




MSP430® Peripheral Driver Library for FR5xx and FR6xx Devices

USER'S GUIDE

Copyright

Copyright © 2012 Texas Instruments Incorporated. All rights reserved. MSP430 and 430ware are registered trademarks of Texas Instruments. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

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



Revision Information

This is version 1.30.00.00 of this document, last updated on 2012-10-25_0 : 43 : 15_0500.

Table of Contents

Copyright	2
Revision Information	2
1 Introduction	5
2 How to create a new project that uses Driverlib	7
3 12-Bit Analog-to-Digital Converter (ADC12_B)	9
3.1 Introduction	9
3.2 API Functions	9
3.3 Programming Example	10
4 Advanced Encryption Standard (AES256)	13
4.1 Introduction	13
4.2 API Functions	13
4.3 Programming Example	14
5 Comparator (COMP_E)	15
5.1 Introduction	15
5.2 API Functions	15
5.3 Programming Example	16
6 Cyclical Redundancy Check (CRC)	19
6.1 Introduction	19
6.2 API Functions	19
6.3 Programming Example	19
7 Clock System (CS)	21
7.1 Introduction	21
7.2 API Functions	22
7.3 Programming Example	23
8 Direct Memory Access (DMA)	25
8.1 Introduction	25
8.2 API Functions	25
8.3 Programming Example	26
9 EUSCI Inter-Integrated Circuit (I2C)	27
9.1 Introduction	27
9.2 API Functions	29
9.3 Programming Example	30
10 EUSCI Synchronous Peripheral Interface (SPI)	31
10.1 Introduction	31
10.2 Functions	31
10.3 Programming Example	32
11 EUSCI UART	33
11.1 Introduction	33
11.2 API Functions	33
11.3 Programming Example	34
12 FRAM Controller	35
12.1 Introduction	35
12.2 API Functions	35
12.3 Programming Example	36
13 GPIO	37
13.1 Introduction	37
13.2 API Functions	38
13.3 Programming Example	38
14 Memory Protection Unit (MPU)	41
14.1 Introduction	41
14.2 API Functions	41
14.3 Programming Example	42
15 32-Bit Hardware Multiplier (MPY32)	43
15.1 Introduction	43
15.2 API Functions	43
15.3 Programming Example	44
16 Power Management Module (PMM)	45
16.1 Introduction	45
16.2 API Functions	45
16.3 Programming Example	45

17	Internal Reference (REF_A)	49
17.1	Introduction	49
17.2	API Functions	49
17.3	Programming Example	50
18	Real-Time Clock (RTC)	51
18.1	Introduction	51
18.2	API Functions	51
18.3	Programming Example	52
19	SFR Module	53
19.1	Introduction	53
19.2	API Functions	53
19.3	Programming Example	53
20	SYS Module	55
20.1	Introduction	55
20.2	API Functions	55
20.3	Programming Example	56
21	TIMER_A	57
21.1	Introduction	57
21.2	API Functions	58
21.3	Programming Example	58
22	TIMER_B	61
22.1	Introduction	61
22.2	API Functions	62
22.3	Programming Example	63
23	Tag Length Value	65
23.1	Introduction	65
23.2	API Functions	65
23.3	Programming Example	65
24	WatchDog Timer (WDT_A)	67
24.1	Introduction	67
24.2	API Functions	67
24.3	Programming Example	67
	IMPORTANT NOTICE	70

1 Introduction

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430 FR5xx/FR6xx 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\MSP430FR5xx_6xx\00_emptyProject\IAR" "C:\ti\msp430\MSP430ware_x_xx_xx_xx\examples\driverlib\MSP430FR5xx_6xx\00_emptyProject\CCS" or the corresponding relative path where MSP430ware is installed. The features of the emptyProject are

- Includes driverlib source files for that family 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 default 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 12-Bit Analog-to-Digital Converter (ADC12_B)

Introduction	9
API Functions	9
Programming Example	10

3.1 Introduction

The 12-Bit Analog-to-Digital (ADC12_B) API provides a set of functions for using the MSP430Ware ADC12_B modules. Functions are provided to initialize the ADC12_B modules, setup signal sources and reference voltages for each memory buffer, and manage interrupts for the ADC12_B modules.

The ADC12_B 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 ADC12_B can also generate digital values from 0 to V_{CC} 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 ADC12_B 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 ksp/s 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 $2/3 \times AV_{CC}$ 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 `adc12_b.c`, with `adc12_b.h` containing the API definitions for use by applications.

3.2 API Functions

The ADC12_B API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxiliary features of the ADC12_B.

The ADC12_B initialization and conversion functions are

- ADC12_B_init
- ADC12_B_memoryConfigure
- ADC12_B_setWindowCompAdvanced
- ADC12_B_setupSamplingTimer
- ADC12_B_disableSamplingTimer
- ADC12_B_startConversion
- ADC12_B_disableConversions
- ADC12_B_getResults
- ADC12_B_isBusy

The ADC12_B interrupts are handled by

- ADC12_B_enableInterrupt
- ADC12_B_disableInterrupt
- ADC12_B_clearInterrupt
- ADC12_B_getInterruptStatus

Auxiliary features of the ADC12_B are handled by

- ADC12_B_setResolution
- ADC12_B_setSampleHoldSignalInversion
- ADC12_B_setDataReadBackFormat
- ADC12_B_enableReferenceBurst
- ADC12_B_disableReferenceBurst
- ADC12_B_setAdcPowerMode
- ADC12_B_getMemoryAddressForDMA
- ADC12_B_enable
- ADC12_B_disable

3.3 Programming Example

The following example shows how to initialize and use the ADC12_B API to start a single channel with single conversion using an external positive reference for the ADC12_B.

```
//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
*/
ADC12_B_init(ADC12_B_BASE,
            ADC12_B_SAMPLEHOLDSOURCE_SC,
            ADC12_B_CLOCKSOURCE_ADC12OSC,
            ADC12_B_CLOCKDIVIDER_1,
            ADC12_B_CLOCKPREDIVIDER__1,
            ADC12_B_MAPINTCH0);

//Enable the ADC12_B module
ADC12_B_enable(ADC12_B_BASE);

/*
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
*/
ADC12_B_setupSamplingTimer(ADC12_B_BASE,
    ADC12_B_CYCLEHOLD_16_CYCLES,
    ADC12_B_CYCLEHOLD_4_CYCLES,
    ADC12_B_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
*/
ADC12_B_memoryConfigure(ADC12_B_BASE,
    ADC12_B_MEMORY_0,
    ADC12_B_INPUT_A0,
    ADC12_B_VREFPOS_EXTPOS_VREFNEG_VSS,
    ADC12_B_NOTENDOFSEQUENCE,
    ADC12_B_WINDOW_COMPARATOR_DISABLE,
    ADC12_B_DIFFERENTIAL_MODE_DISABLE);

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
    */
    ADC12_B_startConversion(ADC12_B_BASE,
        ADC12_B_MEMORY_0,
        ADC12_B_SINGLECHANNEL);

    //Poll for interrupt on memory buffer 0
    while (!ADC12_B_getInterruptStatus(ADC12_B_BASE,
        0,
        ADC12_B_IFG0));

    __no_operation(); // SET BREAKPOINT HERE
}

```


4 Advanced Encryption Standard (AES256)

Introduction	13
API Functions	13
Programming Example	14

4.1 Introduction

The AES256 accelerator module performs encryption and decryption of 128-bit data with 128-bit keys according to the advanced encryption standard (AES256) (FIPS PUB 197) in hardware. The AES256 accelerator features are:

- Encryption and decryption according to AES256 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
- AES256 ready interrupt flag The AES256256 accelerator module performs encryption and decryption of 128-bit data with 128-/192-/256-bit keys according to the advanced encryption standard (AES256) (FIPS PUB 197) in hardware. The AES256 accelerator features are: AES256 encryption Ú 128 bit - 168 cycles Ú 192 bit - 204 cycles Ú 256 bit - 234 cycles AES256 decryption Ú 128 bit - 168 cycles Ú 192 bit - 206 cycles Ú 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
- AES256 ready interrupt flag

This driver is contained in `aes256.c`, with `aes256.h` containing the API definitions for use by applications.

4.2 API Functions

The AES256 module APIs are

- `AES256_setCipherKey()`,
- `AES256256_setCipherKey()`,
- `AES256_encryptData()`,
- `AES256_decryptDataUsingEncryptionKey()`,
- `AES256_generateFirstRoundKey()`,
- `AES256_decryptData()`,
- `AES256_reset()`,
- `AES256_startEncryptData()`,

- AES256_startDecryptDataUsingEncryptionKey(),
- AES256_startDecryptData(),
- AES256_startGenerateFirstRoundKey(),
- AES256_getDataOut()

The AES256 interrupt handler functions

- AES256_enableInterrupt(),
- AES256_disableInterrupt(),
- AES256_clearInterruptFlag(),

4.3 Programming Example

The following example shows some AES256 operations using the APIs

```

unsigned char Data[16] =          {
                                     0x30, 0x30, 0x30, 0x30,
                                     0x30, 0x30, 0x30, 0x30,
                                     0x30, 0x30, 0x30, 0x30,
                                     0x30, 0x30, 0x30, 0x30
                                     };

unsigned char CipherKey[32] =    {
                                     0xAA, 0xBB, 0x02, 0x03,
                                     0x04, 0x05, 0x06, 0x07,
                                     0x08, 0x09, 0x0A, 0x0B,
                                     0x0C, 0x0D, 0x0E, 0x0F,
                                     0x30, 0x31, 0x32, 0x33,
                                     0x34, 0x35, 0x36, 0x37,
                                     0x30, 0x31, 0x32, 0x33,
                                     0x34, 0x35, 0x36, 0x37
                                     };

unsigned char DataAESencrypted[16];          // Encrypted data
unsigned char DataAESdecrypted[16];         // Decrypted data

// Load a cipher key to module
AES256_setCipherKey(AES256_BASE, CipherKey, Key_256BIT);

// Encrypt data with preloaded cipher key
AES256_encryptData(AES256_BASE, Data, DataAESencrypted);

// 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
AES256_decryptDataUsingEncryptionKey(AES256_BASE, DataAESencrypted, DataAESdecrypted);

```

5 Comparator (COMP_E)

Introduction	15
API Functions	15
Programming Example	16

5.1 Introduction

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

The COMP_E 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 COMP_E 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 COMP_E module also has control over the REF module to generate a reference voltage as an input.

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

This driver is contained in `comp_e.c`, with `comp_e.h` containing the API definitions for use by applications.

5.2 API Functions

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

The COMP_E initialization and output functions are

- COMP_E_init
- COMP_E_setReferenceVoltage
- COMP_E_enable
- COMP_E_disable
- COMP_E_outputValue
- COMP_E_setPowerMode

The COMP_E interrupts are handled by

- COMP_E_enableInterrupt
- COMP_E_disableInterrupt
- COMP_E_clearInterrupt
- COMP_E_getInterruptStatus
- COMP_E_interruptSetEdgeDirection
- COMP_E_interruptToggleEdgeDirection

Auxiliary features of the COMP_E are handled by

- COMP_E_enableShortOfInputs
- COMP_E_disableShortOfInputs
- COMP_E_disableInputBuffer
- COMP_E_enableInputBuffer
- COMP_E_IOSwap
- COMP_E_setReferenceAccuracy
- COMP_E_setPowerMode

5.3 Programming Example

The following example shows how to initialize and use the COMP_E 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 E module
/* Base Address of Comparator E,
   Pin CD2 to Positive(+) Terminal,
   Reference Voltage to Negative(-) Terminal,
   Normal Power Mode,
   Output Filter On with minimal delay,
   Non-Inverted Output Polarity
*/
COMP_E_init(COMP_E_BASE,
            COMP_E_INPUT2,
            COMP_E_VREF,
            COMP_E_FILTEROUTPUT_OFF,
            COMP_E_NORMALOUTPUTPOLARITY
            );

// Set the reference voltage that is being supplied to the (-) terminal
/* Base Address of Comparator E,
   Reference Voltage of 2.0 V,
   Upper Limit of 2.0*(32/32) = 2.0V,
   Lower Limit of 2.0*(32/32) = 2.0V
   Static Accuracy
*/
COMP_E_setReferenceVoltage(COMP_E_BASE,
                           COMP_E_VREFBASE2_0V,
                           32,
                           32,
                           COMP_E_ACCURACY_STATIC
                           );

//Disable Input Buffer on P1.2/CD2
/* Base Address of Comparator E,
   Input Buffer port
   Selecting the CEx input pin to the comparator
   multiplexer with the CEx bits automatically
   disables output driver and input buffer for
   that pin, regardless of the state of the
   associated CEPD.x bit
*/
COMP_E_disableInputBuffer(COMP_E_BASE,
                          COMP_E_INPUT2);
// Allow power to Comparator module
COMP_E_enable(COMP_E_BASE);
```



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


6 Cyclical Redundancy Check (CRC)

Introduction	19
API Functions	19
Programming Example	19

6.1 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 `crc.c`, with `crc.h` containing the API definitions for use by applications.

6.2 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`

6.3 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(WDT_A_BASE);
```

Cyclical Redundancy Check (CRC)

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

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

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

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

7 Clock System (CS)

Introduction	21
API Functions	22
Programming Example	23

7.1 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 `cs_a.c`, with `cs_a.h` containing the API definitions for use by applications.

7.2 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

- CS_LFXTStart
- CS_bypassLFXT
- CS_bypassLFXTWithTimeout
- CS_LFXTStartWithTimeout
- CS_LFXTOff
- CS_HFXTStart
- CS_bypassHFXT
- CS_HFXTStartWithTimeout
- CS_bypassHFXTWithTimeout
- CS_HFXTOff

- CS_VLOoff

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 LFXT or HFXT is used and the user intends to call CS_getMCLK, CS_getSMCLK or CS_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.

7.3 Programming Example

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

```
//Set DCO Frequency to 8MHz
CS_setDCOFreq(CS_BASE,CS_DCORSEL_0,CS_DCOFSEL_6);

//configure MCLK, SMCLK to be source by DCOCLK
CS_clockSignalInit(CS_BASE,CS_SMCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
CS_clockSignalInit(CS_BASE,CS_MCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
```


8 Direct Memory Access (DMA)

Introduction	25
API Functions	25
Programming Example	26

8.1 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 transferring, 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 disastrous 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 occurred, while with block or burst-block transfers, once the block is completely transferred the interrupt is asserted.

8.2 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`

8.3 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(DMA_BASE,
         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(DMA_BASE,
                 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(DMA_BASE,
                 DMA_CHANNEL_0,
                 0x1C20,
                 DMA_DIRECTION_INCREMENT);

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

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

9 EUSCI Inter-Integrated Circuit (I2C)

Introduction	27
API Functions	29
Programming Example	30

9.1 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.

9.1.1 Master Operations

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

- **EUSCI_I2C_masterInit**
- **EUSCI_I2C_setSlaveAddress**
- **EUSCI_I2C_setMode**
- **EUSCI_I2C_enable**
- **EUSCI_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 `EUSCI_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 `EUSCI_I2C_setSlaveAddress`. Then the mode of operation (transmit or receive) is chosen using `EUSCI_I2C_setMode`. The I2C module may now be enabled using `EUSCI_I2C_enable`. It is recommended to enable the `EUSCI_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 Transmission

- EUSCI_I2C_masterSendSingleByte

Master Multiple Byte Transmission

- EUSCI_I2C_masterMultiByteSendStart
- EUSCI_I2C_masterMultiByteSendNext
- EUSCI_I2C_masterMultiByteSendStop

Master Single Byte Reception

- EUSCI_I2C_masterReceiveStart
- EUSCI_I2C_masterSingleReceive

Master Multiple Byte Reception

- EUSCI_I2C_masterMultiByteReceiveStart
- EUSCI_I2C_masterMultiByteReceiveNext
- EUSCI_I2C_masterMultiByteReceiveFinish
- EUSCI_I2C_masterMultiByteReceiveStop

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

9.1.2 Slave Operations

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

- **EUSCI_I2C_slaveInit**
- **EUSCI_I2C_setMode**
- **EUSCI_I2C_enable**
- **EUSCI_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 EUSCI_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 EUSCI_I2C_enable. It is recommended 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

- EUSCI_I2C_slaveDataPut

Slave Reception API

- EUSCI_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 `eusci_i2c.c`, with `eusci_i2c.h` containing the API definitions for use by applications.

9.2 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

- `EUSCI_I2C_enableInterrupt`
- `EUSCI_I2C_disableInterrupt`
- `EUSCI_I2C_clearInterruptFlag`
- `EUSCI_I2C_getInterruptStatus`

Status and initialization functions for the I2C modules are

- `EUSCI_I2C_masterInit`
- `EUSCI_I2C_enable`
- `EUSCI_I2C_disable`
- `EUSCI_I2C_isBusBusy`
- `EUSCI_I2C_isBusy`
- `EUSCI_I2C_slaveInit`
- `EUSCI_I2C_interruptStatus`
- `EUSCI_I2C_setSlaveAddress`
- `EUSCI_I2C_setMode`
- `EUSCI_I2C_masterIsSTOPSent`
- `EUSCI_I2C_selectMasterEnvironmentSelect`

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

- `EUSCI_I2C_slaveDataPut`
- `EUSCI_I2C_slaveDataGet`

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

- `EUSCI_I2C_masterSendSingleByte`
- `EUSCI_I2C_masterSendStart`
- `EUSCI_I2C_masterMultiByteSendStart`
- `EUSCI_I2C_masterMultiByteSendNext`
- `EUSCI_I2C_masterMultiByteSendFinish`
- `EUSCI_I2C_masterMultiByteSendStop`
- `EUSCI_I2C_masterMultiByteReceiveNext`

- EUSCI_I2C_masterMultiByteReceiveFinish
- EUSCI_I2C_masterMultiByteReceiveStop
- EUSCI_I2C_masterReceiveStart
- EUSCI_I2C_masterSingleReceive
- EUSCI_I2C_getReceiveBufferAddressForDMA
- EUSCI_I2C_getTransmitBufferAddressForDMA

DMA related

- EUSCI_I2C_getReceiveBufferAddressForDMA
- EUSCI_I2C_getTransmitBufferAddressForDMA

9.3 Programming Example

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

```
//Initialize Slave
EUSCI_I2C_slaveInit(EUSCI_B0_BASE,
                   0x48,
                   EUSCI_I2C_OWN_ADDRESS_OFFSET0,
                   EUSCI_I2C_OWN_ADDRESS_ENABLE
                   );

//Set in receive mode
EUSCI_I2C_setMode(EUSCI_B0_BASE,
                 EUSCI_I2C_TRANSMIT_MODE
                 );

EUSCI_I2C_enable(EUSCI_B0_BASE);

EUSCI_I2C_enableInterrupt(EUSCI_B0_BASE,
                          EUSCI_I2C_TRANSMIT_INTERRUPT0 +
                          EUSCI_I2C_STOP_INTERRUPT
                          );
```

10 EUSCI Synchronous Peripheral Interface (SPI)

Introduction	31
API Functions	31
Programming Example	32

10.1 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 `eusci_spi.c`, with `eusci_spi.h` containing the API definitions for use by applications.

10.2 Functions

To use the module as a master, the user must call `EUSCI_SPI_masterInit()` to configure the SPI Master. This is followed by enabling the SPI module using `EUSCI_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 `EUSCI_SPI_transmitData()` and then when the receive flag is set, the received data is read using `EUSCI_SPI_receiveData()` and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using `EUSCI_SPI_slaveInit()` and this is followed by enabling the module using `EUSCI_SPI_enable()`. Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using `EUSCI_SPI_transmitData()` and this is followed by a data reception by `EUSCI_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

- `EUSCI_SPI_masterInit`
- `EUSCI_SPI_slaveInit`
- `EUSCI_SPI_disable`
- `EUSCI_SPI_enable`
- `EUSCI_SPI_masterChangeClock`
- `EUSCI_SPI_isBusy`

- EUSCI_SPI_select4PinFunctionality
- EUSCI_SPI_changeClockPhasePolarity

Data handling is done by

- EUSCI_SPI_transmitData
- EUSCI_SPI_receiveData

Interrupts from the SPI module are managed using

- EUSCI_SPI_disableInterrupt
- EUSCI_SPI_enableInterrupt
- EUSCI_SPI_getInterruptStatus
- EUSCI_SPI_clearInterruptFlag

DMA related

- EUSCI_SPI_getReceiveBufferAddressForDMA
- EUSCI_SPI_getTransmitBufferAddressForDMA

10.3 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 slave to MSB first, inactive high clock polarity and 3 wire SPI
returnValue = EUSCI_SPI_slaveInit(EUSCI_A0_BASE,
    EUSCI_SPI_MSB_FIRST,
    EUSCI_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,
    EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_HIGH
);

if (STATUS_FAIL == returnValue){
    return;
}

//Enable SPI Module
EUSCI_SPI_enable(EUSCI_A0_BASE);

//Enable Receive interrupt
EUSCI_SPI_enableInterrupt(EUSCI_A0_BASE,
    EUSCI_SPI_RECEIVE_INTERRUPT
);
```


11 EUSCI UART

Introduction	33
API Functions	33
Programming Example	34

11.1 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 `eusci_uart.c`, with `eusci_uart.h` containing the API definitions for use by applications.

11.2 API Functions

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

The EUSCI_UART API is broken into three groups of functions: those that deal with configuration and control of the EUSCI_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 EUSCI_UART are handled by the

- `EUSCI_UART_init()`
- `EUSCI_UART_initAdvance()`
- `EUSCI_UART_enable()`
- `EUSCI_UART_disable()`

- EUSCI_UART_setDormant()
- EUSCI_UART_resetDormant()
- EUSCI_UART_selectDeglitchTime()

Sending and receiving data via the EUSCI_UART is handled by the

- EUSCI_UART_transmitData()
- EUSCI_UART_receiveData()
- EUSCI_UART_transmitAddress()
- EUSCI_UART_transmitBreak()

Managing the EUSCI_UART interrupts and status are handled by the

- EUSCI_UART_enableInterrupt()
- EUSCI_UART_disableInterrupt()
- EUSCI_UART_getInterruptStatus()
- EUSCI_UART_clearInterruptFlag()
- EUSCI_UART_queryStatusFlags()

DMA related

- EUSCI_UART_getReceiveBufferAddressForDMA()
- EUSCI_UART_getTransmitBufferAddressForDMA()

11.3 Programming Example

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

```
// Configure UART
if ( STATUS_FAIL == EUSCI_UART_init(EUSCI_A0_BASE,
    EUSCI_UART_CLOCKSOURCE_ACLK,
    CLOCK_VALUE,
    32768,
    EUSCI_UART_NO_PARITY,
    EUSCI_UART_LSB_FIRST,
    EUSCI_UART_ONE_STOP_BIT,
    EUSCI_UART_MODE,
    EUSCI_UART_LOW_FREQUENCY_BAUDRATE_GENERATION )) {
    return;
}

EUSCI_UART_enable(EUSCI_A0_BASE);

// Enable USCI_A0 RX interrupt
EUSCI_UART_enableInterrupt(EUSCI_A0_BASE,
    EUSCI_UART_RECEIVE_INTERRUPT);
```

12 FRAM Controller

Introduction	35
API Functions	35
Programming Example	36

12.1 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 `fram.c`, with `fram.h` containing the API definitions for use by applications.

12.2 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

12.3 Programming Example

The following example shows some FRAM operations using the APIs

```
//Writes the value of "data", 128 times to FRAM
FRAM_memoryFill132(FRAM_BASE,data,
                   (unsigned long *)FRAM_TEST_START,128);
```

13 GPIO

Introduction	37
API Functions	38
Programming Example	38

13.1 Introduction

The Digital I/O (GPIO) API provides a set of functions for using the MSP430Ware GPIO 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 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 `gpio.c`, with `gpio.h` containing the API definitions for use by applications.

13.2 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_setAsPeripheralModuleFunctionOutputPin()`
- `GPIO_setAsPeripheralModuleFunctionInputPin()`

The GPIO interrupts are handled with

- `GPIO_enableInterrupt()`
- `GPIO_disableInterrupt()`
- `GPIO_clearInterruptFlag()`
- `GPIO_getInterruptStatus()`
- `GPIO_interruptEdgeSelect()`

The GPIO pin state is accessed with

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

13.3 Programming Example

The following example shows how to use the GPIO 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
GPIO_setAsOutputPin(
    GPIO_PORT_P1,
    GPIO_PIN0
);

//Enable P1.4 internal resistance as pull-Up resistance
GPIO_setAsInputPinWithPullUpresistor(
    GPIO_PORT_P1,
    GPIO_PIN4
);

//P1.4 interrupt enabled
GPIO_enableInterrupt(
    GPIO_PORT_P1,
    GPIO_PIN4
);

//P1.4 Hi/Lo edge
GPIO_interruptEdgeSelect(
    GPIO_PORT_P1,
    GPIO_PIN4,
    GPIO_HIGH_TO_LOW_TRANSITION
);

//P1.4 IFG cleared
GPIO_clearInterruptFlag(
    GPIO_PORT_P1,
    GPIO_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
    GPIO_toggleOutputOnPin(
        GPIO_PORT_P1,
        GPIO_PIN0
    );

    //P1.4 IFG cleared
    GPIO_clearInterruptFlag(
        GPIO_PORT_P1,
        GPIO_PIN4
    );
}
}
```


14 Memory Protection Unit (MPU)

Introduction	41
API Functions	41
Programming Example	42

14.1 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 `mpu.c`, with `mpu.h` containing the API definitions for use by applications.

14.2 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_disablePUCOnViolation`
- `MPU_getInterruptStatus`
- `MPU_clearInterruptFlag`
- `MPU_clearAllInterruptFlags`
- `MPU_enableNMlevent`

14.3 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 (MPU_BASE,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 (MPU_BASE,MPU_SECOND_SEG);

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

15 32-Bit Hardware Multiplier (MPY32)

Introduction	43
API Functions	43
Programming Example	44

15.1 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 `mpy32.c`, with `mpy32.h` containing the API definitions for use by applications.

15.2 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

15.3 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(WDT_A_BASE);    // Stop WDT

// Set a 16-bit Operand into the specific Operand 1 register to specify
// unsigned multiplication
MPY32_setOperandOne16Bit(MPY32_BASE,
                        MPY32_MULTIPLY_UNSIGNED,
                        0x1234);

// Set Operand 2 to begin the multiplication operation
MPY32_setOperandTwo16Bit(MPY32_BASE,
                        0x5678);

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

16 Power Management Module (PMM)

Introduction	45
API Functions	45
Programming Example	45

16.1 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.

16.2 API Functions

PMM_enableLowPowerReset() / PMM_disableLowPowerReset() If enabled, SVSH does not re-set device but triggers a system NMI. If disabled, SVSH resets device.

PMM_enableSVSH() / PMM_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 enabled, SVSH is always enabled. Note: this API has different functionality depending on the part.

PMM_regOff() / PMM_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

PMM_clearInterrupt() Clear selected or all interrupt flags for the PMM

PMM_getInterruptStatus() Returns interrupt status of the selected flag in the PMM module

PMM_lockLPM5() / PMM_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 `pmm.c`, with `pmm.h` containing the API definitions for use by applications.

16.3 Programming Example

```
//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.

    */
    PMM_unlockLPM5(PMM_BASE);

        //Get Interrupt Status from the PMMIFG register.
        /* Base Address of Comparator D,
mask:
    PMM_PMMBORIFG
    PMM_PMMRSTIFG,
    PMM_PMPORIFG,
    PMM_SVSLIFG,
    PMM_SVSHIFG
    PMM_PMMLPM5IFG,
return STATUS_SUCCESS (0x01) or STATUS_FAIL (0x00)
    */
    if (PMM_getInterruptStatus(PMM_BASE, PMM_PMMLPM5IFG)) // Was this device in LPMx.5 mode before
    {
        //Clear Interrupt Flag from the PMMIFG register.
        /* Base Address of Comparator D,
mask:
    PMM_PMMBORIFG
    PMM_PMMRSTIFG,
    PMM_PMPORIFG,
    PMM_SVSLIFG,
    PMM_SVSHIFG
    PMM_PMMLPM5IFG,
    PMM_ALL
    */
        PMM_clearInterrupt(PMM_BASE, PMM_PMMLPM5IFG); // Clear the LPMx.5 flag
    }

    if (PMM_getInterruptStatus(PMM_BASE, PMM_PMMRSTIFG)) // Was this reset triggered by the Reset
    {
        PMM_clearInterrupt(PMM_BASE, PMM_PMMRSTIFG); // Clear reset flag

        __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.
        */
        PMM_lockLPM5(PMM_BASE);
        //Trigger a software Brown Out Reset (BOR)
        /*
Base Address of Comparator D,
Forces the devices to perform a BOR.
        */
        PMM_trigBOR(PMM_BASE); // Software trig
    }

    if (PMM_getInterruptStatus(PMM_BASE, PMM_PMMBORIFG)) // Was this reset triggered by the BOR
    {
        PMM_clearInterrupt(PMM_BASE, PMM_PMMBORIFG); // Clear BOR flag

        __delay_cycles(1000000);

        PMM_lockLPM5(PMM_BASE);

        //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.
    */
    PMM_disableSVSH(PMM_BASE);
    //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.
    */
    PMM_disableSVSL(PMM_BASE);
    //Disable Regulator
    /*
Base Address of Comparator D,
Regulator is turned off when going to LPM3/4.
System enters LPM3.5 or LPM4.5, respectively.
    */
    PMM_regOff(PMM_BASE);
    __bis_SR_register(LPM4_bits); // Enter LPM4.5, This automatically locks
                                   // (if not locked already) all GPIO pins
                                   // and will set the LPM5 flag and set
                                   // in the PM5CTL0 register upon wake
}
//-----
while (1)
{
    __no_operation();           // Don't sleep
}
}
```


17 Internal Reference (REF_A)

Introduction	49
API Functions	49
Programming Example	50

17.1 Introduction

The Internal Reference (REF_A) API provides a set of functions for using the MSP430Ware REF_A 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_A 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 `ref_a.c`, with `ref_a.h` containing the API definitions for use by applications.

17.2 API Functions

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

The REF_Aerence voltage of the REF_A module is handled by

- REF_A_setReferenceVoltage()
- REF_A_enableReferenceVoltageOutput()
- REF_A_disableReferenceVoltageOutput()
- REF_A_enableReferenceVoltage()
- REF_A_disableReferenceVoltage()

The internal temperature sensor is handled by

- REF_A_disableTempSensor()
- REF_A_enableTempSensor()

The status of the REF_A module is handled by

- REF_A_getBandgapMode()
- REF_A_isBandgapActive()
- REF_A_isRefGenBusy()

- REF_A_isRefGenActive()
- REF_A_getBufferedBandgapVoltageStatus()
- REF_A_getVariableReferenceVoltageStatus()
- REF_A_setReferenceVoltageOneTimeTrigger()
- REF_A_setBufBandgapVoltageOneTimeTrigger()

17.3 Programming Example

The following example shows how to initialize and use the REF_A API with the ADC12 module to use the internal 2.5V reference and perform a single conversion 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.

```
    //If ref generator busy, WAIT
while (REF_A_isRefGenBusy(REF_A_BASE)) ;
//Select internal ref = 2.5V
REF_A_setReferenceVoltage(REF_A_BASE,
    REF_A_VREF2_5V);
//Internal Reference ON
REF_A_enableReferenceVoltage(REF_A_BASE);

//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
*/
ADC12_B_init(ADC12_B_BASE,
    ADC12_B_SAMPLEHOLDSOURCE_SC,
    ADC12_B_CLOCKSOURCE_ADC12OSC,
    ADC12_B_CLOCKDIVIDER_1,
    ADC12_B_CLOCKPREDIVIDER__1,
    ADC12_B_MAPINTCH0);
```

18 Real-Time Clock (RTC)

Introduction	51
API Functions	51
Programming Example	52

18.1 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_B_A module is used, then Counter mode may also be initialized, 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_B_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 `rtc_b.c`, with `rtc_b.h` containing the API definitions for use by applications.

18.2 API Functions

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

The RTC clock settings are handled by

- `RTC_B_startClock`
- `RTC_B_holdClock`
- `RTC_B_setCalibrationFrequency`
- `RTC_B_setCalibrationData`

The RTC Calendar Mode is initialized and setup by

- `RTC_B_calendarInit`
- `RTC_B_getCalendarTime`
- `RTC_B_getPrescaleValue`
- `RTC_B_setPrescaleValue`

The RTC interrupts are handled by

- `RTC_B_setCalendarAlarm`
- `RTC_B_setCalendarEvent`
- `RTC_B_definePrescaleEvent`

- RTC_B_enableInterrupt
- RTC_B_disableInterrupt
- RTC_B_getInterruptStatus
- RTC_B_clearInterrupt

The RTC conversions are handled by

- RTC_B_convertBCDToBinary
- RTC_B_convertBinaryToBCD

18.3 Programming Example

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

```
//Initialize Calendar Mode of RTC
/*
Base Address of the RTC_B_A
Pass in current time, intialized above
Use BCD as Calendar Register Format
*/
RTC_B_calendarInit(RTC_B_BASE,
    currentTime,
    RTC_B_FORMAT_BCD);

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

//Specify an interrupt to assert every minute
RTC_B_setCalendarEvent(RTC_B_BASE,
    RTC_B_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_B_enableInterrupt(RTC_B_BASE,
    RTCRDYIE + RTCTEVIE + RTCAIE);

//Start RTC Clock
RTC_B_startClock(RTC_B_BASE);

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

19 SFR Module

Introduction	53
API Functions	53
Programming Example	53

19.1 Introduction

The Special Function Registers API provides a set of functions for using the MSP430Ware SFR module. Functions are provided to enable and disable interrupts and control the \sim RST/NMI pin

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

This driver is contained in `sfr.c`, with `sfr.h` containing the API definitions for use by applications.

19.2 API Functions

The SFR API is broken into 2 groups: the SFR interrupts and the SFR \sim RST/NMI pin control

The SFR interrupts are handled by

- `SFR_enableInterrupt`
- `SFR_disableInterrupt`
- `SFR_getInterruptStatus`
- `SFR_clearInterrupt`

The SFR \sim RST/NMI pin is controlled by

- `SFR_setResetPinPullResistor`
- `SFR_setNMIEdge`
- `SFR_setResetNMIPinFunction`

19.3 Programming Example

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

```
do
{
    // Clear SFR Fault Flag
    SFR_clearInterrupt(SFR_BASE,
                      OFIFG);

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


20 SYS Module

Introduction	55
API Functions	55
Programming Example	??

20.1 Introduction

The System Control (SYS) API provides a set of functions for using the MSP430Ware SYS module. Functions are provided to control various SYS controls, setup the BSL, and control the JTAG Mailbox.

This driver is contained in `sys.c`, with `sys.h` containing the API definitions for use by applications.

20.2 API Functions

The SYS API is broken into 3 groups: the various SYS controls, the BSL controls, and the JTAG mailbox controls.

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`

20.3 Programming Example

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

```
SYS_enableBSLProtect (SYS_BASE);
```


21 TIMER_A

Introduction	57
API Functions	58
Programming Example	58

21.1 Introduction

TIMER_A is a 16-bit timer/counter with multiple capture/compare registers. TIMER_A can support multiple capture/compares, PWM outputs, and interval timing. TIMER_A 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.

TIMER_A 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_A can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

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

The TIMER_A may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with `TIMER_A_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_A_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_A_generatePWM()` or a combination of `Timer_initCompare()` and timer start APIs

The TIMER_A API provides a set of functions for dealing with the TIMER_A 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 `TIMER_A.c`, with `TIMER_A.h` containing the API definitions for use by applications.

21.2 API Functions

The TIMER_A 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_A configuration and initialization is handled by

- TIMER_A_startCounter(),
- TIMER_A_configureContinuousMode(),
- TIMER_A_configureUpMode(),
- TIMER_A_configureUpDownMode(),
- TIMER_A_startContinuousMode(),
- TIMER_A_startUpMode(),
- TIMER_A_startUpDownMode(),
- TIMER_A_initCapture(),
- TIMER_A_initCompare(),
- TIMER_A_clear(),
- TIMER_A_stop()

TIMER_A outputs are handled by

- TIMER_A_getSynchronizedCaptureCompareInput(),
- TIMER_A_getOutputForOutputModeOutBitValue(),
- TIMER_A_setOutputForOutputModeOutBitValue(),
- TIMER_A_generatePWM()
- TIMER_A_getCaptureCompareCount()
- TIMER_A_setCompareValue()

The interrupt handler for the TIMER_A interrupt is managed with

- TIMER_A_enableInterrupt(),
- TIMER_A_disableInterrupt(),
- TIMER_A_getInterruptStatus(),
- TIMER_A_enableCaptureCompareInterrupt(),
- TIMER_A_disableCaptureCompareInterrupt(),
- TIMER_A_getCaptureCompareInterruptStatus(),
- TIMER_A_clearCaptureCompareInterruptFlag()
- TIMER_A_clearTimerInterruptFlag()

21.3 Programming Example

The following example shows some TIMER_A operations using the APIs

```
{    //Start TIMER_A
    TIMER_A_configureUpDownMode( TIMER_A1_BASE,
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_A_TAIE_INTERRUPT_DISABLE,
        TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE,
        TIMER_A_DO_CLEAR
    );

    TIMER_A_startCounter( TIMER_A1_BASE,
        TIMER_A_UPDOWN_MODE
    );

    //Initialize compare registers to generate PWM1
    TIMER_A_initCompare(TIMER_A1_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_1,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_A_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE1
    );
    //Initialize compare registers to generate PWM2
    TIMER_A_initCompare(TIMER_A1_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_2,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TIMER_A_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE2
    );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

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


22 TIMER_B

Introduction	61
API Functions	62
Programming Example	63

22.1 Introduction

TIMER_B is a 16-bit timer/counter with multiple capture/compare registers. TIMER_B can support multiple capture/compares, PWM outputs, and interval timing. TIMER_B 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 hardware peripheral.

TIMER_B 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

TIMER_B can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

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

The TIMER_B may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TIMER_B_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_B_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_B_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The TIMER_B API provides a set of functions for dealing with the TIMER_B 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 `TIMER_B.c`, with `TIMER_B.h` containing the API definitions for use by applications.

22.2 API Functions

The TIMER_B 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_B configuration and initialization is handled by

- `TIMER_B_startCounter()`,
- `TIMER_B_configureContinuousMode()`,
- `TIMER_B_configureUpMode()`,
- `TIMER_B_configureUpDownMode()`,
- `TIMER_B_startContinuousMode()`,
- `TIMER_B_startUpMode()`,
- `TIMER_B_startUpDownMode()`,
- `TIMER_B_initCapture()`,
- `TIMER_B_initCompare()`,
- `TIMER_B_clear()`,
- `TIMER_B_stop()`
- `TIMER_B_initCompareLatchLoadEvent()`,
- `TIMER_B_selectLatchingGroup()`,
- `TIMER_B_selectCounterLength()`,

TIMER_B outputs are handled by

- `TIMER_B_getSynchronizedCaptureCompareInput()`,
- `TIMER_B_getOutputForOutputModeOutBitValue()`,
- `TIMER_B_setOutputForOutputModeOutBitValue()`,
- `TIMER_B_generatePWM()`
- `TIMER_B_getCaptureCompareCount()`
- `TIMER_B_setCompareValue()`

The interrupt handler for the TIMER_B interrupt is managed with

- `TIMER_B_enableInterrupt()`,
- `TIMER_B_disableInterrupt()`,
- `TIMER_B_getInterruptStatus()`,

- TIMER_B_enableCaptureCompareInterrupt(),
- TIMER_B_disableCaptureCompareInterrupt(),
- TIMER_B_getCaptureCompareInterruptStatus(),
- TIMER_B_clearCaptureCompareInterruptFlag()
- TIMER_B_clearTimerInterruptFlag()

22.3 Programming Example

The following example shows some TIMER_B operations using the APIs

```
{    //Start TIMER_B
    TIMER_B_configureUpMode(    TIMER_B0_BASE,
        TIMER_B_CLOCKSOURCE_SMCLK,
        TIMER_B_CLOCKSOURCE_DIVIDER_1,
        511,
        TIMER_B_TBIE_INTERRUPT_DISABLE,
        TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE,
        TIMER_B_DO_CLEAR
    );

    TIMER_B_startCounter(    TIMER_B0_BASE,
        TIMER_B_UP_MODE
    );

    //Initialize compare mode to generate PWM1
    TIMER_B_initCompare(TIMER_B0_BASE,
        TIMER_B_CAPTURECOMPARE_REGISTER_1,
        TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TIMER_B_OUTPUTMODE_RESET_SET,
        383
    );

    //Initialize compare mode to generate PWM2
    TIMER_B_initCompare(TIMER_B0_BASE,
        TIMER_B_CAPTURECOMPARE_REGISTER_2,
        TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_B_OUTPUTMODE_RESET_SET,
        128
    );
}
```


23 Tag Length Value

Introduction	65
API Functions	65
Programming Example	65

23.1 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 User's Guide.

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

23.2 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 if the specified interrupt vector is not defined.

23.3 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
            );

```


24 WatchDog Timer (WDT_A)

Introduction	67
API Functions	67
Programming Example	67

24.1 Introduction

The Watchdog Timer (WDT_A) API provides a set of functions for using the MSP430Ware WDT_A 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_A module can generate only 1 kind of interrupt in timer interval mode. If in watchdog mode, then the WDT_A module will assert a reset once the timer has finished.

This driver is contained in `wdt_a.c`, with `wdt_a.h` containing the API definitions for use by applications.

24.2 API Functions

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

- WDT_A_hold
- WDT_A_start
- WDT_A_clearCounter
- WDT_A_watchdogTimerInit
- WDT_A_intervalTimerInit

24.3 Programming Example

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

```
//Initialize WDT_A module in timer interval mode,
//with SMCLK as source at an interval of 32 ms.
WDT_A_intervalTimerInit(WDT_A_BASE,
    WDT_A_CLOCKSOURCE_SMCLK,
    WDT_A_CLOCKDIVIDER_32K);

//Enable Watchdog Interrupt
SFR_enableInterrupt(SFR_BASE,
    WDT_AIE);

//Set P1.0 to output direction
GPIO_setAsOutputPin(
    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

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

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

Copyright © 2012, Texas Instruments Incorporated