUPT_DISABLE,
    TIMER_CAPTURECOMPARE_INTERRUPT_ENABLE,
    TIMER_DO_CLEAR
    );

//RAM controller sector off
ramController_setSectorOff(__MSP430_BASEADDRESS_RC__,
    RAMCONTROL_SECTOR2
    ) ;


//Enter LPM0, enable interrupts
__bis_SR_register(LPM3_bits + GIE);
```

```
    //For debugger
    __no_operation();
}

//******************************************************************************
//
//This is the Timer B0 interrupt vector service routine.
//
//******************************************************************************
#pragma vector=TIMERB0_VECTOR
__interrupt void TIMERB0_ISR (void)
{
    returnValue = ramController_getSectorState(__MSP430_BASEADDRESS_RC__,
        RAMCONTROL_SECTOR0 +
        RAMCONTROL_SECTOR1 +
        RAMCONTROL_SECTOR2 +
        RAMCONTROL_SECTOR3);

}
```

30 Internal Reference (REF)

30.1 Introduction

The Internal Reference (REF) API provides a set of functions for using the MSP430Ware REF modules. Functions are provided to setup and enable use of the Reference voltage, enable or disable the internal temperature sensor, and view the status of the inner workings of the REF module.

The reference module (REF) is responsible for generation of all critical reference voltages that can be used by various analog peripherals in a given device. These include, but are not necessarily limited to, the ADC10_A, ADC12_A, DAC12_A, LCD_B, and COMP_B modules dependent upon the particular device. The heart of the reference system is the bandgap from which all other references are derived by unity or non-inverting gain stages. The REFGEN sub-system consists of the bandgap, the bandgap bias, and the non-inverting buffer stage which generates the three primary voltage reference available in the system, namely 1.5 V, 2.0 V, and 2.5 V. In addition, when enabled, a buffered bandgap voltage is also available.

This driver is contained in `driverlib/5xx_6xx/ref.c`, with `driverlib/5xx_6xx/ref.h` containing the API definitions for use by applications.

30.2 API Functions

The DMA API is broken into three groups of functions: those that deal with the reference voltage, those that handle the internal temperature sensor, and those that return the status of the REF module.

The reference voltage of the REF module is handled by

- REF_setReferenceVoltage
- REF_enableReferenceVoltageOutput
- REF_disableReferenceVoltageOutput
- REF_enableReferenceVoltage
- REF_disableReferenceVoltage

The internal temperature sensor is handled by

- REF_disableTempSensor
- REF_enableTempSensor

The status of the REF module is handled by

- REF_getBandgapMode
- REF_isBandgapActive

- REF_isRefGenBusy
- REF_isRefGen

30.3 Programming Example

The following example shows how to initialize and use the REF API with the ADC12 module to use as a positive reference to the analog signal input.

```
 // By default, REFMSTR=1 => REFCTL is used to configure the internal reference

 // If ref generator busy, WAIT
 while(REF_refGenBusyStatus(__MSP430_BASEADDRESS_REF__));
 // Select internal ref = 2.5V
 REF_setReferenceVoltage(__MSP430_BASEADDRESS_REF__,
                         REF_VREF2_5V);
 // Internal Reference ON
 REF_enableReferenceVoltage(__MSP430_BASEADDRESS_REF__);

 __delay_cycles(75);                       // Delay (~75us) for Ref to settle

 // Initialize the ADC12 Module
 /*
Base address of ADC12 Module
Use internal ADC12 bit as sample/hold signal to start conversion
USE MODOSC 5MHZ Digital Oscillator as clock source
Use default clock divider of 1
  */
 ADC12_init(__MSP430_BASEADDRESS_ADC12_PLUS__,
            ADC12_SAMPLEHOLDSOURCE_SC,
            ADC12_CLOCKSOURCE_ADC12OSC,
            ADC12_CLOCKDIVIDEBY_1);
 /*
Base address of ADC12 Module
For memory buffers 0-7 sample/hold for 64 clock cycles
For memory buffers 8-15 sample/hold for 4 clock cycles (default)
Disable Multiple Sampling
  */
 ADC12_setupSamplingTimer(__MSP430_BASEADDRESS_ADC12_PLUS__,
                          ADC12_CYCLEHOLD_64_CYCLES,
                          ADC12_CYCLEHOLD_4_CYCLES,
                          ADC12_MULTIPLESAMPLESENABLE);

 // Configure Memory Buffer
 /*
Base address of the ADC12 Module
Configure memory buffer 0
Map input A0 to memory buffer 0
Vref+ = Vref+ (INT)
Vref- = AVss
  */
 ADC12_memoryConfigure(__MSP430_BASEADDRESS_ADC12_PLUS__,
                       ADC12_MEMORY_0,
                       ADC12_INPUT_A0,
                       ADC12_VREFPOS_INT,
                       ADC12_VREFNEG_AVSS,
                       ADC12_NOTENDOFSEQUENCE);

 while (1)
 {
   // Enable/Start sampling and conversion
   /*
```

```
Base address of ADC12 Module
Start the conversion into memory buffer 0
Use the single-channel, single-conversion mode
    */
   ADC12_startConversion(__MSP430_BASEADDRESS_ADC12_PLUS__,
                         ADC12_MEMORY_0,
                         ADC12_SINGLECHANNEL);

   // Poll for interrupt on memory buffer 0
   while(!ADC12_interruptStatus(__MSP430_BASEADDRESS_ADC12_PLUS__, ADC12IFG0));

   __no_operation();                       // SET BREAKPOINT HERE
 }
```

31 Internal Reference (REFA)

31.1 Introduction

The Internal Reference (REFA) API provides a set of functions for using the MSP430Ware REFA modules. Functions are provided to setup and enable use of the Reference voltage, enable or disable the internal temperature sensor, and view the status of the inner workings of the REFA module.

The reference module (REF) is responsible for generation of all critical reference voltages that can be used by various analog peripherals in a given device. The heart of the reference system is the bandgap from which all other references are derived by unity or non-inverting gain stages. The REFGEN sub-system consists of the bandgap, the bandgap bias, and the non-inverting buffer stage which generates the three primary voltage reference available in the system, namely 1.2 V, 2.0 V, and 2.5 V. In addition, when enabled, a buffered bandgap voltage is available.

This driver is contained in `driverlib/5xx_6xx/refa.c`, with `driverlib/5xx_6xx/refa.h` containing the API definitions for use by applications.

31.2 API Functions

The DMA API is broken into three groups of functions: those that deal with the refaerence voltage, those that handle the internal temperature sensor, and those that return the status of the REFA module.

The refaerence voltage of the REFA module is handled by

- REFA_setReferenceVoltage()
- REFA_enableReferenceVoltageOutput()
- REFA_disableReferenceVoltageOutput()
- REFA_enableReferenceVoltage()
- REFA_disableReferenceVoltage()

The internal temperature sensor is handled by

- REFA_disableTempSensor()
- REFA_enableTempSensor()

The status of the REFA module is handled by

- REFA_getBandgapMode()
- REFA_isBandgapActive()
- REFA_isRefGenBusy()

- REFA_isRefGenActive()
- REFA_getBufferedBandgapVoltageStatus()
- REFA_getVariableReferenceVoltageStatus()
- REFA_setReferenceVoltageOneTimeTrigger()
- REFA_setBufBandgapVoltageOneTimeTrigger()

31.3 Programming Example

The following example shows how to initialize and use the REFA API with the ADC12 module to use the internal 2.5V reference and perform a single converson on channel A0. The conversion results are stored in ADC12BMEM0. Test by applying a voltage to channel A0, then setting and running to a break point at the "__no_operation()" instruction. To view the conversion results, open an ADC12B register window in debugger and view the contents of ADC12BMEM0.

```
        //Set P1.0 as Ternary Module Function Output.
        /*
Base Address for Port 1
Select Port 1
Set Pin 0 to output Ternary Module Function, (A0, C0, VREFA-, VeREFA-).
        */
        FRGPIO_setAsPeripheralModuleFunctionOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
                FRGPIO_PORT_P1,
                FRGPIO_PIN0,
                FRGPIO_TERNARY_MODULE_FUNCTION
        );

   //If ref generator busy, WAIT
   while (REFA_isRefGenBusy(__MSP430_BASEADDRESS_REF_A__)) ;
   //Select internal ref = 2.5V
   REFA_setReferenceVoltage(__MSP430_BASEADDRESS_REF_A__,
       REFA_VREFA2_5V);
   //Internal Reference ON
   REFA_enableReferenceVoltage(__MSP430_BASEADDRESS_REF_A__);

   //Delay (~75us) for Ref to settle
   __delay_cycles(75);

        //Initialize the ADC12 Module
        /*
Base address of ADC12 Module
Use internal ADC12 bit as sample/hold signal to start conversion
USE MODOSC 5MHZ Digital Oscillator as clock source
Use default clock divider/pre-divider of 1
Map to internal channel 0
        */
        ADC12B_init(__MSP430_BASEADDRESS_ADC12_B__,
                ADC12B_SAMPLEHOLDSOURCE_SC,
                ADC12B_CLOCKSOURCE_ADC12OSC,
                ADC12B_CLOCKDIVIDER_1,
                ADC12B_CLOCKPREDIVIDER__1,
                ADC12B_MAPINTCH0);

        //Enable the ADC12B module
   ADC12B_enable(__MSP430_BASEADDRESS_ADC12_B__);

   /*
Base address of ADC12B Module
For memory buffers 0-7 sample/hold for 64 clock cycles
```

```
For memory buffers 8-15 sample/hold for 4 clock cycles (default)
Disable Multiple Sampling
    */
   ADC12B_setupSamplingTimer(__MSP430_BASEADDRESS_ADC12_B__,
       ADC12B_CYCLEHOLD_64_CYCLES,
       ADC12B_CYCLEHOLD_4_CYCLES,
       ADC12B_MULTIPLESAMPLESDISABLE);

   //Configure Memory Buffer
   /*
Base address of the ADC12B Module
Configure memory buffer 0
Map input A0 to memory buffer 0
Vref+ = Vref+ (INT)
Vref- = AVss
    */
   ADC12B_memoryConfigure(__MSP430_BASEADDRESS_ADC12_B__,
       ADC12B_MEMORY_0,
       ADC12B_INPUT_A0,
       ADC12B_VREFAPOS_INTBUF_VREFANEG_VSS,
       ADC12B_NOTENDOFSEQUENCE);

   while (1)
   {
       //Enable/Start sampling and conversion
       /*
Base address of ADC12B Module
Start the conversion into memory buffer 0
Use the single-channel, single-conversion mode
        */
       ADC12B_startConversion(__MSP430_BASEADDRESS_ADC12_B__,
           ADC12B_MEMORY_0,
           ADC12B_SINGLECHANNEL);

       //Poll for interrupt on memory buffer 0
       while (!ADC12B_getInterruptStatus(__MSP430_BASEADDRESS_ADC12_B__,
                  0,
                  ADC12IFG0));

       //SET BREAKPOINT HERE
       __no_operation();
   }
```

32 Real-Time Clock (RTC)

32.1 Introduction

The Real Time Clock (RTC) API provides a set of functions for using the MSP430Ware RTC modules. Functions are provided to calibrate the clock, initialize the RTC modules in Calendar mode, and setup conditions for, and enable, interrupts for the RTC modules. If an RTC_A module is used, then Counter mode may also be intialized, as well as prescale counters.

The RTC module provides the ability to keep track of the current time and date in calendar mode, or can be setup as a 32-bit counter (RTC_A Only).

The RTC module generates multiple interrupts. There are 2 interrupts that can be defined in calendar mode, and 1 interrupt in counter mode for counter overflow, as well as an interrupt for each prescaler.

This driver is contained in `driverlib/5xx_6xx/rtc.c`, with `driverlib/5xx_6xx/rtc.h` containing the API definitions for use by applications.

32.2 API Functions

The RTC API is broken into 4 groups of functions: clock settings, calender mode, counter mode, and interrupt condition setup and enable functions.

The RTC clock settings are handled by

- RTC_startClock
- RTC_holdClock
- RTC_setCalibrationFrequency
- RTC_setCalibrationData

The RTC Calender Mode is initialized and setup by

- RTC_calenderInit
- RTC_getCalenderTime
- RTC_getPrescaleValue
- RTC_setPrescaleValue

The RTC Counter Mode is initialized and setup by (Available in RTC_A Only)

- RTC_counterInit
- RTC_getCounterValue
- RTC_setCounterValue

- RTC_counterPrescaleInit
- RTC_counterPrescaleHold
- RTC_counterPrescaleStart
- RTC_getPrescaleValue
- RTC_setPrescaleValue

The RTC interrupts are handled by

- RTC_setCalenderAlarm
- RTC_setCalenderEvent
- RTC_definePrescaleEvent
- RTC_enableInterrupt
- RTC_disableInterrupt
- RTC_getInterruptStatus
- RTC_clearInterrupt

The following API are available in RTC_B Only

- RTC_convertBCDToBinary
- RTC_convertBinaryToBCD

32.3 Programming Example

The following example shows how to initialize and use the RTC API to setup Calender Mode with the current time and various interrupts.

```
  //Initialize Calendar Mode of RTC
    /*
 Base Address of the RTC_A
 Pass in current time, intialized above
 Use BCD as Calendar Register Format
     */
    RTC_calendarInit(__MSP430_BASEADDRESS_RTC__,
        currentTime,
        RTC_FORMAT_BCD);

    //Setup Calendar Alarm for 5:00pm on the 5th day of the week.
    //Note: Does not specify day of the week.
    RTC_setCalendarAlarm(__MSP430_BASEADDRESS_RTC__,
        0x00,
        0x17,
        RTC_ALARMCONDITION_OFF,
        0x05);

    //Specify an interrupt to assert every minute
    RTC_setCalendarEvent(__MSP430_BASEADDRESS_RTC__,
        RTC_CALENDAREVENT_MINUTECHANGE);

    //Enable interrupt for RTC Ready Status, which asserts when the RTC
    //Calendar registers are ready to read.
    //Also, enable interrupts for the Calendar alarm and Calendar event.
    RTC_enableInterrupt(__MSP430_BASEADDRESS_RTC__,
        RTCRDYIE + RTCTEVIE + RTCAIE);
```

```
//Start RTC Clock
RTC_startClock(__MSP430_BASEADDRESS_RTC__);

//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

33 SFR-SYS Modules

33.1 Introduction

The Special Function Registers & System Control (SFR_SYS) API provides a set of functions for using the MSP430Ware SFR and SYS modules. Functions are provided to enable interrupts, control the ~RST/NMI pin, control various SYS controls, setup the BSL, and control the JTAG Mailbox.

The SFR_SYS module can enable interrupts to be generated from other peripherals of the device.

This driver is contained in `driverlib/5xx_6xx/sfr_sys.c`, with `driverlib/5xx_6xx/sfr_sys.h` containing the API definitions for use by applications.

33.2 API Functions

The SFR_SYS API is broken into 5 groups: the SFR interrupts, the SFR ~RST/NMI pin control, the various SYS controls, the BSL controls, and the JTAG mailbox controls.

The SFR interrupts are handled by

- SFR_enableInterrupt
- SFR_disableInterrupt
- SFR_getInterruptStatus
- SFR_clearInterrupt

The SFR ~RST/NMI pin is controlled by

- SFR_setResetPinPullResistor
- SFR_setNMIEdge
- SFR_setResetNMIPinFunction

The various SYS controls are handled by

- SYS_enableDedicatedJTAGPins
- SYS_getBSLEntryIndication
- SYS_enablePMMAccessProtect
- SYS_enableRAMBasedInterruptVectors
- SYS_disableRAMBasedInterruptVectors

The BSL controls are handled by

- SYS_enableBSLProtect
- SYS_disableBSLProtect
- SYS_disableBSLMemory
- SYS_enableBSLMemory
- SYS_setRAMAssignedToBSL
- SYS_setBSLSize

The JTAG Mailbox controls are handled by

- SYS_JTAGMailboxInit
- SYS_getJTAGMailboxFlagStatus
- SYS_getJTAGInboxMessage16Bit
- SYS_getJTAGInboxMessage32Bit
- SYS_setJTAGOutgoingMessage16Bit
- SYS_setJTAGOutgoingMessage32Bit
- SYS_clearJTAGMailboxFlagStatus

33.3 Programming Example

The following example shows how to initialize and use the SFR API

```
do
  {
        // Clear XT2,XT1,DCO fault flags
        UCS_clearFaultFlag (__MSP430_BASEADDRESS_UCS__,
                                                UCS_XT2OFFG + UCS_XT1HFOFFG +
                                                UCS_XT1LFOFFG + UCS_DCOFFG
                                                );

        // Clear SFR Fault Flag
    SFR_clearInterrupt(__MSP430_BASEADDRESS_SFR__,
                       OFIFG);

    // Test oscillator fault flag
  }while (SFR_getInterruptStatus(__MSP430_BASEADDRESS_SFR__,OFIFG));
```

34 Synchronous Peripheral Interface (SPI)

34.1 Introduction

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame.

This library provides the API for handling a 3-wire SPI communication

The SPI module can be configured as either a master or a slave device.

The SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock.

This driver is contained in `driverlib/5xx_6xx/spi.c`, with `driverlib/5xx_6xx/spi.h` containing the API definitions for use by applications.

34.2 API Functions

To use the module as a master, the user must call SPI_masterInit() to configure the SPI Master. This is followed by enabling the SPI module using SPI_enable(). The interrupts are then enabled (if needed). **It** is recommended to enable the SPI module before enabling the interrupts. A data transmit is then initiated using SPI_transmitData() and tehn when the receive flag is set, the received data is read using SPI_receiveData() and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using SPI_slaveInit() and this is followed by enabling the module using SPI_enable(). Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using SPI_transmitData() and this is followed by a data reception by SPI_receiveData()

The SPI API is broken into 3 groups of functions: those that deal with status and initialization, those that handle data, and those that manage interrupts.

The status and initialization of the SPI module are managed by

- SPI_masterInit
- SPI_slaveInit
- SPI_disable
- SPI_enable
- SPI_masterChangeClock
- SPI_isBusy

Data handling is done by

- SPI_transmitData

- SPI_receiveData

Interrupts from the SPI module are managed using

- SPI_disableInterrupt
- SPI_enableInterrupt
- SPI_getInterruptStatus
- SPI_clearInterruptFlag

DMA related

- SPI_getReceiveBufferAddressForDMA
- SPI_getTransmitBufferAddressForDMA

34.3 Programming Example

The following example shows how to use the SPI API to configure the SPI module as a master device, and how to do a simple send of data.

```
  //Initialize Master
  returnValue = SPI_masterInit(USCI_A0_BASE, SMCLK, CLK_getSMClk(),
                               SPICLK, MSB_FIRST,
                               CLOCK_POLARITY_INACTIVITYHIGH
                              );

  if(STATUS_FAIL == returnValue)
  {
    return;
  }

  //Enable SPI module
  SPI_enable(USCI_A0_BASE);

  //Enable Receive interrupt
  SPI_enableInterrupt(USCI_A0_BASE, UCRXIE);

  //Configure port pins to reset slave

  // Wait for slave to initialize
  __delay_cycles(100);

  // Initialize data values
  transmitData = 0x00;

  // USCI_A0 TX buffer ready?
  while (!SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

  //Transmit Data to slave
  SPI_transmitData(USCI_A0_BASE, transmitData);

  // CPU off, enable interrupts
  __bis_SR_register(LPM0_bits + GIE);
}

//******************************************************************************
//
// This is the USCI_B0 interrupt vector service routine.
//
```

```
//******************************************************************************
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    // Vector 2 - RXIFG
    case 2:
    // USCI_A0 TX buffer ready?
    while (!SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

    receiveData = SPI_receiveData(USCI_A0_BASE);

    // Increment data
    transmitData++;

    // Send next value
    SPI_transmitData(USCI_A0_BASE, transmitData);

    //Delay between transmissions for slave to process information
    __delay_cycles(40);

    break;
     default: break;
  }
}
```

35 Timer

35.1 Introduction

Timer is a 16-bit timer/counter with multiple capture/compare registers. Timer can support multiple capture/compares, PWM outputs, and interval timing. Timer also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

This peripheral API handles Timer A and Timer B handware peripheral.

Timer features include:

- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Up to seven configurable capture/compare registers
- Configurable outputs with pulse width modulation (PWM) capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer interrupts

Timer can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

Timer Interrupts may be generated on counter overflow conditions and during capture compare events.

The timer may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with Timer_initCompare() and the necessary parameters. The PWM may be customized by selecting a desired timer mode (continuous/up/upDown), duty cycle, output mode, timer period etc. The library also provides a simpler way to generate PWM using Timer_generatePWM() API. However the level of customization and the kinds of PWM generated are limited in this API. Depending on how complex the PWM is and what level of customization is required, the user can use Timer_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The timer API provides a set of functions for dealing with the timer module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured.

This driver is contained in `driverlib/5xx_6xx/timer.c`, with `driverlib/5xx_6xx/timer.h` containing the API definitions for use by applications.

35.2 API Functions

The timer API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

Timer configuration and initialization is handled by

- Timer_startContinousMode(),
- Timer_startUpMode(),
- Timer_startUpDownMode(),
- Timer_initCapture(),
- Timer_initCompare(),
- Timer_clear(),
- Timer_stop()

Timer outputs are handled by

- Timer_getSynchronizedCaptureCompareInput(),
- Timer_getOutputForOutputModeOutBitValue(),
- Timer_setOutputForOutputModeOutBitValue(),
- Timer_generatePWM()
- Timer_getCaptureCompareCount()
- Timer_setCompareValue()

The interrupt handler for the Timer interrupt is managed with

- Timer_enableInterrupt(),
- Timer_disableInterrupt(),
- Timer_getInterruptStatus(),
- Timer_enableCaptureCompareInterrupt(),
- Timer_disableCaptureCompareInterrupt(),
- Timer_getCaptureCompareInterruptStatus(),
- Timer_clearCaptureCompareInterruptFlag()
- Timer_clearTimerInterruptFlag()

35.3 Programming Example

The following example shows some timer operations using the APIs

```
{

        //Set P1.0 to output direction
    GPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
        GPIO_PORT_P1,
        GPIO_PIN0
        );
```

```
    //Start timer in continuous mode sourced by SMCLK
    Timer_startContinousMode( __MSP430_BASEADDRESS_T1A3__,
        TIMER_CLOCKSOURCE_SMCLK,
        TIMER_CLOCKSOURCE_DIVIDER_1,
        TIMER_TAIE_INTERRUPT_DISABLE,
        TIMER_DO_CLEAR
        );

    //Initiaze compare mode
    Timer_initCompare(__MSP430_BASEADDRESS_T1A3__,
        TIMER_CAPTURECOMPARE_REGISTER_0,
        TIMER_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_OUTPUTMODE_OUTBITVALUE,
        COMPARE_VALUE
        );

    //Enter LPM0, enable interrupts
    __bis_SR_register(LPM0_bits + GIE);

    //For debugger
    __no_operation();
}

//------------------------------------------------------------------------------
//
//This is the Timer A0 interrupt vector service routine.
//
//------------------------------------------------------------------------------
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR (void)
{
    //Toggle P1.0
    GPIO_toggleOutputOnPin(
        __MSP430_BASEADDRESS_PORT1_R__,
        GPIO_PORT_P1,
        GPIO_PIN0
        );

    //Add Offset to CCR0
    Timer_setCompareValue(__MSP430_BASEADDRESS_T1A3__,
        TIMER_CAPTURECOMPARE_REGISTER_0,
        COMPARE_VALUE
        );
}
```

36 TimerA

36.1 Introduction

TimerA is a 16-bit timer/counter with multiple capture/compare registers. TimerA can support multiple capture/compares, PWM outputs, and interval timing. TimerA also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

This peripheral API handles Timer A hardware peripheral.

TimerA features include:

- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Up to seven configurable capture/compare registers
- Configurable outputs with pulse width modulation (PWM) capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer interrupts

TimerA can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

TimerA Interrupts may be generated on counter overflow conditions and during capture compare events.

The timerA may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TimerA_initCompare() and the necessary parameters. The PWM may be customized by selecting a desired timer mode (continuous/up/upDown), duty cycle, output mode, timer period etc. The library also provides a simpler way to generate PWM using TimerA_generatePWM() API. However the level of customization and the kinds of PWM generated are limited in this API. Depending on how complex the PWM is and what level of customization is required, the user can use TimerA_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The timerA API provides a set of functions for dealing with the timerA module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured.

This driver is contained in `driverlib/5xx_6xx/timera.c`, with `driverlib/5xx_6xx/timera.h` containing the API definitions for use by applications.

36.2 API Functions

The timerA API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TimerA configuration and initialization is handled by

- TimerA_startCounter(),
- TimerA_configureContinuousMode(),
- TimerA_configureUpMode(),
- TimerA_configureUpDownMode(),
- TimerA_startContinuousMode(),
- TimerA_startUpMode(),
- TimerA_startUpDownMode(),
- TimerA_initCapture(),
- TimerA_initCompare(),
- TimerA_clear(),
- TimerA_stop()

TimerA outputs are handled by

- TimerA_getSynchronizedCaptureCompareInput(),
- TimerA_getOutputForOutputModeOutBitValue(),
- TimerA_setOutputForOutputModeOutBitValue(),
- TimerA_generatePWM()
- TimerA_getCaptureCompareCount()
- TimerA_setCompareValue()

The interrupt handler for the TimerA interrupt is managed with

- TimerA_enableInterrupt(),
- TimerA_disableInterrupt(),
- TimerA_getInterruptStatus(),
- TimerA_enableCaptureCompareInterrupt(),
- TimerA_disableCaptureCompareInterrupt(),
- TimerA_getCaptureCompareInterruptStatus(),
- TimerA_clearCaptureCompareInterruptFlag()
- TimerA_clearTimerInterruptFlag()

36.3 Programming Example

The following example shows some timerA operations using the APIs

```
{       //Start TimerA
    TimerA_configureUpDownMode( __MSP430_BASEADDRESS_T1A3__,
        TIMERA_CLOCKSOURCE_SMCLK,
        TIMERA_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMERA_TAIE_INTERRUPT_DISABLE,
        TIMERA_CCIE_CCR0_INTERRUPT_DISABLE,
        TIMERA_DO_CLEAR
        );

    TimerA_startCounter( __MSP430_BASEADDRESS_T1A3__,
                TIMERA_UPDOWN_MODE
                );

    //Initialze compare registers to generate PWM1
    TimerA_initCompare(__MSP430_BASEADDRESS_T1A3__,
        TIMERA_CAPTURECOMPARE_REGISTER_1,
        TIMERA_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMERA_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE1
        );
    //Initialze compare registers to generate PWM2
    TimerA_initCompare(__MSP430_BASEADDRESS_T1A3__,
        TIMERA_CAPTURECOMPARE_REGISTER_2,
        TIMERA_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TIMERA_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE2
        );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

    //For debugger
    __no_operation();
}
```

37 timerB

37.1 Introduction

timerB is a 16-bit timer/counter with multiple capture/compare registers. timerB can support multiple capture/compares, PWM outputs, and interval timing. timerB also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

This peripheral API handles Timer B harware peripheral.

TimerB features include:

- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Up to seven configurable capture/compare registers
- Configurable outputs with pulse width modulation (PWM) capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer_B interrupts

Differences From Timer_A Timer_B is identical to Timer_A with the following exceptions:

- The length of Timer_B is programmable to be 8, 10, 12, or 16 bits
- Timer_B TBxCCRn registers are double-buffered and can be grouped
- All Timer_B outputs can be put into a high-impedance state
- The SCCI bit function is not implemented in Timer_B

TimerB can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

TimerB Interrupts may be generated on counter overflow conditions and during capture compare events.

The timerB may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with timerB_initCompare() and the necessary parameters. The PWM may be customized by selecting a desired timer mode (continuous/up/upDown), duty cycle, output mode, timer period etc. The library also provides a simpler way to generate PWM using timerB_generatePWM() API. However the level of customization and the kinds of PWM generated are limited in this API. Depending on how complex the PWM is and what level of customization is required, the user can use timerB_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The timerB API provides a set of functions for dealing with the timerB module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured.

This driver is contained in `driverlib/5xx_6xx/timerB.c`, with `driverlib/5xx_6xx/timerB.h` containing the API definitions for use by applications.

37.2 API Functions

The timerB API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TimerB configuration and initialization is handled by

- TimerB_startCounter(),
- TimerB_configureContinuousMode(),
- TimerB_configureUpMode(),
- TimerB_configureUpDownMode(),
- TimerB_startContinuousMode(),
- TimerB_startUpMode(),
- TimerB_startUpDownMode(),
- TimerB_initCapture(),
- TimerB_initCompare(),
- TimerB_clear(),
- TimerB_stop()
- TimerB_initCompareLatchLoadEvent(),
- TimerB_selectLatchingGroup(),
- TimerB_selectCounterLength(),

TimerB outputs are handled by

- TimerB_getSynchronizedCaptureCompareInput(),
- TimerB_getOutputForOutputModeOutBitValue(),
- TimerB_setOutputForOutputModeOutBitValue(),
- TimerB_generatePWM()
- TimerB_getCaptureCompareCount()
- TimerB_setCompareValue()

The interrupt handler for the TimerB interrupt is managed with

- TimerB_enableInterrupt(),
- TimerB_disableInterrupt(),

- TimerB_getInterruptStatus(),
- TimerB_enableCaptureCompareInterrupt(),
- TimerB_disableCaptureCompareInterrupt(),
- TimerB_getCaptureCompareInterruptStatus(),
- TimerB_clearCaptureCompareInterruptFlag()
- TimerB_clearTimerInterruptFlag()

37.3 Programming Example

The following example shows some timerB operations using the APIs

```
{       //Start timerB
        TimerB_configureUpMode(   __MSP430_BASEADDRESS_T0B7__,
          TIMERB_CLOCKSOURCE_SMCLK,
          TIMERB_CLOCKSOURCE_DIVIDER_1,
          511,
          TIMERB_TBIE_INTERRUPT_DISABLE,
          TIMERB_CCIE_CCR0_INTERRUPT_DISABLE,
          TIMERB_DO_CLEAR
          );

        TimerB_startCounter(   __MSP430_BASEADDRESS_T0B7__,
                TIMERB_UP_MODE
                );



        //Initialize compare mode to generate PWM1
        TimerB_initCompare(__MSP430_BASEADDRESS_T0B7__,
          TIMERB_CAPTURECOMPARE_REGISTER_1,
          TIMERB_CAPTURECOMPARE_INTERRUPT_DISABLE,
          TIMERB_OUTPUTMODE_RESET_SET,
          383
          );

        //Initialize compare mode to generate PWM2
        TimerB_initCompare(__MSP430_BASEADDRESS_T0B7__,
          TIMERB_CAPTURECOMPARE_REGISTER_2,
          TIMERB_CAPTURECOMPARE_INTERRUPT_ENABLE,
          TIMERB_OUTPUTMODE_RESET_SET,
          128
          );
}
```

38 timerD

38.1 Introduction

Timer_D is a 16-bit timer/counter with multiple capture/compare registers. Timer_D can support multiple capture/compares, interval timing, and PWM outputs both in general and high resolution modes. Timer_D also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions, from each of the capture/compare registers.

This peripheral API handles Timer D handware peripheral.

timerD features include:

- Asynchronous 16-bit timer/counter with four operating modes and four selectable lengths
- Selectable and configurable clock source
- Configurable capture/compare registers
- Controlling rising and falling PWM edges by combining two neighbor TDCCR registers in one compare channel output
- Configurable outputs with PWM capability
- High-resolution mode with a fine clock frequency up to 16 times the timer input clock frequency
- Double-buffered compare registers with synchronized loading
- Interrupt vector register for fast decoding of all Timer_D interrupts

Differences From Timer_B Timer_D is identical to Timer_B with the following exceptions:

- Timer_D supports high-resolution mode.
- Timer_D supports the combination of two adjacent TDCCRx registers in one capture/compare channel.
- Timer_D supports the dual capture event mode.
- Timer_D supports external fault input, external clear input, and signal. See the TEC chapter for detailed information.
- Timer_D can synchronize with a second timer instance when available. See the TEC chapter for detailed information.

timerD can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

timerD Interrupts may be generated on counter overflow conditions and during capture compare events.

The timerD may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TimerD_initCompare() and the necessary parameters. The

PWM may be customized by selecting a desired timer mode (continuous/up/upDown), duty cycle, output mode, timer period etc. The library also provides a simpler way to generate PWM using TimerD_generatePWM() API. However the level of customization and the kinds of PWM generated are limited in this API. Depending on how complex the PWM is and what level of customization is required, the user can use TimerD_generatePWM() or a combination of TimerD_initCompare() and timer start APIs

The TimerD API provides a set of functions for dealing with the TimerD module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured.

This driver is contained in `driverlib/5xx_6xx/timerd.c`, with `driverlib/5xx_6xx/timerd.h` containing the API definitions for use by applications.

38.2 API Functions

The TimerD API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TimerD configuration and initialization is handled by

- TimerD_startCounter(),
- TimerD_configureContinuousMode(),
- TimerD_configureUpMode(),
- TimerD_configureUpDownMode(),
- TimerD_startContinuousMode(),
- TimerD_startUpMode(),
- TimerD_startUpDownMode(),
- TimerD_initCapture(),
- TimerD_initCompare(),
- TimerD_clear(),
- TimerD_stop(),
- TimerD_configureHighResGeneratorInFreeRunningMode(),
- TimerD_configureHighResGeneratorInRegulatedMode(),
- TimerD_combineTDCCRToGeneratePWM(),
- TimerB_selectLatchingGroup(),
- TimerD_selectCounterLength(),
- TimerD_initCompareLatchLoadEvent(),
- TimerD_disableHighResFastWakeup(),
- TimerD_enableHighResFastWakeup(),
- TimerD_disableHighResClockEnhancedAccuracy(),
- TimerD_enableHighResClockEnhancedAccuracy(),

- TimerD_DisableHighResGeneratorForceON(),
- TimerD_EnableHighResGeneratorForceON(),
- TimerD_selectHighResCoarseClockRange(),
- TimerD_selectHighResClockRange()

TimerD outputs are handled by

- TimerD_getSynchronizedCaptureCompareInput(),
- TimerD_getOutputForOutputModeOutBitValue(),
- TimerD_setOutputForOutputModeOutBitValue(),
- TimerD_generatePWM(),
- TimerD_getCaptureCompareCount(),
- TimerD_setCompareValue(),
- TimerD_getCaptureCompareLatchCount(),
- TimerD_getCaptureCompareInputSignal()

The interrupt handler for the TimerD interrupt is managed with

- TimerD_enableTimerInterrupt(),
- TimerD_disableTimerInterrupt(),
- TimerD_getTimerInterruptStatus(),
- TimerD_enableCaptureCompareInterrupt(),
- TimerD_disableCaptureCompareInterrupt(),
- TimerD_getCaptureCompareInterruptStatus(),
- TimerD_clearCaptureCompareInterruptFlag()
- TimerD_clearTimerInterruptFlag(),
- TimerD_enableHighResInterrupt(),
- TimerD_disableTimerInterrupt(),
- TimerD_getHighResInterruptStatus(),
- TimerD_clearHighResInterruptStatus()

Timer_D High Resolution handling APIs

- TimerD_getHighResInterruptStatus(),
- TimerD_clearHighResInterruptStatus(),
- TimerD_disableHighResFastWakeup(),
- TimerD_enableHighResFastWakeup(),
- TimerD_disableHighResClockEnhancedAccuracy(),
- TimerD_enableHighResClockEnhancedAccuracy(),
- TimerD_DisableHighResGeneratorForceON(),
- TimerD_EnableHighResGeneratorForceON(),
- TimerD_selectHighResCoarseClockRange(),
- TimerD_selectHighResClockRange(),
- TimerD_configureHighResGeneratorInFreeRunningMode(),
- TimerD_configureHighResGeneratorInRegulatedMode()

38.3 Programming Example

The following example shows some TimerD operations using the APIs

```
{       //Start TimerD
    TimerD_configureUpDownMode( __MSP430_BASEADDRESS_T1A3__,
        TimerD_CLOCKSOURCE_SMCLK,
        TimerD_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TimerD_TAIE_INTERRUPT_DISABLE,
        TimerD_CCIE_CCR0_INTERRUPT_DISABLE,
        TimerD_DO_CLEAR
        );

    TimerD_startCounter( __MSP430_BASEADDRESS_T1A3__,
                TimerD_UPDOWN_MODE
                );

    //Initialze compare registers to generate PWM1
    TimerD_initCompare(__MSP430_BASEADDRESS_T1A3__,
        TimerD_CAPTURECOMPARE_REGISTER_1,
        TimerD_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TimerD_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE1
        );
    //Initialze compare registers to generate PWM2
    TimerD_initCompare(__MSP430_BASEADDRESS_T1A3__,
        TimerD_CAPTURECOMPARE_REGISTER_2,
        TimerD_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TimerD_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE2
        );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

    //For debugger
    __no_operation();
}
```

39 Tag Length Value

39.1 Introduction

The TLV structure is a table stored in flash memory that contains device-specific information. This table is read-only and is write-protected. It contains important information for using and calibrating the device. A list of the contents of the TLV is available in the device-specific data sheet (in the Device Descriptors section), and an explanation on its functionality is available in the MSP430x5xx/MSP430x6xx Family UserŠs Guide

This driver is contained in `driverlib/5xx_6xx/tlv.c`, with `driverlib/5xx_6xx/tlv.h` containing the API definitions for use by applications.

39.2 API Functions

The APIs that help in querying the information in the TLV structure are listed

- TLV_getInfo() This function retrieves the value of a tag and the length of the tag.
- TLV_getDeviceType() This function retrieves the unique device ID from the TLV structure.
- TLV_getMemory() The returned value is zero if the end of the memory list is reached.
- TLV_getPeripheral() The returned value is zero if the specified tag value (peripheral) is not available in the device.
- TLV_getInterrupt() The returned value is zero is the specified interrupt vector is not defined.

39.3 Programming Example

The following example shows some tlv operations using the APIs

```
struct s_TLV_Die_Record * pDIEREC;
unsigned char bDieRecord_bytes;

TLV_getInfo(TLV_TAG_DIERECORD,
                        0,
                        &bDieRecord_bytes,
                        (unsigned int **)&pDIEREC
                        );
```

40 UART

40.1 Introduction

The MSP430Ware library for UART mode features include:

- Odd, even, or non-parity
- Independent transmit and receive shift registers
- Separate transmit and receive buffer registers
- LSB-first or MSB-first data transmit and receive
- Built-in idle-line and address-bit communication protocols for multiprocessor systems
- Receiver start-edge detection for auto wake up from LPMx modes
- Status flags for error detection and suppression
- Status flags for address detection
- Independent interrupt capability for receive and transmit

The modes of operations supported by the UART and the library include

- UART mode
- Idle-line multiprocessor mode
- Address-bit multiprocessor mode
- UART mode with automatic baud-rate detection

In UART mode, the USCI transmits and receives characters at a bit rate asynchronous to another device. Timing for each character is based on the selected baud rate of the USCI. The transmit and receive functions use the same baud-rate frequency.

This driver is contained in `driverlib/5xx_6xx/uart.c`, with `driverlib/5xx_6xx/uart.h` containing the API definitions for use by applications.

40.2 API Functions

The UART API provides the set of functions required to implement an interrupt driven UART driver. The UART initialization with the various modes and features is done by the UART_init(). At the end of this fucntion UART is initialized and stays disabled. UART_enable() enables the UART and the module is now ready for transmit and receive. It is recommended to iniailize the UART via UART_init(), enable the required interrupts and then enable UART via UART_enable().

The UART API is broken into three groups of functions: those that deal with configuration and control of the UART modules, those used to send and receive data, and those that deal with interrupt handling and those dealing with DMA.

Configuration and control of the UART are handled by the

- UART_init()
- UART_initAdvance()
- UART_enable()
- UART_disable()
- UART_setDormant()
- UART_resetDormant()

Sending and receiving data via the UART is handled by the

- UART_transmitData()
- UART_receiveData()
- UART_transmitAddress()
- UART_transmitBreak()

Managing the UART interrupts and status are handled by the

- UART_enableInterrupt()
- UART_disableInterrupt()
- UART_getInterruptStatus()
- UART_clearInterruptFlag()
- UART_queryStatusFlags()

DMA related

- UART_getReceiveBufferAddressForDMA()
- UART_getTransmitBufferAddressForDMA()

40.3 Programming Example

The following example shows how to use the UART API to initialize the UART, transmit characters, and receive characters.

```
if ( STATUS_FAIL == UART_init (__MSP430_BASEADDRESS_USCI_A0__,
                               UART_CLOCKSOURCE_SMCLK,
                               UCS_getSMCLK(__MSP430_BASEADDRESS_UCS__),
                               BAUD_RATE,
                               UART_NO_PARITY,
                               UART_LSB_FIRST,
                               UART_ONE_STOP_BIT,
                               UART_MODE,
                               UART_OVERSAMPLING_BAUDRATE_GENERATION ))
  {
        return;
  }

  //Enable UART module for operation
  UART_enable (__MSP430_BASEADDRESS_USCI_A0__);

  //Enable Receive Interrupt
  UART_enableInterrupt (__MSP430_BASEADDRESS_USCI_A0__,
                        UCRXIE);
```

```
  //Transmit data
  UART_transmitData(__MSP430_BASEADDRESS_USCI_A0__,
                    transmitData++
                      );

  // Enter LPM3, interrupts enabled
  __bis_SR_register(LPM3_bits + GIE);
  __no_operation();
}

//******************************************************************************
//
// This is the USCI_A0 interrupt vector service routine.
//
//******************************************************************************
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    // Vector 2 - RXIFG
     case 2:
        // Echo back RXed character, confirm TX buffer is ready first

        // USCI_A0 TX buffer ready?
        while (!UART_interruptStatus(__MSP430_BASEADDRESS_USCI_A0__,
                                     UCTXIFG)
               );

        //Receive echoed data
        receivedData = UART_receiveData(__MSP430_BASEADDRESS_USCI_A0__);

        //Transmit next data
        UART_transmitData(__MSP430_BASEADDRESS_USCI_A0__,
                          transmitData++
                            );

        break;
     default: break;
  }
}
```

41 Unified Clock System (UCS)

41.1 Introduction

The UCS is based on five available clock sources (VLO, REFO, XT1, XT2, and DCO) providing signals to three system clocks (MCLK, SMCLK, ACLK). Different low power modes are achieved by turning off the MCLK, SMCLK, ACLK, and integrated LDO.

- VLO - Internal very-low-power low-frequency oscillator. 10 kHz (ś0.5/řC, ś4/V)
- REFO - Reference oscillator. 32 kHz (ś1%, ś3% over full temp range)
- XT1 (LFXT1, HFXT1) - Ultra-low-power oscillator, compatible with low-frequency 32768-Hz watch crystals and with standard XT1 (LFXT1, HFXT1) crystals, resonators, or external clock sources in the 4-MHz to 32-MHz range, including digital inputs. Most commonly used as 32-kHz watch crystal oscillator.
- XT2 - Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 4-MHz to 32-MHz range, including digital inputs.
- DCO - Internal digitally-controlled oscillator (DCO) that can be stabilized by a frequency lock loop (FLL) that sets the DCO to a specified multiple of a reference frequency.

System Clocks and Functionality on the MSP430 MCLK Master Clock Services the CPU. Commonly sourced by DCO. Is avaiable in Active mode only SMCLK Subsystem Master Clock Services 'fast' system peripherals. Commonly sourced by DCO. Is available in Active mode, LPM0 and LPM1 ACLK Auxiliary Clock Services 'slow' system peripherals. Commonly used for 32-kHz signal.Is available in Active mode, LPM0 to LPM3

System clocks of the MSP430x5xx generation are automatically enabled, regardless of the LPM mode of operation, if they are required for the proper operation of the peripheral module that they source. This additional flexibility of the UCS, along with improved fail-safe logic, provides a robust clocking scheme for all applications.

Fail-Safe logic The UCS fail-safe logic plays an important part in providing a robust clocking scheme for MSP430x5xx and MSP430x6xx applications. This feature hinges on the ability to detect an oscillator fault for the XT1 in both low- and high-frequency modes (XT1LFOFFG and XT1HFOFFG respectively), the high-frequency XT2 (XT2OFFG), and the DCO (DCOFFG). These flags are set and latched when the respective oscillator is enabled but not operating properly; therefore, they must be explicitly cleared in software

The oscillator fault flags on previous MSP430 generations are not latched and are asserted only as long as the failing condition exists. Therefore, an important difference between the families is that the fail-safe behavior in a 5xx-based MSP430 remains active until both the OFIFG and the respective fault flag are cleared in software.

This fail-safe behavior is implemented at the oscillator level, at the system clock level and, consequently, at the module level. Some notable highlights of this behavior are described below. For the full description of fail-safe behavior and conditions, see the MSP430x5xx/MSP430x6xx Family UserŠs Guide (SLAU208).

- Low-frequency crystal oscillator 1 (LFXT1) The low-frequency (32768 Hz) crystal oscillator is the default reference clock to the FLL. An asserted XT1LFOFFG switches the FLL reference from the failing LFXT1 to the internal 32-kHz REFO. This can influence the DCO accuracy, because the FLL crystal ppm specification is typically tighter than the REFO accuracy over temperature and voltage of ś3%.
- System Clocks (ACLK, SMCLK, MCLK) A fault on the oscillator that is sourcing a system clock switches the source from the failing oscillator to the DCO oscillator (DCOCLKDIV). This is true for all clock sources except the LFXT1. As previously described, a fault on the LFXT1 switches the source to the REFO. Since ACLK is the active clock in LPM3 there is a notable difference in the LPM3 current consumption when the REFO is the clock source ($\sim$3 μA active) versus the LFXT1 ($\sim$300 nA active).
- Modules (WDT_A) In watchdog mode, when SMCLK or ACLK fails, the clock source defaults to the VLOCLK.

This driver is contained in `driverlib/5xx_6xx/ucs.c`, with `driverlib/5xx_6xx/ucs.h` containing the API definitions for use by applications.

41.2 API Functions

The UCS API is broken into three groups of functions: those that deal with clock configuration and control

General UCS configuration and initialization is handled by

- UCS_clockSignalInit(),
- UCS_initFLLSettle(),
- UCS_enableClockRequest(),
- UCS_disableClockRequest(),
- UCS_SMCLKOff(),
- UCS_SMCLKOn()

External crystal specific configuration and initialization is handled by

- UCS_setExternalClockSource(),
- UCS_LFXT1Start(),
- UCS_HFXT1Start(),
- UCS_bypassXT1(),
- UCS_LFXT1StartWithTimeout(),
- UCS_HFXT1StartWithTimeout(),
- UCS_bypassXT1WithTimeout(),
- UCS_XT1Off(),
- UCS_XT2Start(),
- UCS_XT2Off(),
- UCS_bypassXT2(),
- UCS_XT2StartWithTimeout(),

- UCS_bypassXT2WithTimeout()
- UCS_clearAllOscFlagsWithTimeout()

UCS_setExternalClockSource must be called if an external crystal XT1 or XT2 is used and the user intends to call UCS_getMCLK, UCS_getSMCLK or UCS_getACLK APIs. If not, it is not necessary to invoke this API.

Failure to invoke UCS_clockSignalInit() sets the clock signals to the default modes ACLK default mode - UCS_XT1CLK_SELECT SMCLK default mode - UCS_DCOCLKDIV_SELECT MCLK default mode - UCS_DCOCLKDIV_SELECT

Also fail-safe mode behavior takes effect when a slected mode fails.

The status and configuration query are done by

- UCS_faultFlagStatus(),
- UCS_clearFaultFlag(),
- UCS_getACLK(),
- UCS_getSMCLK(),
- UCS_getMCLK()

41.3 Programming Example

The following example shows some UCS operations using the APIs

```
// Set DCO FLL reference = REFO
  UCS_clockSignalInit(
                       __MSP430_BASEADDRESS_UCS__,
                       UCS_FLLREF,
                       UCS_REFOCLK_SELECT,
                       UCS_CLOCK_DIVIDER_1
                        );
  // Set ACLK = REFO
 UCS_clockSignalInit(
                       __MSP430_BASEADDRESS_UCS__,
                       UCS_ACLK,
                       UCS_REFOCLK_SELECT,
                       UCS_CLOCK_DIVIDER_1
                        );

  // Set Ratio and Desired MCLK Frequency  and initialize DCO
  UCS_initFLLSettle(
                        __MSP430_BASEADDRESS_UCS__,
                        UCS_MCLK_DESIRED_FREQUENCY_IN_KHZ,
                        UCS_MCLK_FLLREF_RATIO
                      );

  //Verify if the Clock settings are as expected
  clockValue = UCS_getSMCLK (__MSP430_BASEADDRESS_UCS__);

  while(1);
```

42 WatchDog Timer (WDT)

42.1 Introduction

The Watchdog Timer (WDT) API provides a set of functions for using the MSP430Ware WDT modules. Functions are provided to initialize the Watchdog in either timer interval mode, or watchdog mode, with selectable clock sources and dividers to define the timer interval.

The WDT module can generate only 1 kind of interrupt in timer interval mode. If in watchdog mode, then the WDT module will assert a reset once the timer has finished.

This driver is contained in `driverlib/5xx_6xx/wdt.c`, with `driverlib/5xx_6xx/wdt.h` containing the API definitions for use by applications.

42.2 API Functions

The WDT API is one group that controls the WDT module.

- WDT_hold
- WDT_start
- WDT_clearCounter
- WDT_watchdogTimerInit
- WDT_intervalTimerInit

42.3 Programming Example

The following example shows how to initialize and use the WDT API to interrupt about every 32 ms, toggling the LED in the ISR.

```
//Initialize WDT module in timer interval mode,
  //with SMCLK as source at an interval of 32 ms.
  WDT_intervalTimerInit(__MSP430_BASEADDRESS_WDT_A__,
      WDT_CLOCKSOURCE_SMCLK,
      WDT_CLOCKDIVIDER_32K);

  //Enable Watchdog Interupt
  SFR_enableInterrupt(__MSP430_BASEADDRESS_SFR__,
      WDTIE);

  //Set P1.0 to output direction
  GPIO_setAsOutputPin(__MSP430_BASEADDRESS_PORT1_R__,
      GPIO_PORT_P1,
      GPIO_PIN0
      );
```

```
//Enter LPM0, enable interrupts
__bis_SR_register(LPM0_bits + GIE);
//For debugger
__no_operation();
```

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