



TivaWare™ Peripheral Driver Library

USER'S GUIDE

Copyright

Copyright © 2006-2014 Texas Instruments Incorporated. All rights reserved. Tiva and TivaWare are trademarks of Texas Instruments Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. 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
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c



Revision Information

This is version 2.1.0.12573 of this document, last updated on February 07, 2014.

Table of Contents

Copyright	2
Revision Information	2
1 Introduction	7
2 Programming Model	9
2.1 Introduction	9
2.2 Direct Register Access Model	9
2.3 Software Driver Model	10
2.4 Combining The Models	11
3 Analog Comparator	13
3.1 Introduction	13
3.2 API Functions	13
3.3 Programming Example	19
4 Analog to Digital Converter (ADC)	21
4.1 Introduction	21
4.2 API Functions	22
4.3 Programming Example	46
5 AES	47
5.1 Introduction	47
5.2 API Functions	47
5.3 Programming Example	62
6 Controller Area Network (CAN)	65
6.1 Introduction	65
6.2 API Functions	65
6.3 CAN Message Objects	88
6.4 Programming Examples	89
7 CRC	93
7.1 Introduction	93
7.2 API Functions	93
7.3 Programming Example	96
8 DES	99
8.1 Introduction	99
8.2 API Functions	99
8.3 DES Programming Example	108
8.4 TDES Programming Example	110
9 EEPROM	113
9.1 Introduction	113
9.2 API Functions	114
9.3 Programming Example	128
10 Ethernet Controller	129
10.1 Introduction	129
10.2 API Functions	129
10.3 Programming Example	193
11 External Peripheral Interface (EPI)	201
11.1 Introduction	201
11.2 API Functions	201

11.3 Programming Example	231
12 Fan Controller	233
12.1 Introduction	233
12.2 API Functions	233
12.3 Programming Example	235
13 Flash	237
13.1 Introduction	237
13.2 API Functions	237
13.3 Programming Example	245
14 Floating-Point Unit (FPU)	247
14.1 Introduction	247
14.2 API Functions	248
14.3 Programming Example	252
15 GPIO	253
15.1 Introduction	253
15.2 API Functions	254
15.3 Programming Example	286
16 Hibernation Module	289
16.1 Introduction	289
16.2 API Functions	289
16.3 Programming Example	316
17 Inter-Integrated Circuit (I2C)	321
17.1 Introduction	321
17.2 API Functions	322
17.3 Programming Example	349
18 Interrupt Controller (NVIC)	351
18.1 Introduction	351
18.2 API Functions	352
18.3 Programming Example	362
19 LCD Controller (LCD)	365
19.1 Introduction	365
19.2 API Functions	365
19.3 Programming Example	393
20 Memory Protection Unit (MPU)	397
20.1 Introduction	397
20.2 API Functions	397
20.3 Programming Example	404
21 1-Wire Master Module	407
21.1 Introduction	407
21.2 API Functions	407
21.3 Programming Example	415
22 Pulse Width Modulator (PWM)	417
22.1 Introduction	417
22.2 API Functions	417
22.3 Programming Example	439
23 Quadrature Encoder (QEI)	441
23.1 Introduction	441
23.2 API Functions	441

23.3 Programming Example	450
24 SHA/MD5	451
24.1 Introduction	451
24.2 API Functions	451
24.3 Hashing Programming Example	461
24.4 HMAC Programming Example	461
25 Synchronous Serial Interface (SSI)	463
25.1 Introduction	463
25.2 API Functions	463
25.3 Programming Example	476
26 Software CRC Module	479
26.1 Introduction	479
26.2 API Functions	479
26.3 Programming Example	482
27 System Control	483
27.1 Introduction	483
27.2 API Functions	484
27.3 Programming Example	520
28 System Exception Module	523
28.1 Introduction	523
28.2 API Functions	523
28.3 Programming Example	526
29 System Tick (SysTick)	529
29.1 Introduction	529
29.2 API Functions	529
29.3 Programming Example	533
30 Timer	535
30.1 Introduction	535
30.2 API Functions	536
30.3 Programming Example	558
31 UART	559
31.1 Introduction	559
31.2 API Functions	559
31.3 Programming Example	583
32 uDMA Controller	585
32.1 Introduction	585
32.2 API Functions	586
32.3 Programming Example	606
33 USB Controller	609
33.1 Introduction	609
33.2 General USB API Functions	609
33.3 Using USB with the uDMA Controller	652
33.4 Using the integrated USB DMA Controller	656
33.5 USB Link Power Management Functions	671
33.6 USB UTMI Low Pin Interface (ULPI)	684
33.7 Programming Example	688
34 Watchdog Timer	689
34.1 Introduction	689
34.2 API Functions	689

34.3 Programming Example	698
35 Using the ROM	699
35.1 Introduction	699
35.2 Direct ROM Calls	699
35.3 Mapped ROM Calls	700
35.4 Firmware Update	701
36 Error Handling	705
IMPORTANT NOTICE	706

1 Introduction

The Texas Instruments® TivaWare™ Peripheral Driver Library is a set of drivers for accessing the peripherals found on the Tiva™ family of ARM® Cortex™-M based 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.

The Driver Library includes drivers for all classes of Tiva microcontrollers. Some drivers and parameters are only valid for certain classes. See the application report entitled, "Differences Among Tiva Product Classes" for more information.

The following tool chains are supported:

- Keil™ RealView® Microcontroller Development Kit
- MentorGraphics Sourcery CodeBench for ARM EABI
- IAR Embedded Workbench®
- Texas Instruments Code Composer Studio™
- GNU Compiler Collection(GCC)

Source Code Overview

The following is an overview of the organization of the peripheral driver library source code.

EULA.txt	The full text of the End User License Agreement that covers the use of this software package.
driverlib/	This directory contains the source code for the drivers.
hw_* .h	Header files, one per peripheral, that describe all the registers and the bit fields within those registers for each peripheral. These header files are used by the drivers to directly access a peripheral, and can be used by application code to bypass the peripheral driver library API.
inc/	This directory holds the part specific header files used for the direct register access programming model.
makedefs	A set of definitions used by make files.

2 Programming Model

Introduction	9
Direct Register Access Model	9
Software Driver Model	10
Combining The Models	11

2.1 Introduction

The peripheral driver library provides support for two programming models: the direct register access model and the software driver model. Each model can be used independently or combined, based on the needs of the application or the programming environment desired by the developer.

Each programming model has advantages and disadvantages. Use of the direct register access model generally results in smaller and more efficient code than using the software driver model. However, the direct register access model requires detailed knowledge of the operation of each register and bit field, as well as their interactions and any sequencing required for proper operation of the peripheral; the developer is insulated from these details by the software driver model, generally requiring less time to develop applications.

2.2 Direct Register Access Model

In the direct register access model, the peripherals are programmed by the application by writing values directly into the peripheral's registers. A set of macros is provided that simplifies this process. These macros are stored in part-specific header files contained in the `inc` directory; the name of the header file matches the part number (for example, the header file for the TM4C123GH6PM microcontroller is `inc/tm4c123gh6pm.h`). By including the header file that matches the part being used, macros are available for accessing all registers on that part, as well as all bit fields within those registers. No macros are available for registers that do not exist on the part in question, making it difficult to access registers that do not exist.

The defines used by the direct register access model follow a naming convention that makes it easier to know how to use a particular macro. The rules are as follows:

- Values that end in `_R` are used to access the value of a register. For example, `SSI0_CRO_R` is used to access the `CRO` register in the `SSI0` module.
- Values that end in `_M` represent the mask for a multi-bit field in a register. If the value placed in the multi-bit field is a number, there is a macro with the same base name but ending with `_S` (for example, `SSI_CRO_SCR_M` and `SSI_CRO_SCR_S`). If the value placed into the multi-bit field is an enumeration, then there are a set of macros with the same base name but ending with identifiers for the various enumeration values (for example, the `SSI_CRO_FRF_M` macro defines the bit field, and the `SSI_CRO_FRF_NMW`, `SSI_CRO_FRF_TI`, and `SSI_CRO_FRF_MOTO` macros provide the enumerations for the bit field).
- Values that end in `_S` represent the number of bits to shift a value in order to align it with a multi-bit field. These values match the macro with the same base name but ending with `_M`.

- All other macros represent the value of a bit field.
- All register name macros start with the module name and instance number (for example, `SSI0` for the first SSI module) and are followed by the name of the register as it appears in the data sheet (for example, the `CR0` register in the data sheet results in `SSI0_CR0_R`).
- All register bit fields start with the module name, followed by the register name, and then followed by the bit field name as it appears in the data sheet. For example, the `SCR` bit field in the `CR0` register in the `SSI` module is identified by `SSI_CR0_SCR`. . . . In the case where the bit field is a single bit, there is nothing further (for example, `SSI_CR0_SPH` is a single bit in the `CR0` register). If the bit field is more than a single bit, there is a mask value (`_M`) and either a shift (`_S`) if the bit field contains a number or a set of enumerations if not.

Given these definitions, the `CR0` register can be programmed as follows:

```
SSI0_CR0_R = ((5 << SSI_CR0_SCR_S) | SSI_CR0_SPH | SSI_CR0_SPO |
                SSI_CR0_FRF_MOTO | SSI_CR0_DSS_8);
```

Alternatively, the following has the same effect (although it is not as easy to understand):

```
SSI0_CR0_R = 0x000005c7;
```

Extracting the value of the `SCR` field from the `CR0` register is as follows:

```
ulValue = (SSI0_CR0_R & SSI_CR0_SCR_M) >> SSI0_CR0_SCR_S;
```

The GPIO modules have many registers that do not have bit field definitions. For these registers, the register bits represent the individual GPIO pins; so bit zero in these registers corresponds to the **Px0** pin on the part (where **x** is replaced by a GPIO module letter), bit one corresponds to the **Px1** pin, and so on.

The `blinky` example for each board uses the direct register access model to blink the on-board LED.

Note:

The `hw_*.h` header files that are used by the drivers in the library contain many of the same definitions as the header files used for direct register access. As a result, the two cannot both be included into the same source file without the compiler producing warnings about the redefinition of symbols.

2.3 Software Driver Model

In the software driver model, the API provided by the peripheral driver library is used by applications to control the peripherals. Because these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This method provides for rapid development of the application without requiring detailed knowledge of how to program the peripherals.

Corresponding to the direct register access model example, the following call also programs the `CR0` register in the SSI module (though the register name is hidden by the API):

```
SSIConfigSetExpClk(SSI0_BASE, 50000000, SSI_FRF_MOTO_MODE_3,  
                    SSI_MODE_MASTER, 1000000, 8);
```

The resulting value in the CR0 register might not be exactly the same because [SSIConfigSetExpClk\(\)](#) may compute a different value for the SCR bit field than what was used in the direct register access model example.

All example applications other than `blinky` use the software driver model.

The drivers in the peripheral driver library are described in the remaining chapters in this document. They combine to form the software driver model.

2.4 Combining The Models

The direct register access model and software driver model can be used together in a single application, allowing the most appropriate model to be applied as needed to any particular situation within the application. For example, the software driver model can be used to configure the peripherals (because this is not performance critical) and the direct register access model can be used for operation of the peripheral (which may be more performance critical). Or, the software driver model can be used for peripherals that are not performance critical (such as a UART used for data logging) and the direct register access model for performance critical peripherals (such as the ADC module used to capture real-time analog data).

3 Analog Comparator

Introduction	13
API Functions	13
Programming Example	19

3.1 Introduction

The comparator API provides a set of functions for programming and using the analog comparators. A comparator can compare a test voltage against an individual external reference voltage, a shared single external reference voltage, or a shared internal reference voltage. It can provide its output to a device pin, acting as a replacement for an analog comparator on the board, or it can be used to signal the application via interrupts or triggers to the ADC to start capturing a sample sequence. The interrupt generation logic is independent from the ADC triggering logic. As a result, the comparator can generate an interrupt based on one event and an ADC trigger based on another event. For example, an interrupt can be generated on a rising edge and the ADC triggered on a falling edge.

This driver is contained in `driverlib/comp.c`, with `driverlib/comp.h` containing the API declarations for use by applications.

3.2 API Functions

Functions

- void [ComparatorConfigure](#) (uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32Config)
- void [ComparatorIntClear](#) (uint32_t ui32Base, uint32_t ui32Comp)
- void [ComparatorIntDisable](#) (uint32_t ui32Base, uint32_t ui32Comp)
- void [ComparatorIntEnable](#) (uint32_t ui32Base, uint32_t ui32Comp)
- void [ComparatorIntRegister](#) (uint32_t ui32Base, uint32_t ui32Comp, void (*pfnHandler)(void))
- bool [ComparatorIntStatus](#) (uint32_t ui32Base, uint32_t ui32Comp, bool bMasked)
- void [ComparatorIntUnregister](#) (uint32_t ui32Base, uint32_t ui32Comp)
- void [ComparatorRefSet](#) (uint32_t ui32Base, uint32_t ui32Ref)
- bool [ComparatorValueGet](#) (uint32_t ui32Base, uint32_t ui32Comp)

3.2.1 Detailed Description

The comparator API is fairly simple, like the comparators themselves. There are functions for configuring a comparator and reading its output ([ComparatorConfigure\(\)](#), [ComparatorRefSet\(\)](#) and [ComparatorValueGet\(\)](#)) and functions for dealing with an interrupt handler for the comparator ([ComparatorIntRegister\(\)](#), [ComparatorIntUnregister\(\)](#), [ComparatorIntEnable\(\)](#), [ComparatorIntDisable\(\)](#), [ComparatorIntStatus\(\)](#), and [ComparatorIntClear\(\)](#)).

3.2.2 Function Documentation

3.2.2.1 ComparatorConfigure

Configures a comparator.

Prototype:

```
void
ComparatorConfigure(uint32_t ui32Base,
                     uint32_t ui32Comp,
                     uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator to configure.

ui32Config is the configuration of the comparator.

Description:

This function configures a comparator. The **ui32Config** parameter is the result of a logical OR operation between the **COMP_TRIG_xxx**, **COMP_INT_xxx**, **COMP_ASRCP_XXX**, and **COMP_OUTPUT_xxx** values.

The **COMP_TRIG_xxx** term can take on the following values:

- **COMP_TRIG_NONE** to have no trigger to the ADC.
- **COMP_TRIG_HIGH** to trigger the ADC when the comparator output is high.
- **COMP_TRIG_LOW** to trigger the ADC when the comparator output is low.
- **COMP_TRIG_FALL** to trigger the ADC when the comparator output goes low.
- **COMP_TRIG_RISE** to trigger the ADC when the comparator output goes high.
- **COMP_TRIG_BOTH** to trigger the ADC when the comparator output goes low or high.

The **COMP_INT_xxx** term can take on the following values:

- **COMP_INT_HIGH** to generate an interrupt when the comparator output is high.
- **COMP_INT_LOW** to generate an interrupt when the comparator output is low.
- **COMP_INT_FALL** to generate an interrupt when the comparator output goes low.
- **COMP_INT_RISE** to generate an interrupt when the comparator output goes high.
- **COMP_INT_BOTH** to generate an interrupt when the comparator output goes low or high.

The **COMP_ASRCP_xxx** term can take on the following values:

- **COMP_ASRCP_PIN** to use the dedicated Comp+ pin as the reference voltage.
- **COMP_ASRCP_PIN0** to use the Comp0+ pin as the reference voltage (this is the same as **COMP_ASRCP_PIN** for the comparator 0).
- **COMP_ASRCP_REF** to use the internally generated voltage as the reference voltage.

The **COMP_OUTPUT_xxx** term can take on the following values:

- **COMP_OUTPUT_NORMAL** to enable a non-inverted output from the comparator to a device pin.
- **COMP_OUTPUT_INVERT** to enable an inverted output from the comparator to a device pin.

Returns:

None.

3.2.2.2 ComparatorIntClear

Clears a comparator interrupt.

Prototype:

```
void
ComparatorIntClear(uint32_t ui32Base,
                    uint32_t ui32Comp)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

Description:

The comparator interrupt is cleared, so that it no longer asserts. This function must be called in the interrupt handler to keep the handler from being called again immediately upon exit. Note that for a level-triggered interrupt, the interrupt cannot be cleared until it stops asserting.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

3.2.2.3 ComparatorIntDisable

Disables the comparator interrupt.

Prototype:

```
void
ComparatorIntDisable(uint32_t ui32Base,
                      uint32_t ui32Comp)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

Description:

This function disables generation of an interrupt from the specified comparator. Only enabled comparator interrupts can be reflected to the processor.

Returns:

None.

3.2.2.4 ComparatorIntEnable

Enables the comparator interrupt.

Prototype:

```
void  
ComparatorIntEnable(uint32_t ui32Base,  
                     uint32_t ui32Comp)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

Description:

This function enables generation of an interrupt from the specified comparator. Only enabled comparator interrupts can be reflected to the processor.

Returns:

None.

3.2.2.5 ComparatorIntRegister

Registers an interrupt handler for the comparator interrupt.

Prototype:

```
void  
ComparatorIntRegister(uint32_t ui32Base,  
                      uint32_t ui32Comp,  
                      void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

pfnHandler is a pointer to the function to be called when the comparator interrupt occurs.

Description:

This function sets the handler to be called when the comparator interrupt occurs and enables the interrupt in the interrupt controller. It is the interrupt handler's responsibility to clear the interrupt source via [ComparatorIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

3.2.2.6 ComparatorIntStatus

Gets the current interrupt status.

Prototype:

```
bool
ComparatorIntStatus(uint32_t ui32Base,
                     uint32_t ui32Comp,
                     bool bMasked)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function returns the interrupt status for the comparator. Either the raw or the masked interrupt status can be returned.

Returns:

true if the interrupt is asserted and **false** if it is not asserted.

3.2.2.7 ComparatorIntUnregister

Unregisters an interrupt handler for a comparator interrupt.

Prototype:

```
void
ComparatorIntUnregister(uint32_t ui32Base,
                        uint32_t ui32Comp)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

Description:

This function clears the handler to be called when a comparator interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

3.2.2.8 ComparatorRefSet

Sets the internal reference voltage.

Prototype:

```
void
ComparatorRefSet(uint32_t ui32Base,
                  uint32_t ui32Ref)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Ref is the desired reference voltage.

Description:

This function sets the internal reference voltage value. The voltage is specified as one of the following values:

- **COMP_REF_OFF** to turn off the reference voltage
- **COMP_REF_0V** to set the reference voltage to 0 V
- **COMP_REF_0_1375V** to set the reference voltage to 0.1375 V
- **COMP_REF_0_275V** to set the reference voltage to 0.275 V
- **COMP_REF_0_4125V** to set the reference voltage to 0.4125 V
- **COMP_REF_0_55V** to set the reference voltage to 0.55 V
- **COMP_REF_0_6875V** to set the reference voltage to 0.6875 V
- **COMP_REF_0_825V** to set the reference voltage to 0.825 V
- **COMP_REF_0_928125V** to set the reference voltage to 0.928125 V
- **COMP_REF_0_9625V** to set the reference voltage to 0.9625 V
- **COMP_REF_1_03125V** to set the reference voltage to 1.03125 V
- **COMP_REF_1_134375V** to set the reference voltage to 1.134375 V
- **COMP_REF_1_1V** to set the reference voltage to 1.1 V
- **COMP_REF_1_2375V** to set the reference voltage to 1.2375 V
- **COMP_REF_1_340625V** to set the reference voltage to 1.340625 V
- **COMP_REF_1_375V** to set the reference voltage to 1.375 V
- **COMP_REF_1_44375V** to set the reference voltage to 1.44375 V
- **COMP_REF_1_5125V** to set the reference voltage to 1.5125 V
- **COMP_REF_1_546875V** to set the reference voltage to 1.546875 V
- **COMP_REF_1_65V** to set the reference voltage to 1.65 V
- **COMP_REF_1_753125V** to set the reference voltage to 1.753125 V
- **COMP_REF_1_7875V** to set the reference voltage to 1.7875 V
- **COMP_REF_1_85625V** to set the reference voltage to 1.85625 V
- **COMP_REF_1_925V** to set the reference voltage to 1.925 V
- **COMP_REF_1_959375V** to set the reference voltage to 1.959375 V
- **COMP_REF_2_0625V** to set the reference voltage to 2.0625 V
- **COMP_REF_2_165625V** to set the reference voltage to 2.165625 V
- **COMP_REF_2_26875V** to set the reference voltage to 2.26875 V
- **COMP_REF_2_371875V** to set the reference voltage to 2.371875 V

Returns:

None.

3.2.2.9 ComparatorValueGet

Gets the current comparator output value.

Prototype:

```
bool  
ComparatorValueGet(uint32_t ui32Base,  
                    uint32_t ui32Comp)
```

Parameters:

ui32Base is the base address of the comparator module.

ui32Comp is the index of the comparator.

Description:

This function retrieves the current value of the comparator output.

Returns:

Returns **true** if the comparator output is high and **false** if the comparator output is low.

3.3 Programming Example

The following example shows how to use the comparator API to configure the comparator and read its value.

```
//  
// Configure the internal voltage reference.  
//  
ComparatorRefSet(COMP_BASE, COMP_REF_1_65V);  
  
//  
// Configure comparator 0.  
//  
ComparatorConfigure(COMP_BASE, 0,  
                    (COMP_TRIG_NONE | COMP_INT_BOTH |  
                     COMP_ASRCP_REF | COMP_OUTPUT_NORMAL));  
  
//  
// Delay for some time...  
//  
  
//  
// Read the comparator output value.  
//  
ComparatorValueGet(COMP_BASE, 0);
```


4 Analog to Digital Converter (ADC)

Introduction	21
API Functions	22
Programming Example	46

4.1 Introduction

The analog to digital converter (ADC) API provides a set of functions for programming and operating the ADC. Functions are provided to configure the sample sequencers, read the captured data, register a sample sequence interrupt handler, and handle interrupt masking/clearing.

Depending on the features of the individual microcontroller, the ADC supports up to twenty-four input channels plus an internal temperature sensor. Four sampling sequencers, each with configurable trigger events, can be captured. The first sequencer captures up to eight samples, the second and third sequencers capture up to four samples, and the fourth sequencer captures a single sample. Each sample can be the same channel, different channels, or any combination in any order.

The sample sequencers have configurable priorities that determine the order in which they are captured when multiple triggers occur simultaneously. The highest priority sequencer that is currently triggered is sampled first. Care must be taken with triggers that occur frequently (such as the “always” trigger); if their priority is too high, it is possible to starve the lower priority sequencers.

Hardware oversampling of the ADC data is available for improved accuracy. An oversampling factor of 2x, 4x, 8x, 16x, 32x, or 64x is supported, but reduces the throughput of the ADC by a corresponding factor. Hardware oversampling is applied uniformly across all sample sequencers.

Software oversampling of the ADC data is also available (even when hardware oversampling is available). An oversampling factor of 2x, 4x, or 8x is supported, but reduces the depth of the sample sequencers by a corresponding amount. For example, the first sample sequencer captures eight samples; in 4x oversampling mode, it can only capture two samples because the first four samples are used for the first oversampled value and the second four samples are used for the second oversampled value. The amount of software oversampling is configured on a per sample sequencer basis.

A more sophisticated software oversampling can be used to eliminate the reduction of the sample sequencer depth. By increasing the ADC trigger rate by 4x (for example) and averaging four triggers worth of data, 4x oversampling is achieved without any loss of sample sequencer capability. In this case, an increase in the number of ADC triggers (and presumably ADC interrupts) is the consequence. Because this method requires adjustments outside of the ADC driver itself, it is not directly supported by the driver (though nothing in the driver prevents it). The software oversampling APIs should not be used in this case.

This driver is contained in `driverlib/adc.c`, with `driverlib/adc.h` containing the API declarations for use by applications.

4.2 API Functions

Functions

- `bool ADCBusy (uint32_t ui32Base)`
- `uint32_t ADCClockConfigGet (uint32_t ui32Base, uint32_t *pui32ClockDiv)`
- `void ADCClockConfigSet (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32ClockDiv)`
- `void ADCComparatorConfigure (uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32Config)`
- `void ADCComparatorIntClear (uint32_t ui32Base, uint32_t ui32Status)`
- `void ADCComparatorIntDisable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCComparatorIntEnable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `uint32_t ADCComparatorIntStatus (uint32_t ui32Base)`
- `void ADCComparatorRegionSet (uint32_t ui32Base, uint32_t ui32Comp, uint32_t ui32LowRef, uint32_t ui32HighRef)`
- `void ADCComparatorReset (uint32_t ui32Base, uint32_t ui32Comp, bool bTrigger, bool bIn-terrupt)`
- `void ADCHardwareOversampleConfigure (uint32_t ui32Base, uint32_t ui32Factor)`
- `void ADCIntClear (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCIntClearEx (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void ADCIntDisable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCIntDisableEx (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void ADCIntEnable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCIntEnableEx (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void ADCIntRegister (uint32_t ui32Base, uint32_t ui32SequenceNum, void (*pfnHandler)(void))`
- `uint32_t ADCIntStatus (uint32_t ui32Base, uint32_t ui32SequenceNum, bool bMasked)`
- `uint32_t ADCIntStatusEx (uint32_t ui32Base, bool bMasked)`
- `void ADCIntUnregister (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `uint32_t ADCPhaseDelayGet (uint32_t ui32Base)`
- `void ADCPhaseDelaySet (uint32_t ui32Base, uint32_t ui32Phase)`
- `void ADCProcessorTrigger (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `uint32_t ADCReferenceGet (uint32_t ui32Base)`
- `void ADCReferenceSet (uint32_t ui32Base, uint32_t ui32Ref)`
- `void ADCSequenceConfigure (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Trigger, uint32_t ui32Priority)`
- `int32_t ADCSequenceDataGet (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t *pui32Buffer)`
- `void ADCSequenceDisable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCSequenceDMADisable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCSequenceDMAEnable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCSequenceEnable (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `int32_t ADCSequenceOverflow (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCSequenceOverflowClear (uint32_t ui32Base, uint32_t ui32SequenceNum)`
- `void ADCSequenceStepConfigure (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config)`
- `int32_t ADCSequenceUnderflow (uint32_t ui32Base, uint32_t ui32SequenceNum)`

- void [ADCSequenceUnderflowClear](#) (uint32_t ui32Base, uint32_t ui32SequenceNum)
- void [ADCSoftwareOversampleConfigure](#) (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Factor)
- void [ADCSoftwareOversampleDataGet](#) (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t *pu32Buffer, uint32_t ui32Count)
- void [ADCSoftwareOversampleStepConfigure](#) (uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config)

4.2.1 Detailed Description

The analog to digital converter API is broken into three groups of functions: those that deal with the sample sequencers, those that deal with the processor trigger, and those that deal with interrupt handling.

The sample sequencers are configured with [ADCSequenceConfigure\(\)](#) and [ADCSequenceStepConfigure\(\)](#). They are enabled and disabled with [ADCSequenceEnable\(\)](#) and [ADCSequenceDisable\(\)](#). The captured data is obtained with [ADCSequenceDataGet\(\)](#). Sample sequencer FIFO overflow and underflow is managed with [ADCSequenceOverflow\(\)](#), [ADCSequenceOverflowClear\(\)](#), [ADCSequenceUnderflow\(\)](#), and [ADCSequenceUnderflowClear\(\)](#).

Hardware oversampling of the ADC is controlled with [ADCHardwareOversampleConfigure\(\)](#). Software oversampling of the ADC is controlled with [ADCSoftwareOversampleConfigure\(\)](#), [ADCSoftwareOversampleStepConfigure\(\)](#), and [ADCSoftwareOversampleDataGet\(\)](#).

The processor trigger is generated with [ADCProcessorTrigger\(\)](#).

The interrupt handler for the ADC sample sequencer interrupts are managed with [ADCIntRegister\(\)](#) and [ADCIntUnregister\(\)](#). The sample sequencer interrupt sources are managed with [ADCIntDisable\(\)](#), [ADCIntEnable\(\)](#), [ADCIntStatus\(\)](#), and [ADCIntClear\(\)](#).

4.2.2 Function Documentation

4.2.2.1 ADCBusy

Determines whether the ADC is busy or not.

Prototype:

```
bool  
ADCBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the ADC.

Description:

This function allows the caller to determine whether or not the ADC is currently sampling . If **false** is returned, then the ADC is not sampling data.

Use this function to detect that the ADC is finished sampling data before putting the device into deep sleep. Before using this function, it is highly recommended that the event trigger is changed to **ADC_TRIGGER_NEVER** on all enabled sequencers to prevent the ADC from starting after checking the busy status.

Returns:

Returns **true** if the ADC is sampling or **false** if all samples are complete.

4.2.2.2 ADCClockConfigGet

Returns the clock configuration for the ADC.

Prototype:

```
uint32_t  
ADCClockConfigGet(uint32_t ui32Base,  
                  uint32_t *pui32ClockDiv)
```

Parameters:

ui32Base is the base address of the ADC to configure, which must always be **ADC0_BASE**.

pui32ClockDiv is a pointer to the input clock divider for the clock selected by the **ADC_CLOCK_SRC** in use by the ADCs.

Description:

This function returns the ADC clock configuration and the clock divider for the ADCs.

Example: Read the current ADC clock configuration.

```
uint32_t ui32Config, ui32ClockDiv;  
  
//  
// Read the current ADC clock configuration.  
//  
ui32Config = ADCClockConfigGet(ADC0_BASE, &ui32ClockDiv);
```

Returns:

The current clock configuration of the ADC defined as a combination of one of **ADC_CLOCK_SRC_PLL**, **ADC_CLOCK_SRC_PIOSC**, **ADC_CLOCK_SRC莫斯**, or **ADC_CLOCK_SRC_ALTCLK** logical ORed with one of **ADC_CLOCK_RATE_FULL**, **ADC_CLOCK_RATE_HALF**, **ADC_CLOCK_RATE_QUARTER**, or **ADC_CLOCK_RATE_EIGHTH**. See [ADCClockConfigSet\(\)](#) for more information on these values.

4.2.2.3 ADCClockConfigSet

Sets the clock configuration for the ADC.

Prototype:

```
void  
ADCClockConfigSet(uint32_t ui32Base,  
                  uint32_t ui32Config,  
                  uint32_t ui32ClockDiv)
```

Parameters:

ui32Base is the base address of the ADC to configure, which must always be **ADC0_BASE**.

ui32Config is a combination of the **ADC_CLOCK_SRC_** and **ADC_CLOCK_RATE_*** values used to configure the ADC clock input.

ui32ClockDiv is the input clock divider for the clock selected by the **ADC_CLOCK_SRC** value.

Description:

This function is used to configure the input clock to the ADC modules. The clock configuration is shared across ADC units so *ui32Base* must always be **ADC0_BASE**. The *ui32Config* value is logical OR of one of the **ADC_CLOCK_RATE_** and one of the **ADC_CLOCK_SRC_** values defined below. The **ADC_CLOCK_SRC_*** values determine the input clock for the ADC. Not all values are available on all devices so check the device data sheet to determine value configuration options. Regardless of the source, the final frequency for TM4C123x devices must be 16 MHz and for TM4C129x parts after dividing must be between 16 and 32 MHz.

Note:

For TM4C123x devices, if the PLL is enabled, the PLL/25 is used as the ADC clock unless **ADC_CLOCK_SRC_PIOSC** is specified. If the PLL is disabled, the MOSC is used as the clock source unless **ADC_CLOCK_SRC_PIOSC** is specified.

- **ADC_CLOCK_SRC_PLL** - The main PLL output (TM4x129 class only).
- **ADC_CLOCK_SRC_PIOSC** - The internal PIOSC at 16 MHz.
- **ADC_CLOCK_SRC_ALTCLK** - The output of the ALTCLK in the system control module (TM4x129 class only).
- **ADC_CLOCK_SRC_MOSC** - The external MOSC (TM4x129 class only).

ADC_CLOCK_RATE values control how often samples are provided back to the application. The values are the following:

- **ADC_CLOCK_RATE_FULL** - All samples.
- **ADC_CLOCK_RATE_HALF** - Every other sample.
- **ADC_CLOCK_RATE_QUARTER** - Every fourth sample.
- **ADC_CLOCK_RATE_EIGHTH** - Every either sample.

The *ui32ClockDiv* parameter allows for dividing a higher frequency down into the valid range for the ADCs. This parameter is typically only used **ADC_CLOCK_SRC_PLL** option because it is the only clock value that can be with the in the correct range to use the divider. The actual value ranges from 1 to 64.

Example: ADC Clock Configurations

```

// Configure the ADC to use PIOSC divided by one (16 MHz) and sample at
// half the rate.
//
ADCClockConfigSet(ADC0_BASE,  ADC_CLOCK_SRC_PIOSC | ADC_CLOCK_RATE_HALF,  1);

...
//
// Configure the ADC to use PLL at 480 MHz divided by 24 to get an ADC
// clock of 20 MHz.
//
ADCClockConfigSet(ADC0_BASE,  ADC_CLOCK_SRC_PLL | ADC_CLOCK_RATE_FULL,  24);

```

Returns:

None.

4.2.2.4 ADCComparatorConfigure

Configures an ADC digital comparator.

Prototype:

```
void  
ADCComparatorConfigure(uint32_t ui32Base,  
                        uint32_t ui32Comp,  
                        uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32Comp is the index of the comparator to configure.

ui32Config is the configuration of the comparator.

Description:

This function configures a comparator. The **ui32Config** parameter is the result of a logical OR operation between the **ADC_COMP_TRIG_xxx**, and **ADC_COMP_INT_xxx** values.

The **ADC_COMP_TRIG_xxx** term can take on the following values:

- **ADC_COMP_TRIG_NONE** to never trigger PWM fault condition.
- **ADC_COMP_TRIG_LOW_ALWAYS** to always trigger PWM fault condition when ADC output is in the low-band.
- **ADC_COMP_TRIG_LOW_ONCE** to trigger PWM fault condition once when ADC output transitions into the low-band.
- **ADC_COMP_TRIG_LOW_HALWAYS** to always trigger PWM fault condition when ADC output is in the low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_TRIG_LOW_HONCE** to trigger PWM fault condition once when ADC output transitions into low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_TRIG_MID_ALWAYS** to always trigger PWM fault condition when ADC output is in the mid-band.
- **ADC_COMP_TRIG_MID_ONCE** to trigger PWM fault condition once when ADC output transitions into the mid-band.
- **ADC_COMP_TRIG_HIGH_ALWAYS** to always trigger PWM fault condition when ADC output is in the high-band.
- **ADC_COMP_TRIG_HIGH_ONCE** to trigger PWM fault condition once when ADC output transitions into the high-band.
- **ADC_COMP_TRIG_HIGH_HALWAYS** to always trigger PWM fault condition when ADC output is in the high-band only if ADC output has been in the low-band since the last trigger output.
- **ADC_COMP_TRIG_HIGH_HONCE** to trigger PWM fault condition once when ADC output transitions into high-band only if ADC output has been in the low-band since the last trigger output.

The **ADC_COMP_INT_xxx** term can take on the following values:

- **ADC_COMP_INT_NONE** to never generate ADC interrupt.
- **ADC_COMP_INT_LOW_ALWAYS** to always generate ADC interrupt when ADC output is in the low-band.

- **ADC_COMP_INT_LOW_ONCE** to generate ADC interrupt once when ADC output transitions into the low-band.
- **ADC_COMP_INT_LOW_HALWAYS** to always generate ADC interrupt when ADC output is in the low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_INT_LOW_HONCE** to generate ADC interrupt once when ADC output transitions into low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_INT_MID_ALWAYS** to always generate ADC interrupt when ADC output is in the mid-band.
- **ADC_COMP_INT_MID_ONCE** to generate ADC interrupt once when ADC output transitions into the mid-band.
- **ADC_COMP_INT_HIGH_ALWAYS** to always generate ADC interrupt when ADC output is in the high-band.
- **ADC_COMP_INT_HIGH_ONCE** to generate ADC interrupt once when ADC output transitions into the high-band.
- **ADC_COMP_INT_HIGH_HALWAYS** to always generate ADC interrupt when ADC output is in the high-band only if ADC output has been in the low-band since the last trigger output.
- **ADC_COMP_INT_HIGH_HONCE** to generate ADC interrupt once when ADC output transitions into high-band only if ADC output has been in the low-band since the last trigger output.

Returns:

None.

4.2.2.5 ADCComparatorIntClear

Clears sample sequence comparator interrupt source.

Prototype:

```
void
ADCComparatorIntClear(uint32_t ui32Base,
                      uint32_t ui32Status)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32Status is the bit-mapped interrupts status to clear.

Description:

The specified interrupt status is cleared.

Returns:

None.

4.2.2.6 ADCComparatorIntDisable

Disables a sample sequence comparator interrupt.

Prototype:

```
void  
ADCComparatorIntDisable(uint32_t ui32Base,  
                         uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function disables the requested sample sequence comparator interrupt.

Returns:

None.

4.2.2.7 ADCComparatorIntEnable

Enables a sample sequence comparator interrupt.

Prototype:

```
void  
ADCComparatorIntEnable(uint32_t ui32Base,  
                        uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function enables the requested sample sequence comparator interrupt.

Returns:

None.

4.2.2.8 ADCComparatorIntStatus

Gets the current comparator interrupt status.

Prototype:

```
uint32_t  
ADCComparatorIntStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the ADC module.

Description:

This function returns the digital comparator interrupt status bits. This status is sequence agnostic.

Returns:

The current comparator interrupt status.

4.2.2.9 ADCComparatorRegionSet

Defines the ADC digital comparator regions.

Prototype:

```
void
ADCComparatorRegionSet(uint32_t ui32Base,
                      uint32_t ui32Comp,
                      uint32_t ui32LowRef,
                      uint32_t ui32HighRef)
```

Parameters:

- ui32Base*** is the base address of the ADC module.
- ui32Comp*** is the index of the comparator to configure.
- ui32LowRef*** is the reference point for the low/mid band threshold.
- ui32HighRef*** is the reference point for the mid/high band threshold.

Description:

The ADC digital comparator operation is based on three ADC value regions:

- **low-band** is defined as any ADC value less than or equal to the *ui32LowRef* value.
- **mid-band** is defined as any ADC value greater than the *ui32LowRef* value but less than or equal to the *ui32HighRef* value.
- **high-band** is defined as any ADC value greater than the *ui32HighRef* value.

Returns:

None.

4.2.2.10 ADCComparatorReset

Resets the current ADC digital comparator conditions.

Prototype:

```
void
ADCComparatorReset(uint32_t ui32Base,
                    uint32_t ui32Comp,
                    bool bTrigger,
                    bool bInterrupt)
```

Parameters:

- ui32Base*** is the base address of the ADC module.
- ui32Comp*** is the index of the comparator.
- bTrigger*** is the flag to indicate reset of Trigger conditions.
- bInterrupt*** is the flag to indicate reset of Interrupt conditions.

Description:

Because the digital comparator uses current and previous ADC values, this function allows the comparator to be reset to its initial value to prevent stale data from being used when a sequence is enabled.

Returns:

None.

4.2.2.11 ADCHardwareOversampleConfigure

Configures the hardware oversampling factor of the ADC.

Prototype:

```
void  
ADCHardwareOversampleConfigure(uint32_t ui32Base,  
                                uint32_t ui32Factor)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32Factor is the number of samples to be averaged.

Description:

This function configures the hardware oversampling for the ADC, which can be used to provide better resolution on the sampled data. Oversampling is accomplished by averaging multiple samples from the same analog input. Six different oversampling rates are supported; 2x, 4x, 8x, 16x, 32x, and 64x. Specifying an oversampling factor of zero disables hardware oversampling.

Hardware oversampling applies uniformly to all sample sequencers. It does not reduce the depth of the sample sequencers like the software oversampling APIs; each sample written into the sample sequencer FIFO is a fully oversampled analog input reading.

Enabling hardware averaging increases the precision of the ADC at the cost of throughput. For example, enabling 4x oversampling reduces the throughput of a 250 k samples/second ADC to 62.5 k samples/second.

Returns:

None.

4.2.2.12 ADCIntClear

Clears sample sequence interrupt source.

Prototype:

```
void  
ADCIntClear(uint32_t ui32Base,  
            uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

The specified sample sequence interrupt is cleared, so that it no longer asserts. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid

returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

4.2.2.13 ADCIntClearEx

Clears the specified ADC interrupt sources.

Prototype:

```
void
ADCIntClearEx(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the ADC port.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

Clears the interrupt for the specified interrupt source(s).

The *ui32IntFlags* parameter is the logical OR of the **ADC_INT_*** values. See the [ADCIntEnableEx\(\)](#) function for the list of possible **ADC_INT*** values.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

4.2.2.14 ADCIntDisable

Disables a sample sequence interrupt.

Prototype:

```
void
ADCIntDisable(uint32_t ui32Base,
              uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function disables the requested sample sequence interrupt.

Returns:

None.

4.2.2.15 ADCIntDisableEx

Disables ADC interrupt sources.

Prototype:

```
void  
ADCIntDisableEx(uint32_t ui32Base,  
                 uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

This function disables the indicated ADC interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **ADC_INT_SS0** - interrupt due to ADC sample sequence 0.
- **ADC_INT_SS1** - interrupt due to ADC sample sequence 1.
- **ADC_INT_SS2** - interrupt due to ADC sample sequence 2.
- **ADC_INT_SS3** - interrupt due to ADC sample sequence 3.
- **ADC_INT_DMA_SS0** - interrupt due to DMA on ADC sample sequence 0.
- **ADC_INT_DMA_SS1** - interrupt due to DMA on ADC sample sequence 1.
- **ADC_INT_DMA_SS2** - interrupt due to DMA on ADC sample sequence 2.
- **ADC_INT_DMA_SS3** - interrupt due to DMA on ADC sample sequence 3.
- **ADC_INT_DCON_SS0** - interrupt due to digital comparator on ADC sample sequence 0.
- **ADC_INT_DCON_SS1** - interrupt due to digital comparator on ADC sample sequence 1.
- **ADC_INT_DCON_SS2** - interrupt due to digital comparator on ADC sample sequence 2.
- **ADC_INT_DCON_SS3** - interrupt due to digital comparator on ADC sample sequence 3.

Returns:

None.

4.2.2.16 ADCIntEnable

Enables a sample sequence interrupt.

Prototype:

```
void  
ADCIntEnable(uint32_t ui32Base,  
             uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function enables the requested sample sequence interrupt. Any outstanding interrupts are cleared before enabling the sample sequence interrupt.

Returns:

None.

4.2.2.17 ADCIntEnableEx

Enables ADC interrupt sources.

Prototype:

```
void
ADCIntEnableEx(uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

This function enables the indicated ADC interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **ADC_INT_SS0** - interrupt due to ADC sample sequence 0.
- **ADC_INT_SS1** - interrupt due to ADC sample sequence 1.
- **ADC_INT_SS2** - interrupt due to ADC sample sequence 2.
- **ADC_INT_SS3** - interrupt due to ADC sample sequence 3.
- **ADC_INT_DMA_SS0** - interrupt due to DMA on ADC sample sequence 0.
- **ADC_INT_DMA_SS1** - interrupt due to DMA on ADC sample sequence 1.
- **ADC_INT_DMA_SS2** - interrupt due to DMA on ADC sample sequence 2.
- **ADC_INT_DMA_SS3** - interrupt due to DMA on ADC sample sequence 3.
- **ADC_INT_DCON_SS0** - interrupt due to digital comparator on ADC sample sequence 0.
- **ADC_INT_DCON_SS1** - interrupt due to digital comparator on ADC sample sequence 1.
- **ADC_INT_DCON_SS2** - interrupt due to digital comparator on ADC sample sequence 2.
- **ADC_INT_DCON_SS3** - interrupt due to digital comparator on ADC sample sequence 3.

Returns:

None.

4.2.2.18 ADCIntRegister

Registers an interrupt handler for an ADC interrupt.

Prototype:

```
void  
ADCIntRegister(uint32_t ui32Base,  
                uint32_t ui32SequenceNum,  
                void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

pfnHandler is a pointer to the function to be called when the ADC sample sequence interrupt occurs.

Description:

This function sets the handler to be called when a sample sequence interrupt occurs. This function enables the global interrupt in the interrupt controller; the sequence interrupt must be enabled with [ADCIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [ADCIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

4.2.2.19 ADCIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t  
ADCIntStatus(uint32_t ui32Base,  
             uint32_t ui32SequenceNum,  
             bool bMasked)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the specified sample sequence. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current raw or masked interrupt status.

4.2.2.20 ADCIntStatusEx

Gets interrupt status for the specified ADC module.

Prototype:

```
uint32_t
ADCIntStatusEx(uint32_t ui32Base,
               bool bMasked)
```

Parameters:

ui32Base is the base address of the ADC module.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If **bMasked** is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

Returns the current interrupt status for the specified ADC module. The value returned is the logical OR of the **ADC_INT_*** values that are currently active.

4.2.2.21 ADCIntUnregister

Unregisters the interrupt handler for an ADC interrupt.

Prototype:

```
void
ADCIntUnregister(uint32_t ui32Base,
                  uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function unregisters the interrupt handler. This function disables the global interrupt in the interrupt controller; the sequence interrupt must be disabled via [ADCIntDisable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

4.2.2.22 ADCPhaseDelayGet

Gets the phase delay between a trigger and the start of a sequence.

Prototype:

```
uint32_t
ADCPhaseDelayGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the ADC module.

Description:

This function gets the current phase delay between the detection of an ADC trigger event and the start of the sample sequence.

Returns:

Returns the phase delay, specified as one of **ADC_PHASE_0**, **ADC_PHASE_22_5**, **ADC_PHASE_45**, **ADC_PHASE_67_5**, **ADC_PHASE_90**, **ADC_PHASE_112_5**, **ADC_PHASE_135**, **ADC_PHASE_157_5**, **ADC_PHASE_180**, **ADC_PHASE_202_5**, **ADC_PHASE_225**, **ADC_PHASE_247_5**, **ADC_PHASE_270**, **ADC_PHASE_292_5**, **ADC_PHASE_315**, or **ADC_PHASE_337_5**.

4.2.2.23 ADCPhaseDelaySet

Sets the phase delay between a trigger and the start of a sequence.

Prototype:

```
void  
ADCPhaseDelaySet (uint32_t ui32Base,  
                  uint32_t ui32Phase)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32Phase is the phase delay, specified as one of **ADC_PHASE_0**, **ADC_PHASE_22_5**, **ADC_PHASE_45**, **ADC_PHASE_67_5**, **ADC_PHASE_90**, **ADC_PHASE_112_5**, **ADC_PHASE_135**, **ADC_PHASE_157_5**, **ADC_PHASE_180**, **ADC_PHASE_202_5**, **ADC_PHASE_225**, **ADC_PHASE_247_5**, **ADC_PHASE_270**, **ADC_PHASE_292_5**, **ADC_PHASE_315**, or **ADC_PHASE_337_5**.

Description:

This function sets the phase delay between the detection of an ADC trigger event and the start of the sample sequence. By selecting a different phase delay for a pair of ADC modules (such as **ADC_PHASE_0** and **ADC_PHASE_180**) and having each ADC module sample the same analog input, it is possible to increase the sampling rate of the analog input (with samples N, N+2, N+4, and so on, coming from the first ADC and samples N+1, N+3, N+5, and so on, coming from the second ADC). The ADC module has a single phase delay that is applied to all sample sequences within that module.

Note:

This capability is not available on all parts.

Returns:

None.

4.2.2.24 ADCProcessorTrigger

Causes a processor trigger for a sample sequence.

Prototype:

```
void  
ADCProcessorTrigger (uint32_t ui32Base,  
                     uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number, with **ADC_TRIGGER_WAIT** or **ADC_TRIGGER_SIGNAL** optionally ORed into it.

Description:

This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to **ADC_TRIGGER_PROCESSOR**. If **ADC_TRIGGER_WAIT** is ORed into the sequence number, the processor-initiated trigger is delayed until a later processor-initiated trigger to a different ADC module that specifies **ADC_TRIGGER_SIGNAL**, allowing multiple ADCs to start from a processor-initiated trigger in a synchronous manner.

Returns:

None.

4.2.2.25 ADCReferenceGet

Returns the current setting of the ADC reference.

Prototype:

```
uint32_t  
ADCReferenceGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the ADC module.

Description:

Returns the value of the ADC reference setting. The returned value is one of **ADC_REF_INT**, **ADC_REF_EXT_3V**, or **ADC_REF_EXT_1V**.

Note:

The value returned by this function is only meaningful if used on a part that is capable of using an external reference. Consult the data sheet for your part to determine if it has an external reference input.

Returns:

The current setting of the ADC reference.

4.2.2.26 ADCReferenceSet

Selects the ADC reference.

Prototype:

```
void  
ADCReferenceSet(uint32_t ui32Base,  
                uint32_t ui32Ref)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32Ref is the reference to use.

Description:

The ADC reference is set as specified by *ui32Ref*. It must be one of **ADC_REF_INT**, **ADC_REF_EXT_3V**, or **ADC_REF_EXT_1V** for internal or external reference. If **ADC_REF_INT** is chosen, then an internal 3V reference is used and no external reference is needed. If **ADC_REF_EXT_3V** is chosen, then a 3V reference must be supplied to the AVREF pin. If **ADC_REF_EXT_1V** is chosen, then a 1V external reference must be supplied to the AVREF pin.

Note:

The ADC reference can only be selected on parts that have an external reference. Consult the data sheet for your part to determine if there is an external reference.

Returns:

None.

4.2.2.27 ADCSequenceConfigure

Configures the trigger source and priority of a sample sequence.

Prototype:

```
void  
ADCSequenceConfigure(uint32_t ui32Base,  
                      uint32_t ui32SequenceNum,  
                      uint32_t ui32Trigger,  
                      uint32_t ui32Priority)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

ui32Trigger is the trigger source that initiates the sample sequence; must be one of the **ADC_TRIGGER_*** values.

ui32Priority is the relative priority of the sample sequence with respect to the other sample sequences.

Description:

This function configures the initiation criteria for a sample sequence. Valid sample sequencers range from zero to three; sequencer zero captures up to eight samples, sequencers one and two capture up to four samples, and sequencer three captures a single sample. The trigger condition and priority (with respect to other sample sequencer execution) are set.

The *ui32Trigger* parameter can take on the following values:

- **ADC_TRIGGER_PROCESSOR** - A trigger generated by the processor, via the [ADCProcessorTrigger\(\)](#) function.
- **ADC_TRIGGER_COMP0** - A trigger generated by the first analog comparator; configured with [ComparatorConfigure\(\)](#).
- **ADC_TRIGGER_COMP1** - A trigger generated by the second analog comparator; configured with [ComparatorConfigure\(\)](#).
- **ADC_TRIGGER_COMP2** - A trigger generated by the third analog comparator; configured with [ComparatorConfigure\(\)](#).
- **ADC_TRIGGER_EXTERNAL** - A trigger generated by an input from the Port B4 pin. Note that some microcontrollers can select from any GPIO using the [GPIOADCTriggerEnable\(\)](#) function.

- **ADC_TRIGGER_TIMER** - A trigger generated by a timer; configured with [TimerControlTrigger\(\)](#).
- **ADC_TRIGGER_PWM0** - A trigger generated by the first PWM generator; configured with [PWMGenIntTrigEnable\(\)](#).
- **ADC_TRIGGER_PWM1** - A trigger generated by the second PWM generator; configured with [PWMGenIntTrigEnable\(\)](#).
- **ADC_TRIGGER_PWM2** - A trigger generated by the third PWM generator; configured with [PWMGenIntTrigEnable\(\)](#).
- **ADC_TRIGGER_PWM3** - A trigger generated by the fourth PWM generator; configured with [PWMGenIntTrigEnable\(\)](#).
- **ADC_TRIGGER_ALWAYS** - A trigger that is always asserted, causing the sample sequence to capture repeatedly (so long as there is not a higher priority source active).

When **ADC_TRIGGER_PWM0**, **ADC_TRIGGER_PWM1**, **ADC_TRIGGER_PWM2** or **ADC_TRIGGER_PWM3** is specified, one of the following should be ORed into *ui32Trigger* to select the PWM module from which the triggers will be routed for this sequence:

- **ADC_TRIGGER_PWM_MOD0** - Selects PWM module 0 as the source of the PWM0 to PWM3 triggers for this sequence.
- **ADC_TRIGGER_PWM_MOD1** - Selects PWM module 1 as the source of the PWM0 to PWM3 triggers for this sequence.

Note that not all trigger sources are available on all Tiva family members; consult the data sheet for the device in question to determine the availability of triggers.

The *ui32Priority* parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest. Note that when programming the priority among a set of sample sequences, each must have unique priority; it is up to the caller to guarantee the uniqueness of the priorities.

Returns:

None.

4.2.2.28 ADCSequenceDataGet

Gets the captured data for a sample sequence.

Prototype:

```
int32_t
ADCSequenceDataGet(uint32_t ui32Base,
                   uint32_t ui32SequenceNum,
                   uint32_t *pui32Buffer)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

pui32Buffer is the address where the data is stored.

Description:

This function copies data from the specified sample sequencer output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples. This function only returns

the samples that are presently available, which may not be the entire sample sequence if it is in the process of being executed.

Returns:

Returns the number of samples copied to the buffer.

4.2.2.29 ADCSequenceDisable

Disables a sample sequence.

Prototype:

```
void  
ADCSequenceDisable(uint32_t ui32Base,  
                    uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

Prevents the specified sample sequence from being captured when its trigger is detected. A sample sequence must be disabled before it is configured.

Returns:

None.

4.2.2.30 ADCSequenceDMADisable

Disables DMA for sample sequencers.

Prototype:

```
void  
ADCSequenceDMADisable(uint32_t ui32Base,  
                      uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

Prevents the specified sample sequencer from generating DMA requests.

Returns:

None.

4.2.2.31 ADCSequenceDMAEnable

Enables DMA for sample sequencers.

Prototype:

```
void
ADCSequenceDMAEnable(uint32_t ui32Base,
                      uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

Allows DMA requests to be generated based on the FIFO level of the sample sequencer.

Returns:

None.

4.2.2.32 ADCSequenceEnable

Enables a sample sequence.

Prototype:

```
void
ADCSequenceEnable(uint32_t ui32Base,
                  uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

Allows the specified sample sequence to be captured when its trigger is detected. A sample sequence must be configured before it is enabled.

Returns:

None.

4.2.2.33 ADCSequenceOverflow

Determines if a sample sequence overflow occurred.

Prototype:

```
int32_t
ADCSequenceOverflow(uint32_t ui32Base,
                    uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function determines if a sample sequence overflow has occurred. Overflow happens if the captured samples are not read from the FIFO before the next trigger occurs.

Returns:

Returns zero if there was not an overflow, and non-zero if there was.

4.2.2.34 ADCSequenceOverflowClear

Clears the overflow condition on a sample sequence.

Prototype:

```
void  
ADCSequenceOverflowClear(uint32_t ui32Base,  
                        uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function clears an overflow condition on one of the sample sequences. The overflow condition must be cleared in order to detect a subsequent overflow condition (it otherwise causes no harm).

Returns:

None.

4.2.2.35 ADCSequenceStepConfigure

Configure a step of the sample sequencer.

Prototype:

```
void  
ADCSequenceStepConfigure(uint32_t ui32Base,  
                        uint32_t ui32SequenceNum,  
                        uint32_t ui32Step,  
                        uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

ui32Step is the step to be configured.

ui32Config is the configuration of this step; must be a logical OR of **ADC_CTL_TS**, **ADC_CTL_IE**, **ADC_CTL_END**, **ADC_CTL_D**, one of the input channel selects (**ADC_CTL_CH0** through **ADC_CTL_CH23**), and one of the digital comparator selects (**ADC_CTL_CMP0** through **ADC_CTL_CMP7**).

Description:

This function configures the ADC for one step of a sample sequence. The ADC can be configured for single-ended or differential operation (the **ADC_CTL_D** bit selects differential operation when set), the channel to be sampled can be chosen (the **ADC_CTL_CH0** through **ADC_CTL_CH23** values), and the internal temperature sensor can be selected (the **ADC_CTL_TS** bit). Additionally, this step can be defined as the last in the sequence (the

ADC_CTL_END bit) and it can be configured to cause an interrupt when the step is complete (the **ADC_CTL_IE** bit). If the digital comparators are present on the device, this step may also be configured to send the ADC sample to the selected comparator using **ADC_CTL_CMP0** through **ADC_CTL_CMP7**. The configuration is used by the ADC at the appropriate time when the trigger for this sequence occurs.

Note:

If the Digital Comparator is present and enabled using the **ADC_CTL_CMP0** through **ADC_CTL_CMP7** selects, the ADC sample is NOT written into the ADC sequence data FIFO.

The *ui32Step* parameter determines the order in which the samples are captured by the ADC when the trigger occurs. It can range from zero to seven for the first sample sequencer, from zero to three for the second and third sample sequencer, and can only be zero for the fourth sample sequencer.

Differential mode only works with adjacent channel pairs (for example, 0 and 1). The channel select must be the number of the channel pair to sample (for example, **ADC_CTL_CH0** for 0 and 1, or **ADC_CTL_CH1** for 2 and 3) or undefined results are returned by the ADC. Additionally, if differential mode is selected when the temperature sensor is being sampled, undefined results are returned by the ADC.

It is the responsibility of the caller to ensure that a valid configuration is specified; this function does not check the validity of the specified configuration.

Returns:

None.

4.2.2.36 ADCSequenceUnderflow

Determines if a sample sequence underflow occurred.

Prototype:

```
int32_t
ADCSequenceUnderflow(uint32_t ui32Base,
                     uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function determines if a sample sequence underflow has occurred. Underflow happens if too many samples are read from the FIFO.

Returns:

Returns zero if there was not an underflow, and non-zero if there was.

4.2.2.37 ADCSequenceUnderflowClear

Clears the underflow condition on a sample sequence.

Prototype:

```
void  
ADCSequenceUnderflowClear(uint32_t ui32Base,  
                           uint32_t ui32SequenceNum)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

Description:

This function clears an underflow condition on one of the sample sequencers. The underflow condition must be cleared in order to detect a subsequent underflow condition (it otherwise causes no harm).

Returns:

None.

4.2.2.38 ADCSoftwareOversampleConfigure

Configures the software oversampling factor of the ADC.

Prototype:

```
void  
ADCSoftwareOversampleConfigure(uint32_t ui32Base,  
                                 uint32_t ui32SequenceNum,  
                                 uint32_t ui32Factor)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

ui32Factor is the number of samples to be averaged.

Description:

This function configures the software oversampling for the ADC, which can be used to provide better resolution on the sampled data. Oversampling is accomplished by averaging multiple samples from the same analog input. Three different oversampling rates are supported; 2x, 4x, and 8x.

Oversampling is only supported on the sample sequencers that are more than one sample in depth (that is, the fourth sample sequencer is not supported). Oversampling by 2x (for example) divides the depth of the sample sequencer by two; so 2x oversampling on the first sample sequencer can only provide four samples per trigger. This also means that 8x oversampling is only available on the first sample sequencer.

Returns:

None.

4.2.2.39 ADCSoftwareOversampleDataGet

Gets the captured data for a sample sequence using software oversampling.

Prototype:

```
void
ADCSoftwareOversampleDataGet(uint32_t ui32Base,
                             uint32_t ui32SequenceNum,
                             uint32_t *pui32Buffer,
                             uint32_t ui32Count)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

pui32Buffer is the address where the data is stored.

ui32Count is the number of samples to be read.

Description:

This function copies data from the specified sample sequence output FIFO to a memory resident buffer with software oversampling applied. The requested number of samples are copied into the data buffer; if there are not enough samples in the hardware FIFO to satisfy this many oversampled data items, then incorrect results are returned. It is the caller's responsibility to read only the samples that are available and wait until enough data is available, for example as a result of receiving an interrupt.

Returns:

None.

4.2.2.40 ADCSoftwareOversampleStepConfigure

Configures a step of the software oversampled sequencer.

Prototype:

```
void
ADCSoftwareOversampleStepConfigure(uint32_t ui32Base,
                                    uint32_t ui32SequenceNum,
                                    uint32_t ui32Step,
                                    uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the ADC module.

ui32SequenceNum is the sample sequence number.

ui32Step is the step to be configured.

ui32Config is the configuration of this step.

Description:

This function configures a step of the sample sequencer when using the software oversampling feature. The number of steps available depends on the oversampling factor set by [ADCSoftwareOversampleConfigure\(\)](#). The value of **ui32Config** is the same as defined for [ADCSequenceStepConfigure\(\)](#).

Returns:

None.

4.3 Programming Example

The following example shows how to use the ADC API to initialize a sample sequencer for processor triggering, trigger the sample sequence, and then read back the data when it is ready.

```
uint32_t ui32Value;

//
// Enable the first sample sequencer to capture the value of channel 0 when
// the processor trigger occurs.
//
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                        ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC0_BASE, 0);

//
// Trigger the sample sequence.
//
ADCPProcessorTrigger(ADC0_BASE, 0);

//
// Wait until the sample sequence has completed.
//
while(!ADCIntStatus(ADC0_BASE, 0, false))
{
}

//
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

5 AES

Introduction	47
API Functions	47
Programming Example	62

5.1 Introduction

The AES module driver provides a method for performing encryption and decryption operations on blocks of 128-bits of data. The configuration and feature highlights are:

- Supports ECB, CBC, CTR, ICM, CFB, CBC-MAC, GCM, CCM, XTS, F8, and F9 operating modes.
- The cipher block handles keys of 128-bits, 192-bits, and 256 bits.
- In modes that require authentication, a hash tag is generated.
- Controls uDMA triggers for context and data transfers.

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

5.2 API Functions

Functions

- void `AESAuthLengthSet` (uint32_t ui32Base, uint32_t ui32Length)
- void `AESConfigSet` (uint32_t ui32Base, uint32_t ui32Config)
- bool `AESDataAuth` (uint32_t ui32Base, uint32_t *pui32Src, uint32_t ui32Length, uint32_t *pui32Tag)
- bool `AESDataProcess` (uint32_t ui32Base, uint32_t *pui32Src, uint32_t *pui32Dest, uint32_t ui32Length)
- bool `AESDataProcessAuth` (uint32_t ui32Base, uint32_t *pui32Src, uint32_t *pui32Dest, uint32_t ui32Length, uint32_t *pui32AuthSrc, uint32_t ui32AuthLength, uint32_t *pui32Tag)
- void `AESDataRead` (uint32_t ui32Base, uint32_t *pui32Dest)
- bool `AESDataReadNonBlocking` (uint32_t ui32Base, uint32_t *pui32Dest)
- void `AESDataWrite` (uint32_t ui32Base, uint32_t *pui32Src)
- bool `AESDataWriteNonBlocking` (uint32_t ui32Base, uint32_t *pui32Src)
- void `AESDMADisable` (uint32_t ui32Base, uint32_t ui32Flags)
- void `AESDMAEnable` (uint32_t ui32Base, uint32_t ui32Flags)
- void `AESIntClear` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `AESIntDisable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `AESIntEnable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `AESIntRegister` (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t `AESIntStatus` (uint32_t ui32Base, bool bMasked)

- void [AESIntUnregister](#) (uint32_t ui32Base)
- void [AESIVRead](#) (uint32_t ui32Base, uint32_t *pui32IVData)
- void [AESIVSet](#) (uint32_t ui32Base, uint32_t *pui32IVData)
- void [AESKey1Set](#) (uint32_t ui32Base, uint32_t *pui32Key, uint32_t ui32Keysize)
- void [AESKey2Set](#) (uint32_t ui32Base, uint32_t *pui32Key, uint32_t ui32Keysize)
- void [AESKey3Set](#) (uint32_t ui32Base, uint32_t *pui32Key)
- void [AESLengthSet](#) (uint32_t ui32Base, uint64_t ui64Length)
- void [AESReset](#) (uint32_t ui32Base)
- void [AESTagRead](#) (uint32_t ui32Base, uint32_t *pui32TagData)

5.2.1 Detailed Description

The AES API consists of functions for configuring the AES module and processing data.

5.2.2 Function Documentation

5.2.2.1 AESAuthLengthSet

Sets the authentication data length in the AES module.

Prototype:

```
void  
AESAuthLengthSet (uint32_t ui32Base,  
                  uint32_t ui32Length)
```

Parameters:

ui32Base is the base address of the AES module.

ui32Length is the length in bytes.

Description:

This function is only used to write the authentication data length in the combined modes (GCM or CCM) and XTS mode. Supported AAD lengths for CCM are from 0 to ($2^{16} - 28$) bytes. For GCM, any value up to ($2^{32} - 1$) can be used. For XTS mode, this register is used to load j. Loading of j is only required if j != 0. j represents the sequential number of the 128-bit blocks inside the data unit. Consequently, j must be multiplied by 16 when passed to this function, thereby placing the block number in bits [31:4] of the register.

When this function is called, the engine is triggered to start using this context for GCM and CCM.

Returns:

None

5.2.2.2 AESConfigSet

Configures the AES module.

Prototype:

```
void
AESConfigSet(uint32_t ui32Base,
              uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the AES module.

ui32Config is the configuration of the AES module.

Description:

This function configures the AES module based on the specified parameters. It does not change any DMA- or interrupt-related parameters.

The ui32Config parameter is a bit-wise OR of a number of configuration flags. The valid flags are grouped based on their function.

The direction of the operation is specified with only of following flags:

- **AES_CFG_DIR_ENCRYPT** - Encryption mode
- **AES_CFG_DIR_DECRYPT** - Decryption mode

The key size is specified with only one of the following flags:

- **AES_CFG_KEY_SIZE_128BIT** - Key size of 128 bits
- **AES_CFG_KEY_SIZE_192BIT** - Key size of 192 bits
- **AES_CFG_KEY_SIZE_256BIT** - Key size of 256 bits

The mode of operation is specified with only one of the following flags.

- **AES_CFG_MODE_ECB** - Electronic codebook mode
- **AES_CFG_MODE_CBC** - Cipher-block chaining mode
- **AES_CFG_MODE_CFB** - Cipher feedback mode
- **AES_CFG_MODE_CTR** - Counter mode
- **AES_CFG_MODE_ICM** - Integer counter mode
- **AES_CFG_MODE_XTS** - Ciphertext stealing mode
- **AES_CFG_MODE_XTS_TWEAKJL** - XEX-based tweaked-codebook mode with ciphertext stealing with previous/intermediate tweak value and j loaded
- **AES_CFG_MODE_XTS_K2IJL** - XEX-based tweaked-codebook mode with ciphertext stealing with key2, i and j loaded
- **AES_CFG_MODE_XTS_K2ILJ0** - XEX-based tweaked-codebook mode with ciphertext stealing with key2 and i loaded, j = 0
- **AES_CFG_MODE_F8** - F8 mode
- **AES_CFG_MODE_F9** - F9 mode
- **AES_CFG_MODE_CBCMAC** - Cipher block chaining message authentication code mode
- **AES_CFG_MODE_GCM_HLY0ZERO** - Galois/counter mode with GHASH with H loaded, Y0-encrypted forced to zero and counter is not enabled.
- **AES_CFG_MODE_GCM_HLY0CALC** - Galois/counter mode with GHASH with H loaded, Y0-encrypted calculated internally and counter is enabled.
- **AES_CFG_MODE_GCM_HY0CALC** - Galois/Counter mode with autonomous GHASH (both H and Y0-encrypted calculated internally) and counter is enabled.
- **AES_CFG_MODE_CCM** - Counter with CBC-MAC mode

The following defines are used to specify the counter width. It is only required to be defined when using CTR, CCM, or GCM modes, only one of the following defines must be used to specify the counter width length:

- **AES_CFG_CTR_WIDTH_32** - Counter is 32 bits
- **AES_CFG_CTR_WIDTH_64** - Counter is 64 bits
- **AES_CFG_CTR_WIDTH_96** - Counter is 96 bits
- **AES_CFG_CTR_WIDTH_128** - Counter is 128 bits

Only one of the following defines must be used to specify the length field for CCM operations (L):

- **AES_CFG_CCM_L_1** - 1 byte
- **AES_CFG_CCM_L_2** - 2 bytes
- **AES_CFG_CCM_L_3** - 3 bytes
- **AES_CFG_CCM_L_4** - 4 bytes
- **AES_CFG_CCM_L_5** - 5 bytes
- **AES_CFG_CCM_L_6** - 6 bytes
- **AES_CFG_CCM_L_7** - 7 bytes
- **AES_CFG_CCM_L_8** - 8 bytes

Only one of the following defines must be used to specify the length of the authentication field for CCM operations (M) through the *ui32Config* argument in the [AESConfigSet\(\)](#) function:

- **AES_CFG_CCM_M_4** - 4 bytes
- **AES_CFG_CCM_M_6** - 6 bytes
- **AES_CFG_CCM_M_8** - 8 bytes
- **AES_CFG_CCM_M_10** - 10 bytes
- **AES_CFG_CCM_M_12** - 12 bytes
- **AES_CFG_CCM_M_14** - 14 bytes
- **AES_CFG_CCM_M_16** - 16 bytes

Note:

When performing a basic GHASH operation for used with GCM mode, use the **AES_CFG_MODE_GCM_HLY0ZERO** and do not specify a direction.

Returns:

None.

5.2.2.3 AESDataAuth

Used to authenticate blocks of data by generating a hash tag.

Prototype:

```
bool  
AESDataAuth(uint32_t ui32Base,  
            uint32_t *pui32Src,  
            uint32_t ui32Length,  
            uint32_t *pui32Tag)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Src is a pointer to the memory location where the input data is stored. The data must be padded to the 16-byte boundary.

ui32Length is the length of the cryptographic data in bytes.

pui32Tag is a pointer to a 4-word array where the hash tag is written.

Description:

This function processes data to produce a hash tag that can be used for authentication. Before calling this function, ensure that the AES module is properly configured the key, data size, mode, etc. Only CBC-MAC and F9 modes should be used.

Returns:

Returns true if data was processed successfully. Returns false if data processing failed.

5.2.2.4 AESDataProcess

Used to process(transform) blocks of data, either encrypt or decrypt it.

Prototype:

```
bool
AESDataProcess(uint32_t ui32Base,
               uint32_t *pui32Src,
               uint32_t *pui32Dest,
               uint32_t ui32Length)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Src is a pointer to the memory location where the input data is stored. The data must be padded to the 16-byte boundary.

pui32Dest is a pointer to the memory location output is written. The space for written data must be rounded up to the 16-byte boundary.

ui32Length is the length of the cryptographic data in bytes.

Description:

This function iterates the encryption or decryption mechanism number over the data length. Before calling this function, ensure that the AES module is properly configured the key, data size, mode, etc. Only ECB, CBC, CTR, ICM, CFB, XTS and F8 operating modes should be used. The data is processed in 4-word (16-byte) blocks.

Note:

This function only supports values of **ui32Length** less than 2^{32} , because the memory size is restricted to between 0 to 2^{32} bytes.

Returns:

Returns true if data was processed successfully. Returns false if data processing failed.

5.2.2.5 AESDataProcessAuth

Processes and authenticates blocks of data, either encrypt it or decrypts it.

Prototype:

```
bool  
AESDataProcessAuth(uint32_t ui32Base,  
                    uint32_t *pui32Src,  
                    uint32_t *pui32Dest,  
                    uint32_t ui32Length,  
                    uint32_t *pui32AuthSrc,  
                    uint32_t ui32AuthLength,  
                    uint32_t *pui32Tag)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Src is a pointer to the memory location where the input data is stored. The data must be padded to the 16-byte boundary.

pui32Dest is a pointer to the memory location output is written. The space for written data must be rounded up to the 16-byte boundary.

ui32Length is the length of the cryptographic data in bytes.

pui32AuthSrc is a pointer to the memory location where the additional authentication data is stored. The data must be padded to the 16-byte boundary.

ui32AuthLength is the length of the additional authentication data in bytes.

pui32Tag is a pointer to a 4-word array where the hash tag is written.

Description:

This function encrypts or decrypts blocks of data in addition to authentication data. A hash tag is also produced. Before calling this function, ensure that the AES module is properly configured the key, data size, mode, etc. Only CCM and GCM modes should be used.

Returns:

Returns true if data was processed successfully. Returns false if data processing failed.

5.2.2.6 AESDataRead

Reads plaintext/ciphertext from data registers with blocking.

Prototype:

```
void  
AESDataRead(uint32_t ui32Base,  
            uint32_t *pui32Dest)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Dest is a pointer to an array of words.

Description:

This function reads a block of either plaintext or ciphertext out of the AES module. If the output is not ready, the function waits until it is ready. A block is 16 bytes or 4 words.

Returns:

None.

5.2.2.7 AESDataReadNonBlocking

Reads plaintext/ciphertext from data registers without blocking.

Prototype:

```
bool
AESDataReadNonBlocking(uint32_t ui32Base,
                      uint32_t *pui32Dest)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Dest is a pointer to an array of words of data.

Description:

This function reads a block of either plaintext or ciphertext out of the AES module. If the output data is not ready, the function returns false. If the read completed successfully, the function returns true. A block is 16 bytes or 4 words.

Returns:

true or false.

5.2.2.8 AESDataWrite

Writes plaintext/ciphertext to data registers with blocking.

Prototype:

```
void
AESDataWrite(uint32_t ui32Base,
             uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the AES module.

pui32Src is a pointer to an array of bytes.

Description:

This function writes a block of either plaintext or ciphertext into the AES module. If the input is not ready, the function waits until it is ready before performing the write. A block is 16 bytes or 4 words.

Returns:

None.

5.2.2.9 AESDataWriteNonBlocking

Writes plaintext/ciphertext to data registers without blocking.

Prototype:

```
bool
AESDataWriteNonBlocking(uint32_t ui32Base,
                       uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the AES module.
pui32Src is a pointer to an array of words of data.

Description:

This function writes a block of either plaintext or ciphertext into the AES module. If the input is not ready, the function returns false. If the write completed successfully, the function returns true. A block is 16 bytes or 4 words.

Returns:

True or false.

5.2.2.10 AESDMADisable

Disables uDMA requests for the AES module.

Prototype:

```
void  
AESDMADisable(uint32_t ui32Base,  
               uint32_t ui32Flags)
```

Parameters:

ui32Base is the base address of the AES module.
ui32Flags is a bit mask of the uDMA requests to be disabled.

Description:

This function disables the uDMA request sources in the AES module. The *ui32Flags* parameter is the logical OR of any of the following:

- **AES_DMA_DATA_IN**
- **AES_DMA_DATA_OUT**
- **AES_DMA_CONTEXT_IN**
- **AES_DMA_CONTEXT_OUT**

Returns:

None.

5.2.2.11 AESDMAEnable

Enables uDMA requests for the AES module.

Prototype:

```
void  
AESDMAEnable(uint32_t ui32Base,  
              uint32_t ui32Flags)
```

Parameters:

ui32Base is the base address of the AES module.
ui32Flags is a bit mask of the uDMA requests to be enabled.

Description:

This function enables the uDMA request sources in the AES module. The *ui32Flags* parameter is the logical OR of any of the following:

- **AES_DMA_DATA_IN**
- **AES_DMA_DATA_OUT**
- **AES_DMA_CONTEXT_IN**
- **AES_DMA_CONTEXT_OUT**

Returns:

None.

5.2.2.12 AESIntClear

Clears AES module interrupts.

Prototype:

```
void
AESIntClear(uint32_t ui32Base,
            uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the AES module.

ui32IntFlags is a bit mask of the interrupt sources to disable.

Description:

This function clears the interrupt sources in the AES module. The *ui32IntFlags* parameter is the logical OR of any of the following:

- **AES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **AES_INT_DMA_CONTEXT_OUT** - Authentication tag (and IV) DMA done interrupt
- **AES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **AES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Note:

Only the DMA done interrupts can be cleared. The remaining interrupts should be disabled with [AESIntDisable\(\)](#).

Returns:

None.

5.2.2.13 AESIntDisable

Disables AES module interrupts.

Prototype:

```
void
AESIntDisable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the AES module.

ui32IntFlags is a bit mask of the interrupt sources to disable.

Description:

This function disables the interrupt sources in the AES module. The ***ui32IntFlags*** parameter is the logical OR of any of the following:

- **AES_INT_CONTEXT_IN** - Context interrupt
- **AES_INT_CONTEXT_OUT** - Authentication tag (and IV) interrupt
- **AES_INT_DATA_IN** - Data input interrupt
- **AES_INT_DATA_OUT** - Data output interrupt
- **AES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **AES_INT_DMA_CONTEXT_OUT** - Authentication tag (and IV) DMA done interrupt
- **AES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **AES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Note:

The DMA done interrupts are the only interrupts that can be cleared. The remaining interrupts can be disabled instead using [AESIntDisable\(\)](#).

Returns:

None.

5.2.2.14 AESIntEnable

Enables AES module interrupts.

Prototype:

```
void  
AESIntEnable(uint32_t ui32Base,  
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the AES module.

ui32IntFlags is a bit mask of the interrupt sources to enable.

Description:

This function enables the interrupts in the AES module. The ***ui32IntFlags*** parameter is the logical OR of any of the following:

- **AES_INT_CONTEXT_IN** - Context interrupt
- **AES_INT_CONTEXT_OUT** - Authentication tag (and IV) interrupt
- **AES_INT_DATA_IN** - Data input interrupt
- **AES_INT_DATA_OUT** - Data output interrupt
- **AES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **AES_INT_DMA_CONTEXT_OUT** - Authentication tag (and IV) DMA done interrupt
- **AES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **AES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Note:

Interrupts that have been previously been enabled are not disabled when this function is called.

Returns:

None.

5.2.2.15 AESIntRegister

Registers an interrupt handler for the AES module.

Prototype:

```
void
AESIntRegister(uint32_t ui32Base,
               void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the AES module.

pfnHandler is a pointer to the function to be called when the enabled AES interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables AES interrupts on the interrupt controller; specific AES interrupt sources must be enabled using [AESIntEnable\(\)](#). The interrupt handler being registered must clear the source of the interrupt using [AESIntClear\(\)](#).

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, [IntEnable\(\)](#) is used to enable AES interrupts on the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

5.2.2.16 AESIntStatus

Returns the current AES module interrupt status.

Prototype:

```
uint32_t
AESIntStatus(uint32_t ui32Base,
             bool bMasked)
```

Parameters:

ui32Base is the base address of the AES module.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Returns:

Returns a bit mask of the interrupt sources, which is a logical OR of any of the following:

- **AES_INT_CONTEXT_IN** - Context interrupt
- **AES_INT_CONTEXT_OUT** - Authentication tag (and IV) interrupt.
- **AES_INT_DATA_IN** - Data input interrupt
- **AES_INT_DATA_OUT** - Data output interrupt
- **AES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **AES_INT_DMA_CONTEXT_OUT** - Authentication tag (and IV) DMA done interrupt
- **AES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **AES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

5.2.2.17 void AESIntUnregister (uint32_t *ui32Base*)

Unregisters an interrupt handler for the AES module.

Parameters:

ui32Base is the base address of the AES module.

Description:

This function unregisters the previously registered interrupt handler and disables the interrupt in the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

5.2.2.18 AESIVRead

Saves the Initial Vector (IV) registers to a user-defined location.

Prototype:

```
void  
AESIVRead(uint32_t ui32Base,  
          uint32_t *pui32IVData)
```

Parameters:

ui32Base is the base address of the AES module.

pui32IVData is pointer to the location that stores the IV data.

Description:

This function stores the IV for use with authenticated encryption and decryption operations. It is assumed that the AES_CTRL_SAVE_CONTEXT bit is set in the AES_CTRL register.

Returns:

None.

5.2.2.19 AESIVSet

Writes the Initial Vector (IV) register, needed in some of the AES Modes.

Prototype:

```
void
AESIVSet(uint32_t ui32Base,
          uint32_t *pui32IVdata)
```

Parameters:

ui32Base is the base address of the AES module.

pui32IVdata is an array of 4 words (128 bits), containing the IV value to be configured. The least significant word is in the 0th index.

Description:

This function writes the initial vector registers in the AES module.

Returns:

None.

5.2.2.20 AESKey1Set

Writes the key 1 configuration registers, which are used for encryption or decryption.

Prototype:

```
void
AESKey1Set(uint32_t ui32Base,
           uint32_t *pui32Key,
           uint32_t ui32Keysize)
```

Parameters:

ui32Base is the base address for the AES module.

pui32Key is an array of 32-bit words, containing the key to be configured. The least significant word in the 0th index.

ui32Keysize is the size of the key, which must be one of the following values:
AES_CFG_KEY_SIZE_128, **AES_CFG_KEY_SIZE_192**, or **AES_CFG_KEY_SIZE_256**.

Description:

This function writes key 1 configuration registers based on the key size. This function is used in all modes.

Returns:

None.

5.2.2.21 AESKey2Set

Writes the key 2 configuration registers, which are used for encryption or decryption.

Prototype:

```
void
AESKey2Set(uint32_t ui32Base,
```

```
    uint32_t *pui32Key,  
    uint32_t ui32Keysize)
```

Parameters:

ui32Base is the base address for the AES module.

pui32Key is an array of 32-bit words, containing the key to be configured. The least significant word in the 0th index.

ui32Keysize is the size of the key, which must be one of the following values:
AES_CFG_KEY_SIZE_128, **AES_CFG_KEY_SIZE_192**, or **AES_CFG_KEY_SIZE_256**.

Description:

This function writes the key 2 configuration registers based on the key size. This function is used in the F8, F9, XTS, CCM, and CBC-MAC modes.

Returns:

None.

5.2.2.22 AESKey3Set

Writes key 3 configuration registers, which are used for encryption or decryption.

Prototype:

```
void  
AESKey3Set (uint32_t ui32Base,  
            uint32_t *pui32Key)
```

Parameters:

ui32Base is the base address for the AES module.

pui32Key is a pointer to an array of 4 words (128 bits), containing the key to be configured. The least significant word is in the 0th index.

Description:

This function writes the key 2 configuration registers with key 3 data used in CBC-MAC and F8 modes. This key is always 128 bits.

Returns:

None.

5.2.2.23 AESLengthSet

Used to set the write crypto data length in the AES module.

Prototype:

```
void  
AESLengthSet (uint32_t ui32Base,  
              uint64_t ui64Length)
```

Parameters:

ui32Base is the base address of the AES module.

ui64Length is the crypto data length in bytes.

Description:

This function stores the cryptographic data length in blocks for all modes. Data lengths up to $(2^{61} - 1)$ bytes are allowed. For GCM, any value up to $(2^{36} - 2)$ bytes are allowed because a 32-bit block counter is used. For basic modes (ECB/CBC/CTR/ICM/CFB128), zero can be programmed into the length field, indicating that the length is infinite.

When this function is called, the engine is triggered to start using this context.

Note:

This length does not include the authentication-only data used in some modes. Use the [AE-SAuthLengthSet\(\)](#) function to specify the authentication data length.

Returns:

None

5.2.2.24 AESReset

Resets the AES module.

Prototype:

```
void
AESReset(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the AES module.

Description:

This function performs a softreset the AES module.

Returns:

None.

5.2.2.25 AESEtagRead

Saves the tag registers to a user-defined location.

Prototype:

```
void
AESEtagRead(uint32_t ui32Base,
            uint32_t *pui32TagData)
```

Parameters:

ui32Base is the base address of the AES module.

pui32TagData is pointer to the location that stores the tag data.

Description:

This function stores the tag data for use authenticated encryption and decryption operations. It is assumed that the AES_CTRL_SAVE_CONTEXT bit is set in the AES_CTRL register.

Returns:

None.

5.3 Programming Example

The following example sets up the AES module to perform an encryption operation on four blocks of data in CBC mode with an 128-bit key. This example corresponds to vector F.2.1 in NIST document SP 800-38A.

```

// Random data for encryption/decryption.
//
uint32_t g_ui32AESPlainText[16] =
{
    0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373,
    0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45,
    0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520ala,
    0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6
};

// Encryption key
//
uint32_t g_ui32AES128Key[4] =
{
    0x16157e2b, 0xa6d2ae28, 0x8815f7ab, 0x3c4fcf09
};

// Initial value for CBC mode.
//
uint32_t g_ui32AESIV[4] =
{
    0x03020100, 0x07060504, 0xb0a0908, 0xf0e0d0c
};

int
main(void)
{
    uint32_t pui32CipherText[16];

    //
    // Enable the CCM module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_EC0);

    //
    // Wait for the CCM module to be ready.
    //
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_EC0))
    {
    }

    //
    // Reset the AES module before use.
    //
    AESReset(AES_BASE);

    //
    // Configure the AES module.
    //
    AESConfigSet(AES_BASE,
                 AES_CFG_DIR_ENCRYPT |
                 AES_CFG_MODE_CBC |
                 AES_CFG_KEY_SIZE_128BIT);

    //

```

```
// Set the initial value.  
//  
AESIVSet(AES_BASE, g_ui32AESIV);  
  
//  
// Set the encryption key.  
//  
AESKey1Set(AES_BASE, g_ui32AES128Key);  
  
//  
// Encrypt the data.  
//  
// The ciphertext should be:  
// {0xacab4976, 0x46b21981, 0x9b8ee9ce, 0x7d19e912,  
// 0x9bcb8650, 0xee197250, 0x3a11db95, 0xb2787691,  
// 0xb8d6be73, 0x3b74c1e3, 0x9ee61671, 0x16952222,  
// 0xa1caf13f, 0x09ac1f68, 0x30ca0e12, 0xa7e18675}  
//  
AESDataProcess(AES_BASE, g_ui32AESPlainText, pui32CipherText, 64);  
}
```


6 Controller Area Network (CAN)

Introduction	65
API Functions	65
CAN Message Objects	88
Programming Example	89

6.1 Introduction

The Controller Area Network (CAN) APIs provide a set of functions for accessing the Tiva CAN modules. Functions are provided to configure the CAN controllers, configure message objects, and manage CAN interrupts.

The Tiva CAN module provides hardware processing of the CAN data link layer. It can be configured with message filters and preloaded message data so that it can autonomously send and receive messages on the bus and notify the application accordingly. It automatically handles generation and checking of CRCs, error processing, and retransmission of CAN messages.

The message objects are stored in the CAN controller and provide the main interface for the CAN module on the CAN bus. There are 32 message objects that can each be programmed to handle a separate message ID, or can be chained together for a sequence of frames with the same ID. The message identifier filters provide masking that can be programmed to match any or all of the message ID bits, and frame types.

This driver is contained in `driverlib/can.c`, with `driverlib/can.h` containing the API declarations for use by applications.

6.2 API Functions

Data Structures

- [tCANBitClkParms](#)
- [tCANMsgObject](#)

Defines

- [CAN_INT_ERROR](#)
- [CAN_INT_MASTER](#)
- [CAN_INT_STATUS](#)
- [CAN_STATUS_BUS_OFF](#)
- [CAN_STATUS_EPASS](#)
- [CAN_STATUS_EWARN](#)
- [CAN_STATUS_LEC_ACK](#)
- [CAN_STATUS_LEC_BIT0](#)
- [CAN_STATUS_LEC_BIT1](#)
- [CAN_STATUS_LEC_CRC](#)

- [CAN_STATUS_LEC_FORM](#)
- [CAN_STATUS_LEC_MASK](#)
- [CAN_STATUS_LEC_MSK](#)
- [CAN_STATUS_LEC_NONE](#)
- [CAN_STATUS_LEC_STUFF](#)
- [CAN_STATUS_RXOK](#)
- [CAN_STATUS_TXOK](#)
- [MSG_OBJ_DATA_LOST](#)
- [MSG_OBJ_EXTENDED_ID](#)
- [MSG_OBJ_FIFO](#)
- [MSG_OBJ_NEW_DATA](#)
- [MSG_OBJ_NO_FLAGS](#)
- [MSG_OBJ_REMOTE_FRAME](#)
- [MSG_OBJ_RX_INT_ENABLE](#)
- [MSG_OBJ_STATUS_MASK](#)
- [MSG_OBJ_TX_INT_ENABLE](#)
- [MSG_OBJ_USE_DIR_FILTER](#)
- [MSG_OBJ_USE_EXT_FILTER](#)
- [MSG_OBJ_USE_ID_FILTER](#)

Enumerations

- [tCANIntStsReg](#)
- [tCANStsReg](#)
- [tMsgObjType](#)

Functions

- [uint32_t CANBitRateSet \(uint32_t ui32Base, uint32_t ui32SourceClock, uint32_t ui32BitRate\)](#)
- [void CANBitTimingGet \(uint32_t ui32Base, \[tCANBitClkParms\]\(#\) *psClkParms\)](#)
- [void CANBitTimingSet \(uint32_t ui32Base, \[tCANBitClkParms\]\(#\) *psClkParms\)](#)
- [void CANDisable \(uint32_t ui32Base\)](#)
- [CANEnable \(uint32_t ui32Base\)](#)
- [bool CANErrCntrGet \(uint32_t ui32Base, uint32_t *pui32RxCount, uint32_t *pui32TxCount\)](#)
- [void CANInit \(uint32_t ui32Base\)](#)
- [void CANIntClear \(uint32_t ui32Base, uint32_t ui32IntClr\)](#)
- [void CANIntDisable \(uint32_t ui32Base, uint32_t ui32IntFlags\)](#)
- [void CANIntEnable \(uint32_t ui32Base, uint32_t ui32IntFlags\)](#)
- [void CANIntRegister \(uint32_t ui32Base, void \(*pfnHandler\)\(void\)\)](#)
- [uint32_t CANIntStatus \(uint32_t ui32Base, \[tCANIntStsReg\]\(#\) eIntStsReg\)](#)
- [void CANIntUnregister \(uint32_t ui32Base\)](#)
- [void CANMessageClear \(uint32_t ui32Base, uint32_t ui32ObjID\)](#)
- [void CANMessageGet \(uint32_t ui32Base, uint32_t ui32ObjID, \[tCANMsgObject\]\(#\) *psMsgObject, bool bClrPendingInt\)](#)

- void [CANMessageSet](#) (uint32_t ui32Base, uint32_t ui32ObjID, tCANMsgObject *psMsgObject, tMsgObjType eMsgType)
- bool [CANRetryGet](#) (uint32_t ui32Base)
- void [CANRetrySet](#) (uint32_t ui32Base, bool bAutoRetry)
- uint32_t [CANStatusGet](#) (uint32_t ui32Base, tCANStsReg eStatusReg)

6.2.1 Detailed Description

The CAN APIs provide all of the functions needed by the application to implement an interrupt-driven CAN stack. These functions may be used to control any of the available CAN ports on a Tiva microcontroller, and can be used with one port without causing conflicts with the other port.

The CAN module is disabled by default, so the [CANInit\(\)](#) function must be called before any other CAN functions are called. This call initializes the message objects to a safe state prior to enabling the controller on the CAN bus. Also, the bit timing values must be programmed prior to enabling the CAN controller. The [CANSetBitTiming\(\)](#) function should be called with the appropriate bit timing values for the CAN bus. Once these two functions have been called, a CAN controller can be enabled using [CANEnable\(\)](#) and later disabled using [CANDisable\(\)](#) if needed. Calling [CANDisable\(\)](#) does not reinitialize a CAN controller, so it can be used to temporarily remove a CAN controller from the bus.

The CAN controller is highly configurable and can be programmed to automatically transmit and receive CAN messages under certain conditions. Message objects allow the application to perform some actions automatically without interaction from the microcontroller. Some examples of these actions are the following:

- Send a data frame immediately
- Send a data frame when a matching remote frame is seen on the CAN bus
- Receive a specific data frame
- Receive data frames that match a certain identifier pattern

To configure message objects to perform any of these actions, the application must first set up one of the 32 message objects using [CANMessageSet\(\)](#). This function must be used to configure a message object to send data, or to configure a message object to receive data. Each message object can be configured to generate interrupts on transmission or reception of CAN messages.

When data is received from the CAN bus, the application can use the [CANMessageGet\(\)](#) function to read the received message. This function can also be used to read a message object that is already configured in order to populate a message structure prior to making changes to the configuration of a message object. Reading the message object using this function also clears any pending interrupt on the message object.

Once a message object has been configured using [CANMessageSet\(\)](#), the message object has been allocated and continues to perform its programmed function unless it is released by a call to [CANMessageClear\(\)](#). The application is not required to clear out a message object before setting it with a new configuration, because each time [CANMessageSet\(\)](#) is called, it overwrites any previously programmed configuration.

The 32 message objects are identical except for priority. The lowest numbered message objects have the highest priority. Priority affects operation in two ways. First, if multiple actions are ready at the same time, the one with the highest priority message object occurs first. And second, when

multiple message objects have interrupts pending, the highest priority is presented first when reading the interrupt status. It is up to the application to manage the 32 message objects as a resource, and determine the best method for allocating and releasing them.

The CAN controller can generate interrupts on several conditions:

- When any message object transmits a message
- When any message object receives a message
- On warning conditions such as an error counter reaching a limit or occurrence of various bus errors
- On controller error conditions such as entering the bus-off state

An interrupt handler must be installed in order to process CAN interrupts. If dynamic interrupt configuration is desired, the [CANIntRegister\(\)](#) can be used to register the interrupt handler. This function places the vector in a RAM-based vector table. However, if the application uses a pre-loaded vector table in flash, then the CAN controller handler should be entered in the appropriate slot in the vector table. In this case, [CANIntRegister\(\)](#) is not needed, but the interrupt must be enabled on the host processor master interrupt controller using the [IntEnable\(\)](#) function. The CAN module interrupts are enabled using the [CANIntEnable\(\)](#) function. They can be disabled by using the [CANIntDisable\(\)](#) function.

Once CAN interrupts are enabled, the handler is invoked whenever a CAN interrupt is triggered. The handler can determine which condition caused the interrupt by using the [CANIntStatus\(\)](#) function. Multiple conditions can be pending when an interrupt occurs, so the handler must be designed to process all pending interrupt conditions before exiting. Each interrupt condition must be cleared before exiting the handler. There are two ways to do this. The [CANIntClear\(\)](#) function clears a specific interrupt condition without further action required by the handler. However, the handler can also clear the condition by performing certain actions. If the interrupt is a status interrupt, the interrupt can be cleared by reading the status register with [CANStatusGet\(\)](#). If the interrupt is caused by one of the message objects, then it can be cleared by reading the message object using [CANMessageGet\(\)](#).

There are several status registers that can be used to help the application manage the controller. The status registers are read using the [CANStatusGet\(\)](#) function. There is a controller status register that provides general status information such as error or warning conditions. There are also several status registers that provide information about all of the message objects at once using a 32-bit bit map of the status, with one bit representing each message object. These status registers can be used to determine:

- Which message objects have unprocessed received data
- Which message objects have pending transmission requests
- Which message objects are allocated for use

Bus error conditions when using CAN require special handling by the application, especially in cases where the CAN controller has gone into a bus-off condition. The CAN specification requires that a controller that has seen its error counters go above 256 transmit errors removes itself from the bus and enters a bus-off state. This state is indicated when the [CANStatusGet\(\)](#) function returns the value [**CAN_STATUS_BUS_OFF**](#). There are other warning levels ([**CAN_STATUS_EWARN**](#) and [**CAN_STATUS_EPASS**](#)) that occur before a bus-off condition that indicate something is wrong on the CAN bus. After entering the bus-off condition, the CAN controller automatically disables itself just as if the application had called [CANDisable\(\)](#). To exit the bus-off condition, the application must call [CANEnable\(\)](#) and then wait for the [**CAN_STATUS_BUS_OFF**](#) condition to clear. If the bus-off condition does not clear, then there is likely some physical condition or bit timing issue causing the

controller to be unable to function properly. There is no way to shorten this sequence as this is the method for recovering from a bus-off condition specified in the CAN 2.0 specification.

Example:

```
if(CANstatusGet(CANO_BASE) & CAN_STATUS_BUS_OFF)
{
    //
    // Enable the controller again to allow it to start decrementing the
    // error counter allowing the bus off condition to clear.
    //
    CANEnable();

    //
    // Wait for the bus off condition to clear. This condition can be
    // polled elsewhere depending on the application. But no CAN messages
    // can be sent until this condition clears.
    //
    while(CANStatusGet(CANO_BASE) & CAN_STATUS_BUS_OFF)
    {
    }
}
```

6.2.2 Data Structure Documentation

6.2.2.1 tCANBitClkParms

Definition:

```
typedef struct
{
    uint32_t ui32SyncPropPhase1Seg;
    uint32_t ui32Phase2Seg;
    uint32_t ui32SJW;
    uint32_t ui32QuantumPrescaler;
}
tCANBitClkParms
```

Members:

ui32SyncPropPhase1Seg This value holds the sum of the Synchronization, Propagation, and Phase Buffer 1 segments, measured in time quanta. The valid values for this setting range from 2 to 16.

ui32Phase2Seg This value holds the Phase Buffer 2 segment in time quanta. The valid values for this setting range from 1 to 8.

ui32SJW This value holds the Resynchronization Jump Width in time quanta. The valid values for this setting range from 1 to 4.

ui32QuantumPrescaler This value holds the CAN_CLK divider used to determine time quanta. The valid values for this setting range from 1 to 1023.

Description:

This structure is used for encapsulating the values associated with setting up the bit timing for a CAN controller. The structure is used when calling the CANGetBitTiming and CANSetBitTiming functions.

6.2.2.2 tCANMsgObject

Definition:

```
typedef struct
{
    uint32_t ui32MsgID;
    uint32_t ui32MsgIDMask;
    uint32_t ui32Flags;
    uint32_t ui32MsgLen;
    uint8_t *pui8MsgData;
}
tCANMsgObject
```

Members:

ui32MsgID The CAN message identifier used for 11 or 29 bit identifiers.

ui32MsgIDMask The message identifier mask used when identifier filtering is enabled.

ui32Flags This value holds various status flags and settings specified by tCANObjFlags.

ui32MsgLen This value is the number of bytes of data in the message object.

pui8MsgData This is a pointer to the message object's data.

Description:

The structure used for encapsulating all the items associated with a CAN message object in the CAN controller.

6.2.3 Define Documentation

6.2.3.1 CAN_INT_ERROR

Definition:

```
#define CAN_INT_ERROR
```

Description:

This flag is used to allow a CAN controller to generate error interrupts.

6.2.3.2 CAN_INT_MASTER

Definition:

```
#define CAN_INT_MASTER
```

Description:

This flag is used to allow a CAN controller to generate any CAN interrupts. If this is not set, then no interrupts are generated by the CAN controller.

6.2.3.3 CAN_INT_STATUS

Definition:

```
#define CAN_INT_STATUS
```

Description:

This flag is used to allow a CAN controller to generate status interrupts.

6.2.3.4 CAN_STATUS_BUS_OFF

Definition:

```
#define CAN_STATUS_BUS_OFF
```

Description:

CAN controller has entered a Bus Off state.

6.2.3.5 CAN_STATUS_EPASS

Definition:

```
#define CAN_STATUS_EPASS
```

Description:

CAN controller error level has reached error passive level.

6.2.3.6 CAN_STATUS_EWARN

Definition:

```
#define CAN_STATUS_EWARN
```

Description:

CAN controller error level has reached warning level.

6.2.3.7 CAN_STATUS_LEC_ACK

Definition:

```
#define CAN_STATUS_LEC_ACK
```

Description:

An acknowledge error has occurred.

6.2.3.8 CAN_STATUS_LEC_BIT0

Definition:

```
#define CAN_STATUS_LEC_BIT0
```

Description:

The bus remained a bit level of 0 for longer than is allowed.

6.2.3.9 CAN_STATUS_LEC_BIT1

Definition:

```
#define CAN_STATUS_LEC_BIT1
```

Description:

The bus remained a bit level of 1 for longer than is allowed.

6.2.3.10 CAN_STATUS_LEC_CRC

Definition:

```
#define CAN_STATUS_LEC_CRC
```

Description:

A CRC error has occurred.

6.2.3.11 CAN_STATUS_LEC_FORM

Definition:

```
#define CAN_STATUS_LEC_FORM
```

Description:

A formatting error has occurred.

6.2.3.12 CAN_STATUS_LEC_MASK

Definition:

```
#define CAN_STATUS_LEC_MASK
```

Description:

This is the mask for the CAN Last Error Code (LEC).

6.2.3.13 CAN_STATUS_LEC_MSK

Definition:

```
#define CAN_STATUS_LEC_MSK
```

Description:

This is the mask for the last error code field.

6.2.3.14 CAN_STATUS_LEC_NONE

Definition:

```
#define CAN_STATUS_LEC_NONE
```

Description:

There was no error.

6.2.3.15 CAN_STATUS_LEC_STUFF

Definition:

```
#define CAN_STATUS_LEC_STUFF
```

Description:

A bit stuffing error has occurred.

6.2.3.16 CAN_STATUS_RXOK

Definition:

```
#define CAN_STATUS_RXOK
```

Description:

A message was received successfully since the last read of this status.

6.2.3.17 CAN_STATUS_TXOK

Definition:

```
#define CAN_STATUS_TXOK
```

Description:

A message was transmitted successfully since the last read of this status.

6.2.3.18 MSG_OBJ_DATA_LOST

Definition:

```
#define MSG_OBJ_DATA_LOST
```

Description:

This indicates that data was lost since this message object was last read.

6.2.3.19 MSG_OBJ_EXTENDED_ID

Definition:

```
#define MSG_OBJ_EXTENDED_ID
```

Description:

This indicates that a message object is using an extended identifier.

6.2.3.20 MSG_OBJ_FIFO

Definition:

```
#define MSG_OBJ_FIFO
```

Description:

This indicates that this message object is part of a FIFO structure and not the final message object in a FIFO.

6.2.3.21 MSG_OBJ_NEW_DATA

Definition:

```
#define MSG_OBJ_NEW_DATA
```

Description:

This indicates that new data was available in the message object.

6.2.3.22 MSG_OBJ_NO_FLAGS

Definition:

```
#define MSG_OBJ_NO_FLAGS
```

Description:

This indicates that a message object has no flags set.

6.2.3.23 MSG_OBJ_REMOTE_FRAME

Definition:

```
#define MSG_OBJ_REMOTE_FRAME
```

Description:

This indicates that a message object is a remote frame.

6.2.3.24 MSG_OBJ_RX_INT_ENABLE

Definition:

```
#define MSG_OBJ_RX_INT_ENABLE
```

Description:

This indicates that receive interrupts are enabled.

6.2.3.25 MSG_OBJ_STATUS_MASK

Definition:

```
#define MSG_OBJ_STATUS_MASK
```

Description:

This define is used with the flag values to allow checking only status flags and not configuration flags.

6.2.3.26 MSG_OBJ_TX_INT_ENABLE

Definition:

```
#define MSG_OBJ_TX_INT_ENABLE
```

Description:

This indicates that transmit interrupts are enabled.

6.2.3.27 MSG_OBJ_USE_DIR_FILTER

Definition:

```
#define MSG_OBJ_USE_DIR_FILTER
```

Description:

This indicates that a message object uses or is using filtering based on the direction of the transfer. If the direction filtering is used, then ID filtering must also be enabled.

6.2.3.28 MSG_OBJ_USE_EXT_FILTER

Definition:

```
#define MSG_OBJ_USE_EXT_FILTER
```

Description:

This indicates that a message object uses or is using message identifier filtering based on the extended identifier. If the extended identifier filtering is used, then ID filtering must also be enabled.

6.2.3.29 MSG_OBJ_USE_ID_FILTER

Definition:

```
#define MSG_OBJ_USE_ID_FILTER
```

Description:

This indicates that a message object is using filtering based on the object's message identifier.

6.2.4 Enumeration Documentation

6.2.4.1 tCANIntStsReg

Description:

This data type is used to identify the interrupt status register. This is used when calling the [CANIntStatus\(\)](#) function.

Enumerators:

CAN_INT_STS_CAUSE Read the CAN interrupt status information.

CAN_INT_STS_OBJECT Read a message object's interrupt status.

6.2.4.2 tCANStsReg

Description:

This data type is used to identify which of several status registers to read when calling the [CANStatusGet\(\)](#) function.

Enumerators:

CAN_STS_CONTROL Read the full CAN controller status.

CAN_STS_TXREQUEST Read the full 32-bit mask of message objects with a transmit request set.

CAN_STS_NEWDAT Read the full 32-bit mask of message objects with new data available.

CAN_STS_MSGVAL Read the full 32-bit mask of message objects that are enabled.

6.2.4.3 tMsgObjType

Description:

This definition is used to determine the type of message object that is set up via a call to the [CANMessageSet\(\)](#) API.

Enumerators:

MSG_OBJ_TYPE_TX Transmit message object.

MSG_OBJ_TYPE_TX_REMOTE Transmit remote request message object.

MSG_OBJ_TYPE_RX Receive message object.

MSG_OBJ_TYPE_RX_REMOTE Receive remote request message object.

MSG_OBJ_TYPE_RXTX_REMOTE Remote frame receive remote, with auto-transmit message object.

6.2.5 Function Documentation

6.2.5.1 CANBitRateSet

Sets the CAN bit timing values to a nominal setting based on a desired bit rate.

Prototype:

```
uint32_t  
CANBitRateSet(uint32_t ui32Base,  
              uint32_t ui32SourceClock,  
              uint32_t ui32BitRate)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32SourceClock is the system clock for the device in Hz.

ui32BitRate is the desired bit rate.

Description:

This function sets the CAN bit timing for the bit rate passed in the *ui32BitRate* parameter based on the *ui32SourceClock* parameter. Because the CAN clock is based off of the system clock, the calling function must pass in the source clock rate either by retrieving it from [SysCtlClockGet\(\)](#) or using a specific value in Hz. The CAN bit timing is calculated assuming a minimal amount of propagation delay, which works for most cases where the network length is short. If tighter timing requirements or longer network lengths are needed, then the [CANBitTimingSet\(\)](#) function is available for full customization of all of the CAN bit timing values. Because not all bit rates can be matched exactly, the bit rate is set to the value closest to the desired bit rate without being higher than the *ui32BitRate* value.

Note:

On some devices the source clock is fixed at 8MHz so the *ui32SourceClock* must be set to 8000000.

Returns:

This function returns the bit rate that the CAN controller was configured to use or it returns 0 to indicate that the bit rate was not changed because the requested bit rate was not valid.

6.2.5.2 CANBitTimingGet

Reads the current settings for the CAN controller bit timing.

Prototype:

```
void
CANBitTimingGet(uint32_t ui32Base,
                tCANBitClkParms *psClkParms)
```

Parameters:

ui32Base is the base address of the CAN controller.

psClkParms is a pointer to a structure to hold the timing parameters.

Description:

This function reads the current configuration of the CAN controller bit clock timing and stores the resulting information in the structure supplied by the caller. Refer to [CANBitTimingSet\(\)](#) for the meaning of the values that are returned in the structure pointed to by *psClkParms*.

Returns:

None.

6.2.5.3 CANBitTimingSet

Configures the CAN controller bit timing.

Prototype:

```
void
CANBitTimingSet(uint32_t ui32Base,
                 tCANBitClkParms *psClkParms)
```

Parameters:

ui32Base is the base address of the CAN controller.

psClkParms points to the structure with the clock parameters.

Description:

Configures the various timing parameters for the CAN bus bit timing: Propagation segment, Phase Buffer 1 segment, Phase Buffer 2 segment, and the Synchronization Jump Width. The values for Propagation and Phase Buffer 1 segments are derived from the combination *psClkParms->ui32SyncPropPhase1Seg* parameter. Phase Buffer 2 is determined from the *psClkParms->ui32Phase2Seg* parameter. These two parameters, along with *psClkParms->ui32SJW* are based in units of bit time quanta. The actual quantum time is determined by the *psClkParms->ui32QuantumPrescaler* value, which specifies the divisor for the CAN module clock.

The total bit time, in quanta, is the sum of the two Seg parameters, as follows:

$$\text{bit_time_q} = \text{ui32SyncPropPhase1Seg} + \text{ui32Phase2Seg} + 1$$

Note that the Sync_Seg is always one quantum in duration, and is added to derive the correct duration of Prop_Seg and Phase1_Seg.

The equation to determine the actual bit rate is as follows:

CAN Clock / ((ui32SyncPropPhase1Seg + ui32Phase2Seg + 1) * (ui32QuantumPrescaler))

Thus with *ui32SyncPropPhase1Seg* = 4, *ui32Phase2Seg* = 1, *ui32QuantumPrescaler* = 2 and an 8 MHz CAN clock, the bit rate is (8 MHz) / ((5 + 2 + 1) * 2) or 500 Kbit/sec.

Returns:

None.

6.2.5.4 CANDisable

Disables the CAN controller.

Prototype:

```
void  
CANDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the CAN controller to disable.

Description:

Disables the CAN controller for message processing. When disabled, the controller no longer automatically processes data on the CAN bus. The controller can be restarted by calling [CANEnable\(\)](#). The state of the CAN controller and the message objects in the controller are left as they were before this call was made.

Returns:

None.

6.2.5.5 CANEnable

Enables the CAN controller.

Prototype:

```
void  
CANEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the CAN controller to enable.

Description:

Enables the CAN controller for message processing. Once enabled, the controller automatically transmits any pending frames, and processes any received frames. The controller can be stopped by calling [CANDisable\(\)](#). Prior to calling [CANEnable\(\)](#), [CANInit\(\)](#) must have been called to initialize the controller and the CAN bus clock must be configured by calling [CANBitTimingSet\(\)](#).

Returns:

None.

6.2.5.6 CANErrCntrGet

Reads the CAN controller error counter register.

Prototype:

```
bool
CANErrCntrGet(uint32_t ui32Base,
               uint32_t *pui32RxCount,
               uint32_t *pui32TxCount)
```

Parameters:

ui32Base is the base address of the CAN controller.

pui32RxCount is a pointer to storage for the receive error counter.

pui32TxCount is a pointer to storage for the transmit error counter.

Description:

This function reads the error counter register and returns the transmit and receive error counts to the caller along with a flag indicating if the controller receive counter has reached the error passive limit. The values of the receive and transmit error counters are returned through the pointers provided as parameters.

After this call, ***pui32RxCount** holds the current receive error count and ***pui32TxCount** holds the current transmit error count.

Returns:

Returns **true** if the receive error count has reached the error passive limit, and **false** if the error count is below the error passive limit.

6.2.5.7 CANInit

Initializes the CAN controller after reset.

Prototype:

```
void
CANInit(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the CAN controller.

Description:

After reset, the CAN controller is left in the disabled state. However, the memory used for message objects contains undefined values and must be cleared prior to enabling the CAN controller the first time. This prevents unwanted transmission or reception of data before the message objects are configured. This function must be called before enabling the controller the first time.

Returns:

None.

6.2.5.8 CANIntClear

Clears a CAN interrupt source.

Prototype:

```
void  
CANIntClear(uint32_t ui32Base,  
            uint32_t ui32IntClr)
```

Parameters:

ui32Base is the base address of the CAN controller.
ui32IntClr is a value indicating which interrupt source to clear.

Description:

This function can be used to clear a specific interrupt source. The *ui32IntClr* parameter must be one of the following values:

- **CAN_INT_INTID_STATUS** - Clears a status interrupt.
- 1-32 - Clears the specified message object interrupt

It is not necessary to use this function to clear an interrupt. This function is only used if the application wants to clear an interrupt source without taking the normal interrupt action.

Normally, the status interrupt is cleared by reading the controller status using [CANStatusGet\(\)](#). A specific message object interrupt is normally cleared by reading the message object using [CANMessageGet\(\)](#).

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

6.2.5.9 CANIntDisable

Disables individual CAN controller interrupt sources.

Prototype:

```
void  
CANIntDisable(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.
ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the specified CAN controller interrupt sources. Only enabled interrupt sources can cause a processor interrupt.

The *ui32IntFlags* parameter has the same definition as in the [CANIntEnable\(\)](#) function.

Returns:

None.

6.2.5.10 CANIntEnable

Enables individual CAN controller interrupt sources.

Prototype:

```
void
CANIntEnable(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables specific interrupt sources of the CAN controller. Only enabled sources cause a processor interrupt.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **CAN_INT_ERROR** - a controller error condition has occurred
- **CAN_INT_STATUS** - a message transfer has completed, or a bus error has been detected
- **CAN_INT_MASTER** - allow CAN controller to generate interrupts

In order to generate any interrupts, **CAN_INT_MASTER** must be enabled. Further, for any particular transaction from a message object to generate an interrupt, that message object must have interrupts enabled (see [CANMessageSet\(\)](#)). **CAN_INT_ERROR** generates an interrupt if the controller enters the “bus off” condition, or if the error counters reach a limit. **CAN_INT_STATUS** generates an interrupt under quite a few status conditions and may provide more interrupts than the application needs to handle. When an interrupt occurs, use [CANIntStatus\(\)](#) to determine the cause.

Returns:

None.

6.2.5.11 CANIntRegister

Registers an interrupt handler for the CAN controller.

Prototype:

```
void
CANIntRegister(uint32_t ui32Base,
                void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the CAN controller.

pfnHandler is a pointer to the function to be called when the enabled CAN interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables CAN interrupts on the interrupt controller; specific CAN interrupt sources must be enabled using [CANIntEnable\(\)](#). The interrupt handler being registered must clear the source of the interrupt using [CANIntClear\(\)](#).

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, [IntEnable\(\)](#) is used to enable CAN interrupts on the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

6.2.5.12 CANIntStatus

Returns the current CAN controller interrupt status.

Prototype:

```
uint32_t  
CANIntStatus(uint32_t ui32Base,  
             tCANIntStsReg eIntStsReg)
```

Parameters:

ui32Base is the base address of the CAN controller.

eIntStsReg indicates which interrupt status register to read

Description:

This function returns the value of one of two interrupt status registers. The interrupt status register read is determined by the **eIntStsReg** parameter, which can have one of the following values:

- **CAN_INT_STS_CAUSE** - indicates the cause of the interrupt
- **CAN_INT_STS_OBJECT** - indicates pending interrupts of all message objects

CAN_INT_STS_CAUSE returns the value of the controller interrupt register and indicates the cause of the interrupt. The value returned is **CAN_INT_INTID_STATUS** if the cause is a status interrupt. In this case, the status register is read with the [CANStatusGet\(\)](#) function. Calling this function to read the status also clears the status interrupt. If the value of the interrupt register is in the range 1-32, then this indicates the number of the highest priority message object that has an interrupt pending. The message object interrupt can be cleared by using the [CANIntClear\(\)](#) function, or by reading the message using [CANMessageGet\(\)](#) in the case of a received message. The interrupt handler can read the interrupt status again to make sure all pending interrupts are cleared before returning from the interrupt.

CAN_INT_STS_OBJECT returns a bit mask indicating which message objects have pending interrupts. This value can be used to discover all of the pending interrupts at once, as opposed to repeatedly reading the interrupt register by using **CAN_INT_STS_CAUSE**.

Returns:

Returns the value of one of the interrupt status registers.

6.2.5.13 CANIntUnregister

Unregisters an interrupt handler for the CAN controller.

Prototype:

```
void
CANIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function unregisters the previously registered interrupt handler and disables the interrupt in the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

6.2.5.14 CANMessageClear

Clears a message object so that it is no longer used.

Prototype:

```
void
CANMessageClear(uint32_t ui32Base,
                 uint32_t ui32ObjID)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32ObjID is the message object number to disable (1-32).

Description:

This function frees the specified message object from use. Once a message object has been “cleared,” it no longer automatically sends or receives messages, nor does it generate interrupts.

Returns:

None.

6.2.5.15 CANMessageGet

Reads a CAN message from one of the message object buffers.

Prototype:

```
void
CANMessageGet(uint32_t ui32Base,
              uint32_t ui32ObjID,
              tCANMsgObject *psMsgObject,
              bool bClrPendingInt)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32ObjID is the object number to read (1-32).

psMsgObject points to a structure containing message object fields.

bClrPendingInt indicates whether an associated interrupt should be cleared.

Description:

This function is used to read the contents of one of the 32 message objects in the CAN controller and return it to the caller. The data returned is stored in the fields of the caller-supplied structure pointed to by *psMsgObject*. The data consists of all of the parts of a CAN message, plus some control and status information.

Normally, this function is used to read a message object that has received and stored a CAN message with a certain identifier. However, this function could also be used to read the contents of a message object in order to load the fields of the structure in case only part of the structure must be changed from a previous setting.

When using [CANMessageGet\(\)](#), all of the same fields of the structure are populated in the same way as when the [CANMessageSet\(\)](#) function is used, with the following exceptions:

psMsgObject->ui32Flags:

- **MSG_OBJ_NEW_DATA** indicates if this data is new since the last time it was read
- **MSG_OBJ_DATA_LOST** indicates that at least one message was received on this message object and not read by the host before being overwritten.

Returns:

None.

6.2.5.16 CANMessageSet

Configures a message object in the CAN controller.

Prototype:

```
void  
CANMessageSet(uint32_t ui32Base,  
              uint32_t ui32ObjID,  
              tCANMsgObject *psMsgObject,  
              tMsgObjType eMsgType)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32ObjID is the object number to configure (1-32).

psMsgObject is a pointer to a structure containing message object settings.

eMsgType indicates the type of message for this object.

Description:

This function is used to configure any one of the 32 message objects in the CAN controller. A message object can be configured to be any type of CAN message object as well as to use automatic transmission and reception. This call also allows the message object to be configured to generate interrupts on completion of message receipt or transmission. The message object can also be configured with a filter/mask so that actions are only taken when a message that meets certain parameters is seen on the CAN bus.

The *eMsgType* parameter must be one of the following values:

- **MSG_OBJ_TYPE_TX** - CAN transmit message object.
- **MSG_OBJ_TYPE_TX_REMOTE** - CAN transmit remote request message object.
- **MSG_OBJ_TYPE_RX** - CAN receive message object.
- **MSG_OBJ_TYPE_RX_REMOTE** - CAN receive remote request message object.
- **MSG_OBJ_TYPE_RXTX_REMOTE** - CAN remote frame receive remote, then transmit message object.

The message object pointed to by *psMsgObject* must be populated by the caller, as follows:

- *ui32MsgID* - contains the message ID, either 11 or 29 bits.
- *ui32MsgIDMask* - mask of bits from *ui32MsgID* that must match if identifier filtering is enabled.
- *ui32Flags*
 - Set **MSG_OBJ_TX_INT_ENABLE** flag to enable interrupt on transmission.
 - Set **MSG_OBJ_RX_INT_ENABLE** flag to enable interrupt on receipt.
 - Set **MSG_OBJ_USE_ID_FILTER** flag to enable filtering based on the identifier mask specified by *ui32MsgIDMask*.
- *ui32MsgLen* - the number of bytes in the message data. This parameter must be non-zero even for a remote frame; it must match the expected bytes of data in the responding data frame.
- *pui8MsgData* - points to a buffer containing up to 8 bytes of data for a data frame.

Example: To send a data frame or remote frame (in response to a remote request), take the following steps:

1. Set *eMsgType* to **MSG_OBJ_TYPE_TX**.
2. Set *psMsgObject->ui32MsgID* to the message ID.
3. Set *psMsgObject->ui32Flags*. Make sure to set **MSG_OBJ_TX_INT_ENABLE** to allow an interrupt to be generated when the message is sent.
4. Set *psMsgObject->ui32MsgLen* to the number of bytes in the data frame.
5. Set *psMsgObject->pui8MsgData* to point to an array containing the bytes to send in the message.
6. Call this function with *ui32ObjID* set to one of the 32 object buffers.

Example: To receive a specific data frame, take the following steps:

1. Set *eMsgObjType* to **MSG_OBJ_TYPE_RX**.
2. Set *psMsgObject->ui32MsgID* to the full message ID, or a partial mask to use partial ID matching.
3. Set *psMsgObject->ui32MsgIDMask* bits that are used for masking during comparison.
4. Set *psMsgObject->ui32Flags* as follows:
 - Set **MSG_OBJ_RX_INT_ENABLE** flag to be interrupted when the data frame is received.
 - Set **MSG_OBJ_USE_ID_FILTER** flag to enable identifier-based filtering.
5. Set *psMsgObject->ui32MsgLen* to the number of bytes in the expected data frame.
6. The buffer pointed to by *psMsgObject->pui8MsgData* is not used by this call as no data is present at the time of the call.
7. Call this function with *ui32ObjID* set to one of the 32 object buffers.

If you specify a message object buffer that already contains a message definition, it is overwritten.

Returns:

None.

6.2.5.17 CANRetryGet

Returns the current setting for automatic retransmission.

Prototype:

```
bool  
CANRetryGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the CAN controller.

Description:

This function reads the current setting for automatic retransmission in the CAN controller and returns it to the caller.

Returns:

Returns **true** if automatic retransmission is enabled, **false** otherwise.

6.2.5.18 CANRetrySet

Sets the CAN controller automatic retransmission behavior.

Prototype:

```
void  
CANRetrySet (uint32_t ui32Base,  
             bool bAutoRetry)
```

Parameters:

ui32Base is the base address of the CAN controller.

bAutoRetry enables automatic retransmission.

Description:

This function enables or disables automatic retransmission of messages with detected errors. If **bAutoRetry** is **true**, then automatic retransmission is enabled, otherwise it is disabled.

Returns:

None.

6.2.5.19 CANStatusGet

Reads one of the controller status registers.

Prototype:

```
uint32_t  
CANStatusGet (uint32_t ui32Base,  
              tCANstsReg eStatusReg)
```

Parameters:

ui32Base is the base address of the CAN controller.

eStatusReg is the status register to read.

Description:

This function reads a status register of the CAN controller and returns it to the caller. The different status registers are:

- **CAN_STS_CONTROL** - the main controller status
- **CAN_STS_TXREQUEST** - bit mask of objects pending transmission
- **CAN_STS_NEWDAT** - bit mask of objects with new data
- **CAN_STS_MSGVAL** - bit mask of objects with valid configuration

When reading the main controller status register, a pending status interrupt is cleared. This parameter is used in the interrupt handler for the CAN controller if the cause is a status interrupt. The controller status register fields are as follows:

- **CAN_STATUS_BUS_OFF** - controller is in bus-off condition
- **CAN_STATUS_EWARN** - an error counter has reached a limit of at least 96
- **CAN_STATUS_EPASS** - CAN controller is in the error passive state
- **CAN_STATUS_RXOK** - a message was received successfully (independent of any message filtering).
- **CAN_STATUS_TXOK** - a message was successfully transmitted
- **CAN_STATUS_LEC_MSK** - mask of last error code bits (3 bits)
- **CAN_STATUS_LEC_NONE** - no error
- **CAN_STATUS_LEC_STUFF** - stuffing error detected
- **CAN_STATUS_LEC_FORM** - a format error occurred in the fixed format part of a message
- **CAN_STATUS_LEC_ACK** - a transmitted message was not acknowledged
- **CAN_STATUS_LEC_BIT1** - dominant level detected when trying to send in recessive mode
- **CAN_STATUS_LEC_BIT0** - recessive level detected when trying to send in dominant mode
- **CAN_STATUS_LEC_CRC** - CRC error in received message

The remaining status registers consist of 32-bit-wide bit maps to the message objects. They can be used to quickly obtain information about the status of all the message objects without needing to query each one. They contain the following information:

- **CAN_STS_TXREQUEST** - if a message object's TXRQST bit is set, a transmission is pending on that object. The application can use this information to determine which objects are still waiting to send a message.
- **CAN_STS_NEWDAT** - if a message object's NEWDAT bit is set, a new message has been received in that object, and has not yet been picked up by the host application
- **CAN_STS_MSGVAL** - if a message object's MSGVAL bit is set, the object has a valid configuration programmed. The host application can use this information to determine which message objects are empty/unused.

Returns:

Returns the value of the status register.

6.3 CAN Message Objects

This section explains how to configure the CAN message objects in various modes using the [CANMessageSet\(\)](#) and [CANMessageGet\(\)](#) APIs. The configuration of a message object is determined by two parameters that are passed into the [CANMessageSet\(\)](#) API. These are the [tCANMsgObject](#) structure and the [tMsgObjType](#) type field. It is important to note that the [ulObjID](#) parameter is the index of one of the 32 message objects that are available and is not the message object's identifier.

Message objects can be defined as one of five types based on the needs of the application. They are defined in the [tMsgObjType](#) enumeration and can only be one of those values. The simplest of the message object types are [MSG_OBJ_TYPE_TX](#) and [MSG_OBJ_TYPE_RX](#) which are used to send or receive messages for a given message identifier or a range of identifiers. The message type [MSG_OBJ_TYPE_TX_REMOTE](#) is used to transmit a remote request for data from another CAN node on the network. These message objects do not transmit any data but once they send the request, they automatically turn into receive message object and wait for data from a remote CAN device. The message type [MSG_OBJ_TYPE_RX_REMOTE](#) is the receiving end of a remote request, and receives remote requests for data and generates an interrupt to let the application know when to supply and transmit data back to the CAN controller that issued the remote request for data. The message type [MSG_OBJ_TYPE_RX_TX_REMOTE](#) is similar to the [MSG_OBJ_TYPE_RX_REMOTE](#) except that it automatically responds with data that the application placed in the message object.

The remaining information used to configure a CAN message object is contained in the [tCANMsgObject](#) structure which is used when calling [CANMessageSet\(\)](#) or is filled by data read from the message object when calling [CANMessageGet\(\)](#). The CAN message identifier is simply stored into the [ulMsgID](#) member of the [tCANMsgObject](#) structure and is the 11- or 20-bit CAN identifier for this message object. The [ulMsgIDMask](#) is the mask that is used in combination with the [ulMsgID](#) value to determine a match when the [MSG_OBJ_USE_ID_FILTER](#) flag is set for a message object. The [ulMsgIDMask](#) is ignored if [MSG_OBJ_USE_ID_FILTER](#) flag is not set. The last of the configuration parameters are specified in the [ulFlags](#) which are defined as a combination of the [MSG_OBJ_*](#) values. The [MSG_OBJ_TX_INT_ENABLE](#) and [MSG_OBJ_RX_INT_ENABLE](#) flags enable transmit complete or receive data interrupts. If the CAN network is only using extended (20-bit) identifiers, then the [MSG_OBJ_EXTENDED_ID](#) flag should be specified. The [CANMessageSet\(\)](#) function forces this flag to be set if the length of the identifier is greater than an 11-bit identifier can hold. The [MSG_OBJ_USE_ID_FILTER](#) is used to enable filtering based on the message identifiers as message are seen by the CAN controller. The combination of [ulMsgID](#) and [ulMsgIDMask](#) determines if a message is accepted for a given message object. In some cases it may be necessary to add a filter based on the direction of the message, so in these cases, the [MSG_OBJ_USE_DIR_FILTER](#) is used to only accept the direction specified in the message type. Another additional filter flag is [MSG_OBJ_USE_EXT_FILTER](#) which filters on only extended identifiers. In a mixed 11-bit and 20-bit identifier system, this parameter prevents an 11-bit identifier from being confused with a 20-bit identifier of the same value. It is not necessary to specify this parameter if there are only extended identifiers being used in the system. To determine if the incoming message identifier matches a given message object, the incoming message identifier is ANDed with [ulMsgIDMask](#) and compared with [ulMsgID](#). The "C" logic would be the following:

```
if((IncomingID & ulMsgIDMask) == ulMsgID)
{
    // Accept the message.
}
else
{
    // Ignore the message.
}
```

The last of the flags to affect [CANMessageSet\(\)](#) is the MSG_OBJ_FIFO flag. This flag is used when combining multiple message objects in a FIFO. This flag is useful when an application must receive more than the 8 bytes of data that can be received by a single CAN message object. It can also be used to reduce the likelihood of causing an overrun of data on a single message object that may be receiving data faster than the application can handle when using a single message object. If multiple message objects are going to be used in a FIFO, they must be read in sequential order based on the message object number and have the exact same message identifiers and filtering values. All but the last of the message objects in a FIFO should have the MSG_OBJ_FIFO flag set and the last message object in the FIFO should not have the MSG_OBJ_FIFO flag set, indicating that it is the last entry in the FIFO. See the CAN FIFO configuration example in the Programming Examples section of this document.

The remaining flags are all used when calling [CANMessageGet\(\)](#) when reading data or checking the status of a message object. If the MSG_OBJ_NEW_DATA flag is set in the [tCANMsgObject](#) ulFlags variable then the data returned was new and not stale data from a previous call to [CANMessageGet\(\)](#). If the MSG_OBJ_DATA_LOST flag is set, then data was lost since this message object was last read with [CANMessageGet\(\)](#). The MSG_OBJ_REMOTE_FRAME flag is set if the message object was configured as a remote message object and a remote request was received.

When sending or receiving data, the last two variables define the size and a pointer to the data used by [CANMessageGet\(\)](#) and [CANMessageSet\(\)](#). The ulMsgLen variable in [tCANMsgObject](#) specifies the number of bytes to send when calling [CANMessageSet\(\)](#) and the number of bytes to read when calling [CANMessageGet\(\)](#). The pucMsgData variable in [tCANMsgObject](#) is the pointer to the data to send ulMsgLen bytes, or the pointer to the buffer to read ulMsgLen bytes into.

6.4 Programming Examples

This example code sends out data from CAN controller 0 to be received by CAN controller 1. In order to actually receive the data, an external cable must be connected between the two ports. In this example, both controllers are configured for 1 Mbit operation.

```
tCANBitClkParms CANBitClk;
tCANMsgObject sMsgObjectRx;
tCANMsgObject sMsgObjectTx;
uint8_t pui8BufferIn[8];
uint8_t pui8BufferOut[8];

//
// Reset the state of all the message objects and the state of the CAN
// module to a known state.
//
CANInit(CAN0_BASE);
CANInit(CAN1_BASE);

//
// Configure the controller for 1 Mbit operation.
//
CANSetBitTiming(CAN1_BASE, &CANBitClk);

//
// Take the CAN0 device out of INIT state.
//
CANEnable(CAN0_BASE);
CANEnable(CAN1_BASE);

//
```

```
// Configure a receive object.  
//  
sMsgObjectRx.ulMsgID = (0x400);  
sMsgObjectRx.ulMsgIDMask = 0x7f8;  
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER | MSG_OBJ_FIFO;  
  
//  
// The first three message objects have the MSG_OBJ_FIFO set to indicate  
// that they are part of a FIFO.  
//  
CANMessageSet(CAN0_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
CANMessageSet(CAN0_BASE, 2, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
CANMessageSet(CAN0_BASE, 3, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
  
//  
// Last message object does not have the MSG_OBJ_FIFO set to indicate that  
// this is the last message.  
//  
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;  
CANMessageSet(CAN0_BASE, 4, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
  
//  
// Configure and start transmit of message object.  
//  
sMsgObjectTx.ulMsgID = 0x400;  
sMsgObjectTx.ulFlags = 0;  
sMsgObjectTx.ulMsgLen = 8;  
sMsgObjectTx.pucMsgData = pui8BufferOut;  
CANMessageSet(CAN0_BASE, 2, &sMsgObjectTx, MSG_OBJ_TYPE_TX);  
  
//  
// Wait for new data to become available.  
//  
while((CANStatusGet(CAN1_BASE, CAN_STS_NEWDAT) & 1) == 0)  
{  
    //  
    // Read the message out of the message object.  
    //  
    CANMessageGet(CAN1_BASE, 1, &sMsgObjectRx, true);  
}  
  
//  
// Process new data in sMsgObjectRx.pucMsgData.  
//  
...
```

This example code configures a set of CAN message objects in FIFO mode using CAN controller 0.

```
tCANBitClkParms CANBitClk;  
tCANMsgObject sMsgObjectRx;  
uint8_t pui8BufferIn[8];  
uint8_t pui8BufferOut[8];  
  
//  
// Reset the state of all the message objects and the state of the CAN  
// module to a known state.  
//  
CANInit(CAN0_BASE);  
  
//  
// Configure the controller for 1 Mbit operation.  
//  
CANBitRateSet(CAN0_BASE, 8000000, 1000000);
```

```
//  
// Take the CAN0 device out of INIT state.  
//  
CANEnable(CAN0_BASE);  
  
//  
// Configure a receive object as a CAN FIFO to receive message objects with  
// message ID 0x400-0x407.  
//  
sMsgObjectRx.ulMsgID = (0x400);  
sMsgObjectRx.ulMsgIDMask = 0x7f8;  
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER | MSG_OBJ_FIFO;  
  
//  
// The first three message objects have the MSG_OBJ_FIFO set to indicate  
// that they are part of a FIFO.  
//  
CANMessageSet(CAN0_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
CANMessageSet(CAN0_BASE, 2, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
CANMessageSet(CAN0_BASE, 3, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
  
//  
// Last message object does not have the MSG_OBJ_FIFO set to indicate that  
// this is the last message.  
//  
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;  
CANMessageSet(CAN0_BASE, 4, &sMsgObjectRx, MSG_OBJ_TYPE_RX);  
  
...
```


7 CRC

Introduction	93
API Functions	93
Programming Example	96

7.1 Introduction

The CRC module driver provides a method for generating CRC checksums of various types. The configuration and feature highlights are:

- Seed value for CRC operations is either all zeroes, all ones or a user defined value.
- Accepts data as bytes or 4-byte words.
- Optionally performs pre- and post-processing on the input data and checksum.

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

7.2 API Functions

Functions

- `void CRCCConfigSet (uint32_t ui32Base, uint32_t ui32CRCCConfig)`
- `uint32_t CRCDataProcess (uint32_t ui32Base, uint32_t *pui32DataIn, uint32_t ui32DataLength, bool bPPResult)`
- `void CRCDataWrite (uint32_t ui32Base, uint32_t ui32Data)`
- `uint32_t CRCResultRead (uint32_t ui32Base, bool bPPResult)`
- `void CRCSeedSet (uint32_t ui32Base, uint32_t ui32Seed)`

7.2.1 Detailed Description

The CRC API consists of functions for configuring the CRC module, processing data, and reading the resultant checksum.

7.2.2 Function Documentation

7.2.2.1 CRCCConfigSet

Set the configuration of CRC functionality with the EC module.

Prototype:

```
void  
CRCCConfigSet(uint32_t ui32Base,  
              uint32_t ui32CRCCConfig)
```

Parameters:

ui32Base is the base address of the EC module.

ui32CRCCConfig is the configuration of the CRC engine.

Description:

This function configures the operation of the CRC engine within the EC module. The configuration is specified with the *ui32CRCCConfig* argument. It is the logical OR of any of the following options:

CRC Initialization Value

- **CRC_CFG_INIT_SEED** - Initialize with seed value
- **CRC_CFG_INIT_0** - Initialize to all '0s'
- **CRC_CFG_INIT_1** - Initialize to all '1s'

Input Data Size

- **CRC_CFG_SIZE_8BIT** - Input data size of 8 bits
- **CRC_CFG_SIZE_32BIT** - Input data size of 32 bits

Post Process Reverse/Inverse

- **CRC_CFG_RESINV** - Result inverse enable
- **CRC_CFG_OBR** - Output reverse enable

Input Bit Reverse

- **CRC_CFG_IBR** - Bit reverse enable

Endian Control

- **CRC_CFG_ENDIAN_SBHW** - Swap byte in half-word
- **CRC_CFG_ENDIAN_SHW** - Swap half-word

Operation Type

- **CRC_CFG_TYPE_P8005** - Polynomial 0x8005
- **CRC_CFG_TYPE_P1021** - Polynomial 0x1021
- **CRC_CFG_TYPE_P4C11DB7** - Polynomial 0x4C11DB7
- **CRC_CFG_TYPE_P1EDC6F41** - Polynomial 0x1EDC6F41
- **CRC_CFG_TYPE_TCPCHKSUM** - TCP checksum

Returns:

None.

7.2.2.2 CRCDATAProcess

Process data to generate a CRC with the EC module.

Prototype:

```
uint32_t
CRCDATAProcess(uint32_t ui32Base,
                uint32_t *pui32DataIn,
                uint32_t ui32DataLength,
                bool bPPResult)
```

Parameters:

ui32Base is the base address of the EC module.

pui32DataIn is a pointer to an array of data that is processed.

ui32DataLength is the number of data items that are processed to produce the CRC.

bPPResult is **true** to read the post-processed result, or **false** to read the unmodified result.

Description:

This function processes an array of data to produce a CRC result.

The data in the array pointed to be *pui32DataIn* is either an array of bytes or an array of words depending on the selection of the input data size options **CRC_CFG_SIZE_8BIT** and **CRC_CFG_SIZE_32BIT**.

This function returns either the unmodified CRC result or the post-processed CRC result from the EC module. The post-processing options are selectable through **CRC_CFG_RESINV** and **CRC_CFG_OBR** parameters.

Returns:

The CRC result.

7.2.2.3 CRCDATAWrite

Write data into the EC module for CRC operations.

Prototype:

```
void
CRCDATAWrite(uint32_t ui32Base,
              uint32_t ui32Data)
```

Parameters:

ui32Base is the base address of the EC module.

ui32Data is the data to be written.

Description:

This function writes either 8 or 32 bits of data into the EC module for CRC operations. The distinction between 8 and 32 bits of data is made when the **CRC_CFG_SIZE_8BIT** or **CRC_CFG_SIZE_32BIT** flag is set using the [CRCCConfigSet\(\)](#) function.

When writing 8 bits of data, ensure the data is in the least significant byte position. The remaining bytes should be written with zero. For example, when writing 0xAB, *ui32Data* should be 0x000000AB.

Returns:

None

7.2.2.4 CRCResultRead

Reads the result of a CRC operation in the EC module.

Prototype:

```
uint32_t  
CRCResultRead(uint32_t ui32Base,  
              bool bPPResult)
```

Parameters:

ui32Base is the base address of the EC module.

bPPResult is **true** to read the post-processed result, or **false** to read the unmodified result.

Description:

This function reads either the unmodified CRC result or the post processed CRC result from the EC module. The post-processing options are selectable through **CRC_CFG_RESINV** and **CRC_CFG_OBR** parameters in the [CRCCConfigSet\(\)](#) function.

Returns:

The CRC result.

7.2.2.5 CRCSseedSet

Write the seed value for CRC operations in the EC module.

Prototype:

```
void  
CRCSseedSet(uint32_t ui32Base,  
            uint32_t ui32Seed)
```

Parameters:

ui32Base is the base address of the EC module.

ui32Seed is the seed value.

Description:

This function writes the seed value for use with CRC operations in the EC module. This value is the start value for CRC operations. If this value is not written, then the residual seed from the previous operation is used as the starting value.

Note:

The seed must be written only if **CRC_CFG_INIT_SEED** is set with the [CRCCConfigSet\(\)](#) function.

7.3 Programming Example

The following example sets up the CRC for basic CRC32 operation with a starting seed of zero.

```
uint32_t g_ui32Result;  
  
//
```

```
// Random data for generating CRC.  
//  
uint32_t g_ui32RandomData[16] =  
{  
    0x8a5f1b22, 0xcb935d29, 0xcc1ac092, 0x5dad8c9e,  
    0x6a83b39f, 0x8607dc60, 0xda0ba4d2, 0xf49b0fa2,  
    0xaf35d524, 0xffa8001d, 0xbcc931e8, 0x4a2c99ef,  
    0x7fa297ab, 0xab943bae, 0x07c61cc4, 0x47c8627d  
};  
  
int  
main(void)  
{  
    //  
    // Enable the CRC module.  
    //  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_EC0);  
  
    //  
    // Wait for the CRC module to be ready.  
    //  
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_EC0))  
    {  
    }  
  
    //  
    // Configure the CRC module.  
    //  
    CRCConfigSet(EC_BASE,  
        CRC_CFG_INIT_SEED |  
        CRC_CFG_TYPE_P4C11DB7 |  
        CRC_CFG_SIZE_32BIT);  
  
    //  
    // Set the seed value.  
    //  
    CRCSeedSet(EC_BASE, 0x5a5a5a5a);  
  
    //  
    // Process the data and get the result. The result should be  
    // 0x75fd6f5c.  
    //  
    g_ui32Result = CRCDATAProcess(EC_BASE, g_ui32RandomData, 16, false);  
}
```


8 DES

Introduction	99
API Functions	99
Programming Example	108

8.1 Introduction

The DES module driver provides a method for performing encryption and decryption operations on blocks of 64-bits of data. The configuration and feature highlights are:

- Supports ECB, CBC, and CFB operating modes.
- Supports DES and TDES(3EDE) operating modes.

This driver is contained in `driverlib/des.c`, with `driverlib/des.h` containing the API declarations for use by applications.

8.2 API Functions

Functions

- void `DESConfigSet` (uint32_t ui32Base, uint32_t ui32Config)
- bool `DESDataProcess` (uint32_t ui32Base, uint32_t *pui32Src, uint32_t *pui32Dest, uint32_t ui32Length)
- void `DESDataRead` (uint32_t ui32Base, uint32_t *pui32Dest)
- bool `DESDataReadNonBlocking` (uint32_t ui32Base, uint32_t *pui32Dest)
- void `DESDataWrite` (uint32_t ui32Base, uint32_t *pui32Src)
- bool `DESDataWriteNonBlocking` (uint32_t ui32Base, uint32_t *pui32Src)
- void `DESDMADisable` (uint32_t ui32Base, uint32_t ui32Flags)
- void `DESDMAEnable` (uint32_t ui32Base, uint32_t ui32Flags)
- void `DESIntClear` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `DESIntDisable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `DESIntEnable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `DESIntRegister` (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t `DESIntStatus` (uint32_t ui32Base, bool bMasked)
- void `DESIntUnregister` (uint32_t ui32Base)
- bool `DESIVSet` (uint32_t ui32Base, uint32_t *pui32IVdata)
- void `DESKeySet` (uint32_t ui32Base, uint32_t *pui32Key)
- void `DESLengthSet` (uint32_t ui32Base, uint32_t ui32Length)
- void `DESReset` (uint32_t ui32Base)

8.2.1 Detailed Description

The DES API consists of functions for configuring the DES module and processing data.

8.2.2 Function Documentation

8.2.2.1 DESConfigSet

Configures the DES module for operation.

Prototype:

```
void  
DESConfigSet(uint32_t ui32Base,  
             uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the DES module.

ui32Config is the configuration of the DES module.

Description:

This function configures the DES module for operation.

The *ui32Config* parameter is a bit-wise OR of a number of configuration flags. The valid flags are grouped below based on their function.

The direction of the operation is specified with one of the following two flags. Only one is permitted.

- **DES_CFG_DIR_ENCRYPT** - Encryption
- **DES_CFG_DIR_DECRYPT** - Decryption

The operational mode of the DES engine is specified with one of the following flags. Only one is permitted.

- **DES_CFG_MODE_ECB** - Electronic Codebook Mode
- **DES_CFG_MODE_CBC** - Cipher-Block Chaining Mode
- **DES_CFG_MODE_CFB** - Cipher Feedback Mode

The selection of single DES or triple DES is specified with one of the following two flags. Only one is permitted.

- **DES_CFG_SINGLE** - Single DES
- **DES_CFG_TRIPLE** - Triple DES

Returns:

None.

8.2.2.2 DESDataProcess

Processes blocks of data through the DES module.

Prototype:

```
bool  
DESDataProcess(uint32_t ui32Base,  
               uint32_t *pui32Src,  
               uint32_t *pui32Dest,  
               uint32_t ui32Length)
```

Parameters:

ui32Base is the base address of the DES module.

pui32Src is a pointer to an array of words that contains the source data for processing.

pui32Dest is a pointer to an array of words consisting of the processed data.

ui32Length is the length of the cryptographic data in bytes. It must be a multiple of eight.

Description:

This function takes the data contained in the pui32Src array and processes it using the DES engine. The resulting data is stored in the pui32Dest array. The function blocks until all of the data has been processed. If processing is successful, the function returns true.

Note:

This functions assumes that the DES module has been configured, and initialization values and keys have been written.

Returns:

true or false.

8.2.2.3 DESDataRead

Reads plaintext/ciphertext from data registers with blocking.

Prototype:

```
void
DESDataRead(uint32_t ui32Base,
            uint32_t *pui32Dest)
```

Parameters:

ui32Base is the base address of the DES module.

pui32Dest is a pointer to an array of bytes.

Description:

This function waits until the DES module is finished and encrypted or decrypted data is ready. The output data is then stored in the pui32Dest array.

Returns:

None

8.2.2.4 DESDataReadNonBlocking

Reads plaintext/ciphertext from data registers without blocking

Prototype:

```
bool
DESDataReadNonBlocking(uint32_t ui32Base,
                       uint32_t *pui32Dest)
```

Parameters:

ui32Base is the base address of the DES module.

pui32Dest is a pointer to an array of 2 words.

Description:

This function returns true if the data was ready when the function was called. If the data was not ready, false is returned.

Returns:

True or false.

8.2.2.5 DESDataWrite

Writes plaintext/ciphertext to data registers without blocking

Prototype:

```
void  
DESDataWrite(uint32_t ui32Base,  
             uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the DES module.
pui32Src is a pointer to an array of bytes.

Description:

This function waits until the DES module is ready before writing the data contained in the pui32Src array.

Returns:

None.

8.2.2.6 DESDataWriteNonBlocking

Writes plaintext/ciphertext to data registers without blocking

Prototype:

```
bool  
DESDataWriteNonBlocking(uint32_t ui32Base,  
                        uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the DES module.
pui32Src is a pointer to an array of 2 words.

Description:

This function returns false if the DES module is not ready to accept data. It returns true if the data was written successfully.

Returns:

true or false.

8.2.2.7 DESDMADisable

Disables DMA request sources in the DES module.

Prototype:

```
void  
DESDMADisable(uint32_t ui32Base,  
              uint32_t ui32Flags)
```

Parameters:

ui32Base is the base address of the DES module.

ui32Flags is a bit mask of the DMA requests to be disabled.

Description:

This function disables DMA request sources in the DES module. The *ui32Flags* parameter should be the logical OR of any of the following:

- **DES_DMA_CONTEXT_IN** - Context In
- **DES_DMA_DATA_OUT** - Data Out
- **DES_DMA_DATA_IN** - Data In

Returns:

None.

8.2.2.8 DESDMAEnable

Enables DMA request sources in the DES module.

Prototype:

```
void  
DESDMAEnable(uint32_t ui32Base,  
             uint32_t ui32Flags)
```

Parameters:

ui32Base is the base address of the DES module.

ui32Flags is a bit mask of the DMA requests to be enabled.

Description:

This function enables DMA request sources in the DES module. The *ui32Flags* parameter should be the logical OR of any of the following:

- **DES_DMA_CONTEXT_IN** - Context In
- **DES_DMA_DATA_OUT** - Data Out
- **DES_DMA_DATA_IN** - Data In

Returns:

None.

8.2.2.9 DESIntClear

Clears interrupts in the DES module.

Prototype:

```
void  
DESIntClear(uint32_t ui32Base,  
            uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the DES module.

ui32IntFlags is a bit mask of the interrupts to be disabled.

Description:

This function disables interrupt sources in the DES module. *ui32IntFlags* should be a logical OR of one or more of the following values:

- **DES_INT_DMA_CONTEXT_IN** - Context interrupt
- **DES_INT_DMA_DATA_IN** - Data input interrupt
- **DES_INT_DMA_DATA_OUT** - Data output interrupt

Note:

The DMA done interrupts are the only interrupts that can be cleared. The remaining interrupts can be disabled instead using [DESIntDisable\(\)](#).

Returns:

None.

8.2.2.10 DESIntDisable

Disables interrupts in the DES module.

Prototype:

```
void  
DESIntDisable(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the DES module.

ui32IntFlags is a bit mask of the interrupts to be disabled.

Description:

This function disables interrupt sources in the DES module. *ui32IntFlags* should be a logical OR of one or more of the following values:

- **DES_INT_CONTEXT_IN** - Context interrupt
- **DES_INT_DATA_IN** - Data input interrupt
- **DES_INT_DATA_OUT** - Data output interrupt
- **DES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **DES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **DES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Returns:

None.

8.2.2.11 DESIntEnable

Enables interrupts in the DES module.

Prototype:

```
void
DESIntEnable(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the DES module.

ui32IntFlags is a bit mask of the interrupts to be enabled.

Description:

ui32IntFlags should be a logical OR of one or more of the following values:

- **DES_INT_CONTEXT_IN** - Context interrupt
- **DES_INT_DATA_IN** - Data input interrupt
- **DES_INT_DATA_OUT** - Data output interrupt
- **DES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **DES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **DES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Returns:

None.

8.2.2.12 DESIntRegister

Registers an interrupt handler for the DES module.

Prototype:

```
void
DESIntRegister(uint32_t ui32Base,
                void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the DES module.

pfnHandler is a pointer to the function to be called when the enabled DES interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables DES interrupts on the interrupt controller; specific DES interrupt sources must be enabled using [DESIntEnable\(\)](#). The interrupt handler being registered must clear the source of the interrupt using [DESIntClear\(\)](#).

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, [IntEnable\(\)](#) should be used to enable DES interrupts on the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

8.2.2.13 DESIntStatus

Returns the current interrupt status of the DES module.

Prototype:

```
uint32_t  
DESIntStatus(uint32_t ui32Base,  
             bool bMasked)
```

Parameters:

ui32Base is the base address of the DES module.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function gets the current interrupt status of the DES module. The value returned is a logical OR of the following values:

- **DES_INT_CONTEXT_IN** - Context interrupt
- **DES_INT_DATA_IN** - Data input interrupt
- **DES_INT_DATA_OUT_INT** - Data output interrupt
- **DES_INT_DMA_CONTEXT_IN** - Context DMA done interrupt
- **DES_INT_DMA_DATA_IN** - Data input DMA done interrupt
- **DES_INT_DMA_DATA_OUT** - Data output DMA done interrupt

Returns:

A bit mask of the current interrupt status.

8.2.2.14 DESIntUnregister

Unregisters an interrupt handler for the DES module.

Prototype:

```
void  
DESIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the DES module.

Description:

This function unregisters the previously registered interrupt handler and disables the interrupt in the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

8.2.2.15 DESIVSet

Sets the initialization vector in the DES module.

Prototype:

```
bool
DESIVSet(uint32_t ui32Base,
         uint32_t *pui32IVdata)
```

Parameters:

ui32Base is the base address of the DES module.

pui32IVdata is a pointer to an array of 64 bits (2 words) of data to be written into the initialization vectors registers.

Description:

This function sets the initialization vector in the DES module. It returns true if the registers were successfully written. If the context registers cannot be written at the time the function was called, then false is returned.

Returns:

True or false.

8.2.2.16 DESKeySet

Sets the key used for DES operations.

Prototype:

```
void
DESKeySet(uint32_t ui32Base,
          uint32_t *pui32Key)
```

Parameters:

ui32Base is the base address of the DES module.

pui32Key is a pointer to an array that holds the key

Description:

This function sets the key used for DES operations.

pui32Key should be 64 bits long (2 words) if single DES is being used or 192 bits (6 words) if triple DES is being used.

Returns:

None.

8.2.2.17 DESLengthSet

Sets the cryptographic data length in the DES module.

Prototype:

```
void  
DESLengthSet(uint32_t ui32Base,  
             uint32_t ui32Length)
```

Parameters:

ui32Base is the base address of the DES module.

ui32Length is the length of the data in bytes.

Description:

This function writes the cryptographic data length into the DES module. When this register is written, the engine is triggered to start using this context.

Note:

Data lengths up to (2³² - 1) bytes are allowed.

Returns:

None.

8.2.2.18 DESReset

Resets the DES Module.

Prototype:

```
void  
DESReset(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the DES module.

Description:

This function performs a soft-reset sequence of the DES module.

Returns:

None.

8.3 DES Programming Example

The following example sets up the DES module to perform an encryption operation on four blocks of data in CBC mode with an 64-bit key.

```
//  
// Random data for encryption/decryption.  
//  
uint32_t g_ui32DESPlainText[16] = {  
    0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373,  
    0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45,
```

```

0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520ala,
0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6
};

uint32_t g_ui32DESKey[2] = {
    0xc7f51c87, 0x8076211f
};

uint32_t g_ui32DESIV[2] = {
    0x6d8ecac4, 0x3b27c885
};

int
main(void)
{
    pui32DESCipherText[16];

    //
    // Enable the CCM module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ECO);

    //
    // Wait for the CCM module to be ready.
    //
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ECO))
    {
    }

    //
    // Reset the DES module before use.
    //
    DESReset(DES_BASE);

    //
    // Configure the DES module.
    //
    DESConfigSet(DES_BASE,
        DES_CFG_DIR_ENCRYPT |
        DES_CFG_MODE_CBC |
        DES_CFG_SINGLE);

    //
    // Set the key.
    //
    DESKeySet(DES_BASE, g_ui32DESKey);

    //
    // Write the initial value registers.
    //
    DESIVSet(DES_BASE, g_ui32DESIV);

    //
    // Perform the encryption.
    //
    // The ciphertext should be:
    // {0x95a74bd5, 0x8595094a, 0xf116bd1d, 0x2aed0a67,
    // 0x4b4e730b, 0x163335ca, 0x8554d039, 0xb9f7e301,
    // 0x599421e2, 0x5db5db40, 0x17fc1ce2, 0xf048de81,
    // 0xabdf0d51, 0x6ce768f1, 0x8233fbdb, 0x3efe7bae}
    //
    DESDataProcess(DES_BASE, g_ui32DESPlaintext, g_ui32DESCipherText, 64);
}

```

8.4 TDES Programming Example

The following example sets up the TDES module to perform an decryption operation on four blocks of data in CBC mode with an 192-bit key.

```
//  
// Random data for encryption/decryption.  
//  
uint32_t g_ui32TDESCipherText[16] = {  
    0x24c69385, 0xb338be54, 0x6eeeb276, 0x1a952b4e,  
    0x7242ce4b, 0x9ec147cf, 0x765916ee, 0x3d25e685,  
    0xfe5865b4, 0xf2238cb8, 0x2a5b68d5, 0x0f79a41a,  
    0x6f4a7601, 0x7a57235f, 0xce84d08a, 0x1a34d011  
};  
  
uint32_t g_ui32TDESKey[6] = {  
    0xc7f51c87, 0x8076211f, 0x5de5c871, 0xa243cf7e,  
    0xd25fdb75, 0xad73068f  
};  
  
uint32_t g_ui32TDESIV[2] = {  
    0x6d8ecac4, 0x3b27c885  
};  
  
int  
main(void)  
{  
    uint32_t pui32TDESCipherText[16];  
  
    //  
    // Enable the CRC module.  
    //  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ECO);  
  
    //  
    // Wait for the CRC module to be ready.  
    //  
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ECO))  
    {  
    }  
  
    //  
    // Reset the DES module before use.  
    //  
    DESReset(DES_BASE);  
  
    //  
    // Configure the DES module.  
    //  
    DESConfigSet(DES_BASE,  
        DES_CFG_DIR_ENCRYPT |  
        DES_CFG_MODE_CBC |  
        DES_CFG_TRIPLE);  
  
    //  
    // Set the key.  
    //  
    DESKeySet(DES_BASE, g_ui32TDESKey);  
  
    //  
    // Write the initial value registers.  
    //  
    DESIVSet(DES_BASE, g_ui32TDESIV);
```

```
//  
// Perform the decryption.  
//  
// The ciphertext should be:  
// {0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373,  
// 0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45,  
// 0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520ala,  
// 0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6}  
//  
DESDataProcess(DES_BASE, g_ui32TDESPlainText, g_ui32TDESCipherText, 64);  
};
```


9 EEPROM

Introduction	113
API Functions	114
Programming Example	128

9.1 Introduction

The EEPROM API provides a set of functions for interacting with the on-chip EEPROM providing easy-to-use non-volatile data storage. Functions are provided to program and erase the EEPROM, configure the EEPROM protection, and handle the EEPROM interrupt.

The EEPROM can be programmed on a word-by-word basis and, unlike flash, the application need not explicitly erase a word or page before writing a new value to it.

The EEPROM controller has the ability to generate an interrupt when an invalid access is attempted (such as reading from a protected block). This interrupt can be used to validate the operation of a program; the interrupt prevents invalid accesses from being silently ignored, hiding potential bugs. An interrupt can also be generated when an erase or programming operation has completed.

The size of the available EEPROM and the number of blocks it contains varies between different members of the Tiva family. API functions [EEPROMSizeGet\(\)](#) and [EEPROMBlockCountGet\(\)](#) are provided to allow this information to be determined at runtime.

Data protection is supported at both the device and block levels with configurable passwords used to control read and write access. Additionally, blocks may be configured to allow access only while the CPU is running in supervisor mode. A second protection mechanism allows one or more EEPROM blocks to be made completely inaccessible to software until the next system reset.

This driver is contained in `driverlib/eeprom.c`, with `driverlib/eeprom.h` containing the API declarations for use by applications.

9.1.1 EEPROM Protection

The EEPROM device is organized into a number of blocks each of which may be configured with various protection options to control an application's ability to read and/or write data. Additionally, protection options set on the first block of the device, block 0, affect access to the EEPROM as a whole, allowing global options to be set on block 0 and individual block protection to be layered on top of this.

Each block may be configured for two protection states, one which is in effect when the block is locked and a second which applies when the block is unlocked. Unlocking is performed by writing a 32- to 96-bit password which has previously been set and committed by the user.

If a password is set on block 0, all other blocks in the device and the registers which control them are inaccessible until block 0 is unlocked. At this point, the protection set on each individual block applies with those blocks being individually lockable via their own passwords.

The EEPROM driver allows three specific protection modes to be set on each block. These modes are defined by the following labels from `eeprom.h` which define the protection provided if the block has no password set, if it has a password set and is not unlocked and if it has a password set and is unlocked. Additionally, `EEPROM_PROT_SUPERVISOR_ONLY` may be ORed with each of these

labels when calling [EEPROMBlockProtectSet\(\)](#) to prevent all accesses to the block when the CPU is executing in user mode.

9.1.1.1 EEPROM_PROT_RW_LRO_URW

If no password is set for the block, this protection level allows both read and write access to the block data.

If a password is set for the block and the block is locked, this protection level allows only read access to the block data.

If a password is set for the block and the block is unlocked, this protection level allows both read and write access to the block data.

9.1.1.2 EEPROM_PROT_NA_LNA_URW

If no password is set for the block, this protection level prevents the block data from being read or written.

If a password is set for the block and the block is locked, this protection level prevents the block data from being read or written.

If a password is set for the block and the block is unlocked, this protection level allows both read and write access to the block data.

9.1.1.3 EEPROM_PROT_RO_LNA_URO

If no password is set for the block, this protection level allows only read access to the block data.

If a password is set for the block and the block is locked, this protection level prevents the block data from being read or written.

If a password is set for the block and the block is unlocked, this protection level allows only read access to the block data.

9.2 API Functions

Defines

- [EEPROM_INIT_ERROR](#)
- [EEPROM_INIT_OK](#)
- [EEPROM_INT_PROGRAM](#)
- [EEPROM_PROT_NA_LNA_URW](#)
- [EEPROM_PROT_RO_LNA_URO](#)
- [EEPROM_PROT_RW_LRO_URW](#)
- [EEPROM_PROT_SUPERVISOR_ONLY](#)
- [EEPROM_RC_NOPERM](#)
- [EEPROM_RC_WKCOPY](#)

- [EEPROM_RC_WKERASE](#)
- [EEPROM_RC_WORKING](#)
- [EEPROM_RC_WRBUSY](#)
- [EEPROMAddrFromBlock\(ui32Block\)](#)
- [EEPROMBlockFromAddr\(ui32Addr\)](#)

Functions

- [uint32_t EEPROMBlockCountGet \(void\)](#)
- [void EEPROMBlockHide \(uint32_t ui32Block\)](#)
- [uint32_t EEPROMBlockLock \(uint32_t ui32Block\)](#)
- [uint32_t EEPROMBlockPasswordSet \(uint32_t ui32Block, uint32_t *pui32Password, uint32_t ui32Count\)](#)
- [uint32_t EEPROMBlockProtectGet \(uint32_t ui32Block\)](#)
- [uint32_t EEPROMBlockProtectSet \(uint32_t ui32Block, uint32_t ui32Protect\)](#)
- [uint32_t EEPROMBlockUnlock \(uint32_t ui32Block, uint32_t *pui32Password, uint32_t ui32Count\)](#)
- [uint32_t EEPROMInit \(void\)](#)
- [void EEPROMIntClear \(uint32_t ui32IntFlags\)](#)
- [void EEPROMIntDisable \(uint32_t ui32IntFlags\)](#)
- [void EEPROMIntEnable \(uint32_t ui32IntFlags\)](#)
- [uint32_t EEPROMIntStatus \(bool bMasked\)](#)
- [uint32_t EEPROMMassErase \(void\)](#)
- [uint32_t EEPROMProgram \(uint32_t *pui32Data, uint32_t ui32Address, uint32_t ui32Count\)](#)
- [uint32_t EEPROMProgramNonBlocking \(uint32_t ui32Data, uint32_t ui32Address\)](#)
- [void EEPROMRead \(uint32_t *pui32Data, uint32_t ui32Address, uint32_t ui32Count\)](#)
- [uint32_t EEPROMSizeGet \(void\)](#)
- [uint32_t EEPROMStatusGet \(void\)](#)

9.2.1 Detailed Description

The EEPROM API is broken into four groups of functions: those that deal with reading the EEPROM, those that deal with programming the EEPROM, those that deal with EEPROM protection, and those that deal with interrupt handling.

EEPROM reading is managed with [EEPROMRead\(\)](#).

EEPROM programming is managed with [EEPROMMassErase\(\)](#), [EEPROMProgram\(\)](#) and [EEPROMProgramNonBlocking\(\)](#).

EEPROM protection is managed with [EEPROMBlockProtectGet\(\)](#), [EEPROMBlockProtectSet\(\)](#), [EEPROMBlockPasswordSet\(\)](#), [EEPROMBlockLock\(\)](#), [EEPROMBlockUnlock\(\)](#) and [EEPROMBlockHide\(\)](#).

Interrupt handling is managed with [EEPROMIntEnable\(\)](#), [EEPROMIntDisable\(\)](#), [EEPROMIntStatus\(\)](#), and [EEPROMIntClear\(\)](#).

An additional function, [EEPROMSizeGet\(\)](#) is provided to allow an application to query the size of the device storage and the number of blocks it contains.

9.2.2 Define Documentation

9.2.2.1 EEPROM_INIT_ERROR

Definition:

```
#define EEPROM_INIT_ERROR
```

Description:

This value may be returned from a call to [EEPROMInit\(\)](#). It indicates that a previous data or protection write operation was interrupted by a reset event and that the EEPROM peripheral was unable to clean up after the problem. This situation may be resolved with another reset or may be fatal depending upon the cause of the problem. For example, if the voltage to the part is unstable, retrying once the voltage has stabilized may clear the error.

9.2.2.2 EEPROM_INIT_OK

Definition:

```
#define EEPROM_INIT_OK
```

Description:

This value may be returned from a call to [EEPROMInit\(\)](#). It indicates that no previous write operations were interrupted by a reset event and that the EEPROM peripheral is ready for use.

9.2.2.3 EEPROM_INT_PROGRAM

Definition:

```
#define EEPROM_INT_PROGRAM
```

Description:

This value may be passed to [EEPROMIntEnable\(\)](#) and [EEPROMIntDisable\(\)](#) and is returned by [EEPROMIntStatus\(\)](#) if an EEPROM interrupt is currently being signaled.

9.2.2.4 EEPROM_PROT_NA_LNA_URW

Definition:

```
#define EEPROM_PROT_NA_LNA_URW
```

Description:

This value may be passed to [EEPROMBlockProtectSet\(\)](#) or returned from [EEPROMBlockProtectGet\(\)](#). It indicates that the block should offer neither read nor write access unless it is protected by a password and unlocked.

9.2.2.5 EEPROM_PROT_RO_LNA_URO

Definition:

```
#define EEPROM_PROT_RO_LNA_URO
```

Description:

This value may be passed to [EEPROMBlockProtectSet\(\)](#) or returned from [EEPROMBlockProtectGet\(\)](#). It indicates that the block should offer read-only access when no password is set or when a password is set and the block is unlocked. When a password is set and the block is locked, neither read nor write access is permitted.

9.2.2.6 EEPROM_PROT_RW_LRO_URW

Definition:

```
#define EEPROM_PROT_RW_LRO_URW
```

Description:

This value may be passed to [EEPROMBlockProtectSet\(\)](#) or returned from [EEPROMBlockProtectGet\(\)](#). It indicates that the block should offer read/write access when no password is set or when a password is set and the block is unlocked, and read-only access when a password is set but the block is locked.

9.2.2.7 EEPROM_PROT_SUPERVISOR_ONLY

Definition:

```
#define EEPROM_PROT_SUPERVISOR_ONLY
```

Description:

This bit may be ORed with the protection option passed to [EEPROMBlockProtectSet\(\)](#) or returned from [EEPROMBlockProtectGet\(\)](#). It restricts EEPROM access to threads running in supervisor mode and prevents access to an EEPROM block when the CPU is in user mode.

9.2.2.8 EEPROM_RC_NOPERM

Definition:

```
#define EEPROM_RC_NOPERM
```

Description:

This return code bit indicates that an attempt was made to write a value but the destination permissions disallow write operations. This may be due to the destination block being locked, access protection set to prohibit writes or an attempt to write a password when one is already written.

9.2.2.9 EEPROM_RC_WKCOPY

Definition:

```
#define EEPROM_RC_WKCOPY
```

Description:

This return code bit indicates that the EEPROM programming state machine is currently copying to or from the internal copy buffer to make room for a newly written value. It is provided as a status indicator and does not indicate an error.

9.2.2.10 EEPROM_RC_WKERASE

Definition:

```
#define EEPROM_RC_WKERASE
```

Description:

This return code bit indicates that the EEPROM programming state machine is currently erasing the internal copy buffer. It is provided as a status indicator and does not indicate an error.

9.2.2.11 EEPROM_RC_WORKING

Definition:

```
#define EEPROM_RC_WORKING
```

Description:

This return code bit indicates that the EEPROM programming state machine is currently working. No new write operations should be attempted until this bit is clear.

9.2.2.12 EEPROM_RC_WRBUSY

Definition:

```
#define EEPROM_RC_WRBUSY
```

Description:

This return code bit indicates that an attempt was made to read from the EEPROM while a write operation was in progress.

9.2.2.13 EEPROMAddrFromBlock

Returns the offset address of the first word in an EEPROM block.

Definition:

```
#define EEPROMAddrFromBlock(ui32Block)
```

Parameters:

ui32Block is the index of the EEPROM block whose first word address is to be returned.

Description:

This macro may be used to determine the address of the first word in a given EEPROM block. The address returned is expressed as a byte offset from the base of EEPROM storage.

Returns:

Returns the address of the first word in the given EEPROM block.

9.2.2.14 EEPROMBlockFromAddr

Returns the EEPROM block number containing a given offset address.

Definition:

```
#define EEPROMBlockFromAddr(ui32Addr)
```

Parameters:

ui32Addr is the linear, byte address of the EEPROM location whose block number is to be returned. This is a zero-based offset from the start of the EEPROM storage.

Description:

This macro may be used to translate an EEPROM address offset into a block number suitable for use in any of the driver's block protection functions. The address provided is expressed as a byte offset from the base of the EEPROM.

Returns:

Returns the zero-based block number which contains the passed address.

9.2.3 Function Documentation

9.2.3.1 EEPROMBlockCountGet

Determines the number of blocks in the EEPROM.

Prototype:

```
uint32_t
EEPROMBlockCountGet(void)
```

Description:

This function may be called to determine the number of blocks in the EEPROM. Each block is the same size and the number of bytes of storage contained in a block may be determined by dividing the size of the device, obtained via a call to the [EEPROMSizeGet\(\)](#) function, by the number of blocks returned by this function.

Returns:

Returns the total number of blocks in the device EEPROM.

9.2.3.2 EEPROMBlockHide

Hides an EEPROM block until the next reset.

Prototype:

```
void
EEPROMBlockHide(uint32_t ui32Block)
```

Parameters:

ui32Block is the EEPROM block number which is to be hidden.

Description:

This function hides an EEPROM block other than block 0. Once hidden, a block is completely inaccessible until the next reset. This mechanism allows initialization code to have access to data which is to be hidden from the rest of the application. Unlike applications using passwords, an application making use of block hiding need not contain any embedded passwords which could be found through disassembly.

Returns:

None.

9.2.3.3 EEPROMBlockLock

Locks a password-protected EEPROM block.

Prototype:

```
uint32_t  
EEPROMBlockLock(uint32_t ui32Block)
```

Parameters:

ui32Block is the EEPROM block number which is to be locked.

Description:

This function locks an EEPROM block that has previously been protected by writing a password. Access to the block once it is locked is determined by the protection settings applied via a previous call to the [EEPROMBlockProtectSet\(\)](#) function. If no password has previously been set for the block, this function has no effect.

Locking block 0 has the effect of making all other blocks in the EEPROM inaccessible.

Returns:

Returns the lock state for the block on exit, 1 if unlocked (as would be the case if no password was set) or 0 if locked.

9.2.3.4 EEPROMBlockPasswordSet

Sets the password used to protect an EEPROM block.

Prototype:

```
uint32_t  
EEPROMBlockPasswordSet(uint32_t ui32Block,  
                      uint32_t *pui32Password,  
                      uint32_t ui32Count)
```

Parameters:

ui32Block is the EEPROM block number for which the password is to be set.

pui32Password points to an array of `uint32_t` values comprising the password to set. Each element may be any 32-bit value other than 0xFFFFFFFF. This array must contain the number of elements given by the `ui32Count` parameter.

ui32Count provides the number of `uint32_t`s in the `ui32Password`. Valid values are 1, 2 and 3.

Description:

This function allows the password used to unlock an EEPROM block to be set. Valid passwords may be either 32, 64 or 96 bits comprising words with any value other than 0xFFFFFFFF. The password may only be set once. Any further attempts to set the password result in an error. Once the password is set, the block remains unlocked until [EEPROMBlockLock\(\)](#) is called for that block or block 0, or a reset occurs.

If a password is set on block 0, this affects locking of the peripheral as a whole. When block 0 is locked, all other EEPROM blocks are inaccessible until block 0 is unlocked. Once block 0 is unlocked, other blocks become accessible according to any passwords set on those blocks and the protection set for that block via a call to [EEPROMBlockProtectSet\(\)](#).

Returns:

Returns a logical OR combination of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING** to indicate status and error conditions.

9.2.3.5 EEPROMBlockProtectGet

Returns the current protection level for an EEPROM block.

Prototype:

```
uint32_t
EEPROMBlockProtectGet(uint32_t ui32Block)
```

Parameters:

ui32Block is the block number for which the protection level is to be queried.

Description:

This function returns the current protection settings for a given EEPROM block. If block 0 is currently locked, it must be unlocked prior to calling this function to query the protection setting for other blocks.

Returns:

Returns one of **EEPROM_PROT_RW_LRO_URW**, **EEPROM_PROT_NA_LNA_URW** or **EEPROM_PROT_RO_LNA_URO** optionally OR-ed with **EEPROM_PROT_SUPERVISOR_ONLY**.

9.2.3.6 EEPROMBlockProtectSet

Set the current protection options for an EEPROM block.

Prototype:

```
uint32_t
EEPROMBlockProtectSet(uint32_t ui32Block,
                      uint32_t ui32Protect)
```

Parameters:

ui32Block is the block number for which the protection options are to be set.

ui32Protect consists of one of the values **EEPROM_PROT_RW_LRO_URW**, **EEPROM_PROT_NA_LNA_URW** or **EEPROM_PROT_RO_LNA_URO** optionally ORed with **EEPROM_PROT_SUPERVISOR_ONLY**.

Description:

This function sets the protection settings for a given EEPROM block assuming no protection settings have previously been written. Note that protection settings applied to block 0 have special meaning and control access to the EEPROM peripheral as a whole. Protection settings applied to blocks numbered 1 and above are layered above any protection set on block 0 such that the effective protection on each block is the logical OR of the protection flags set for block 0 and for the target block. This protocol allows global protection options to be set for the whole device via block 0 and more restrictive protection settings to be set on a block-by-block basis.

The protection flags indicate access permissions as follow:

EEPROM_PROT_SUPERVISOR_ONLY restricts access to the block to threads running in supervisor mode. If clear, both user and supervisor threads can access the block.

EEPROM_PROT_RW_LRO_URW provides read/write access to the block if no password is set or if a password is set and the block is unlocked. If the block is locked, only read access is permitted.

EEPROM_PROT_NA_LNA_URW provides neither read nor write access unless a password is set and the block is unlocked. If the block is unlocked, both read and write access are permitted.

EEPROM_PROT_RO_LNA_URO provides read access to the block if no password is set or if a password is set and the block is unlocked. If the block is password protected and locked, neither read nor write access is permitted.

Returns:

Returns a logical OR combination of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING** to indicate status and error conditions.

9.2.3.7 EEPROMBlockUnlock

Unlocks a password-protected EEPROM block.

Prototype:

```
uint32_t  
EEPROMBlockUnlock(uint32_t ui32Block,  
                  uint32_t *pui32Password,  
                  uint32_t ui32Count)
```

Parameters:

ui32Block is the EEPROM block number which is to be unlocked.

pui32Password points to an array of `uint32_t` values containing the password for the block. Each element must match the password originally set via a call to [EEPROMBlockPasswordSet\(\)](#).

ui32Count provides the number of elements in the `pui32Password` array and must match the value originally passed to [EEPROMBlockPasswordSet\(\)](#). Valid values are 1, 2 and 3.

Description:

This function unlocks an EEPROM block that has previously been protected by writing a password. Access to the block once it is unlocked is determined by the protection settings applied via a previous call to the [EEPROMBlockProtectSet\(\)](#) function.

To successfully unlock an EEPROM block, the password provided must match the password provided on the original call to [EEPROMBlockPasswordSet\(\)](#). If an incorrect password is provided, the block remains locked.

Unlocking block 0 has the effect of making all other blocks in the device accessible according to their own access protection settings. When block 0 is locked, all other EEPROM blocks are inaccessible.

Returns:

Returns the lock state for the block on exit, 1 if unlocked or 0 if locked.

9.2.3.8 EEPROMInit

Performs any necessary recovery in case of power failures during write.

Prototype:

```
uint32_t
EEPROMInit(void)
```

Description:

This function **must** be called after [SysCtlPeripheralEnable\(\)](#) and before the EEPROM is accessed. It is used to check for errors in the EEPROM state such as from power failure during a previous write operation. The function detects these errors and performs as much recovery as possible.

If **EEPROM_INIT_ERROR** is returned, the EEPROM was unable to recover its state. If power is stable when this occurs, this indicates a fatal error and is likely an indication that the EEPROM memory has exceeded its specified lifetime write/erase specification. If the supply voltage is unstable when this return code is observed, retrying the operation once the voltage is stabilized may clear the error.

Failure to call this function after a reset may lead to incorrect operation or permanent data loss if the EEPROM is later written.

Returns:

Returns **EEPROM_INIT_OK** if no errors were detected or **EEPROM_INIT_ERROR** if the EEPROM peripheral cannot currently recover from an interrupted write or erase operation.

9.2.3.9 EEPROMIntClear

Clears the EEPROM interrupt.

Prototype:

```
void
EEPROMIntClear(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags indicates which interrupt sources to clear. Currently, the only valid value is **EEPROM_INT_PROGRAM**.

Description:

This function allows an application to clear the EEPROM interrupt.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

9.2.3.10 EEPROMIntDisable

Disables the EEPROM interrupt.

Prototype:

```
void  
EEPROMIntDisable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags indicates which EEPROM interrupt source to disable. This must be **EPP-ROM_INT_PROGRAM** currently.

Description:

This function disables the EEPROM interrupt and prevents calls to the interrupt vector when any EEPROM write or erase operation completes. The EEPROM peripheral shares a single interrupt vector with the flash memory subsystem, **INT_FLASH**. This function is provided as a convenience but the EEPROM interrupt can also be disabled using a call to [FlashIntDisable\(\)](#) passing **FLASH_INT_EEPROM** in the *ui32IntFlags* parameter.

Returns:

None.

9.2.3.11 EEPROMIntEnable

Enables the EEPROM interrupt.

Prototype:

```
void  
EEPROMIntEnable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags indicates which EEPROM interrupt source to enable. This must be **EPP-ROM_INT_PROGRAM** currently.

Description:

This function enables the EEPROM interrupt. When enabled, an interrupt is generated when any EEPROM write or erase operation completes. The EEPROM peripheral shares a single interrupt vector with the flash memory subsystem, **INT_FLASH**. This function is provided as a convenience but the EEPROM interrupt can also be enabled using a call to [FlashIntEnable\(\)](#) passing **FLASH_INT_EEPROM** in the *ui32IntFlags* parameter.

Returns:

None.

9.2.3.12 EEPROMIntStatus

Reports the state of the EEPROM interrupt.

Prototype:

```
uint32_t
EEPROMIntStatus(bool bMasked)
```

Parameters:

bMasked determines whether the masked or unmasked state of the interrupt is to be returned.
If bMasked is **true**, the masked state is returned, otherwise the unmasked state is returned.

Description:

This function allows an application to query the state of the EEPROM interrupt. If active, the interrupt may be cleared by calling [EEPROMIntClear\(\)](#).

Returns:

Returns **EEPROM_INT_PROGRAM** if an interrupt is being signaled or 0 otherwise.

9.2.3.13 EEPROMMassErase

Erases the EEPROM and returns it to the factory default condition.

Prototype:

```
uint32_t
EEPROMMassErase(void)
```

Description:

This function completely erases the EEPROM and removes any and all access protection on its blocks, leaving the device in the factory default condition. After this operation, all EEPROM words contain the value 0xFFFFFFFF and all blocks are accessible for both read and write operations in all CPU modes. No passwords are active.

The function is synchronous and does not return until the erase operation has completed.

Returns:

Returns 0 on success or non-zero values on failure. Failure codes are logical OR combinations of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

9.2.3.14 EEPROMProgram

Writes data to the EEPROM.

Prototype:

```
uint32_t
EEPROMProgram(uint32_t *pui32Data,
```

```
    uint32_t ui32Address,  
    uint32_t ui32Count)
```

Parameters:

pui32Data points to the first word of data to write to the EEPROM.

ui32Address defines the byte address within the EEPROM that the data is to be written to.
This value must be a multiple of 4.

ui32Count defines the number of bytes of data that is to be written. This value must be a
multiple of 4.

Description:

This function may be called to write data into the EEPROM at a given word-aligned address.
The call is synchronous and returns only after all data has been written or an error occurs.

Returns:

Returns 0 on success or non-zero values on failure. Failure codes are logical OR combinations
of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEP-
ROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

9.2.3.15 EEPROMProgramNonBlocking

Writes a word to the EEPROM.

Prototype:

```
uint32_t  
EEPROMProgramNonBlocking(uint32_t ui32Data,  
                         uint32_t ui32Address)
```

Parameters:

ui32Data is the word to write to the EEPROM.

ui32Address defines the byte address within the EEPROM to which the data is to be written.
This value must be a multiple of 4.

Description:

This function is intended to allow EEPROM programming under interrupt control. It may be
called to start the process of writing a single word of data into the EEPROM at a given word-
aligned address. The call is asynchronous and returns immediately without waiting for the
write to complete. Completion of the operation is signaled by means of an interrupt from the
EEPROM module. The EEPROM peripheral shares a single interrupt vector with the flash
memory subsystem, **INT_FLASH**.

Returns:

Returns status and error information in the form of a logical OR combinations
of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEP-
ROM_RC_WKERASE** and **EEPROM_RC_WORKING**. Flags **EEPROM_RC_WKCOPY**, **EEP-
ROM_RC_WKERASE**, and **EEPROM_RC_WORKING** are expected in normal operation and
do not indicate an error.

9.2.3.16 EEPROMRead

Reads data from the EEPROM.

Prototype:

```
void
EEPROMRead(uint32_t *pui32Data,
            uint32_t ui32Address,
            uint32_t ui32Count)
```

Parameters:

pui32Data is a pointer to storage for the data read from the EEPROM. This pointer must point to at least *ui32Count* bytes of available memory.

ui32Address is the byte address within the EEPROM from which data is to be read. This value must be a multiple of 4.

ui32Count is the number of bytes of data to read from the EEPROM. This value must be a multiple of 4.

Description:

This function may be called to read a number of words of data from a word-aligned address within the EEPROM. Data read is copied into the buffer pointed to by the *pui32Data* parameter.

Returns:

None.

9.2.3.17 EEPROMSizeGet

Determines the size of the EEPROM.

Prototype:

```
uint32_t
EEPROMSizeGet (void)
```

Description:

This function returns the size of the EEPROM in bytes.

Returns:

Returns the total number of bytes in the EEPROM.

9.2.3.18 EEPROMStatusGet

Returns status on the last EEPROM program or erase operation.

Prototype:

```
uint32_t
EEPROMStatusGet (void)
```

Description:

This function returns the current status of the last program or erase operation performed by the EEPROM. It is intended to provide error information to applications programming or setting EEPROM protection options under interrupt control.

Returns:

Returns 0 if the last program or erase operation completed without any errors. If an operation is ongoing or an error occurred, the return value is a logical OR combination of **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

9.3 Programming Example

The following example shows how to use the EEPROM API to write a block of data and read it back.

```
uint32_t pui32Data[2];
uint32_t pui32Read[2];

//
// Program some data into the EEPROM at address 0x400.
//
pui32Data[0] = 0x12345678;
pui32Data[1] = 0x56789abc;
EEPROMProgram(pui32Data, 0x400, sizeof(pui32Data));

//
// Read it back.
//
EEPROMRead(pui32Read, 0x400, sizeof(pui32Read));
```

10 Ethernet Controller

Introduction	129
API Functions	129
Programming Example	193

10.1 Introduction

The Tiva Ethernet controller consists of a fully integrated media access controller (MAC) and a network physical (PHY) interface device. The Ethernet controller conforms to IEEE 802.3 specifications and fully supports 10BASE-T and 100BASE-TX standards. Additionally, external PHYs may be connected via either MII or RMII interfaces. Note that this document describes the Ethernet MAC found in Tiva devices which differs markedly from that found in older LM3S devices. The new MAC architecture provides very much improved data handling and throughput using a DMA-based engine in addition to many new hardware features including automatic checksum calculation and insertion, hardware perfect and hash packet filtering, low power operation with remote wakeup and wake-on-LAN capability, VLAN tagging and IEEE1588 types 1 and 2 support. As a result, the API provided has been completely redesigned and cannot be used with older parts.

The Ethernet MAC API provides the set of functions required to implement an interrupt-driven Ethernet driver for the Tiva Ethernet MAC. Functions are provided to configure and control the MAC, to access the register set on the PHY, to transmit and receive Ethernet packets using the MAC's integrated DMA engine, to control timestamp handling for IEEE1588, to configure and control low power operation, to configure and control VLAN tagging, and to configure and control the peripheral interrupts.

This driver is contained in `driverlib/emac.c`, with `driverlib/emac.h` containing the API declarations for use by applications.

10.2 API Functions

Data Structures

- `tEMACDES3`
- `tEMACDMADescriptor`
- `tEMACWakeUpFrameFilter`

Functions

- `uint32_t EMACAddrFilterGet (uint32_t ui32Base, uint32_t ui32Index)`
- `void EMACAddrFilterSet (uint32_t ui32Base, uint32_t ui32Index, uint32_t ui32Config)`
- `void EMACAddrGet (uint32_t ui32Base, uint32_t ui32Index, uint8_t *pui8MACAddr)`
- `void EMACAddrSet (uint32_t ui32Base, uint32_t ui32Index, const uint8_t *pui8MACAddr)`
- `void EMACConfigGet (uint32_t ui32Base, uint32_t *pui32Config, uint32_t *pui32Mode, uint32_t *pui32RxMaxFrameSize)`

- void **EMACConfigSet** (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32ModeFlags, uint32_t ui32RxMaxFrameSize)
- uint32_t **EMACDMAStateGet** (uint32_t ui32Base)
- uint32_t **EMACFrameFilterGet** (uint32_t ui32Base)
- void **EMACFrameFilterSet** (uint32_t ui32Base, uint32_t ui32FilterOpts)
- uint32_t **EMACHashFilterBitCalculate** (uint8_t *pui8MACAddr)
- void **EMACHashFilterGet** (uint32_t ui32Base, uint32_t *pui32HashHi, uint32_t *pui32HashLo)
- void **EMACHashFilterSet** (uint32_t ui32Base, uint32_t ui32HashHi, uint32_t ui32HashLo)
- void **EMACInit** (uint32_t ui32Base, uint32_t ui32SysClk, uint32_t ui32BusConfig, uint32_t ui32RxBurst, uint32_t ui32TxBurst, uint32_t ui32DescSkipSize)
- void **EMACIntClear** (uint32_t ui32Base, uint32_t ui32IntFlags)
- void **EMACIntDisable** (uint32_t ui32Base, uint32_t ui32IntFlags)
- void **EMACIntEnable** (uint32_t ui32Base, uint32_t ui32IntFlags)
- void **EMACIntRegister** (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t **EMACIntStatus** (uint32_t ui32Base, bool bMasked)
- void **EMACIntUnregister** (uint32_t ui32Base)
- uint32_t **EMACNumAddrGet** (uint32_t ui32Base)
- void **EMACPHYConfigSet** (uint32_t ui32Base, uint32_t ui32Config)
- uint16_t **EMACPHYExtendedRead** (uint32_t ui32Base, uint8_t ui8PhyAddr, uint16_t ui16RegAddr)
- void **EMACPHYExtendedWrite** (uint32_t ui32Base, uint8_t ui8PhyAddr, uint16_t ui16RegAddr, uint16_t ui16Value)
- void **EMACPHYPowerOff** (uint32_t ui32Base, uint8_t ui8PhyAddr)
- void **EMACPHYPowerOn** (uint32_t ui32Base, uint8_t ui8PhyAddr)
- uint16_t **EMACPHYRead** (uint32_t ui32Base, uint8_t ui8PhyAddr, uint8_t ui8RegAddr)
- void **EMACPHYWrite** (uint32_t ui32Base, uint8_t ui8PhyAddr, uint8_t ui8RegAddr, uint16_t ui16Data)
- uint32_t **EMACPowerManagementControlGet** (uint32_t ui32Base)
- void **EMACPowerManagementControlSet** (uint32_t ui32Base, uint32_t ui32Flags)
- uint32_t **EMACPowerManagementStatusGet** (uint32_t ui32Base)
- void **EMACRemoteWakeUpFrameFilterGet** (uint32_t ui32Base, tEMACWakeUpFrameFilter *pFilter)
- void **EMACRemoteWakeUpFrameFilterSet** (uint32_t ui32Base, const tEMACWakeUpFrameFilter *pFilter)
- void **EMACReset** (uint32_t ui32Base)
- void **EMACRxDisable** (uint32_t ui32Base)
- uint8_t * **EMACRxDMACurrentBufferGet** (uint32_t ui32Base)
- tEMACDMADescriptor * **EMACRxDMACurrentDescriptorGet** (uint32_t ui32Base)
- tEMACDMADescriptor * **EMACRxDMADescriptorListGet** (uint32_t ui32Base)
- void **EMACRxDMADescriptorListSet** (uint32_t ui32Base, tEMACDMADescriptor *pDescriptor)
- void **EMACRxDMAPollDemand** (uint32_t ui32Base)
- void **EMACRxEnable** (uint32_t ui32Base)
- void **EMACRxWatchdogTimerSet** (uint32_t ui32Base, uint8_t ui8Timeout)
- uint32_t **EMACStatusGet** (uint32_t ui32Base)
- void **EMACTimestampAddendSet** (uint32_t ui32Base, uint32_t ui32Increment)
- uint32_t **EMACTimestampConfigGet** (uint32_t ui32Base, uint32_t *pui32SubSecondInc)

- void [EMACTimestampConfigSet](#) (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32SubSecondInc)
- void [EMACTimestampDisable](#) (uint32_t ui32Base)
- void [EMACTimestampEnable](#) (uint32_t ui32Base)
- uint32_t [EMACTimestampIntStatus](#) (uint32_t ui32Base)
- void [EMACTimestampPPSCmd](#) (uint32_t ui32Base, uint8_t ui8Cmd)
- void [EMACTimestampPPSCommandModeSet](#) (uint32_t ui32Base, uint32_t ui32Config)
- void [EMACTimestampPPSPeriodSet](#) (uint32_t ui32Base, uint32_t ui32Period, uint32_t ui32Width)
- void [EMACTimestampPPSSimpleModeSet](#) (uint32_t ui32Base, uint32_t ui32FreqConfig)
- void [EMACTimestampSysTimeGet](#) (uint32_t ui32Base, uint32_t *pu32Seconds, uint32_t *pu32SubSeconds)
- void [EMACTimestampSysTimeSet](#) (uint32_t ui32Base, uint32_t ui32Seconds, uint32_t ui32SubSeconds)
- void [EMACTimestampSysTimeUpdate](#) (uint32_t ui32Base, uint32_t ui32Seconds, uint32_t ui32SubSeconds, bool blnc)
- void [EMACTimestampTargetIntDisable](#) (uint32_t ui32Base)
- void [EMACTimestampTargetIntEnable](#) (uint32_t ui32Base)
- void [EMACTimestampTargetSet](#) (uint32_t ui32Base, uint32_t ui32Seconds, uint32_t ui32SubSeconds)
- void [EMACTxDisable](#) (uint32_t ui32Base)
- uint8_t * [EMACTxDMACurrentBufferGet](#) (uint32_t ui32Base)
- tEMACDMADescriptor * [EMACTxDMACurrentDescriptorGet](#) (uint32_t ui32Base)
- tEMACDMADescriptor * [EMACTxDMADescriptorListGet](#) (uint32_t ui32Base)
- void [EMACTxDMADescriptorListSet](#) (uint32_t ui32Base, tEMACDMADescriptor *pDescriptor)
- void [EMACTxDMAPollDemand](#) (uint32_t ui32Base)
- void [EMACTxEnable](#) (uint32_t ui32Base)
- void [EMACTxFlush](#) (uint32_t ui32Base)
- uint32_t [EMACVLANHashFilterBitCalculate](#) (uint16_t ui16Tag)
- uint32_t [EMACVLANHashFilterGet](#) (uint32_t ui32Base)
- void [EMACVLANHashFilterSet](#) (uint32_t ui32Base, uint32_t ui32Hash)
- uint32_t [EMACVLANRxConfigGet](#) (uint32_t ui32Base, uint16_t *pu16Tag)
- void [EMACVLANRxConfigSet](#) (uint32_t ui32Base, uint16_t ui16Tag, uint32_t ui32Config)
- uint32_t [EMACVLANTxConfigGet](#) (uint32_t ui32Base, uint16_t *pu16Tag)
- void [EMACVLANTxConfigSet](#) (uint32_t ui32Base, uint16_t ui16Tag, uint32_t ui32Config)

10.2.1 Detailed Description

The Ethernet MAC driver API consists of 9 groups of functions:

Initialization and configuration of the MAC and PHY are controlled using [EMACInit\(\)](#), [EMACReset\(\)](#), [EMACPHYConfigSet\(\)](#), [EMACConfigSet\(\)](#), [EMACConfigGet\(\)](#), [EMACAddrSet\(\)](#), [EMACAddrGet\(\)](#) and [EMACNumAddrGet\(\)](#).

Packet filtering options are set and queried using [EMACFrameFilterSet\(\)](#), [EMACFrameFilterGet\(\)](#), [EMACHashFilterSet\(\)](#), [EMACHashFilterGet\(\)](#), [EMACHashFilterBitCalculate\(\)](#), [EMACAddrFilterSet\(\)](#) and [EMACAddrFilterGet\(\)](#).

Transmit and receive DMA descriptors are managed using [EMACTxDMAPollDemand\(\)](#), [EMACRxDMAPollDemand\(\)](#), [EMACRxDMADescriptorListSet\(\)](#), [EMACRxDMADescriptorListGet\(\)](#), [EMACRxDMACurrentDescriptorGet\(\)](#), [EMACRxDMACurrentBufferGet\(\)](#), [EMACTxDMADescriptorListSet\(\)](#), [EMACTxDMADescriptorListGet\(\)](#), [EMACTxDMACurrentDescriptorGet\(\)](#) and [EMACTxDMACurrentBufferGet\(\)](#).

Overall control of the transmitter and receiver are handled using [EMACRxWatchdogTimerSet\(\)](#), [EMACStatusGet\(\)](#), [EMACDMAStateGet\(\)](#), [EMACTxFlush\(\)](#), [EMACTxEnable\(\)](#), [EMACTxDisable\(\)](#), [EMACRxEnable\(\)](#) and [EMACRxDisable\(\)](#).

Interrupt management is controlled using [EMACIntEnable\(\)](#), [EMACIntDisable\(\)](#), [EMACIntStatus\(\)](#), [EMACIntClear\(\)](#), [EMACIntRegister\(\)](#) and [EMACIntUnregister\(\)](#).

The PHY, either internal or external, is controlled using [EMACPHYWrite\(\)](#), [EMACPHYExtendedWrite\(\)](#), [EMACPHYRead\(\)](#), [EMACPHYExtendedRead\(\)](#), [EMACPHYPowerOff\(\)](#) and [EMACPHYPowerOn\(\)](#).

IEEE1588, Precision Time Protocol timestamping, the integrated PTPD clock and the PPS output signal are controlled using [EMACTimestampConfigSet\(\)](#), [EMACTimestampConfigGet\(\)](#), [EMACTimestampAddendSet\(\)](#), [EMACTimestampEnable\(\)](#), [EMACTimestampDisable\(\)](#), [EMACTimestampSysTimeSet\(\)](#), [EMACTimestampSysTimeGet\(\)](#), [EMACTimestampSysTimeUpdate\(\)](#), [EMACTimestampTargetSet\(\)](#), [EMACTimestampTargetIntEnable\(\)](#), [EMACTimestampTargetIntDisable\(\)](#), [EMACTimestampIntStatus\(\)](#), [EMACTimestampPPSSimpleModeSet\(\)](#), [EMACTimestampPPSCommandModeSet\(\)](#), [EMACTimestampPPSCmd\(\)](#) and [EMACTimestampPPSPeriodSet\(\)](#).

Control of 802.1Q VLAN packet tagging is handled using [EMACVLANRxConfigSet\(\)](#), [EMACVLANRxConfigGet\(\)](#), [EMACVLANTxConfigSet\(\)](#), [EMACVLANTxConfigGet\(\)](#), [EMACVLANHashFilterBitCalculate\(\)](#), [EMACVLANHashFilterSet\(\)](#) and [EMACVLANHashFilterGet\(\)](#).

Handling of remote wakeup packets and power management options are controlled using [EMACRemoteWakeUpFrameFilterSet\(\)](#), [EMACRemoteWakeUpFrameFilterGet\(\)](#), [EMACPowerManagementControlSet\(\)](#), [EMACPowerManagementControlGet\(\)](#) and [EMACPowerManagementStatusGet\(\)](#).

10.2.2 Ethernet MAC Data Transfer

Data is transferred between system SRAM and the Ethernet MAC using independent transmit and receive DMA engines. Each engine is controlled using a list of descriptor structures stored in SRAM and containing frame data buffer pointers, control bits and status information. Two options exist for controlling the arrangement of the descriptor list. Descriptors may be arranged in a ring with a fixed spacing between the start of each descriptor and a control bit in the last descriptor to tell the hardware to return to the head of the list, or they may be configured as a linked list with a pointer in each descriptor directing the hardware to the next descriptor that is to be processed.

Although the hardware supports two distinct descriptor formats for both transmit and receive, a basic 4-word descriptor and an enhanced 8-word descriptor, the DriverLib EMAC driver includes type definitions and labels for only the enhanced descriptor format. Enhanced descriptors allow support for many commonly-used advanced features such as TCP/IP/UDP checksum insertion, VLAN tagging and frame timestamping so using this descriptor format throughout prevents complexity and confusion that could arise due to attempts to handle two somewhat-incompatible formats within the same code. Applications wishing to use the basic descriptor format may do so but must be careful not to use the various descriptor-related types and labels defined in emac.h because many of these will be incorrect for the shorter descriptor format.

The hardware moves through the descriptor lists sequentially until it discovers a descriptor marked

as owned by software at which point it stops and waits for the descriptor to be made available to it. Ownership of a given descriptor for both the transmit and receive cases is controlled by the most significant bit of the first descriptor word. When this bit is set, the hardware owns the descriptor and will read its content and use it to control transmission or reception of a frame. When clear, the software owns the descriptor and it is safe for the software to read or write the descriptor content without fear of treading on an ongoing hardware operation.

Management of transmit and receive descriptor lists is the responsibility of software above the EMAC API. While the API provides function calls to set the list start pointers, query the current descriptor and tell the hardware to start and stop reading the list, the actual descriptor contents must be handled above the EMAC layer, typically in the Ethernet interrupt handler which must track the current descriptor position in each ring and ensure that the correct descriptors are written for frame transmission or read for frame reception.

To transmit a frame, software must determine the next descriptor in the transmit list which is not currently owned by the hardware (has the **DES0_TX_CTRL_OWN** bit in the first descriptor word clear). A pointer to the frame to be transmitted is then written to the third word of the descriptor (**pvBuffer1**) and its length to the second word (**ui32Count**). If the descriptor list uses the ring structure rather than the linked list structure, a second buffer may be linked to the same descriptor using the fourth descriptor word (**DES3.pvBuffer2**) and bits [28:16] of the second word to store its size. Various flags controlling checksum insertion or replacement options, source address insertion or replacement and VLAN tagging are written into the first and second words of the descriptor (**ui32CtrlStatus** and **ui32Count**) before the **DES0_TX_CTRL_OWN** bit in the first word is set to hand the descriptor over to the hardware. If the transmit DMA was stopped waiting for the next descriptor to become available, a call to [EMACTxDMAPollDemand\(\)](#) will then ensure that the DMA restarts and transmits the new frame. Once transmission is completed, the hardware clears the **DES0_TX_CTRL_OWN** bit in the descriptor, returning it to the software and, optionally, raises an interrupt.

Similarly, to receive a frame, software must determine the next descriptor in the receive list which is not currently owned by the hardware (has the **DES0_RX_CTRL_OWN** bit in the first descriptor word clear). A pointer to an empty buffer into which data from the next received frame will be written must be written to the third word of the descriptor. The buffer size is written into the second word, taking care to preserve the **DES1_RX_CTRL_CHAINED** and **DES1_RX_CTRL_END_OF_RING** control bits also found there. Again, if the ring structure is used for the descriptor list, a second buffer may be attached to the descriptor using fields in words 3 and 2 to hold the pointer and size. The descriptor is then passed to the hardware by setting **DES0_RX_CTRL_OWN** in the first descriptor word. If the receiver had previously stopped due to a lack of available descriptors, a call to [EMACRxDMAPollDemand\(\)](#) will cause it to restart.

When a frame is received, the hardware will write its content into the next available receive buffer. If the buffer is smaller than the frame, reception continues in the next available buffer (either the second buffer attached to the current descriptor if a descriptor ring is in use or the first buffer attached to the next descriptor). Once the frame is completed, additional status is written into the receive descriptors to indicate the packet type, the buffer containing the start of the frame and the end of the frame, any errors detected and, optionally, IEEE1588 timestamps, before the **DES0_RX_CTRL_OWN** bits in affected descriptors are cleared and those descriptors become available to the software again.

10.2.3 Data Structure Documentation

10.2.3.1 tEMACDES3

Definition:

```
typedef union
{
    tEMACDMADescriptor *pLink;
    void *pvBuffer2;
}
tEMACDES3
```

Members:

pLink When DMA descriptors are used in chained mode, this field is used to provide a link to the next descriptor.

pvBuffer2 When the DMA descriptors are unchained, this field may be used to point to a second buffer containing data for transmission or providing storage for a received frame.

Description:

A union used to describe the two overlapping fields forming the third word of the Ethernet DMA descriptor.

10.2.3.2 tEMACDMADescriptor

Definition:

```
typedef struct
{
    uint32_t ui32CtrlStatus;
    uint32_t ui32Count;
    void *pvBuffer1;
    tEMACDES3 DES3;
    uint32_t ui32ExtRxStatus;
    uint32_t ui32Reserved;
    uint32_t ui32IEEE1588TimeLo;
    uint32_t ui32IEEE1588TimeHi;
}
tEMACDMADescriptor
```

Members:

ui32CtrlStatus The first DMA descriptor word contains various control and status bits depending upon whether the descriptor is in the transmit or receive queue. Bit 31 is always the “OWN” bit which, when set, indicates that the hardware has control of the descriptor.

ui32Count The second descriptor word contains information on the size of the buffer or buffers attached to the descriptor and various additional control bits.

pvBuffer1 The third descriptor word contains a pointer to the buffer containing data to transmit or into which received data should be written. This pointer must refer to a buffer in internal SRAM. Pointers to flash or EPI-connected memory may not be used and will result in the MAC reporting a bus error.

DES3 The fourth descriptor word contains either a pointer to the next descriptor in the ring or a pointer to a second data buffer. The meaning of the word is controlled by the “CHAINED” control bit which appears in the first word of the transmit descriptor or the second word of the receive descriptor.

- ui32ExtRxStatus*** The fifth descriptor word is reserved for transmit descriptors but used to report extended status in a receive descriptor.
- ui32Reserved*** The sixth descriptor word is reserved for both transmit and receive descriptors.
- ui32IEEE1588TimeLo*** The seventh descriptor word contains the low 32 bits of the 64-bit timestamp captured for transmitted or received data. The value is set only when the transmitted or received data contains the end of a packet. Availability of the timestamp is indicated via a status bit in the first descriptor word.
- ui32IEEE1588TimeHi*** The eighth descriptor word contains the high 32 bits of the 64-bit timestamp captured for transmitted or received data.

Description:

A structure defining a single Ethernet DMA buffer descriptor.

10.2.3.3 tEMACWakeUpFrameFilter

Definition:

```
typedef struct
{
    uint32_t pui32ByteMask[4];
    uint8_t pui8Command[4];
    uint8_t pui8Offset[4];
    uint16_t pui16CRC[4];
}
tEMACWakeUpFrameFilter
```

Members:

- pui32ByteMask*** A byte mask for each filter defining which bytes from a sequence of 31 (bit 31 must be clear in each mask) are used to filter incoming packets. A 1 indicates that the relevant byte is used to update the CRC16 for the filter, a 0 indicates that the byte is ignored.
- pui8Command*** Defines whether each filter is enabled and, if so, whether it filters multicast or unicast frames. Valid values are one of EMAC_RWU_FILTER_ENABLE or EMAC_RWU_FILTER_DISABLE ORed with one of EMAC_RWU_FILTER_UNICAST or EMAC_RWU_FILTER_MULTICAST.
- pui8Offset*** Determines the byte offset within the frame at which the filter starts examining bytes. The minimum value for each offset is 12. The first byte of a frame is offset 0.
- pui16CRC*** The CRC16 value that is expected for each filter if it passes. The CRC is calculated using all bytes indicated by the filter's mask.

Description:

This structure defines up to 4 filters that can be used to define specific frames which will cause the MAC to wake up from sleep mode.

10.2.4 Function Documentation

10.2.4.1 EMACAddrFilterGet

Gets filtering parameters associated with one of the configured MAC addresses.

Prototype:

```
uint32_t  
EMACAddrFilterGet (uint32_t ui32Base,  
                    uint32_t ui32Index)
```

Parameters:

ui32Base is the base address of the controller.

ui32Index is the index of the MAC address slot for which the filter is to be queried.

Description:

This function returns filtering parameters associated with one of the MAC address slots that the controller supports. This configuration is used when perfect filtering (rather than hash table filtering) is selected.

Valid values for *ui32Index* are from 1 to (number of MAC address slots - 1). The number of supported MAC address slots may be found by calling [EMACNumAddrGet\(\)](#). MAC index 0 is the local MAC address and does not have filtering parameters associated with it.

Returns:

Returns the filter configuration as the logical OR of the following labels:

- **EMAC_FILTER_ADDR_ENABLE** indicates that this MAC address is enabled and is used when performing perfect filtering. If this flag is absent, the MAC address at the given index is disabled and is not used in filtering.
- **EMAC_FILTER_SOURCE_ADDR** indicates that the MAC address at the given index is compared to the source address of incoming frames while performing perfect filtering. If absent, the MAC address is compared against the destination address.
- **EMAC_FILTER_MASK_BYTE_6** indicates that the MAC ignores the sixth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_5** indicates that the MAC ignores the fifth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_4** indicates that the MAC ignores the fourth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_3** indicates that the MAC ignores the third byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_2** indicates that the MAC ignores the second byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_1** indicates that the MAC ignores the first byte of the source or destination address when filtering.

10.2.4.2 EMACAddrFilterSet

Sets filtering parameters associated with one of the configured MAC addresses.

Prototype:

```
void  
EMACAddrFilterSet (uint32_t ui32Base,  
                   uint32_t ui32Index,  
                   uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the controller.

ui32Index is the index of the MAC address slot for which the filter is to be set.

ui32Config sets the filter parameters for the given MAC address.

Description:

This function sets filtering parameters associated with one of the MAC address slots that the controller supports. This configuration is used when perfect filtering (rather than hash table filtering) is selected.

Valid values for ***ui32Index*** are from 1 to (number of MAC address slots - 1). The number of supported MAC address slots may be found by calling [EMACNumAddrGet\(\)](#). MAC index 0 is the local MAC address and does not have filtering parameters associated with it.

The ***ui32Config*** parameter determines how the given MAC address is used when filtering incoming Ethernet frames. It is comprised of a logical OR of the fields:

- **EMAC_FILTER_ADDR_ENABLE** indicates that this MAC address is enabled and should be used when performing perfect filtering. If this flag is absent, the MAC address at the given index is disabled and is not used in filtering.
- **EMAC_FILTER_SOURCE_ADDR** indicates that the MAC address at the given index is compared to the source address of incoming frames while performing perfect filtering. If absent, the MAC address is compared against the destination address.
- **EMAC_FILTER_MASK_BYTE_6** indicates that the MAC should ignore the sixth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_5** indicates that the MAC should ignore the fifth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_4** indicates that the MAC should ignore the fourth byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_3** indicates that the MAC should ignore the third byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_2** indicates that the MAC should ignore the second byte of the source or destination address when filtering.
- **EMAC_FILTER_MASK_BYTE_1** indicates that the MAC should ignore the first byte of the source or destination address when filtering.

Returns:

None.

10.2.4.3 EMACAddrGet

Gets one of the MAC addresses stored in the Ethernet controller.

Prototype:

```
void
EMACAddrGet(uint32_t ui32Base,
            uint32_t ui32Index,
            uint8_t *pui8MACAddr)
```

Parameters:

ui32Base is the base address of the controller.

ui32Index is the zero-based index of the MAC address to return.

pui8MACAddr is the pointer to the location in which to store the array of MAC-48 address octets.

Description:

This function reads the currently programmed MAC address into the *pui8MACAddr* buffer. The *ui32Index* parameter defines which of the hardware's MAC addresses to return. The number of MAC addresses supported by the controller may be queried using a call to [EMACNumAddrGet\(\)](#). Index 0 refers to the MAC address of the local node. Other indices are used to define MAC addresses when filtering incoming packets.

The address is written to the *pui8MACAddr* array ordered with the first byte to be transmitted in the first array entry. For example, if the address is written in its usual form with the Organizationally Unique Identifier (OUI) shown first as:

AC-DE-48-00-00-80

the data is returned with 0xAC in the first byte of the array, 0xDE in the second, 0x48 in the third and so on.

Returns:

None.

10.2.4.4 EMACAddrSet

Sets the MAC address of the Ethernet controller.

Prototype:

```
void  
EMACAddrSet (uint32_t ui32Base,  
              uint32_t ui32Index,  
              const uint8_t *pui8MACAddr)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

ui32Index is the zero-based index of the MAC address to set.

pui8MACAddr is the pointer to the array of MAC-48 address octets.

Description:

This function programs the IEEE-defined MAC-48 address specified in *pui8MACAddr* into the Ethernet controller. This address is used by the Ethernet controller for hardware-level filtering of incoming Ethernet packets (when promiscuous mode is not enabled). Index 0 is used to hold the local node's MAC address which is inserted into all transmitted packets.

The controller may support several Ethernet MAC address slots, each of which may be programmed independently and used to filter incoming packets. The number of MAC addresses that the hardware supports may be queried using a call to [EMACNumAddrGet\(\)](#). The value of the *ui32Index* parameter must lie in the range from 0 to (number of MAC addresses - 1) inclusive.

The MAC-48 address is defined as 6 octets, illustrated by the following example address. The numbers are shown in hexadecimal format.

AC-DE-48-00-00-80

In this representation, the first three octets (AC-DE-48) are the Organizationally Unique Identifier (OUI). This is a number assigned by the IEEE to an organization that requests a block of

MAC addresses. The last three octets (00-00-80) are a 24-bit number managed by the OUI owner to uniquely identify a piece of hardware within that organization that is to be connected to the Ethernet.

In this representation, the octets are transmitted from left to right, with the “AC” octet being transmitted first and the “80” octet being transmitted last. Within an octet, the bits are transmitted LSB to MSB. For this address, the first bit to be transmitted would be “0”, the LSB of “AC”, and the last bit to be transmitted would be “1”, the MSB of “80”.

The address passed to this function in the *pui8MACAddr* array is ordered with the first byte to be transmitted in the first array entry. For example, the address given above could be represented using the following array:

```
uint8_t g_pui8MACAddr[] = { 0xAC, 0xDE, 0x48, 0x00, 0x00, 0x80 };
```

If the MAC address set by this function is currently enabled, it remains enabled following this call. Similarly, any filter configured for the MAC address remains unaffected by a change in the address.

Returns:

None.

10.2.4.5 EMACConfigGet

Returns the Ethernet MAC’s current basic configuration parameters.

Prototype:

```
void
EMACConfigGet(uint32_t ui32Base,
               uint32_t *pui32Config,
               uint32_t *pui32Mode,
               uint32_t *pui32RxMaxFrameSize)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

pui32Config points to storage that is written with Ethernet MAC configuration.

pui32Mode points to storage that is written with Ethernet MAC mode information.

pui32RxMaxFrameSize points to storage that is written with the maximum receive frame size.

Description:

This function is called to query the basic operating parameters for the MAC and its DMA engines.

The *pui32Config* parameter is written with the logical OR of various fields and flags. The first field describes which MAC address is used during insertion or replacement for all transmitted frames. Valid options are

- **EMAC_CONFIG_USE_MACADDR1**
- **EMAC_CONFIG_USE_MACADDR0**

The interframe gap between transmitted frames is given using one of the following values:

- **EMAC_CONFIG_IF_GAP_96BITS**
- **EMAC_CONFIG_IF_GAP_88BITS**

- **EMAC_CONFIG_IF_GAP_80BITS**
- **EMAC_CONFIG_IF_GAP_72BITS**
- **EMAC_CONFIG_IF_GAP_64BITS**
- **EMAC_CONFIG_IF_GAP_56BITS**
- **EMAC_CONFIG_IF_GAP_48BITS**
- **EMAC_CONFIG_IF_GAP_40BITS**

The number of bytes of preamble added to the beginning of every transmitted frame is described using one of the following values:

- **EMAC_CONFIG_7BYTE_PREAMBLE**
- **EMAC_CONFIG_5BYTE_PREAMBLE**
- **EMAC_CONFIG_3BYTE_PREAMBLE**

The back-off limit determines the range of the random time that the MAC delays after a collision and before attempting to retransmit a frame. One of the following values provides the currently selected limit. In each case the retransmission delay in terms of 512 bit time slots, is the lower of ($2^{**} N$) and a random number between 0 and the reported backoff-limit.

- **EMAC_CONFIG_BO_LIMIT_1024**
- **EMAC_CONFIG_BO_LIMIT_256**
- **EMAC_CONFIG_BO_LIMIT_16**
- **EMAC_CONFIG_BO_LIMIT_2**

Handling of insertion or replacement of the source address in all transmitted frames is described by one of the following fields:

- **EMAC_CONFIG_SA_INSERT** causes the MAC address (0 or 1 depending on whether **EMAC_CONFIG_USE_MACADDR0** or **EMAC_CONFIG_USE_MACADDR1** was specified) to be inserted into all transmitted frames.
- **EMAC_CONFIG_SA_REPLACE** causes the MAC address to be replaced with the selected address in all transmitted frames.
- **EMAC_CONFIG_SA_FROM_DESCRIPTOR** causes control of source address insertion or deletion to be controlled by fields in the DMA transmit descriptor, allowing control on a frame-by-frame basis.

Whether the interface attempts to operate in full- or half-duplex mode is reported by one of the following flags:

- **EMAC_CONFIG_FULL_DUPLEX**
- **EMAC_CONFIG_HALF_DUPLEX**

The following additional flags may also be included:

- **EMAC_CONFIG_2K_PACKETS** indicates that IEEE802.3as support for 2K packets is enabled. When present, the MAC considers all frames up to 2000 bytes in length as normal packets. When **EMAC_CONFIG_JUMBO_ENABLE** is not reported, all frames larger than 2000 bytes are treated as Giant frames. The value of this flag should be ignored if **EMAC_CONFIG_JUMBO_ENABLE** is also reported.
- **EMAC_CONFIG_STRIP_CRC** indicates that the 4-byte CRC of all Ethernet type frames is being stripped and dropped before the frame is forwarded to the application.
- **EMAC_CONFIG_JABBER_DISABLE** indicates that the the jabber timer on the transmitter is disabled, allowing frames of up to 16384 bytes to be transmitted. If this flag is absent, the MAC does not allow more than 2048 (or 10240 if **EMAC_CONFIG_JUMBO_ENABLE** is reported) bytes to be sent in any one frame.

- **EMAC_CONFIG_JUMBO_ENABLE** indicates that Jumbo Frames of up to 9018 (or 9022 if using VLAN tagging) are enabled.
- **EMAC_CONFIG_CS_DISABLE** indicates that Carrier Sense is disabled during transmission when operating in half-duplex mode.
- **EMAC_CONFIG_100MBPS** indicates that the MAC is using 100Mbps signaling to communicate with the PHY.
- **EMAC_CONFIG_RX_OWN_DISABLE** indicates that reception of transmitted frames is disabled when operating in half-duplex mode.
- **EMAC_CONFIG_LOOPBACK** indicates that internal loopback is enabled.
- **EMAC_CONFIG_CHECKSUM_OFFLOAD** indicates that IPv4 header checksum checking and IPv4 or IPv6 TCP, UDP or ICMP payload checksum checking is enabled. The results of the checksum calculations are reported via status fields in the DMA receive descriptors.
- **EMAC_CONFIG_RETRY_DISABLE** indicates that retransmission is disabled in cases where half-duplex mode is in use and a collision occurs. This condition causes the current frame to be ignored and a frame abort to be reported in the transmit frame status.
- **EMAC_CONFIG_AUTO_CRC_STRIPPING** indicates that the last 4 bytes (frame check sequence) from all Ether type frames are being stripped before frames are forwarded to the application.
- **EMAC_CONFIG_DEFERRAL_CHK_ENABLE** indicates that transmit deferral checking is disabled in half-duplex mode. When enabled, the transmitter reports an error if it is unable to transmit a frame for more than 24288 bit times (or 155680 bit times in Jumbo frame mode) due to an active carrier sense signal on the MII.
- **EMAC_CONFIG_TX_ENABLED** indicates that the MAC transmitter is currently enabled.
- **EMAC_CONFIG_RX_ENABLED** indicates that the MAC receiver is currently enabled.

The *pui32ModeFlags* parameter is written with operating parameters related to the internal MAC FIFOs. It comprises a logical OR of the following fields. The first field reports the transmit FIFO threshold. Transmission of a frame begins when this amount of data or a full frame exists in the transmit FIFO. This field should be ignored if **EMAC_MODE_TX_STORE_FORWARD** is also reported. One of the following values is reported:

- **EMAC_MODE_TX_THRESHOLD_16_BYTES**
- **EMAC_MODE_TX_THRESHOLD_24_BYTES**
- **EMAC_MODE_TX_THRESHOLD_32_BYTES**
- **EMAC_MODE_TX_THRESHOLD_40_BYTES**
- **EMAC_MODE_TX_THRESHOLD_64_BYTES**
- **EMAC_MODE_TX_THRESHOLD_128_BYTES**
- **EMAC_MODE_TX_THRESHOLD_192_BYTES**
- **EMAC_MODE_TX_THRESHOLD_256_BYTES**

The second field reports the receive FIFO threshold. DMA transfers of received data begin either when the receive FIFO contains a full frame or this number of bytes. This field should be ignored if **EMAC_MODE_RX_STORE_FORWARD** is included. One of the following values is reported:

- **EMAC_MODE_RX_THRESHOLD_64_BYTES**
- **EMAC_MODE_RX_THRESHOLD_32_BYTES**
- **EMAC_MODE_RX_THRESHOLD_96_BYTES**
- **EMAC_MODE_RX_THRESHOLD_128_BYTES**

The following additional flags may be included:

- **EMAC_MODE_KEEP_BAD_CRC** indicates that frames with TCP/IP checksum errors are being forwarded to the application if those frames do not have any errors (including FCS errors) in the Ethernet framing. In these cases, the frames have errors only in the payload. If this flag is not reported, all frames with any detected error are discarded unless **EMAC_MODE_RX_ERROR_FRAMES** is also reported.
- **EMAC_MODE_RX_STORE_FORWARD** indicates that the receive DMA is configured to read frames from the FIFO only after the complete frame has been written to it. If this mode is enabled, the receive threshold is ignored.
- **EMAC_MODE_RX_FLUSH_DISABLE** indicates that the flushing of received frames is disabled in cases where receive descriptors or buffers are unavailable.
- **EMAC_MODE_TX_STORE_FORWARD** indicates that the transmitter is configured to transmit a frame only after the whole frame has been written to the transmit FIFO. If this mode is enabled, the transmit threshold is ignored.
- **EMAC_MODE_RX_ERROR_FRAMES** indicates that all frames other than runt error frames are being forwarded to the receive DMA regardless of any errors detected in the frames.
- **EMAC_MODE_RX_UNDERSIZED_FRAMES** indicates that undersized frames (frames shorter than 64 bytes but with no errors) are being forwarded to the application. If this option is not reported, all undersized frames are dropped by the receiver unless it has already started transferring them to the receive FIFO due to the receive threshold setting.
- **EMAC_MODE_OPERATE_2ND_FRAME** indicates that the transmit DMA is configured to operate on a second frame while waiting for the previous frame to be transmitted and associated status and timestamps to be reported. If absent, the transmit DMA works on a single frame at any one time, waiting for that frame to be transmitted and its status to be received before moving on to the next frame.
- **EMAC_MODE_TX_DMA_ENABLED** indicates that the transmit DMA engine is currently enabled.
- **EMAC_MODE_RX_DMA_ENABLED** indicates that the receive DMA engine is currently enabled.

The *ui32RxMaxFrameSize* is written with the currently configured maximum receive packet size. Packets larger than this are flagged as being in error.

Returns:

None.

10.2.4.6 EMACConfigSet

Configures basic Ethernet MAC operation parameters.

Prototype:

```
void  
EMACConfigSet (uint32_t ui32Base,  
                uint32_t ui32Config,  
                uint32_t ui32ModeFlags,  
                uint32_t ui32RxMaxFrameSize)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

ui32Config provides various flags and values configuring the MAC.

ui32ModeFlags provides configuration relating to the transmit and receive DMA engines.
ui32RxMaxFrameSize sets the maximum receive frame size above which an error is reported.

Description:

This function is called to configure basic operating parameters for the MAC and its DMA engines.

The ***ui32Config*** parameter is the logical OR of various fields and flags. The first field determines which MAC address is used during insertion or replacement for all transmitted frames. Valid options are

- **EMAC_CONFIG_USE_MACADDR1** and
- **EMAC_CONFIG_USE_MACADDR0**

The interframe gap between transmitted frames is controlled using one of the following values:

- **EMAC_CONFIG_IF_GAP_96BITS**
- **EMAC_CONFIG_IF_GAP_88BITS**
- **EMAC_CONFIG_IF_GAP_80BITS**
- **EMAC_CONFIG_IF_GAP_72BITS**
- **EMAC_CONFIG_IF_GAP_64BITS**
- **EMAC_CONFIG_IF_GAP_56BITS**
- **EMAC_CONFIG_IF_GAP_48BITS**
- **EMAC_CONFIG_IF_GAP_40BITS**

The number of bytes of preamble added to the beginning of every transmitted frame is selected using one of the following values:

- **EMAC_CONFIG_7BYTE_PREAMBLE**
- **EMAC_CONFIG_5BYTE_PREAMBLE**
- **EMAC_CONFIG_3BYTE_PREAMBLE**

The back-off limit determines the range of the random time that the MAC delays after a collision and before attempting to retransmit a frame. One of the following values must be used to select this limit. In each case, the retransmission delay in terms of 512 bit time slots, is the lower of $(2^{**} N)$ and a random number between 0 and the selected backoff-limit.

- **EMAC_CONFIG_BO_LIMIT_1024**
- **EMAC_CONFIG_BO_LIMIT_256**
- **EMAC_CONFIG_BO_LIMIT_16**
- **EMAC_CONFIG_BO_LIMIT_2**

Control over insertion or replacement of the source address in all transmitted frames is provided by using one of the following fields:

- **EMAC_CONFIG_SA_INSERT** causes the MAC address (0 or 1 depending on whether **EMAC_CONFIG_USE_MACADDR0** or **EMAC_CONFIG_USE_MACADDR1** was specified) to be inserted into all transmitted frames.
- **EMAC_CONFIG_SA_REPLACE** causes the MAC address to be replaced with the selected address in all transmitted frames.
- **EMAC_CONFIG_SA_FROM_DESCRIPTOR** causes control of source address insertion or deletion to be controlled by fields in the DMA transmit descriptor, allowing control on a frame-by-frame basis.

Whether the interface attempts to operate in full- or half-duplex mode is controlled by one of the following flags:

- **EMAC_CONFIG_FULL_DUPLEX**
- **EMAC_CONFIG_HALF_DUPLEX**

The following additional flags may also be specified:

- **EMAC_CONFIG_2K_PACKETS** enables IEEE802.3as support for 2K packets. When specified, the MAC considers all frames up to 2000 bytes in length as normal packets. When **EMAC_CONFIG_JUMBO_ENABLE** is not specified, all frames larger than 2000 bytes are treated as Giant frames. This flag is ignored if **EMAC_CONFIG_JUMBO_ENABLE** is specified.
- **EMAC_CONFIG_STRIP_CRC** causes the 4-byte CRC of all Ethernet type frames to be stripped and dropped before the frame is forwarded to the application.
- **EMAC_CONFIG_JABBER_DISABLE** disables the jabber timer on the transmitter and enables frames of up to 16384 bytes to be transmitted. If this flag is absent, the MAC does not allow more than 2048 (or 10240 if **EMAC_CONFIG_JUMBO_ENABLE** is specified) bytes to be sent in any one frame.
- **EMAC_CONFIG_JUMBO_ENABLE** enables Jumbo Frames, allowing frames of up to 9018 (or 9022 if using VLAN tagging) to be handled without reporting giant frame errors.
- **EMAC_CONFIG_100MBPS** forces the MAC to communicate with the PHY using 100Mbps signaling. If this option is not specified, the MAC uses 10Mbps signaling. This speed setting is important when using an external RMII PHY where the selected rate must match the PHY's setting which may have been made as a result of auto-negotiation. When using the internal PHY or an external MII PHY, the signaling rate is controlled by the PHY- provided transmit and receive clocks.
- **EMAC_CONFIG_CS_DISABLE** disables Carrier Sense during transmission when operating in half-duplex mode.
- **EMAC_CONFIG_RX_OWN_DISABLE** disables reception of transmitted frames when operating in half-duplex mode.
- **EMAC_CONFIG_LOOPBACK** enables internal loopback.
- **EMAC_CONFIG_CHECKSUM_OFFLOAD** enables IPv4 header checksum checking and IPv4 or IPv6 TCP, UDP or ICMP payload checksum checking. The results of the checksum calculations are reported via status fields in the DMA receive descriptors.
- **EMAC_CONFIG_RETRY_DISABLE** disables retransmission in cases where half-duplex mode is in use and a collision occurs. This condition causes the current frame to be ignored and a frame abort to be reported in the transmit frame status.
- **EMAC_CONFIG_AUTO_CRC_STRIPPING** strips the last 4 bytes (frame check sequence) from all Ether type frames before forwarding the frames to the application.
- **EMAC_CONFIG_DEFERRAL_CHK_ENABLE** enables transmit deferral checking in half-duplex mode. When enabled, the transmitter reports an error if it is unable to transmit a frame for more than 24288 bit times (or 155680 bit times in Jumbo frame mode) due to an active carrier sense signal on the MII.

The *ui32ModeFlags* parameter sets operating parameters related to the internal MAC FIFOs. It comprises a logical OR of the following fields. The first selects the transmit FIFO threshold. Transmission of a frame begins when this amount of data or a full frame exists in the transmit FIFO. This field is ignored if **EMAC_MODE_TX_STORE_FORWARD** is included. One of the following must be specified:

- **EMAC_MODE_TX_THRESHOLD_16_BYTES**

- **EMAC_MODE_TX_THRESHOLD_24_BYTES**
- **EMAC_MODE_TX_THRESHOLD_32_BYTES**
- **EMAC_MODE_TX_THRESHOLD_40_BYTES**
- **EMAC_MODE_TX_THRESHOLD_64_BYTES**
- **EMAC_MODE_TX_THRESHOLD_128_BYTES**
- **EMAC_MODE_TX_THRESHOLD_192_BYTES**
- **EMAC_MODE_TX_THRESHOLD_256_BYTES**

The second field controls the receive FIFO threshold. DMA transfers of received data begin either when the receive FIFO contains a full frame or this number of bytes. This field is ignored if **EMAC_MODE_RX_STORE_FORWARD** is included. One of the following must be specified:

- **EMAC_MODE_RX_THRESHOLD_64_BYTES**
- **EMAC_MODE_RX_THRESHOLD_32_BYTES**
- **EMAC_MODE_RX_THRESHOLD_96_BYTES**
- **EMAC_MODE_RX_THRESHOLD_128_BYTES**

The following additional flags may be specified:

- **EMAC_MODE_KEEP_BAD_CRC** causes frames with TCP/IP checksum errors to be forwarded to the application if those frames do not have any errors (including FCS errors) in the Ethernet framing. In these cases, the frames have errors only in the payload. If this flag is not specified, all frames with any detected error are discarded unless **EMAC_MODE_RX_ERROR_FRAMES** is also specified.
- **EMAC_MODE_RX_STORE_FORWARD** causes the receive DMA to read frames from the FIFO only after the complete frame has been written to it. If this mode is enabled, the receive threshold is ignored.
- **EMAC_MODE_RX_FLUSH_DISABLE** disables the flushing of received frames in cases where receive descriptors or buffers are unavailable.
- **EMAC_MODE_TX_STORE_FORWARD** causes the transmitter to start transmitting a frame only after the whole frame has been written to the transmit FIFO. If this mode is enabled, the transmit threshold is ignored.
- **EMAC_MODE_RX_ERROR_FRAMES** causes all frames other than runt error frames to be forwarded to the receive DMA regardless of any errors detected in the frames.
- **EMAC_MODE_RX_UNDERSIZED_FRAMES** causes undersized frames (frames shorter than 64 bytes but with no errors) to the application. If this option is not selected, all undersized frames are dropped by the receiver unless it has already started transferring them to the receive FIFO due to the receive threshold setting.
- **EMAC_MODE_OPERATE_2ND_FRAME** enables the transmit DMA to operate on a second frame while waiting for the previous frame to be transmitted and associated status and timestamps to be reported. If absent, the transmit DMA works on a single frame at any one time, waiting for that frame to be transmitted and its status to be received before moving on to the next frame.

The *ui32RxMaxFrameSize* parameter may be used to override the default setting for the maximum number of bytes that can be received in a frame before that frame is flagged as being in error. If the parameter is set to 0, the default hardware settings are applied. If non-zero, any frame received which is longer than the *ui32RxMaxFrameSize*, regardless of whether the MAC is configured for normal or Jumbo frame operation, is flagged as an error.

Returns:

None.

10.2.4.7 EMACDMAStateGet

Returns the current states of the Ethernet MAC transmit and receive DMA engines.

Prototype:

```
uint32_t  
EMACDMAStateGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be used to query the current states of the transmit and receive DMA engines. The return value contains two fields, one providing the transmit state and the other the receive state. Macros **EMAC_TX_DMA_STATE()** and **EMAC_RX_DMA_STATE()** may be used to extract these fields from the returned value. Alternatively, masks **EMAC_DMA_TXSTAT_MASK** and **EMAC_DMA_RXSTAT_MASK** may be used directly to mask out the individual states from the returned value.

Returns:

Returns the states of the transmit and receive DMA engines. These states are ORed together into a single word containing one of:

- **EMAC_DMA_TXSTAT_STOPPED** indicating that the transmit engine is stopped.
- **EMAC_DMA_TXSTAT_RUN_FETCH_DESC** indicating that the transmit engine is fetching the next descriptor.
- **EMAC_DMA_TXSTAT_RUN_WAIT_STATUS** indicating that the transmit engine is waiting for status from the MAC.
- **EMAC_DMA_TXSTAT_RUN_READING** indicating that the transmit engine is currently transferring data from memory to the MAC transmit FIFO.
- **EMAC_DMA_TXSTAT_RUN_CLOSE_DESC** indicating that the transmit engine is closing the descriptor after transmission of the buffer data.
- **EMAC_DMA_TXSTAT_TS_WRITE** indicating that the transmit engine is currently writing timestamp information to the descriptor.
- **EMAC_DMA_TXSTAT_SUSPENDED** indicating that the transmit engine is suspended due to the next descriptor being unavailable (owned by the host) or a transmit buffer underflow.

and one of:

- **EMAC_DMA_RXSTAT_STOPPED** indicating that the receive engine is stopped.
- **EMAC_DMA_RXSTAT_RUN_FETCH_DESC** indicating that the receive engine is fetching the next descriptor.
- **EMAC_DMA_RXSTAT_RUN_WAIT_PACKET** indicating that the receive engine is waiting for the next packet.
- **EMAC_DMA_RXSTAT_SUSPENDED** indicating that the receive engine is suspended due to the next descriptor being unavailable.
- **EMAC_DMA_RXSTAT_RUN_CLOSE_DESC** indicating that the receive engine is closing the descriptor after receiving a buffer of data.
- **EMAC_DMA_RXSTAT_TS_WRITE** indicating that the transmit engine is currently writing timestamp information to the descriptor.

- **EMAC_DMA_RXSTAT_RUN RECEIVING** indicating that the receive engine is currently transferring data from the MAC receive FIFO to memory.

Additionally, a DMA bus error may be signaled using **EMAC_DMA_ERROR**. If this flag is present, the source of the error is identified using one of the following values which may be extracted from the return value using **EMAC_DMA_ERR_MASK**:

- **EMAC_DMA_ERR_RX_DATA_WRITE** indicates that an error occurred when writing received data to memory.
- **EMAC_DMA_ERR_TX_DATA_READ** indicates that an error occurred when reading data from memory for transmission.
- **EMAC_DMA_ERR_RX_DESC_WRITE** indicates that an error occurred when writing to the receive descriptor.
- **EMAC_DMA_ERR_TX_DESC_WRITE** indicates that an error occurred when writing to the transmit descriptor.
- **EMAC_DMA_ERR_RX_DESC_READ** indicates that an error occurred when reading the receive descriptor.
- **EMAC_DMA_ERR_TX_DESC_READ** indicates that an error occurred when reading the transmit descriptor.

10.2.4.8 EMACFrameFilterGet

Returns the current Ethernet frame filtering settings.

Prototype:

```
uint32_t
EMACFrameFilterGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be called to retrieve the frame filtering configuration set using a prior call to [EMACFrameFilterSet\(\)](#).

Returns:

Returns a value comprising the logical OR of various flags indicating the frame filtering options in use. Possible flags are:

- **EMAC_FRMFILTER_RX_ALL** indicates that the MAC is configured to pass all received frames regardless of whether or not they pass any address filter that is configured. The receive status word in the relevant DMA descriptor is updated to indicate whether the configured filter passed or failed for the frame.
- **EMAC_FRMFILTER_VLAN** indicates that the MAC is configured to drop any frames which do not pass the VLAN tag comparison.
- **EMAC_FRMFILTER_HASH_AND_PERFECT** indicates that the MAC is configured to pass frames if they match either the hash filter or the perfect filter. If this flag is absent, frames passing based on the result of a single filter, the perfect filter if **EMAC_FRMFILTER_HASH_MULTICAST** or **EMAC_FRMFILTER_HASH_UNICAST** are clear or the hash filter otherwise.

- **EMAC_FRMFILTER_SADDR** indicates that the MAC is configured to drop received frames when the source address field in the frame does not match the values programmed into the enabled SA registers.
- **EMAC_FRMFILTER_INV_SADDR** enables inverse source address filtering. When this option is specified, frames for which the SA does not match the SA registers are marked as passing the source address filter.
- **EMAC_FRMFILTER_BROADCAST** indicates that the MAC is configured to discard all incoming broadcast frames.
- **EMAC_FRMFILTER_PASS_MULTICAST** indicates that the MAC is configured to pass all incoming frames with multicast destinations addresses.
- **EMAC_FRMFILTER_INV_DADDR** indicates that the sense of the destination address filtering for both unicast and multicast frames is inverted.
- **EMAC_FRMFILTER_HASH_MULTICAST** indicates that destination address filtering of received multicast frames is enabled using the hash table. If absent, perfect destination address filtering is used. If used in conjunction with **EMAC_FRMFILTER_HASH_AND_PERFECT**, this flag indicates that the hash filter should be used for incoming multicast packets along with the perfect filter.
- **EMAC_FRMFILTER_HASH_UNICAST** indicates that destination address filtering of received unicast frames is enabled using the hash table. If absent, perfect destination address filtering is used. If used in conjunction with **EMAC_FRMFILTER_HASH_AND_PERFECT**, this flag indicates that the hash filter should be used for incoming unicast packets along with the perfect filter.
- **EMAC_FRMFILTER_PROMISCUOUS** indicates that the MAC is configured to operate in promiscuous mode where all received frames are passed to the application and the SA and DA filter status bits of the descriptor receive status word are always cleared.

Control frame filtering configuration is indicated by one of the following values which may be extracted from the returned value using the mask **EMAC_FRMFILTER_PASS_MASK**:

- **EMAC_FRMFILTER_PASS_NO_CTRL** prevents any control frame from reaching the application.
- **EMAC_FRMFILTER_PASS_NO_PAUSE** passes all control frames other than PAUSE even if they fail the configured address filter.
- **EMAC_FRMFILTER_PASS_ALL_CTRL** passes all control frames, including PAUSE even if they fail the configured address filter.
- **EMAC_FRMFILTER_PASS_ADDR_CTRL** passes all control frames only if they pass the configured address filter.

10.2.4.9 EMACFrameFilterSet

Sets options related to Ethernet frame filtering.

Prototype:

```
void  
EMACFrameFilterSet (uint32_t ui32Base,  
                    uint32_t ui32FilterOpts)
```

Parameters:

ui32Base is the base address of the controller.

ui32FilterOpts is a logical OR of flags defining the required MAC address filtering options.

Description:

This function allows various filtering options to be defined and allows an application to control which frames are received based on various criteria related to the frame source and destination MAC addresses or VLAN tagging.

The *ui32FilterOpts* parameter is a logical OR of any of the following flags:

- **EMAC_FRMFILTER_RX_ALL** configures the MAC to pass all received frames regardless of whether or not they pass any address filter that is configured. The receive status word in the relevant DMA descriptor is updated to indicate whether the configured filter passed or failed for the frame.
- **EMAC_FRMFILTER_VLAN** configures the MAC to drop any frames that do not pass the VLAN tag comparison.
- **EMAC_FRMFILTER_HASH_AND_PERFECT** configures the MAC to filter frames based on both any perfect filters set and the hash filter if enabled using **EMAC_FRMFILTER_HASH_UNICAST** or **EMAC_FRMFILTER_HASH_MULTICAST**. In this case, only if both filters fail is the packet rejected. If this option is absent, only one of the filter types is used, as controlled by **EMAC_FRMFILTER_HASH_UNICAST** and **EMAC_FRMFILTER_HASH_MULTICAST** for unicast and multicast frames respectively.
- **EMAC_FRMFILTER_SADDR** configures the MAC to drop received frames when the source address field in the frame does not match the values programmed into the enabled SA registers.
- **EMAC_FRMFILTER_INV_SADDR** enables inverse source address filtering. When this option is specified, frames for which the SA does not match the SA registers are marked as passing the source address filter.
- **EMAC_FRMFILTER_BROADCAST** configures the MAC to discard all incoming broadcast frames.
- **EMAC_FRMFILTER_PASS_MULTICAST** configures the MAC to pass all incoming frames with multicast destinations addresses.
- **EMAC_FRMFILTER_INV_DADDR** inverts the sense of the destination address filtering for both unicast and multicast frames.
- **EMAC_FRMFILTER_HASH_MULTICAST** enables destination address filtering of received multicast frames using the hash table. If absent, perfect destination address filtering is used. If used in conjunction with **EMAC_FRMFILTER_HASH_AND_PERFECT**, this flag indicates that the hash filter should be used for incoming multicast packets along with the perfect filter.
- **EMAC_FRMFILTER_HASH_UNICAST** enables destination address filtering of received unicast frames using the hash table. If absent, perfect destination address filtering is used. If used in conjunction with **EMAC_FRMFILTER_HASH_AND_PERFECT**, this flag indicates that the hash filter should be used for incoming unicast packets along with the perfect filter.
- **EMAC_FRMFILTER_PROMISCUOUS** configures the MAC to operate in promiscuous mode where all received frames are passed to the application and the SA and DA filter status bits of the descriptor receive status word are always cleared.

Control frame filtering may be configured by ORing one of the following values into *ui32FilterOpts*:

- **EMAC_FRMFILTER_PASS_NO_CTRL** prevents any control frame from reaching the application.

- **EMAC_FRMFILTER_PASS_NO_PAUSE** passes all control frames other than PAUSE even if they fail the configured address filter.
- **EMAC_FRMFILTER_PASS_ALL_CTRL** passes all control frames, including PAUSE even if they fail the configured address filter.
- **EMAC_FRMFILTER_PASS_ADDR_CTRL** passes all control frames only if they pass the configured address filter.

Returns:

None.

10.2.4.10 EMACHashFilterBitCalculate

Returns the bit number to set in the MAC hash filter corresponding to a given MAC address.

Prototype:

```
uint32_t  
EMACHashFilterBitCalculate(uint8_t *pui8MACAddr)
```

Parameters:

pui8MACAddr points to a buffer containing the 6-byte MAC address for which the hash filter bit is to be determined.

Description:

This function may be used to determine which bit in the MAC address hash filter to set to describe a given 6-byte MAC address. The returned value is a 6-bit number where bit 5 indicates which of the two hash table words is affected and the bottom 5 bits indicate the bit number to set within that word. For example, if 0x22 (100010b) is returned, this indicates that bit 2 of word 1 (*ui32HashHi* as passed to [EMACHashFilterSet\(\)](#)) must be set to describe the passed MAC address.

Returns:

Returns the bit number to set in the MAC hash table to describe the passed MAC address.

10.2.4.11 EMACHashFilterGet

Returns the current MAC address hash filter table.

Prototype:

```
void  
EMACHashFilterGet(uint32_t ui32Base,  
                  uint32_t *pui32HashHi,  
                  uint32_t *pui32HashLo)
```

Parameters:

ui32Base is the base address of the controller.

pui32HashHi points to storage to be written with the upper 32 bits of the current 64-bit hash filter table.

pui32HashLo points to storage to be written with the lower 32 bits of the current 64-bit hash filter table.

Description:

This function may be used to retrieve the current 64-bit hash filter table from the MAC prior to making changes and setting the new hash filter via a call to [EMACHashFilterSet\(\)](#).

Hash table filtering allows many different MAC addresses to be filtered simultaneously at the cost of some false-positive results in the form of packets passing the filter when their MAC address was not one of those required. A CRC of the packet source or destination MAC address is calculated and the bottom 6 bits are used as a bit index into the 64-bit hash filter table. If the bit in the hash table is set, the filter is considered to have passed. If the bit is clear, the filter fails and the packet is rejected (assuming normal rather than inverse filtering is configured).

Returns:

None.

10.2.4.12 EMACHashFilterSet

Sets the MAC address hash filter table.

Prototype:

```
void  
EMACHashFilterSet (uint32_t ui32Base,  
                    uint32_t ui32HashHi,  
                    uint32_t ui32HashLo)
```

Parameters:

ui32Base is the base address of the controller.

ui32HashHi is the upper 32 bits of the current 64-bit hash filter table to set.

ui32HashLo is the lower 32 bits of the current 64-bit hash filter table to set.

Description:

This function may be used to set the current 64-bit hash filter table used by the MAC to filter incoming packets when hash filtering is enabled. Hash filtering is enabled by passing **EMAC_FRMFILTER_HASH_UNICAST** and/or **EMAC_FRMFILTER_HASH_MULTICAST** in the *ui32FilterOpts* parameter to [EMACFrameFilterSet\(\)](#). The current hash filter may be retrieved by calling [EMACHashFilterGet\(\)](#).

Hash table filtering allows many different MAC addresses to be filtered simultaneously at the cost of some false-positive results (in the form of packets passing the filter when their MAC address was not one of those required). A CRC of the packet source or destination MAC address is calculated and the bottom 6 bits are used as a bit index into the 64-bit hash filter table. If the bit in the hash table is set, the filter is considered to have passed. If the bit is clear, the filter fails and the packet is rejected (assuming normal rather than inverse filtering is configured).

Returns:

None.

10.2.4.13 EMACInit

Initializes the Ethernet MAC and sets bus-related DMA parameters.

Prototype:

```
void  
EMACInit (uint32_t ui32Base,  
           uint32_t ui32SysClk,  
           uint32_t ui32BusConfig,  
           uint32_t ui32RxBurst,  
           uint32_t ui32TxBurst,  
           uint32_t ui32DescSkipSize)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

ui32SysClk is the current system clock frequency in Hertz.

ui32BusConfig defines the bus operating mode for the Ethernet MAC DMA controller.

ui32RxBurst is the maximum receive burst size in words.

ui32TxBurst is the maximum transmit burst size in words.

ui32DescSkipSize is the number of 32-bit words to skip between two unchained DMA descriptors. Values in the range 0 to 31 are valid.

Description:

This function sets bus-related parameters for the Ethernet MAC DMA engines. It must be called after [EMACPHYConfigSet\(\)](#) and called again after any subsequent call to [EMACPHYConfigSet\(\)](#).

The **ui32BusConfig** parameter is the logical OR of various fields. The first sets the DMA channel priority weight:

- **EMAC_BCONFIG_DMA_PRIO_WEIGHT_1**
- **EMAC_BCONFIG_DMA_PRIO_WEIGHT_2**
- **EMAC_BCONFIG_DMA_PRIO_WEIGHT_3**
- **EMAC_BCONFIG_DMA_PRIO_WEIGHT_4**

The second field sets the receive and transmit priorities used when arbitrating between the Rx and Tx DMA. The priorities are Rx:Tx unless **EMAC_BCONFIG_TX_PRIORITY** is also specified, in which case they become Tx:Rx. The priority provided here is ignored if **EMAC_BCONFIG_PRIORITY_FIXED** is specified.

- **EMAC_BCONFIG_PRIORITY_1_1**
- **EMAC_BCONFIG_PRIORITY_2_1**
- **EMAC_BCONFIG_PRIORITY_3_1**
- **EMAC_BCONFIG_PRIORITY_4_1**

The following additional flags may also be defined:

- **EMAC_BCONFIG_TX_PRIORITY** indicates that the transmit DMA should be higher priority in all arbitration for the system-side bus. If this is not defined, the receive DMA has higher priority.
- **EMAC_BCONFIG_ADDR_ALIGNED** works in tandem with **EMAC_BCONFIG_FIXED_BURST** to control address alignment of AHB bursts. When both flags are specified, all bursts are aligned to the start address least significant bits. If **EMAC_BCONFIG_FIXED_BURST** is not specified, the first burst is unaligned but subsequent bursts are aligned to the address.
- **EMAC_BCONFIG_ALT_DESCRIPTOR** indicates that the DMA engine should use the alternate descriptor format as defined in type [tEMACDMADescriptor](#). If absent, the basic descriptor type is used. Alternate descriptors are required if using IEEE 1588-2008

advanced timestamping, VLAN or TCP/UDP/ICMP CRC insertion features. Note that, for clarity, emac.h does not contain type definitions for the basic descriptor type. Please see the part datasheet for information on basic descriptor structures.

- **EMAC_BCONFIG_PRIORITY_FIXED** indicates that a fixed priority scheme should be employed when arbitrating between the transmit and receive DMA for system-side bus access. In this case, the receive channel always has priority unless **EMAC_BCONFIG_TX_PRIORITY** is set, in which case the transmit channel has priority. If **EMAC_BCONFIG_PRIORITY_FIXED** is not specified, a weighted round-robin arbitration scheme is used with the weighting defined using **EMAC_BCONFIG_PRIORITY_1_1**, **EMAC_BCONFIG_PRIORITY_2_1**, **EMAC_BCONFIG_PRIORITY_3_1** or **EMAC_BCONFIG_PRIORITY_4_1**, and **EMAC_BCONFIG_TX_PRIORITY**.
- **EMAC_BCONFIG_FIXED_BURST** indicates that fixed burst transfers should be used.
- **EMAC_BCONFIG_MIXED_BURST** indicates that the DMA engine should use mixed burst types depending on the length of data to be transferred across the system bus.

The *ui32RxBurst* and *ui32TxBurst* parameters indicate the maximum number of words that the relevant DMA should transfer in a single transaction. Valid values are 1, 2, 4, 8, 16 and 32. Any other value results in undefined behavior.

The *ui32DescSkipSize* parameter is used when the descriptor lists are using ring mode (where descriptors are contiguous in memory with the last descriptor marked with the **END_OF_RING** flag) rather than chained mode (where each descriptor includes a field that points to the next descriptor in the list). In ring mode, the hardware uses the *ui32DescSkipSize* to skip past any application-defined fields after the end of the hardware-defined descriptor fields. The parameter value indicates the number of 32-bit words to skip after the last field of the hardware-defined descriptor to get to the first field of the next descriptor. When using arrays of either the **tEMACDMADescriptor** or **tEMACAltDMADescriptor** types defined for this driver, *ui32DescSkipSize* must be set to 1 to skip the *pvNext* pointer added to the end of each of these structures. Applications may modify these structure definitions to include their own application-specific data and modify *ui32DescSkipSize* appropriately if desired.

Returns:

None.

10.2.4.14 EMACIntClear

Clears individual Ethernet MAC interrupt sources.

Prototype:

```
void
EMACIntClear(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the Ethernet MAC.

ui32IntFlags is the bit mask of the interrupt sources to be cleared.

Description:

This function disables the indicated Ethernet MAC interrupt sources.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **EMAC_INT_PHY** indicates that the PHY has signaled a change of state. Software must read and write the appropriate PHY registers to enable, disable and clear particular notifications.
- **EMAC_INT_EARLY_RECEIVE** indicates that the DMA engine has filled the first data buffer of a packet.
- **EMAC_INT_BUS_ERROR** indicates that a fatal bus error has occurred and that the DMA engine has been disabled.
- **EMAC_INT_EARLY_TRANSMIT** indicates that a frame to be transmitted has been fully written from memory into the MAC transmit FIFO.
- **EMAC_INT_RX_WATCHDOG** indicates that a frame with length greater than 2048 bytes (of 10240 bytes in Jumbo Frame mode) was received.
- **EMAC_INT_RX_STOPPED** indicates that the receive process has entered the stopped state.
- **EMAC_INT_RX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's receive descriptor list and the DMA cannot, therefore, acquire a buffer. The receive process is suspended and can be resumed by changing the descriptor ownership and calling [EMACRxDMAPollDemand\(\)](#).
- **EMAC_INT_RECEIVE** indicates that reception of a frame has completed and all requested status has been written to the appropriate DMA receive descriptor.
- **EMAC_INT_TX_UNDERFLOW** indicates that the transmitter experienced an underflow during transmission. The transmit process is suspended.
- **EMAC_INT_RX_OVERFLOW** indicates that an overflow was experienced during reception.
- **EMAC_INT_TX_JABBER** indicates that the transmit jabber timer expired. This condition occurs when the frame size exceeds 2048 bytes (or 10240 bytes in Jumbo Frame mode) and causes the transmit process to abort and enter the Stopped state.
- **EMAC_INT_TX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's transmit descriptor list and that the DMA cannot, therefore, acquire a buffer. Transmission is suspended and can be resumed by changing the descriptor ownership and calling [EMACTxDMAPollDemand\(\)](#).
- **EMAC_INT_TX_STOPPED** indicates that the transmit process has stopped.
- **EMAC_INT_TRANSMIT** indicates that transmission of a frame has completed and that all requested status has been updated in the descriptor.

Summary interrupt bits **EMAC_INT_NORMAL_INT** and **EMAC_INT_ABNORMAL_INT** are cleared automatically by the driver if any of their constituent sources are cleared. Applications do not need to explicitly clear these bits.

Returns:

None.

10.2.4.15 EMACIntDisable

Disables individual Ethernet MAC interrupt sources.

Prototype:

```
void  
EMACIntDisable(uint32_t ui32Base,  
               uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the Ethernet MAC.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated Ethernet MAC interrupt sources.

The ***ui32IntFlags*** parameter is the logical OR of any of the following:

- **EMAC_INT_PHY** indicates that the PHY has signaled a change of state. Software must read and write the appropriate PHY registers to enable and disable particular notifications.
- **EMAC_INT_EARLY_RECEIVE** indicates that the DMA engine has filled the first data buffer of a packet.
- **EMAC_INT_BUS_ERROR** indicates that a fatal bus error has occurred and that the DMA engine has been disabled.
- **EMAC_INT_EARLY_TRANSMIT** indicates that a frame to be transmitted has been fully written from memory into the MAC transmit FIFO.
- **EMAC_INT_RX_WATCHDOG** indicates that a frame with length greater than 2048 bytes (of 10240 bytes in Jumbo Frame mode) was received.
- **EMAC_INT_RX_STOPPED** indicates that the receive process has entered the stopped state.
- **EMAC_INT_RX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's receive descriptor list and the DMA cannot, therefore, acquire a buffer. The receive process is suspended and can be resumed by changing the descriptor ownership and calling [EMACRxDMAPollDemand\(\)](#).
- **EMAC_INT_RECEIVE** indicates that reception of a frame has completed and all requested status has been written to the appropriate DMA receive descriptor.
- **EMAC_INT_TX_UNDERFLOW** indicates that the transmitter experienced an underflow during transmission. The transmit process is suspended.
- **EMAC_INT_RX_OVERFLOW** indicates that an overflow was experienced during reception.
- **EMAC_INT_TX_JABBER** indicates that the transmit jabber timer expired. This condition occurs when the frame size exceeds 2048 bytes (or 10240 bytes in Jumbo Frame mode) and causes the transmit process to abort and enter the Stopped state.
- **EMAC_INT_TX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's transmit descriptor list and that the DMA cannot, therefore, acquire a buffer. Transmission is suspended and can be resumed by changing the descriptor ownership and calling [EMACTxDMAPollDemand\(\)](#).
- **EMAC_INT_TX_STOPPED** indicates that the transmit process has stopped.
- **EMAC_INT_TRANSMIT** indicates that transmission of a frame has completed and that all requested status has been updated in the descriptor.
- **EMAC_INT_TIMESTAMP** indicates that an interrupt from the timestamp module has occurred. This precise source of the interrupt can be determined by calling [EMACTimestampIntStatus\(\)](#), which also clears this bit.

Summary interrupt bits **EMAC_INT_NORMAL_INT** and **EMAC_INT_ABNORMAL_INT** are disabled automatically by the driver if none of their constituent sources are enabled. Applications do not need to explicitly disable these bits.

Note:

Timestamp-related interrupts from the IEEE 1588 module must be disabled independently by using a call to [EMACTimestampTargetIntDisable\(\)](#).

Returns:

None.

10.2.4.16 EMACIntEnable

Enables individual Ethernet MAC interrupt sources.

Prototype:

```
void  
EMACIntEnable(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the Ethernet MAC.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated Ethernet MAC interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **EMAC_INT_PHY** indicates that the PHY has signaled a change of state. Software must read and write the appropriate PHY registers to enable and disable particular notifications.
- **EMAC_INT_EARLY_RECEIVE** indicates that the DMA engine has filled the first data buffer of a packet.
- **EMAC_INT_BUS_ERROR** indicates that a fatal bus error has occurred and that the DMA engine has been disabled.
- **EMAC_INT_EARLY_TRANSMIT** indicates that a frame to be transmitted has been fully written from memory into the MAC transmit FIFO.
- **EMAC_INT_RX_WATCHDOG** indicates that a frame with length greater than 2048 bytes (of 10240 bytes in Jumbo Frame mode) was received.
- **EMAC_INT_RX_STOPPED** indicates that the receive process has entered the stopped state.
- **EMAC_INT_RX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's receive descriptor list and the DMA cannot, therefore, acquire a buffer. The receive process is suspended and can be resumed by changing the descriptor ownership and calling [EMACRxDMAPollDemand\(\)](#).
- **EMAC_INT_RECEIVE** indicates that reception of a frame has completed and all requested status has been written to the appropriate DMA receive descriptor.
- **EMAC_INT_TX_UNDERFLOW** indicates that the transmitter experienced an underflow during transmission. The transmit process is suspended.
- **EMAC_INT_RX_OVERFLOW** indicates that an overflow was experienced during reception.
- **EMAC_INT_TX_JABBER** indicates that the transmit jabber timer expired. This condition occurs when the frame size exceeds 2048 bytes (or 10240 bytes in Jumbo Frame mode) and causes the transmit process to abort and enter the Stopped state.
- **EMAC_INT_TX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's transmit descriptor list and that the DMA cannot, therefore, acquire a buffer. Transmission is suspended and can be resumed by changing the descriptor ownership and calling [EMACTxDMAPollDemand\(\)](#).

- **EMAC_INT_TX_STOPPED** indicates that the transmit process has stopped.
- **EMAC_INT_TRANSMIT** indicates that transmission of a frame has completed and that all requested status has been updated in the descriptor.

Summary interrupt bits **EMAC_INT_NORMAL_INT** and **EMAC_INT_ABNORMAL_INT** are enabled automatically by the driver if any of their constituent sources are enabled. Applications do not need to explicitly enable these bits.

Note:

Timestamp-related interrupts from the IEEE 1588 module must be enabled independently by using a call to [EMACTimestampTargetIntEnable\(\)](#).

Returns:

None.

10.2.4.17 EMACIntRegister

Registers an interrupt handler for an Ethernet interrupt.

Prototype:

```
void  
EMACIntRegister(uint32_t ui32Base,  
                 void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the controller.

pfnHandler is a pointer to the function to be called when the enabled Ethernet interrupts occur.

Description:

This function sets the handler to be called when the Ethernet interrupt occurs. This function enables the global interrupt in the interrupt controller; specific Ethernet interrupts must be enabled via [EMACIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

10.2.4.18 EMACIntStatus

Gets the current Ethernet MAC interrupt status.

Prototype:

```
uint32_t  
EMACIntStatus(uint32_t ui32Base,  
              bool bMasked)
```

Parameters:

ui32Base is the base address of the Ethernet MAC.

bMasked is true to return the masked interrupt status or **false** to return the unmasked status.

Description:

This function returns the interrupt status for the Ethernet MAC. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status as the logical OR of any of the following:

- **EMAC_INT_PHY** indicates that the PHY interrupt has occurred. Software must read the relevant PHY interrupt status register to determine the cause.
- **EMAC_INT_EARLY_RECEIVE** indicates that the DMA engine has filled the first data buffer of a packet.
- **EMAC_INT_BUS_ERROR** indicates that a fatal bus error has occurred and that the DMA engine has been disabled. The cause of the error can be determined by calling [EMACDMAS-stateGet\(\)](#).
- **EMAC_INT_EARLY_TRANSMIT** indicates that a frame to be transmitted has been fully written from memory into the MAC transmit FIFO.
- **EMAC_INT_RX_WATCHDOG** indicates that a frame with length greater than 2048 bytes (of 10240 bytes in Jumbo Frame mode) was received.
- **EMAC_INT_RX_STOPPED** indicates that the receive process has entered the stopped state.
- **EMAC_INT_RX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's receive descriptor list and the DMA cannot, therefore, acquire a buffer. The receive process is suspended and can be resumed by changing the descriptor ownership and calling [EMACRxDMAPollDemand\(\)](#).
- **EMAC_INT_RECEIVE** indicates that reception of a frame has completed and all requested status has been written to the appropriate DMA receive descriptor.
- **EMAC_INT_TX_UNDERFLOW** indicates that the transmitter experienced an underflow during transmission. The transmit process is suspended.
- **EMAC_INT_RX_OVERFLOW** indicates that an overflow was experienced during reception.
- **EMAC_INT_TX_JABBER** indicates that the transmit jabber timer expired. This condition occurs when the frame size exceeds 2048 bytes (or 10240 bytes in Jumbo Frame mode) and causes the transmit process to abort and enter the Stopped state.
- **EMAC_INT_TX_NO_BUFFER** indicates that the host owns the next buffer in the DMA's transmit descriptor list and that the DMA cannot, therefore, acquire a buffer. Transmission is suspended and can be resumed by changing the descriptor ownership and calling [EMAC-TxDMAPollDemand\(\)](#).
- **EMAC_INT_TX_STOPPED** indicates that the transmit process has stopped.
- **EMAC_INT_TRANSMIT** indicates that transmission of a frame has completed and that all requested status has been updated in the descriptor.
- **EMAC_INT_NORMAL_INT** is a summary interrupt comprising the logical OR of the masked state of **EMAC_INT_TRANSMIT**, **EMAC_INT_RECEIVE**, **EMAC_INT_TX_NO_BUFFER** and **EMAC_INT_EARLY_RECEIVE**.
- **EMAC_INT_ABNORMAL_INT** is a summary interrupt comprising the logical OR of the masked state of **EMAC_INT_TX_STOPPED**, **EMAC_INT_TX_JABBER**, **EMAC_INT_RX_OVERFLOW**, **EMAC_INT_TX_UNDERFLOW**, **EMAC_INT_RX_NO_BUFFER**, **EMAC_INT_RX_STOPPED**, **EMAC_INT_RX_WATCHDOG**, **EMAC_INT_EARLY_TRANSMIT** and **EMAC_INT_BUS_ERROR**.

10.2.4.19 EMACIntUnregister

Unregisters an interrupt handler for an Ethernet interrupt.

Prototype:

```
void  
EMACIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function unregisters the interrupt handler. This function disables the global interrupt in the interrupt controller so that the interrupt handler is no longer called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

10.2.4.20 EMACNumAddrGet

Returns the number of MAC addresses supported by the Ethernet controller.

Prototype:

```
uint32_t  
EMACNumAddrGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

Description:

This function may be used to determine the number of MAC addresses that the given controller supports. MAC address slots may be used when performing perfect (rather than hash table) filtering of packets.

Returns:

Returns the number of supported MAC addresses.

10.2.4.21 EMACPHYConfigSet

Selects the Ethernet PHY in use.

Prototype:

```
void  
EMACPHYConfigSet(uint32_t ui32Base,  
                  uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

ui32Config selects the PHY in use and, when using the internal PHY, allows various PHY parameters to be configured.

Description:

This function must be called prior to [EMACInit\(\)](#) and [EMACConfigSet\(\)](#) to select the Ethernet PHY to be used. If the internal PHY is selected, the function also allows configuration of various PHY parameters. Note that the Ethernet MAC is reset during this function call because parameters used by this function are latched by the hardware only on a MAC reset. The call sequence to select and configure the PHY, therefore, must be as follows:

```
// Enable and reset the MAC.  
SysCtlPeripheralEnable(SYSCONFIG_PERIPH_EMAC0);  
SysCtlPeripheralReset(SYSCONFIG_PERIPH_EMAC0);  
if(<using internal PHY>)  
{  
    // Enable and reset the internal PHY.  
    SysCtlPeripheralEnable(SYSCONFIG_PERIPH_EPHY0);  
    SysCtlPeripheralReset(SYSCONFIG_PERIPH_EPHY0);  
}  
  
// Ensure the MAC is completed its reset.  
while(!MAP_SysCtlPeripheralReady(SYSCONFIG_PERIPH_EMAC0))  
{  
}  
  
// Set the PHY type and configuration options.  
EMACPHYConfigSet(EMAC0_BASE, <config>);  
  
// Initialize and configure the MAC.  
EMACInit(EMAC0_BASE, <system clock rate>, <bus config>,  
         <Rx burst size>, <Tx burst size>, <desc skip>);  
EMACConfigSet(EMAC0_BASE, <parameters>);
```

The *ui32Config* parameter must specify one of the following values:

- **EMAC_PHY_TYPE_INTERNAL** selects the internal Ethernet PHY.
- **EMAC_PHY_TYPE_EXTERNAL_MII** selects an external PHY connected via the MII interface.
- **EMAC_PHY_TYPE_EXTERNAL_RMII** selects an external PHY connected via the RMII interface.

If **EMAC_PHY_TYPE_INTERNAL** is selected, the following flags may be ORed into *ui32Config* to control various PHY features and modes. These flags are ignored if an external PHY is selected.

- **EMAC_PHY_INT_NIB_TXERR_DET_DIS** disables odd nibble transmit error detection (sets the default value of PHY register MR10, bit 1).
- **EMAC_PHY_INT_RX_ER_DURING_IDLE** enables receive error detection during idle (sets the default value of PHY register MR10, bit 2).
- **EMAC_PHY_INT_ISOLATE_MII_LLOSS** ties the MII outputs low if no link is established in 100B-T and full duplex modes (sets the default value of PHY register MR10, bit 3).
- **EMAC_PHY_INT_LINK_LOSS_RECOVERY** enables link loss recovery (sets the default value of PHY register MR9, bit 7).
- **EMAC_PHY_INT_TDRRUN** enables execution of the TDR procedure after a link down event (sets the default value of PHY register MR9, bit 8).
- **EMAC_PHY_INT_LD_ON_RX_ERR_COUNT** enables link down if the receiver error count reaches 32 within a 10-us interval (sets the default value of PHY register MR11 bit 3).

- **EMAC_PHY_INT_LD_ON_MTL3_ERR_COUNT** enables link down if the MTL3 error count reaches 20 in a 10 us-interval (sets the default value of PHY register MR11 bit 2).
- **EMAC_PHY_INT_LD_ON_LOW_SNR** enables link down if the low SNR threshold is crossed 20 times in a 10 us-interval (sets the default value of PHY register MR11 bit 1).
- **EMAC_PHY_INT_LD_ON_SIGNAL_ENERGY** enables link down if energy detector indicates Energy Loss (sets the default value of PHY register MR11 bit 0).
- **EMAC_PHY_INT_POLARITY_SWAP** inverts the polarity on both TPTD and TPRD pairs (sets the default value of PHY register MR11 bit 5).
- **EMAC_PHY_INT_MDI_SWAP** swaps the MDI pairs putting receive on the TPTD pair and transmit on TPRD (sets the default value of PHY register MR11 bit 6).
- **EMAC_PHY_INT_ROBUST_MDIX** enables robust auto MDI-X resolution (sets the default value of PHY register MR9 bit 5).
- **EMAC_PHY_INT_FAST_MDIX** enables fast auto-MDI/MDIX resolution (sets the default value of PHY register MR9 bit 6).
- **EMAC_PHY_INT_MDIX_EN** enables auto-MDI/MDIX crossover (sets the default value of PHY register MR9 bit 14).
- **EMAC_PHY_INT_FAST_RXDV_DETECT** enables fast RXDV detection (set the default value of PHY register MR9 bit 1).
- **EMAC_PHY_INT_FAST_L_UP_DETECT** enables fast link-up time during parallel detection (sets the default value of PHY register MR10 bit 6)
- **EMAC_PHY_INT_EXT_FULL_DUPLEX** forces full-duplex while working with a link partner in forced 100B-TX (sets the default value of PHY register MR10 bit 5).
- **EMAC_PHY_INT_FAST_AN_80_50_35** enables fast auto-negotiation using break link, link fail inhibit and wait timers set to 80, 50 and 35 respectively (sets the default value of PHY register MR9 bits [4:2] to 3b100).
- **EMAC_PHY_INT_FAST_AN_120_75_50** enables fast auto-negotiation using break link, link fail inhibit and wait timers set to 120, 75 and 50 respectively (sets the default value of PHY register MR9 bits [4:2] to 3b101).
- **EMAC_PHY_INT_FAST_AN_140_150_100** enables fast auto-negotiation using break link, link fail inhibit and wait timers set to 140, 150 and 100 respectively (sets the default value of PHY register MR9 bits [4:2] to 3b110).
- **EMAC_PHY_FORCE_10B_T_HALF_DUPLEX** disables auto-negotiation and forces operation in 10Base-T, half duplex mode (sets the default value of PHY register MR9 bits [13:11] to 3b000).
- **EMAC_PHY_FORCE_10B_T_FULL_DUPLEX** disables auto-negotiation and forces operation in 10Base-T, full duplex mode (sets the default value of PHY register MR9 bits [13:11] to 3b001).
- **EMAC_PHY_FORCE_100B_T_HALF_DUPLEX** disables auto-negotiation and forces operation in 100Base-T, half duplex mode (sets the default value of PHY register MR9 bits [13:11] to 3b010).
- **EMAC_PHY_FORCE_100B_T_FULL_DUPLEX** disables auto-negotiation and forces operation in 100Base-T, full duplex mode (sets the default value of PHY register MR9 bits [13:11] to 3b011).
- **EMAC_PHY_AN_10B_T_HALF_DUPLEX** enables auto-negotiation and advertises 10Base-T, half duplex mode (sets the default value of PHY register MR9 bits [13:11] to 3b100).
- **EMAC_PHY_AN_10B_T_FULL_DUPLEX** enables auto-negotiation and advertises 10Base-T half or full duplex modes (sets the default value of PHY register MR9 bits [13:11] to 3b101).

- **EMAC_PHY_AN_100B_T_HALF_DUPLEX** enables auto-negotiation and advertises 10Base-T half or full duplex, and 100Base-T half duplex modes (sets the default value of PHY register MR9 bits [13:11] to 3b110).
- **EMAC_PHY_AN_100B_T_FULL_DUPLEX** enables auto-negotiation and advertises 10Base-T half or full duplex, and 100Base-T half or full duplex modes (sets the default value of PHY register MR9 bits [13:11] to 3b111).
- **EMAC_PHY_INT_HOLD** prevents the PHY from transmitting energy on the line.

As a side effect of this function, the Ethernet MAC is reset so any previous MAC configuration is lost.

Returns:

None.

10.2.4.22 EMACPHYExtendedRead

Reads from an extended PHY register.

Prototype:

```
uint16_t  
EMACPHYExtendedRead(uint32_t ui32Base,  
                     uint8_t ui8PhyAddr,  
                     uint16_t ui16RegAddr)
```

Parameters:

ui32Base is the base address of the controller.

ui8PhyAddr is the physical address of the PHY to access.

ui16RegAddr is the address of the PHY extended register to be accessed.

Description:

When using the internal PHY or when connected to an external PHY supporting extended registers, this function returns the contents of the extended PHY register specified by **ui16RegAddr**.

Returns:

Returns the 16-bit value read from the PHY.

10.2.4.23 EMACPHYExtendedWrite

Writes a value to an extended PHY register.

Prototype:

```
void  
EMACPHYExtendedWrite(uint32_t ui32Base,  
                      uint8_t ui8PhyAddr,  
                      uint16_t ui16RegAddr,  
                      uint16_t ui16Value)
```

Parameters:

ui32Base is the base address of the controller.

ui8PhyAddr is the physical address of the PHY to access.

ui16RegAddr is the address of the PHY extended register to be accessed.

ui16Value is the value to write to the register.

Description:

When using the internal PHY or when connected to an external PHY supporting extended registers, this function allows a value to be written to the extended PHY register specified by *ui16RegAddr*.

Returns:

None.

10.2.4.24 EMACPHYPowerOff

Powers off the Ethernet PHY.

Prototype:

```
void  
EMACPHYPowerOff(uint32_t ui32Base,  
                 uint8_t ui8PhyAddr)
```

Parameters:

ui32Base is the base address of the controller.

ui8PhyAddr is the physical address of the PHY to power down.

Description:

This function powers off the Ethernet PHY, reducing the current consumption of the device. While in the powered-off state, the Ethernet controller is unable to connect to Ethernet.

Returns:

None.

10.2.4.25 EMACPHYPowerOn

Powers on the Ethernet PHY.

Prototype:

```
void  
EMACPHYPowerOn(uint32_t ui32Base,  
                uint8_t ui8PhyAddr)
```

Parameters:

ui32Base is the base address of the controller.

ui8PhyAddr is the physical address of the PHY to power up.

Description:

This function powers on the Ethernet PHY, enabling it return to normal operation. By default, the PHY is powered on, so this function is only called if [EMACPHYPowerOff\(\)](#) has previously been called.

Returns:

None.

10.2.4.26 EMACPHYRead

Reads from a PHY register.

Prototype:

```
uint16_t  
EMACPHYRead(uint32_t ui32Base,  
            uint8_t ui8PhyAddr,  
            uint8_t ui8RegAddr)
```

Parameters:

ui32Base is the base address of the controller.
ui8PhyAddr is the physical address of the PHY to access.
ui8RegAddr is the address of the PHY register to be accessed.

Description:

This function returns the contents of the PHY register specified by *ui8RegAddr*.

Returns:

Returns the 16-bit value read from the PHY.

10.2.4.27 EMACPHYWrite

Writes to the PHY register.

Prototype:

```
void  
EMACPHYWrite(uint32_t ui32Base,  
             uint8_t ui8PhyAddr,  
             uint8_t ui8RegAddr,  
             uint16_t ui16Data)
```

Parameters:

ui32Base is the base address of the controller.
ui8PhyAddr is the physical address of the PHY to access.
ui8RegAddr is the address of the PHY register to be accessed.
ui16Data is the data to be written to the PHY register.

Description:

This function writes the *ui16Data* value to the PHY register specified by *ui8RegAddr*.

Returns:

None.

10.2.4.28 EMACPowerManagementControlGet

Queries the current Ethernet MAC remote wake-up configuration.

Prototype:

```
uint32_t  
EMACPowerManagementControlGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function allows the MAC's remote wake-up settings to be queried. These settings determine which types of frame should trigger a remote wake-up event

Returns:

Returns a logical OR of the following flags:

- **EMAC_PMT_GLOBAL_UNICAST_ENABLE** indicates that the MAC wakes up when any unicast frame matching the MAC destination address filter is received.
- **EMAC_PMT_WAKEUP_PACKET_ENABLE** indicates that the MAC wakes up when any received frame matches the remote wake-up filter configured via a call to [EMACRemoteWake-UpFrameFilterSet\(\)](#).
- **EMAC_PMT_MAGIC_PACKET_ENABLE** indicates that the MAC wakes up when a standard Wake-on-LAN "magic packet" is received. The magic packet contains 6 bytes of 0xFF followed immediately by 16 repetitions of the destination MAC address.
- **EMAC_PMT_POWER_DOWN** indicates that the MAC is currently in power-down mode and is waiting for an incoming frame matching the remote wake-up frames as described by other returned flags and via the remote wake-up filter.

10.2.4.29 EMACPowerManagementControlSet

Sets the Ethernet MAC remote wake-up configuration.

Prototype:

```
void
EMACPowerManagementControlSet(uint32_t ui32Base,
                             uint32_t ui32Flags)
```

Parameters:

ui32Base is the base address of the controller.

ui32Flags defines which types of frame should trigger a remote wake-up and allows the MAC to be put into power-down mode.

Description:

This function allows the MAC's remote wake-up features to be configured, determining which types of frame should trigger a wake-up event and allowing an application to place the MAC in power-down mode. In this mode, the MAC ignores all received frames until one matching a configured remote wake-up frame is received, at which point the MAC automatically exits power-down mode and continues to receive frames.

The *ui32Flags* parameter is a logical OR of the following flags:

- **EMAC_PMT_GLOBAL_UNICAST_ENABLE** instructs the MAC to wake up when any unicast frame matching the MAC destination address filter is received.
- **EMAC_PMT_WAKEUP_PACKET_ENABLE** instructs the MAC to wake up when any received frame matches the remote wake-up filter configured via a call to [EMACRemoteWake-UpFrameFilterSet\(\)](#).

- **EMAC_PMT_MAGIC_PACKET_ENABLE** instructs the MAC to wake up when a standard Wake-on-LAN "magic packet" is received. The magic packet contains 6 bytes of 0xFF followed immediately by 16 repetitions of the destination MAC address.
- **EMAC_PMT_POWER_DOWN** instructs the MAC to enter power-down mode and wait for an incoming frame matching the remote wake-up frames as described by other flags and via the remote wake-up filter. This flag should only be set if at least one other flag is specified to configure a wake-up frame type.

When the MAC is in power-down mode, software may exit the mode by calling this function with the **EMAC_PMT_POWER_DOWN** flag absent from *ui32Flags*. If a configured wake-up frame is received while in power-down mode, the **EMAC_INT_POWER_MGMT** interrupt is signaled and may be cleared by reading the status using [EMACPowerManagementStatusGet\(\)](#).

Note:

While it is possible to gate the clock to the MAC while it is in power-down mode, doing so prevents the reading of the registers required to determine the interrupt status and also prevents power-down mode from exiting via another call to this function.

Returns:

None.

10.2.4.30 EMACPowerManagementStatusGet

Queries the current Ethernet MAC remote wake-up status.

Prototype:

```
uint32_t  
EMACPowerManagementStatusGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function returns information on the remote wake-up state of the Ethernet MAC. If the MAC has been woken up since the last call, the returned value indicates the type of received frame that caused the MAC to exit power-down state.

Returns:

Returns a logical OR of the following flags:

- **EMAC_PMT_POWER_DOWN** indicates that the MAC is currently in power-down mode.
- **EMAC_PMT_WAKEUP_PACKET RECEIVED** indicates that the MAC exited power-down mode due to a remote wake-up frame being received. This function call clears this flag.
- **EMAC_PMT_MAGIC_PACKET RECEIVED** indicates that the MAC exited power-down mode due to a wake-on-LAN magic packet being received. This function call clears this flag.

10.2.4.31 EMACRemoteWakeUpFrameFilterGet

Returns the current remote wake-up frame filter configuration.

Prototype:

```
void
EMACRemoteWakeUpFrameFilterGet(uint32_t ui32Base,
                               tEMACWakeUpFrameFilter *pFilter)
```

Parameters:

ui32Base is the base address of the controller.

pFilter points to the structure that is written with the current remote wake-up frame filter information.

Description:

This function may be used to read the current wake-up frame filter settings. The data returned by the function describes wake-up frames in terms of a CRC calculated on up to 31 payload bytes in the frame. The actual bytes used in the CRC calculation are defined by means of a bit mask where a “1” indicates that a byte in the frame should contribute to the CRC calculation and a “0” indicates that the byte should be skipped, and an offset from the start of the frame to the payload byte that represents the first byte in the 31-byte CRC-checked sequence.

The *pFilter* parameter points to storage that is written with a structure containing the information defining the frame filters. This structure contains the following fields, each of which is replicated 4 times, once for each possible wake-up frame:

- **pui32ByteMask** defines whether a given byte in the chosen 31-byte sequence within the frame should contribute to the CRC calculation or not. A 1 indicates that the byte should contribute to the calculation, a 0 causes the byte to be skipped.
- **pui8Command** contains flags defining whether this filter is enabled and, if so, whether it refers to unicast or multicast packets. Valid values are one of **EMAC_RWU_FILTER_MULTICAST** or **EMAC_RWU_FILTER_UNICAST** ORed with one of **EMAC_RWU_FILTER_ENABLE** or **EMAC_RWU_FILTER_DISABLE**.
- **pui8Offset** defines the zero-based index of the byte within the frame at which CRC checking defined by **pui32ByteMask** begins. Alternatively, this value can be thought of as the number of bytes in the frame that the MAC skips before accumulating the CRC based on the pattern in **pui32ByteMask**.
- **pui16CRC** provides the value of the calculated CRC for a valid remote wake-up frame. If the incoming frame is processed according to the filter values provided and the final CRC calculation equals this value, the frame is considered to be a valid remote wake-up frame.

Note that this filter uses CRC16 rather than CRC32 as used in frame checksums.

Returns:

None.

10.2.4.32 EMACRemoteWakeUpFrameFilterSet

Sets values defining up to four frames used to trigger a remote wake-up.

Prototype:

```
void
EMACRemoteWakeUpFrameFilterSet(uint32_t ui32Base,
                               const tEMACWakeUpFrameFilter *pFilter)
```

Parameters:

ui32Base is the base address of the controller.

pFilter points to the structure containing remote wake-up frame filter information.

Description:

This function may be used to define up to four different frames that are considered by the Ethernet MAC to be remote wake-up signals. The data passed to the function describes a wake-up frame in terms of a CRC calculated on up to 31 payload bytes in the frame. The actual bytes used in the CRC calculation are defined by means of a bit mask where a “1” indicates that a byte in the frame should contribute to the CRC calculation and a “0” indicates that the byte should be skipped, as well as an offset from the start of the frame to the payload byte that represents the first byte in the 31-byte CRC-checked sequence.

The *pFilter* parameter points to a structure containing the information necessary to set up the filters. This structure contains the following fields, each of which is replicated 4 times, once for each possible wake-up frame:

- **pui32ByteMask** defines whether a given byte in the chosen 31-byte sequence within the frame should contribute to the CRC calculation or not. A 1 indicates that the byte should contribute to the calculation, a 0 causes the byte to be skipped.
- **pui8Command** contains flags defining whether this filter is enabled and, if so, whether it refers to unicast or multicast packets. Valid values are one of **EMAC_RWU_FILTER_MULTICAST** or **EMAC_RWU_FILTER_UNICAST** ORed with one of **EMAC_RWU_FILTER_ENABLE** or **EMAC_RWU_FILTER_DISABLE**.
- **pui8Offset** defines the zero-based index of the byte within the frame at which CRC checking defined by **pui32ByteMask** begins. Alternatively, this value can be thought of as the number of bytes in the frame that the MAC skips before accumulating the CRC based on the pattern in **pui32ByteMask**.
- **pui16CRC** provides the value of the calculated CRC for a valid remote wake-up frame. If the incoming frame is processed according to the filter values provided and the final CRC calculation equals this value, the frame is considered to be a valid remote wake-up frame.

Note that this filter uses CRC16 rather than CRC32 as used in frame checksums. The required CRC uses a direct algorithm with polynomial 0x8005, initial seed value 0xFFFF, no final XOR and reversed data order. CRCs for use in this function may be determined using the online calculator found at <http://www.zorc.breitbandkatze.de/crc.html>.

Returns:

None.

10.2.4.33 EMACReset

Resets the Ethernet MAC.

Prototype:

```
void  
EMACReset (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

Description:

This function performs a reset of the Ethernet MAC by resetting all logic and returning all registers to their default values. The function returns only after the hardware indicates that the reset has completed.

Note:

To ensure that the reset completes, the selected PHY clock must be enabled when this function is called. If the PHY clock is absent, this function does not return.

Returns:

None.

10.2.4.34 EMACRxDisable

Disables the Ethernet controller receiver.

Prototype:

```
void  
EMACRxDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

When terminating operations on the Ethernet interface, this function should be called. This function disables the receiver.

Returns:

None.

10.2.4.35 EMACRxDMACurrentBufferGet

Returns the current DMA receive buffer pointer.

Prototype:

```
uint8_t *  
EMACRxDMACurrentBufferGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be called to determine which buffer the receive DMA engine is currently writing to.

Returns:

Returns the receive buffer address currently being written by the DMA engine.

10.2.4.36 EMACRxDMACurrentDescriptorGet

Returns the current DMA receive descriptor pointer.

Prototype:

```
tEMACDMADescriptor *  
EMACRxDMACurrentDescriptorGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function returns a pointer to the current Ethernet receive descriptor read by the DMA.

Returns:

Returns a pointer to the start of the current receive DMA descriptor.

10.2.4.37 EMACRxDMADescriptorListGet

Returns a pointer to the start of the DMA receive descriptor list.

Prototype:

```
tEMACDMADescriptor *  
EMACRxDMADescriptorListGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function returns a pointer to the head of the Ethernet MAC's receive DMA descriptor list. This value corresponds to the pointer originally set using a call to [EMACRxDMADescriptorList-Set\(\)](#).

Returns:

Returns a pointer to the start of the DMA receive descriptor list.

10.2.4.38 EMACRxDMADescriptorListSet

Sets the DMA receive descriptor list pointer.

Prototype:

```
void  
EMACRxDMADescriptorListSet(uint32_t ui32Base,  
                           tEMACDMADescriptor *pDescriptor)
```

Parameters:

ui32Base is the base address of the controller.

pDescriptor points to the first DMA descriptor in the list to be passed to the receive DMA engine.

Description:

This function sets the Ethernet MAC's receive DMA descriptor list pointer. The *pDescriptor* pointer must point to one or more descriptor structures.

When multiple descriptors are provided, they can be either chained or unchained. Chained descriptors are indicated by setting the **DES0_TX_CTRL_CHAINED** or **DES1_RX_CTRL_CHAINED** bit in the relevant word of the transmit or receive descriptor. If this bit is clear, unchained descriptors are assumed.

Chained descriptors use a link pointer in each descriptor to point to the next descriptor in the chain.

Unchained descriptors are assumed to be contiguous in memory with a consistent offset between the start of one descriptor and the next. If unchained descriptors are used, the *pvLink* field in the descriptor becomes available to store a second buffer pointer, allowing each descriptor to point to two buffers rather than one. In this case, the *ui32DescSkipSize* parameter to [EMACInit\(\)](#) must previously have been set to the number of words between the end of one descriptor and the start of the next. This value must be 0 in cases where a packed array of [tEMACDMADescriptor](#) structures is used. If the application wishes to add new state fields to the end of the descriptor structure, the skip size should be set to accommodate the newly sized structure.

Applications are responsible for initializing all descriptor fields appropriately before passing the descriptor list to the hardware.

Returns:

None.

10.2.4.39 EMACRxDMAPollDemand

Orders the MAC DMA controller to attempt to acquire the next receive descriptor.

Prototype:

```
void  
EMACRxDMAPollDemand(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

Description:

This function must be called to restart the receiver if it has been suspended due to the current receive DMA descriptor being owned by the host. Once the application reads any data from the descriptor and marks it as being owned by the MAC DMA, this function causes the hardware to attempt to acquire the descriptor before writing the next received packet into its buffer(s).

Returns:

None.

10.2.4.40 EMACRxEnable

Enables the Ethernet controller receiver.

Prototype:

```
void  
EMACRxEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

When starting operations on the Ethernet interface, this function should be called to enable the receiver after all configuration has been completed.

Returns:

None.

10.2.4.41 EMACRxWatchdogTimerSet

Sets the receive interrupt watchdog timer period.

Prototype:

```
void  
EMACRxWatchdogTimerSet(uint32_t ui32Base,  
                        uint8_t ui8Timeout)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

ui8Timeout is the desired timeout expressed as a number of 256 system clock periods.

Description:

This function configures the receive interrupt watchdog timer. The *uiTimeout* parameter specifies the number of 256 system clock periods that elapse before the timer expires. In cases where the DMA has transferred a frame using a descriptor that has **DES1_RX_CTRL_DISABLE_INT** set, the watchdog causes a receive interrupt to be generated when it times out. The watchdog timer is reset whenever a packet is transferred to memory using a DMA descriptor that does not disable the receive interrupt.

To disable the receive interrupt watchdog function, set *ui8Timeout* to 0.

Returns:

None.

10.2.4.42 EMACStatusGet

Returns the current Ethernet MAC status.

Prototype:

```
uint32_t  
EMACStatusGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

Description:

This function returns information on the current status of all the main modules in the MAC transmit and receive data paths.

Returns:

Returns the current MAC status as a logical OR of any of the following flags:

- **EMAC_STATUS_TX_NOT_EMPTY**
- **EMAC_STATUS_TX_WRITING_FIFO**
- **EMAC_STATUS_TX_PAUSED**
- **EMAC_STATUS_MAC_NOT_IDLE**
- **EMAC_STATUS_RWC_ACTIVE**
- **EMAC_STATUS_RPE_ACTIVE**

The transmit frame controller status can be extracted from the returned value by ANDing with **EMAC_STATUS_TFC_STATE_MASK** and is one of the following:

- **EMAC_STATUS_TFC_STATE_IDLE**
- **EMAC_STATUS_TFC_STATE_WAITING**
- **EMAC_STATUS_TFC_STATE_PAUSING**
- **EMAC_STATUS_TFC_STATE_WRITING**

The transmit FIFO read controller status can be extracted from the returned value by ANDing with **EMAC_STATUS_TRC_STATE_MASK** and is one of the following:

- **EMAC_STATUS_TRC_STATE_IDLE**
- **EMAC_STATUS_TRC_STATE_READING**
- **EMAC_STATUS_TRC_STATE_WAITING**
- **EMAC_STATUS_TRC_STATE_STATUS**

The current receive FIFO levels can be extracted from the returned value by ANDing with **EMAC_STATUS_RX_FIFO_LEVEL_MASK** and is one of the following:

- **EMAC_STATUS_RX_FIFO_EMPTY** indicating that the FIFO is empty.
- **EMAC_STATUS_RX_FIFO_BELOW** indicating that the FIFO fill level is below the flow-control deactivate threshold.
- **EMAC_STATUS_RX_FIFO_ABOVE** indicating that the FIFO fill level is above the flow-control activate threshold.
- **EMAC_STATUS_RX_FIFO_FULL** indicating that the FIFO is full.

The current receive FIFO state can be extracted from the returned value by ANDing with **EMAC_STATUS_RX_FIFO_STATE_MASK** and is one of the following:

- **EMAC_STATUS_RX_FIFO_IDLE**
- **EMAC_STATUS_RX_FIFO_READING**
- **EMAC_STATUS_RX_FIFO_STATUS**
- **EMAC_STATUS_RX_FIFO_FLUSHING**

10.2.4.43 EMACTimestampAddendSet

Adjusts the system time update rate when using the fine correction method.

Prototype:

```
void
EMACTimestampAddendSet(uint32_t ui32Base,
                      uint32_t ui32Increment)
```

Parameters:

ui32Base is the base address of the controller.

ui32Increment is the number to add to the accumulator register on each tick of the 25-MHz main oscillator.

Description:

This function is used to control the rate of update of the system time when in fine update mode. Fine correction mode is selected if *EMAC_TS_UPDATE_FINE* is supplied in the *ui32Config* parameter passed to a previous call to [EMACTimestampConfigSet\(\)](#). Fine update mode is typically used when synchronizing the local clock to the IEEE 1588 master clock. The sub-second counter is incremented by the number passed to [EMACTimestampConfigSet\(\)](#) in the *ui32SubSecondInc* parameter each time a 32-bit accumulator register generates a carry. The accumulator register is incremented by the "addend" value on each main oscillator tick, and this addend value is modified to allow fine control over the rate of change of the timestamp counter. The addend value is calculated using the ratio of the main oscillator clock rate and the desired IEEE 1588 clock rate and the *ui32SubSecondInc* value is set to correspond to the desired IEEE 1588 clock rate.

As an example, using digital rollover mode and a 25-MHz main oscillator clock with a desired IEEE 1588 clock accuracy of 12.5 MHz, and having made a previous call to [EMACTimestampConfigSet\(\)](#) with *ui32SubSecondInc* set to the 12.5-MHz clock period of 80 ns, the initial *ui32Increment* value would be set to 0x80000000 to generate a carry on every second main oscillator tick. Because the system time updates each time the accumulator overflows, small changes in the *ui32Increment* value can be used to very finely control the system time rate.

Returns:

None.

See also:

[EMACTimestampConfigSet\(\)](#)

10.2.4.44 EMACTimestampConfigGet

Returns the current IEEE 1588 timestamping configuration.

Prototype:

```
uint32_t  
EMACTimestampConfigGet (uint32_t ui32Base,  
                      uint32_t *pui32SubSecondInc)
```

Parameters:

ui32Base is the base address of the controller.

pui32SubSecondInc points to storage that is written with the current subsecond increment value for the IEEE 1588 clock.

Description:

This function may be used to retrieve the current MAC timestamping configuration.

See also:

[EMACTimestampConfigSet\(\)](#)

Returns:

Returns the current timestamping configuration as a logical OR of the following flags:

- **EMAC_TS_PTP_VERSION_2** indicates that the MAC is processing PTP version 2 messages.
If this flag is absent, PTP version 1 messages are expected.

- **EMAC_TS_DIGITAL_ROLLOVER** causes the clock's subsecond value to roll over at 0x3BA9C9FF (99999999 decimal). In this mode, it can be considered as a nanosecond counter with each digit representing 1 ns. If this flag is absent, the subsecond value rolls over at 0xFFFFFFFF, effectively counting increments of 0.465 ns.
- **EMAC_TS_MAC_FILTER_ENABLE** indicates that incoming PTP messages are filtered using any of the configured MAC addresses. Messages with a destination address programmed into the MAC address filter are passed, others are discarded. If this flag is absent, the MAC address is ignored.
- **EMAC_TS_UPDATE_FINE** implements the fine update method that causes the IEEE 1588 clock to advance by the the value returned in the **pui32SubSecondInc* parameter each time a carry is generated from the addend accumulator register. If this flag is absent, the coarse update method is in use and the clock is advanced by the **pui32SubSecondInc* value on each system clock tick.
- **EMAC_TS_SYNC_ONLY** indicates that timestamps are only generated for SYNC messages.
- **EMAC_TS_DELAYREQ_ONLY** indicates that timestamps are only generated for Delay_Req messages.
- **EMAC_TS_ALL** indicates that timestamps are generated for all IEEE 1588 messages.
- **EMAC_TS_SYNC_PDREQ_PDRESP** timestamps only SYNC, Pdelay_Req and Pdelay_Resp messages.
- **EMAC_TS_DREQ_PDREQ_PDRESP** indicates that timestamps are only generated for Delay_Req, Pdelay_Req and Pdelay_Resp messages.
- **EMAC_TS_SYNC_DELAYREQ** indicates that timestamps are only generated for Delay_Req messages.
- **EMAC_TS_PDREQ_PDRESP** indicates that timestamps are only generated for Pdelay_Req and Pdelay_Resp messages.
- **EMAC_TS_PROCESS_IPV4_UDP** indicates that PTP packets encapsulated in UDP over IPv4 packets are being processed. If absent, the MAC ignores these frames.
- **EMAC_TS_PROCESS_IPV6_UDP** indicates that PTP packets encapsulated in UDP over IPv6 packets are being processed. If absent, the MAC ignores these frames.
- **EMAC_TS_PROCESS_ETHERNET** indicates that PTP packets encapsulated directly in Ethernet frames are being processsd. If absent, the MAC ignores these frames.
- **EMAC_TS_ALL_RX_FRAMES** indicates that timestamping is enabled for all frames received by the MAC, regardless of type.

If **EMAC_TS_ALL_RX_FRAMES** and none of the options specifying subsets of PTP packets to timestamp are set, the MAC is configured to timestamp SYNC, Follow_Up, Delay_Req and Delay_Resp messages only.

10.2.4.45 EMACTimestampConfigSet

Configures the Ethernet MAC's IEEE 1588 timestamping options.

Prototype:

```
void
EMACTimestampConfigSet(uint32_t ui32Base,
                      uint32_t ui32Config,
                      uint32_t ui32SubSecondInc)
```

Parameters:

ui32Base is the base address of the controller.

ui32Config contains flags selecting particular configuration options.

ui32SubSecondInc is the number that the IEEE 1588 subsecond clock should increment on each tick.

Description:

This function is used to configure the operation of the Ethernet MAC's internal timestamping clock. This clock is used to timestamp incoming and outgoing packets and as an accurate system time reference when IEEE 1588 Precision Time Protocol is in use.

The *ui32Config* parameter contains a collection of flags selecting the desired options. Valid flags are:

One of the following to determine whether IEEE 1588 version 1 or version 2 packet format is to be processed:

- **EMAC_TS_PTP_VERSION_2**
- **EMAC_TS_PTP_VERSION_1**

One of the following to determine how the IEEE 1588 clock's subsecond value should be interpreted and handled:

- **EMAC_TS_DIGITAL_ROLLOVER** causes the clock's subsecond value to roll over at 0x3BA9C9FF (99999999 decimal). In this mode, it can be considered as a nanosecond counter with each digit representing 1 ns.
- **EMAC_TS_BINARY_ROLLOVER** causes the clock's subsecond value to roll over at 0xFFFFFFFF. In this mode, the subsecond value counts 0.465 ns periods.

One of the following to enable or disable MAC address filtering. When enabled, PTP frames are filtered unless the destination MAC address matches any of the currently programmed MAC addresses.

- **EMAC_TS_MAC_FILTER_ENABLE**
- **EMAC_TS_MAC_FILTER_DISABLE**

One of the following to determine how the clock is updated:

- **EMAC_TS_UPDATE_COARSE** causes the IEEE 1588 clock to advance by the value supplied in the *ui32SubSecondInc* parameter on each main oscillator clock cycle.
- **EMAC_TS_UPDATE_FINE** selects the fine update method which causes the IEEE 1588 clock to advance by the the value supplied in the *ui32SubSecondInc* parameter each time a carry is generated from the addend accumulator register.

One of the following to determine which IEEE 1588 messages are timestamped:

- **EMAC_TS_SYNC_FOLLOW_DREQ_DRESP** timestamps SYNC, Follow_Up, Delay_Req and Delay_Resp messages.
- **EMAC_TS_SYNC_ONLY** timestamps only SYNC messages.
- **EMAC_TS_DELAYREQ_ONLY** timestamps only Delay_Req messages.
- **EMAC_TS_ALL** timestamps all IEEE 1588 messages.
- **EMAC_TS_SYNC_PDREQ_PDRESP** timestamps only SYNC, Pdelay_Req and Pdelay_Resp messages.
- **EMAC_TS_DREQ_PDREQ_PDRESP** timestamps only Delay_Req, Pdelay_Req and Pdelay_Resp messages.

- **EMAC_TS_SYNC_DELAYREQ** timestamps only Delay_Req messages.
- **EMAC_TS_PDREQ_PDRESP** timestamps only Pdelay_Req and Pdelay_Resp messages.

Optional, additional flags are:

- **EMAC_TS_PROCESS_IPV4_UDP** processes PTP packets encapsulated in UDP over IPv4 packets. If absent, the MAC ignores these frames.
- **EMAC_TS_PROCESS_IPV6_UDP** processes PTP packets encapsulated in UDP over IPv6 packets. If absent, the MAC ignores these frames.
- **EMAC_TS_PROCESS_ETHERNET** processes PTP packets encapsulated directly in Ethernet frames. If absent, the MAC ignores these frames.
- **EMAC_TS_ALL_RX_FRAMES** enables timestamping for all frames received by the MAC, regardless of type.

The *ui32SubSecondInc* controls the rate at which the timestamp clock's subsecond count increments. Its meaning depends on which of **EMAC_TS_DIGITAL_ROLLOVER** or **EMAC_TS_BINARY_ROLLOVER** and **EMAC_TS_UPDATE_FINE** or **EMAC_TS_UPDATE_COARSE** were included in *ui32Config*.

The timestamp second counter is incremented each time the subsecond counter rolls over. In digital rollover mode, the subsecond counter acts as a simple 31-bit counter, rolling over to 0 after reaching 0x7FFFFFFF. In this case, each lsb of the subsecond counter represents 0.465 ns (assuming the definition of 1 second resolution for the seconds counter). When binary rollover mode is selected, the subsecond counter acts as a nanosecond counter and rolls over to 0 after reaching 999, 999, 999 making each lsb represent 1 nanosecond.

In coarse update mode, the timestamp subsecond counter is incremented by *ui32SubSecondInc* on each main oscillator clock tick. Setting *ui32SubSecondInc* to the main oscillator clock period in either 1 ns or 0.465 ns units ensures that the time stamp, read as seconds and subseconds, increments at the same rate as the main oscillator clock. For example, if the main oscillator is 25 MHz, *ui32SubSecondInc* is set to 40 if digital rollover mode is selected or $(40 / 0.465) = 86$ in binary rollover mode.

In fine update mode, the subsecond increment value must be set according to the desired accuracy of the recovered IEEE 1588 clock which must be lower than the system clock rate. Fine update mode is typically used when synchronizing the local clock to the IEEE 1588 master clock. The subsecond counter is incremented by *ui32SubSecondInc* counts each time a 32-bit accumulator register generates a carry. The accumulator register is incremented by the addend value on each main oscillator tick and this addend value is modified to allow fine control over the rate of change of the timestamp counter. The addend value is calculated using the ratio of the main oscillator clock rate and the desired IEEE 1588 clock rate and the *ui32SubSecondInc* value is set to correspond to the desired IEEE 1588 clock rate. As an example, using digital rollover mode and a 25-MHz main oscillator clock with a desired IEEE 1588 clock accuracy of 12.5 MHz, we would set *ui32SubSecondInc* to the 12.5-MHz clock period of 80 ns and set the initial addend value to 0x80000000 to generate a carry on every second system clock.

See also:

[EMACTimestampAddendSet\(\)](#)

Returns:

None.

10.2.4.46 EMACTimestampDisable

Disables packet timestamping and stops the system clock.

Prototype:

```
void  
EMACTimestampDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function is used to stop the system clock used to timestamp Ethernet frames and to disable timestamping.

Returns:

None.

10.2.4.47 EMACTimestampEnable

Enables packet timestamping and starts the system clock running.

Prototype:

```
void  
EMACTimestampEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function is used to enable the system clock used to timestamp Ethernet frames and to enable that timestamping.

Returns:

None.

10.2.4.48 EMACTimestampIntStatus

Reads the status of the Ethernet system time interrupt.

Prototype:

```
uint32_t  
EMACTimestampIntStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

When an Ethernet interrupt occurs and **EMAC_INT_TIMESTAMP** is reported by [EMACIntStatus\(\)](#), this function must be called to read and clear the timer interrupt status.

Returns:

The return value is the logical OR of the values **EMAC_TS_INT_TS_SEC_OVERFLOW** and **EMAC_TS_INT_TARGET_REACHED**.

- **EMAC_TS_INT_TS_SEC_OVERFLOW** indicates that the second counter in the hardware timer has rolled over.
- **EMAC_TS_INT_TARGET_REACHED** indicates that the system time incremented past the value set in an earlier call to [EMACTimestampTargetSet\(\)](#). When this occurs, a new target time may be set and the interrupt re-enabled using calls to [EMACTimestampTargetSet\(\)](#) and [EMACTimestampTargetIntEnable\(\)](#).

10.2.4.49 EMACTimestampPPSCommand

Sends a command to control the PPS output from the Ethernet MAC.

Prototype:

```
void
EMACTimestampPPSCommand(uint32_t ui32Base,
                        uint8_t ui8Cmd)
```

Parameters:

ui32Base is the base address of the controller.

ui8Cmd identifies the command to be sent.

Description:

This function may be used to send a command to the MAC PPS (Pulse Per Second) controller when it is operating in command mode. Command mode is selected by calling [EMACTimestampPPSCommandModeSet\(\)](#). Valid commands are as follow:

- **EMAC_PPS_COMMAND_NONE** indicates no command.
- **EMAC_PPS_COMMAND_START_SINGLE** indicates that a single high pulse should be generated when the system time reaches the current target time.
- **EMAC_PPS_COMMAND_START_TRAIN** indicates that a train of pulses should be started when the system time reaches the current target time.
- **EMAC_PPS_COMMAND_CANCEL_START** cancels any pending start command if the system time has not yet reached the programmed target time.
- **EMAC_PPS_COMMAND_STOP_AT_TIME** indicates that the current pulse train should be stopped when the system time reaches the current target time.
- **EMAC_PPS_COMMAND_STOP_NOW** indicates that the current pulse train should be stopped immediately.
- **EMAC_PPS_COMMAND_CANCEL_STOP** cancels any pending stop command if the system time has not yet reached the programmed target time.

In all cases, the width of the pulses generated is governed by the *ui32Width* parameter passed to [EMACTimestampPPSPeriodSet\(\)](#). If a command starts a train of pulses, the period of the pulses is governed by the *ui32Period* parameter passed to the same function. Target times associated with PPS commands are set by calling [EMACTimestampTargetSet\(\)](#).

Returns:

None.

10.2.4.50 EMACTimestampPPSCommandModeSet

Configures the Ethernet MAC PPS output in command mode.

Prototype:

```
void  
EMACTimestampPPSCommandModeSet(uint32_t ui32Base,  
                                uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the controller.

ui32Config determines how the system target time is used.

Description:

The simple mode of operation offered by the PPS (Pulse Per Second) engine may be too restrictive for some applications. The second mode, however, allows complex pulse trains to be generated using commands that tell the engine to send individual pulses or start and stop trains if pulses. In this mode, the pulse width and period may be set arbitrarily based on ticks of the clock used to update the system time. Commands are triggered at specific times using the target time last set using a call to [EMACTimestampTargetSet\(\)](#).

The *ui32Config* parameter may be used to control whether the target time is used to trigger commands only or can also generate an interrupt to the CPU. Valid values are:

- **EMAC_PPS_TARGET_INT** configures the target time to only raise an interrupt and not to trigger any pending PPS command.
- **EMAC_PPS_TARGET_PPS** configures the target time to trigger a pending PPS command but not raise an interrupt.
- **EMAC_PPS_TARGET_BOTH** configures the target time to trigger any pending PPS command and also raise an interrupt.

To use command mode, an application must call this function to enable the mode, then call:

- [EMACTimestampPPSPeriodSet\(\)](#) to set the desired pulse width and period then
- [EMACTimestampTargetSet\(\)](#) to set the time at which the next command is executed, and finally
- [EMACTimestampPPSCommand\(\)](#) to send a command to cause the pulse or pulse train to be started at the required time.

Returns:

None.

10.2.4.51 EMACTimestampPPSPeriodSet

Sets the period and width of the pulses on the Ethernet MAC PPS output.

Prototype:

```
void  
EMACTimestampPPSPeriodSet(uint32_t ui32Base,  
                           uint32_t ui32Period,  
                           uint32_t ui32Width)
```

Parameters:

ui32Base is the base address of the controller.

ui32Period is the period of the PPS output expressed in terms of system time update ticks.

ui32Width is the width of the high portion of the PPS output expressed in terms of system time update ticks.

Description:

This function may be used to control the period and duty cycle of the signal output on the Ethernet MAC PPS pin when the PPS generator is operating in command mode and a command to send one or more pulses has been executed. Command mode is selected by calling [EMACTimestampPPSCommandModeSet\(\)](#).

In simple mode, the PPS output signal frequency is controlled by the ***ui32FreqConfig*** parameter passed to [EMACTimestampPPSSimpleModeSet\(\)](#).

The ***ui32Period*** and ***ui32Width*** parameters are expressed in terms of system time update ticks. When the system time is operating in coarse update mode, each tick is equivalent to the system clock. In fine update mode, a tick occurs every time the 32-bit system time accumulator overflows and this, in turn, is determined by the value passed to the function [EMACTimestampAddendSet\(\)](#). Regardless of the tick source, each tick increments the actual system time, queried using [EMACTimestampSysTimeGet\(\)](#) by the subsecond increment value passed in the ***ui32SubSecondInc*** to [EMACTimestampConfigSet\(\)](#).

Returns:

None.

10.2.4.52 EMACTimestampPPSSimpleModeSet

Configures the Ethernet MAC PPS output in simple mode.

Prototype:

```
void
EMACTimestampPPSSimpleModeSet(uint32_t ui32Base,
                             uint32_t ui32FreqConfig)
```

Parameters:

ui32Base is the base address of the controller.

ui32FreqConfig determines the frequency of the output generated on the PPS pin.

Description:

This function configures the Ethernet MAC PPS (Pulse Per Second) engine to operate in its simple mode which allows the generation of a few, fixed frequencies and pulse widths on the PPS pin. If more complex pulse train generation is required, the MAC also provides a command-based PPS control mode that can be selected by calling [EMACTimestampPPSCommandModeSet\(\)](#).

The ***ui32FreqConfig*** parameter may take one of the following values:

- **EMAC_PPS_SINGLE_PULSE** generates a single high pulse on the PPS output once per second. The pulse width is the same as the system clock period.
- **EMAC_PPS_1HZ** generates a 1Hz signal on the PPS output. This option is not available if the system time subsecond counter is currently configured to operate in binary rollover mode.

- **EMAC_PPS_2HZ**, **EMAC_PPS_4HZ**, **EMAC_PPS_8HZ**, **EMAC_PPS_16HZ**, **EMAC_PPS_32HZ**, **EMAC_PPS_64HZ**, **EMAC_PPS_128HZ**, **EMAC_PPS_256HZ**, **EMAC_PPS_512HZ**, **EMAC_PPS_1024HZ**, **EMAC_PPS_2048HZ**, **EMAC_PPS_4096HZ**, **EMAC_PPS_8192HZ**, **EMAC_PPS_16384HZ** generate the requested frequency on the PPS output in both binary and digital rollover modes.
- **EMAC_PPS_32768HZ** generates a 32KHz signal on the PPS output. This option is not available if the system time subsecond counter is currently configured to operate in digital rollover mode.

Except when **EMAC_PPS_SINGLE_PULSE** is specified, the signal generated on PPS has a duty cycle of 50% when binary rollover mode is used for the system time subsecond count. In digital mode, the output frequency averages the value requested and is resynchronized each second. For example, if **EMAC_PPS_4HZ** is selected in digital rollover mode, the output generates three clocks with 50 percent duty cycle and 268 ms period followed by a fourth clock of 195 ms period, 134 ms low and 61 ms high.

Returns:

None.

10.2.4.53 EMACTimestampSysTimeGet

Gets the current system time.

Prototype:

```
void  
EMACTimestampSysTimeGet(uint32_t ui32Base,  
                        uint32_t *pui32Seconds,  
                        uint32_t *pui32SubSeconds)
```

Parameters:

ui32Base is the base address of the controller.

pui32Seconds points to storage for the current seconds value.

pui32SubSeconds points to storage for the current subseconds value.

Description:

This function may be used to get the current system time.

The meaning of *ui32SubSeconds* depends on the current system time configuration. If [EMACTimestampConfigSet\(\)](#) was previously called with the *EMAC_TS_DIGITAL_ROLLOVER* configuration option, each bit in the *ui32SubSeconds* value represents 1 ns. If *EMAC_TS_BINARY_ROLLOVER* was specified instead, a *ui32SubSeconds* bit represents 0.46 ns.

Returns:

None.

10.2.4.54 EMACTimestampSysTimeSet

Sets the current system time.

Prototype:

```
void
EMACTimestampSysTimeSet(uint32_t ui32Base,
                        uint32_t ui32Seconds,
                        uint32_t ui32SubSeconds)
```

Parameters:

ui32Base is the base address of the controller.

ui32Seconds is the seconds value of the new system clock setting.

ui32SubSeconds is the subseconds value of the new system clock setting.

Description:

This function may be used to set the current system time. The system clock is set to the value passed in the *ui32Seconds* and *ui32SubSeconds* parameters.

The meaning of *ui32SubSeconds* depends on the current system time configuration. If [EMACTimestampConfigSet\(\)](#) was previously called with the *EMAC_TS_DIGITAL_ROLLOVER* configuration option, each bit in the *ui32SubSeconds* value represents 1 ns. If *EMAC_TS_BINARY_ROLLOVER* was specified instead, a *ui32SubSeconds* bit represents 0.46 ns.

Returns:

None.

10.2.4.55 EMACTimestampSysTimeUpdate

Adjusts the current system time upwards or downwards by a given amount.

Prototype:

```
void
EMACTimestampSysTimeUpdate(uint32_t ui32Base,
                           uint32_t ui32Seconds,
                           uint32_t ui32SubSeconds,
                           bool bInc)
```

Parameters:

ui32Base is the base address of the controller.

ui32Seconds is the seconds value of the time update to apply.

ui32SubSeconds is the subseconds value of the time update to apply.

bInc defines the direction of the update.

Description:

This function may be used to adjust the current system time either upwards or downwards by a given amount. The size of the adjustment is given by the *ui32Seconds* and *ui32SubSeconds* parameter and the direction by the *bInc* parameter. When *bInc* is *true*, the system time is advanced by the interval given. When it is *false*, the time is retarded by the interval.

The meaning of *ui32SubSeconds* depends on the current system time configuration. If [EMAC-TimestampConfigSet\(\)](#) was previously called with the *EMAC_TS_DIGITAL_ROLLOVER* configuration option, each bit in the subsecond value represents 1 ns. If *EMAC_TS_BINARY_ROLLOVER* was specified instead, a subsecond bit represents 0.46 ns.

Returns:

None.

10.2.4.56 EMACTimestampTargetIntDisable

Disables the Ethernet system time interrupt.

Prototype:

```
void  
EMACTimestampTargetIntDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be used to disable any pending Ethernet system time interrupt previously scheduled using calls to [EMACTimestampTargetSet\(\)](#) and [EMACTimestampTargetIntEnable\(\)](#).

Returns:

None.

10.2.4.57 EMACTimestampTargetIntEnable

Enables the Ethernet system time interrupt.

Prototype:

```
void  
EMACTimestampTargetIntEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be used after [EMACTimestampTargetSet\(\)](#) to schedule an interrupt at some future time. The time reference for the function is the IEEE 1588 time as returned by [EMAC-TimestampSysTimeGet\(\)](#). To generate an interrupt when the system time exceeds a given value, call this function to set the desired time, then [EMACTimestampTargetIntEnable\(\)](#) to enable the interrupt. When the system time increments past the target time, an Ethernet interrupt with status **EMAC_INT_TIMESTAMP** is generated.

Returns:

None.

10.2.4.58 EMACTimestampTargetSet

Sets the target system time at which the next Ethernet timer interrupt is generated.

Prototype:

```
void  
EMACTimestampTargetSet(uint32_t ui32Base,  
                      uint32_t ui32Seconds,  
                      uint32_t ui32SubSeconds)
```

Parameters:

ui32Base is the base address of the controller.

ui32Seconds is the second value of the desired target time.

ui32SubSeconds is the subseconds value of the desired target time.

Description:

This function may be used to schedule an interrupt at some future time. The time reference for the function is the IEEE 1588 time as returned by [EMACTimestampSysTimeGet\(\)](#). To generate an interrupt when the system time exceeds a given value, call this function to set the desired time, then [EMACTimestampTargetIntEnable\(\)](#) to enable the interrupt. When the system time increments past the target time, an Ethernet interrupt with status **EMAC_INT_TIMESTAMP** is generated.

The accuracy of the interrupt timing depends on the Ethernet timer update frequency and the subsecond increment value currently in use. The interrupt is generated on the first timer increment that causes the system time to be greater than or equal to the target time set.

Returns:

None.

10.2.4.59 EMACTxDisable

Disables the Ethernet controller transmitter.

Prototype:

```
void  
EMACTxDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

When terminating operations on the Ethernet interface, this function should be called. This function disables the transmitter.

Returns:

None.

10.2.4.60 EMACTxDMACurrentBufferGet

Returns the current DMA transmit buffer pointer.

Prototype:

```
uint8_t *  
EMACTxDMACurrentBufferGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be called to determine which buffer the transmit DMA engine is currently reading from.

Returns:

Returns the transmit buffer address currently being read by the DMA engine.

10.2.4.61 EMACTxDMACurrentDescriptorGet

Returns the current DMA transmit descriptor pointer.

Prototype:

```
tEMACDMADescriptor *  
EMACTxDMACurrentDescriptorGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function returns a pointer to the current Ethernet transmit descriptor read by the DMA.

Returns:

Returns a pointer to the start of the current transmit DMA descriptor.

10.2.4.62 EMACTxDMADescriptorListGet

Returns a pointer to the start of the DMA transmit descriptor list.

Prototype:

```
tEMACDMADescriptor *  
EMACTxDMADescriptorListGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function returns a pointer to the head of the Ethernet MAC's transmit DMA descriptor list.

This value corresponds to the pointer originally set using a call to [EMACTxDMADescriptorList-Set\(\)](#).

Returns:

Returns a pointer to the start of the DMA transmit descriptor list.

10.2.4.63 EMACTxDMADescriptorListSet

Sets the DMA transmit descriptor list pointer.

Prototype:

```
void  
EMACTxDMADescriptorListSet(uint32_t ui32Base,  
                           tEMACDMADescriptor *pDescriptor)
```

Parameters:

ui32Base is the base address of the controller.

pDescriptor points to the first DMA descriptor in the list to be passed to the transmit DMA engine.

Description:

This function sets the Ethernet MAC's transmit DMA descriptor list pointer. The *pDescriptor* pointer must point to one or more descriptor structures.

When multiple descriptors are provided, they can be either chained or unchained. Chained descriptors are indicated by setting the **DES0_TX_CTRL_CHAINED** or **DES1_RX_CTRL_CHAINED** bit in the relevant word of the transmit or receive descriptor. If this bit is clear, unchained descriptors are assumed.

Chained descriptors use a link pointer in each descriptor to point to the next descriptor in the chain.

Unchained descriptors are assumed to be contiguous in memory with a consistent offset between the start of one descriptor and the next. If unchained descriptors are used, the *pvLink* field in the descriptor becomes available to store a second buffer pointer, allowing each descriptor to point to two buffers rather than one. In this case, the *ui32DescSkipSize* parameter to [EMACInit\(\)](#) must previously have been set to the number of words between the end of one descriptor and the start of the next. This value must be 0 in cases where a packed array of [**tEMACDMADescriptor**](#) structures is used. If the application wishes to add new state fields to the end of the descriptor structure, the skip size should be set to accommodate the newly sized structure.

Applications are responsible for initializing all descriptor fields appropriately before passing the descriptor list to the hardware.

Returns:

None.

10.2.4.64 EMACTxDMAPollDemand

Orders the MAC DMA controller to attempt to acquire the next transmit descriptor.

Prototype:

```
void
EMACTxDMAPollDemand(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the Ethernet controller.

Description:

This function must be called to restart the transmitter if it has been suspended due to the current transmit DMA descriptor being owned by the host. Once the application writes new values to the descriptor and marks it as being owned by the MAC DMA, this function causes the hardware to attempt to acquire the descriptor and start transmission of the new data.

Returns:

None.

10.2.4.65 EMACTxEnable

Enables the Ethernet controller transmitter.

Prototype:

```
void  
EMACTxEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

When starting operations on the Ethernet interface, this function should be called to enable the transmitter after all configuration has been completed.

Returns:

None.

10.2.4.66 EMACTxFlush

Flushes the Ethernet controller transmit FIFO.

Prototype:

```
void  
EMACTxFlush(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function flushes any data currently held in the Ethernet transmit FIFO. Data that has already been passed to the MAC for transmission is transmitted, possibly resulting in a transmit underflow or runt frame transmission.

Returns:

None.

10.2.4.67 EMACVLANHashFilterBitCalculate

Returns the bit number to set in the VLAN hash filter corresponding to a given tag.

Prototype:

```
uint32_t  
EMACVLANHashFilterBitCalculate(uint16_t ui16Tag)
```

Parameters:

ui16Tag is the VLAN tag for which the hash filter bit number is to be determined.

Description:

This function may be used to determine which bit in the VLAN hash filter to set to describe a given 12- or 16-bit VLAN tag. The returned value is a 4-bit value indicating the bit number to set within the 16-bit VLAN hash filter. For example, if 0x02 is returned, this indicates that bit 2 of the hash filter must be set to pass the supplied VLAN tag.

Returns:

Returns the bit number to set in the VLAN hash filter to describe the passed tag.

10.2.4.68 EMACVLANHashFilterGet

Returns the current value of the hash filter used to control reception of VLAN-tagged frames.

Prototype:

```
uint32_t
EMACVLANHashFilterGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function allows the current VLAN tag hash filter value to be returned. Additional VLAN tags may be added to this filter by setting the appropriate bits, determined by calling [EMACVLANHashFilterBitCalculate\(\)](#), and then calling [EMACVLANHashFilterSet\(\)](#) to set the new filter value.

Returns:

Returns the current value of the VLAN hash filter.

10.2.4.69 EMACVLANHashFilterSet

Sets the hash filter used to control reception of VLAN-tagged frames.

Prototype:

```
void
EMACVLANHashFilterSet(uint32_t ui32Base,
                      uint32_t ui32Hash)
```

Parameters:

ui32Base is the base address of the controller.

ui32Hash is the hash filter value to set.

Description:

This function allows the VLAG tag hash filter to be set. By using hash filtering, several different VLAN tags can be filtered very easily at the cost of some false positive results that must be removed by software.

The hash filter value passed in *ui32Hash* may be built up by calling [EMACVLANHashFilterBitCalculate\(\)](#) for each VLAN tag that is to pass the filter and then set each of the bits for which the numbers are returned by that function. Care must be taken when clearing bits in the hash filter due to the fact that there is a many-to-one correspondence between VLAN tags and hash filter bits.

Returns:

None

10.2.4.70 EMACVLANRxConfigGet

Returns the currently-set options related to reception of VLAN-tagged frames.

Prototype:

```
uint32_t  
EMACVLANRxConfigGet(uint32_t ui32Base,  
                      uint16_t *pui16Tag)
```

Parameters:

ui32Base is the base address of the controller.

pui16Tag points to storage which is written with the currently configured VLAN tag used for perfect filtering.

Description:

This function returns information on how the receiver is currently handling IEEE 802.1Q VLAN-tagged frames.

See also:

[EMACVLANRxConfigSet\(\)](#)

Returns:

Returns flags defining how VLAN-tagged frames are handled. The value is a logical OR of the following flags:

- **EMAC_VLAN_RX_HASH_ENABLE** indicates that hash filtering is enabled for VLAN tags. If this flag is absent, perfect filtering using the tag returned in ***pui16Tag** is performed.
- **EMAC_VLAN_RX_SVLAN_ENABLE** indicates that the receiver recognizes S-VLAN (Type = 0x88A8) frames as valid VLAN-tagged frames. If absent, only frames with type 0x8100 are considered valid VLAN frames.
- **EMAC_VLAN_RX_INVERSE_MATCH** indicates that the receiver passes all VLAN frames for which the tags do not match the ***pui16Tag** value. If this flag is absent, only tagged frames matching ***pui16Tag** are passed.
- **EMAC_VLAN_RX_12BIT_TAG** indicates that the receiver is comparing only the bottom 12 bits of ***pui16Tag** when performing either perfect or hash filtering of VLAN frames. If this flag is absent, all 16 bits of the frame tag are examined when filtering. If this flag is set and ***pui16Tag** has all bottom 12 bits clear, the receiver passes all frames with types 0x8100 or 0x88A8 regardless of the tag values they contain.

10.2.4.71 EMACVLANRxConfigSet

Sets options related to reception of VLAN-tagged frames.

Prototype:

```
void  
EMACVLANRxConfigSet(uint32_t ui32Base,  
                      uint16_t ui16Tag,  
                      uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the controller.

ui16Tag is the IEEE 802.1Q VLAN tag expected for incoming frames.

ui32Config determines how the receiver handles VLAN-tagged frames.

Description:

This function configures the receiver's handling of IEEE 802.1Q VLAN tagged frames. Incoming tagged frames are filtered using either a perfect filter or a hash filter. When hash filtering is disabled, VLAN frames tagged with the value of *ui16Tag* pass the filter and all others are rejected. The tag comparison may involve all 16 bits or only the 12-bit VLAN ID portion of the tag.

The *ui32Config* parameter is a logical OR of the following values:

- **EMAC_VLAN_RX_HASH_ENABLE** enables hash filtering for VLAN tags. If this flag is absent, perfect filtering using the tag supplied in *ui16Tag* is performed. The hash filter may be set using [EMACVLANHashFilterSet\(\)](#), and [EMACVLANHashFilterBitCalculate\(\)](#) may be used to determine which bits to set in the filter for given VLAN tags.
- **EMAC_VLAN_RX_SVLAN_ENABLE** causes the receiver to recognize S-VLAN (Type = 0x88A8) frames as valid VLAN-tagged frames. If absent, only frames with type 0x8100 are considered valid VLAN frames.
- **EMAC_VLAN_RX_INVERSE_MATCH** causes the receiver to pass all VLAN frames for which the tags do not match the supplied *ui16Tag* value. If this flag is absent, only tagged frames matching *ui16Tag* are passed.
- **EMAC_VLAN_RX_12BIT_TAG** causes the receiver to compare only the bottom 12 bits of *ui16Tag* when performing either perfect or hash filtering of VLAN frames. If this flag is absent, all 16 bits of the frame tag are examined when filtering. If this flag is set and *ui16Tag* has all bottom 12 bits clear, the receiver passes all frames with types 0x8100 or 0x88A8 regardless of the tag values they contain.

Note:

To ensure that VLAN frames that fail the tag filter are dropped by the MAC, [EMACFrameFilterSet\(\)](#) must be called with the **EMAC_FRMFILTER_VLAN** flag set in the *ui32FilterOpts* parameter. If this flag is not set, failing VLAN packets are received by the application, but bit 10 of RDES0 (**EMAC_FRMFILTER_VLAN**) is clear indicating that the packet did not match the current VLAG tag filter.

See also:

[EMACVLANRxConfigGet\(\)](#)

Returns:

None

10.2.4.72 EMACVLANTxConfigGet

Returns currently-selected options related to transmission of VLAN-tagged frames.

Prototype:

```
uint32_t
EMACVLANTxConfigGet(uint32_t ui32Base,
                     uint16_t *pui16Tag)
```

Parameters:

ui32Base is the base address of the controller.

pui16Tag points to storage that is written with the VLAN tag currently being used for insertion or replacement.

Description:

This function returns information on the current settings related to VLAN tagging of transmitted frames.

See also:

[EMACVLANTxConfigSet\(\)](#)

Returns:

Returns flags describing the current VLAN configuration relating to frame transmission. The return value is a logical OR of the following values:

- **EMAC_VLAN_TX_SVLAN** indicates that the S-VLAN type (0x88A8) is being used when inserting or replacing tags in transmitted frames. If this label is absent, C-VLAN type (0x8100) is being used.
- **EMAC_VLAN_TX_USE_VLC** indicates that the transmitter is processing VLAN frames according to the VLAN control (VLC) value returned here. If this tag is absent, VLAN handling is controlled by fields in the transmit descriptor.

If **EMAC_VLAN_TX_USE_VLC** is returned, one of the following four labels is also included to define the transmit VLAN tag handling. Note that this value may be extracted from the return value using the mask **EMAC_VLAN_TX_VLC_MASK**.

- **EMAC_VLAN_TX_VLC_NONE** indicates that the transmitter is not performing VLAN tag insertion, deletion or replacement.
- **EMAC_VLAN_TX_VLC_DELETE** indicates that the transmitter is removing VLAN tags from all transmitted frames which contain them.
- **EMAC_VLAN_TX_VLC_INSERT** indicates that the transmitter is inserting a VLAN type and tag into all outgoing frames regardless of whether or not they already contain a VLAN tag.
- **EMAC_VLAN_TX_VLC_REPLACE** indicates that the transmitter is replacing the VLAN tag in all transmitted frames of type 0x8100 or 0x88A8 with the value returned in **ui16Tag*.

10.2.4.73 EMACVLANTxConfigSet

Sets options related to transmission of VLAN-tagged frames.

Prototype:

```
void  
EMACVLANTxConfigSet(uint32_t ui32Base,  
                     uint16_t ui16Tag,  
                     uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the controller.

ui16Tag is the VLAN tag to be used when inserting or replacing tags in transmitted frames.

ui32Config determines the VLAN-related processing performed by the transmitter.

Description:

This function is used to configure transmitter options relating to IEEE 802.1Q VLAN tagging. The transmitter may be set to insert tagging into untagged frames or replace existing tags with new values.

The *ui16Tag* parameter contains the VLAN tag to be used in outgoing tagged frames. The *ui32Config* parameter is a logical OR of the following labels:

- **EMAC_VLAN_TX_SVLAN** uses the S-VLAN type (0x88A8) when inserting or replacing tags in transmitted frames. If this label is absent, C-VLAN type (0x8100) is used.
- **EMAC_VLAN_TX_USE_VLC** informs the transmitter that the VLAN tag handling should be defined by the VLAN control (VLC) value provided in this function call. If this tag is absent, VLAN handling is controlled by fields in the transmit descriptor.

If **EMAC_VLAN_TX_USE_VLC** is set, one of the following four labels must also be included to define the transmit VLAN tag handling:

- **EMAC_VLAN_TX_VLC_NONE** instructs the transmitter to perform no VLAN tag insertion, deletion or replacement.
- **EMAC_VLAN_TX_VLC_DELETE** instructs the transmitter to remove VLAN tags from all transmitted frames that contain them. As a result, bytes 13, 14, 15 and 16 are removed from all frames with types 0x8100 or 0x88A8.
- **EMAC_VLAN_TX_VLC_INSERT** instructs the transmitter to insert a VLAN type and tag into all outgoing frames regardless of whether or not they already contain a VLAN tag.
- **EMAC_VLAN_TX_VLC_REPLACE** instructs the transmitter to replace the VLAN tag in all frames of type 0x8100 or 0x88A8 with the value provided to this function in the *ui16Tag* parameter.

Returns:

None

10.3 Programming Example

The following example shows how to use the this API to initialize the Ethernet controller to transmit and receive packets. Note that this is a very much simplified example which shows only the basic flow required. A full implementation would contain rather more error checking and recovery code.

```
////////////////////////////////////////////////////////////////////////
//
// Ethernet DMA descriptors.
//
// The MAC hardware needs a minimum of 3 receive descriptors to operate. The
// number used will be application-dependent and should be tuned for best
// performance.
//
////////////////////////////////////////////////////////////////////////
#define NUM_TX_DESCRIPTOROS 3
#define NUM_RX_DESCRIPTOROS 3
tEMACDMADescriptor g_psRxDescriptor[NUM_TX_DESCRIPTOROS];
tEMACDMADescriptor g_psTxDescriptor[NUM_RX_DESCRIPTOROS];

uint32_t g_ui32RxDescIndex;
uint32_t g_ui32TxDescIndex;

////////////////////////////////////////////////////////////////////////
//
// Transmit and receive buffers. These will typically be allocated within your
// network stack somewhere.
//
////////////////////////////////////////////////////////////////////////
#define RX_BUFFER_SIZE 1536
uint8_t g_ppui8RxBuffer[NUM_RX_DESCRIPTOROS][RX_BUFFER_SIZE];
////////////////////////////////////////////////////////////////////////
```

```
//  
// Read a packet from the DMA receive buffer and return the number of bytes  
// read.  
//  
//*****  
int32_t  
ProcessReceivedPacket(void)  
{  
    int_fast32_t i32FrameLen;  
  
    //  
    // By default, we assume we got a bad frame.  
    //  
    i32FrameLen = 0;  
  
    //  
    // Make sure that we own the receive descriptor.  
    //  
    if(!(g_psRxDescriptor[g_ui32RxDescIndex].ui32CtrlStatus & DES0_RX_CTRL_OWN))  
    {  
        //  
        // We own the receive descriptor so check to see if it contains a valid  
        // frame.  
        //  
        if(!(g_psRxDescriptor[g_ui32RxDescIndex].ui32CtrlStatus &  
            DES0_RX_STAT_ERR))  
        {  
            //  
            // We have a valid frame. First check that the "last descriptor"  
            // flag is set. We sized the receive buffer such that it can  
            // always hold a valid frame so this flag should never be clear at  
            // this point but...  
            //  
            if(g_psRxDescriptor[g_ui32RxDescIndex].ui32CtrlStatus &  
                DES0_RX_STAT_LAST_DESC)  
            {  
                //  
                // What size is the received frame?  
                //  
                i32FrameLen =  
                    ((g_psRxDescriptor[g_ui32RxDescIndex].ui32CtrlStatus &  
                        DES0_RX_STAT_FRAME_LENGTH_M) >>  
                        DES0_RX_STAT_FRAME_LENGTH_S);  
  
                //  
                // Pass the received buffer up to the application to handle.  
                //  
                ApplicationProcessFrame(i32FrameLen,  
                    g_psRxDescriptor[g_ui32RxDescIndex].pvBuffer1);  
            }  
        }  
        //  
        // Now that we are finished dealing with this descriptor, hand  
        // it back to the hardware. Note that we assume  
        // ApplicationProcessFrame() is finished with the buffer at this point  
        // so it is safe to reuse.  
        //  
        g_psRxDescriptor[g_ui32RxDescIndex].ui32CtrlStatus =  
            DES0_RX_CTRL_OWN;  
  
        //  
        // Move on to the next descriptor in the chain.  
        //  
        g_ui32RxDescIndex++;
```

```

        if(g_ui32RxDescIndex == NUM_RX_DESCRIPTORS)
        {
            g_ui32RxDescIndex = 0;
        }
    }

    //
    // Return the Frame Length
    //
    return(i32FrameLen);
}

//*****
// The interrupt handler for the Ethernet interrupt.
//
//*****
void
EthernetIntHandler(void)
{
    uint32_t ui32Temp;

    //
    // Read and Clear the interrupt.
    //
    ui32Temp = EMACIntStatus(EMAC0_BASE, true);
    EMACIntClear(EMAC0_BASE, ui32Temp);

    //
    // Check to see if an RX Interrupt has occurred.
    //
    if(ui32Temp & EMAC_INT_RECEIVE)
    {
        //
        // Indicate that a packet has been received.
        //
        ProcessReceivedPacket();
    }
}

//*****
// Transmit a packet from the supplied buffer. This function would be called
// directly by the application. pui8Buf points to the Ethernet frame to send
// and i32BufLen contains the number of bytes in the frame.
//
//*****
static int32_t
PacketTransmit(uint8_t *pui8Buf, int32_t i32BufLen)
{
    //
    // Wait for the transmit descriptor to free up.
    //
    while(g_psTxDescriptor[g_ui32TxDescIndex].ui32CtrlStatus &
          DES0_TX_CTRL_OWN)
    {
        //
        // Spin and waste time.
        //
    }

    //
    // Move to the next descriptor.
    //
    g_ui32TxDescIndex++;
    if(g_ui32TxDescIndex == NUM_TX_DESCRIPTORS)

```

```
{  
    g_ui32TxDescIndex = 0;  
}  
  
//  
// Fill in the packet size and pointer, and tell the transmitter to start  
// work.  
//  
g_psTxDescriptor[g_ui32TxDescIndex].ui32Count = (uint32_t)i32BufLen;  
g_psTxDescriptor[g_ui32TxDescIndex].pvBufferl = pui8Buf;  
g_psTxDescriptor[g_ui32TxDescIndex].ui32CtrlStatus =  
    (DES0_TX_CTRL_LAST SEG | DES0_TX_CTRL_FIRST SEG |  
     DES0_TX_CTRL_INTERRUPT | DES0_TX_CTRL_IP_ALL_CKHSUMS |  
     DES0_TX_CTRL_CHAINED | DES0_TX_CTRL_OWN);  
  
//  
// Tell the DMA to reacquire the descriptor now that we've filled it in.  
// This call is benign if the transmitter hasn't stalled and checking  
// the state takes longer than just issuing a poll demand so we do this  
// for all packets.  
//  
EMACTxDMAPollDemand(EMAC0_BASE);  
  
//  
// Return the number of bytes sent.  
//  
return(i32BufLen);  
}  
  
//*****  
//  
// Initialize the transmit and receive DMA descriptors.  
//  
//*****  
void  
InitDescriptors(uint32_t ui32Base)  
{  
    uint32_t ui32Loop;  
  
    //  
    // Initialize each of the transmit descriptors. Note that we leave the  
    // buffer pointer and size empty and the OWN bit clear here since we have  
    // not set up any transmissions yet.  
    //  
    for(ui32Loop = 0; ui32Loop < NUM_TX_DESCRIPTOR; ui32Loop++)  
    {  
        g_psTxDescriptor[ui32Loop].ui32Count = DES1_TX_CTRL_SADDR_INSERT;  
        g_psTxDescriptor[ui32Loop].DES3.pLink =  
            (ui32Loop == (NUM_TX_DESCRIPTOR - 1)) ?  
                g_psTxDescriptor : &g_psTxDescriptor[ui32Loop + 1];  
        g_psTxDescriptor[ui32Loop].ui32CtrlStatus =  
            (DES0_TX_CTRL_LAST SEG | DES0_TX_CTRL_FIRST SEG |  
             DES0_TX_CTRL_INTERRUPT | DES0_TX_CTRL_CHAINED |  
             DES0_TX_CTRL_IP_ALL_CKHSUMS);  
    }  
  
    //  
    // Initialize each of the receive descriptors. We clear the OWN bit here  
    // to make sure that the receiver doesn't start writing anything  
    // immediately.  
    //  
    for(ui32Loop = 0; ui32Loop < NUM_RX_DESCRIPTOR; ui32Loop++)  
    {  
        g_psRxDescriptor[ui32Loop].ui32CtrlStatus = 0;  
        g_psRxDescriptor[ui32Loop].ui32Count =  
            (DES1_RX_CTRL_CHAINED |
```

```

        (RX_BUFFER_SIZE << DES1_RX_CTRL_BUFF1_SIZE_S));
g_psRxDescriptor[ui32Loop].pvBuffer1 = g_ppui8RxBuffer[ui32Loop];
g_psRxDescriptor[ui32Loop].DES3.pLink =
    (ui32Loop == (NUM_RX_DESCRIPTOR - 1)) ?
        g_psRxDescriptor : &g_psRxDescriptor[ui32Loop + 1];
}

//
// Set the descriptor pointers in the hardware.
//
EMACRxDMADescriptorListSet(ui32Base, g_psRxDescriptor);
EMACTxDMADescriptorListSet(ui32Base, g_psTxDescriptor);

//
// Start from the beginning of both descriptor chains. We actually set
// the transmit descriptor index to the last descriptor in the chain
// since it will be incremented before use and this means the first
// transmission we perform will use the correct descriptor.
//
g_ui32RxDescIndex = 0;
g_ui32TxDescIndex = NUM_TX_DESCRIPTOR - 1;
}

//*****
// This example demonstrates the use of the Ethernet Controller.
//
//*****
int
main(void)
{
    uint32_t ui32User0, ui32User1, ui32Loop, ui32SysClock;
    uint8_t ui8PHYAddr;
    uint8_t pui8MACAddr[6];

    //
    // Run from the PLL at 120 MHz.
    //
    ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN |
                                         SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

    //
    // Configure the device pins.
    //
    PinoutSet();

    //
    // Read the MAC address from the user registers.
    //
    FlashUserGet(&ui32User0, &ui32User1);
    if((ui32User0 == 0xffffffff) || (ui32User1 == 0xffffffff))
    {
        //
        // We should never get here. This is an error if the MAC address has
        // not been programmed into the device. Exit the program.
        //
        while(1)
        {
        }
    }

    //
    // Convert the 24/24 split MAC address from NV ram into a 32/16 split MAC
    // address needed to program the hardware registers, then program the MAC
}

```

```
// address into the Ethernet Controller registers.  
//  
pui8MACAddr[0] = ((ui32User0 >> 0) & 0xff);  
pui8MACAddr[1] = ((ui32User0 >> 8) & 0xff);  
pui8MACAddr[2] = ((ui32User0 >> 16) & 0xff);  
pui8MACAddr[3] = ((ui32User1 >> 0) & 0xff);  
pui8MACAddr[4] = ((ui32User1 >> 8) & 0xff);  
pui8MACAddr[5] = ((ui32User1 >> 16) & 0xff);  
  
//  
// Enable and reset the Ethernet modules.  
//  
SysCtlPeripheralEnable(SYSCTL_PERIPH_EMAC0);  
SysCtlPeripheralEnable(SYSCTL_PERIPH_EPHY0);  
SysCtlPeripheralReset(SYSCTL_PERIPH_EMAC0);  
SysCtlPeripheralReset(SYSCTL_PERIPH_EPHY0);  
  
//  
// Wait for the MAC to be ready.  
//  
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_EMAC0))  
{  
}  
  
//  
// Configure for use with the internal PHY.  
//  
ui8PHYAddr = 0;  
EMACPHYConfigSet(EMAC0_BASE,  
    (EMAC_PHY_TYPE_INTERNAL |  
     EMAC_PHY_INT_MDIX_EN |  
     EMAC_PHY_AN_100B_T_FULL_DUPLEX));  
  
//  
// Reset the MAC to latch the PHY configuration.  
//  
EMACReset(EMAC0_BASE);  
  
//  
// Initialize the MAC and set the DMA mode.  
//  
EMACInit(EMAC0_BASE, ui32SysClock,  
    EMAC_BCONFIG_MIXED_BURST | EMAC_BCONFIG_PRIORITY_FIXED, 4, 4,  
    0);  
  
//  
// Set MAC configuration options.  
//  
EMACConfigSet(EMAC0_BASE,  
    (EMAC_CONFIG_FULL_DUPLEX |  
     EMAC_CONFIG_CHECKSUM_OFFLOAD |  
     EMAC_CONFIG_7BYTE_PREAMBLE |  
     EMAC_CONFIG_IF_GAP_96BITS |  
     EMAC_CONFIG_USE_MACADDR0 |  
     EMAC_CONFIG_SA_FROM_DESCRIPTOR |  
     EMAC_CONFIG_BO_LIMIT_1024),  
    (EMAC_MODE_RX_STORE_FORWARD |  
     EMAC_MODE_TX_STORE_FORWARD |  
     EMAC_MODE_TX_THRESHOLD_64_BYTES |  
     EMAC_MODE_RX_THRESHOLD_64_BYTES), 0);  
  
//  
// Initialize the Ethernet DMA descriptors.  
//  
InitDescriptors(EMAC0_BASE);
```

```

//
// Program the hardware with its MAC address (for filtering).
//
EMACAddrSet(EMAC0_BASE, 0, pui8MACAddr);

//
// Wait for the link to become active.
//
while((EMACPHYRead(EMAC0_BASE, ui8PHYAddr, EPHY_BMSR) &
       EPHY_BMSR_LINKSTAT) == 0)
{
}

//
// Set MAC filtering options. We receive all broadcast and multicast
// packets along with those addressed specifically for us.
//
EMACFrameFilterSet(EMAC0_BASE, (EMAC_FRMFILTER_SADDR |
                                 EMAC_FRMFILTER_PASS_MULTICAST |
                                 EMAC_FRMFILTER_PASS_NO_CTRL));

//
// Clear any pending interrupts.
//
EMACIntClear(EMAC0_BASE, EMACIntStatus(EMAC0_BASE, false));

//
// Mark the receive descriptors as available to the DMA to start
// the receive processing.
//
for(ui32Loop = 0; ui32Loop < NUM_RX_DESCRIPTORs; ui32Loop++)
{
    g_psRxDescriptor[ui32Loop].ui32CtrlStatus |= DES0_RX_CTRL_OWN;
}

//
// Enable the Ethernet MAC transmitter and receiver.
//
EMACTxEnable(EMAC0_BASE);
EMACRxEnable(EMAC0_BASE);

//
// Enable the Ethernet interrupt.
//
IntEnable(INT_EMAC0);

//
// Enable the Ethernet RX Packet interrupt source.
//
EMACIntEnable(EMAC0_BASE, EMAC_INT_RECEIVE);

//
// Application main loop continues...
//
while(1)
{
    //
    // Do main loop things...
    //
}
}

```


11 External Peripheral Interface (EPI)

Introduction	201
API Functions	201
Programming Example	231

11.1 Introduction

The EPI API provides functions to use the EPI module available in the Tiva microcontroller. The EPI module provides a physical interface for external peripherals and memories. The EPI can be configured to support several types of external interfaces and different sized address and data buses.

Some features of the EPI module are:

- configurable interface modes including SDRAM, HostBus, and simple read/write protocols
- configurable address and data sizes
- configurable bus cycle timing
- blocking and non-blocking reads and writes
- FIFO for streaming reads
- interrupt and uDMA support

This driver is contained in `driverlib/epi.c`, with `driverlib/epi.h` containing the API declarations for use by applications.

11.2 API Functions

Functions

- void `EPIAddressMapSet` (uint32_t ui32Base, uint32_t ui32Map)
- void `EPICfgGPModeSet` (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32FrameCount, uint32_t ui32MaxWait)
- void `EPICfgHB16CSSet` (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32Config)
- void `EPICfgHB16Set` (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32MaxWait)
- void `EPICfgHB16TimingSet` (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32Config)
- void `EPICfgHB8CSSet` (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32Config)
- void `EPICfgHB8Set` (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32MaxWait)
- void `EPICfgHB8TimingSet` (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32Config)
- void `EPICfgSDRAMSet` (uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32Refresh)
- void `EPIDividerCSSet` (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32Divider)
- void `EPIDividerSet` (uint32_t ui32Base, uint32_t ui32Divider)
- void `EPIDMATxCount` (uint32_t ui32Base, uint32_t ui32Count)
- void `EPIFIFOConfig` (uint32_t ui32Base, uint32_t ui32Config)
- void `EPIIntDisable` (uint32_t ui32Base, uint32_t ui32IntFlags)

- void [EPIIntEnable](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [EPIIntErrorClear](#) (uint32_t ui32Base, uint32_t ui32ErrFlags)
- uint32_t [EPIIntErrorStatus](#) (uint32_t ui32Base)
- void [EPIIntRegister](#) (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t [EPIIntStatus](#) (uint32_t ui32Base, bool bMasked)
- void [EPIIntUnregister](#) (uint32_t ui32Base)
- void [EPIModeSet](#) (uint32_t ui32Base, uint32_t ui32Mode)
- uint32_t [EPINonBlockingReadAvail](#) (uint32_t ui32Base)
- void [EPINonBlockingReadConfigure](#) (uint32_t ui32Base, uint32_t ui32Channel, uint32_t ui32DataSize, uint32_t ui32Address)
- uint32_t [EPINonBlockingReadCount](#) (uint32_t ui32Base, uint32_t ui32Channel)
- uint32_t [EPINonBlockingReadGet16](#) (uint32_t ui32Base, uint32_t ui32Count, uint16_t *pui16Buf)
- uint32_t [EPINonBlockingReadGet32](#) (uint32_t ui32Base, uint32_t ui32Count, uint32_t *pui32Buf)
- uint32_t [EPINonBlockingReadGet8](#) (uint32_t ui32Base, uint32_t ui32Count, uint8_t *pui8Buf)
- void [EPINonBlockingReadStart](#) (uint32_t ui32Base, uint32_t ui32Channel, uint32_t ui32Count)
- void [EPINonBlockingReadStop](#) (uint32_t ui32Base, uint32_t ui32Channel)
- uint32_t [EPIPSRAMConfigRegGet](#) (uint32_t ui32Base, uint32_t ui32CS)
- bool [EPIPSRAMConfigRegGetNonBlocking](#) (uint32_t ui32Base, uint32_t ui32CS, uint32_t *pui32CR)
- void [EPIPSRAMConfigRegRead](#) (uint32_t ui32Base, uint32_t ui32CS)
- void [EPIPSRAMConfigRegSet](#) (uint32_t ui32Base, uint32_t ui32CS, uint32_t ui32CR)
- uint8_t [EPIWorkaroundByteRead](#) (uint8_t *pui8Addr)
- void [EPIWorkaroundByteWrite](#) (uint8_t *pui8Addr, uint8_t ui8Value)
- uint16_t [EPIWorkaroundHWordRead](#) (uint16_t *pui16Addr)
- void [EPIWorkaroundHWordWrite](#) (uint16_t *pui16Addr, uint16_t ui16Value)
- uint32_t [EPIWorkaroundWordRead](#) (uint32_t *pui32Addr)
- void [EPIWorkaroundWordWrite](#) (uint32_t *pui32Addr, uint32_t ui32Value)
- uint32_t [EPIWriteFIFOCountGet](#) (uint32_t ui32Base)

11.2.1 Detailed Description

The function [EPIModeSet\(\)](#) is used to select the interface mode. The clock divider is set with the [EPIDividerSet\(\)](#) function which determines the speed of the external bus. The external device is mapped into the processor memory or peripheral space using the [EPIAddressMapSet\(\)](#) function.

Once the mode is selected, the interface is configured with one of the configuration functions. If SDRAM mode is chosen, then the function [EPIConfigSDRAMSet\(\)](#) is used to configure the SDRAM interface. If Host-Bus 8 mode is chosen, then [EPIConfigHB8Set\(\)](#) is used. If Host-Bus 16 mode is chosen, then [EPIConfigHB16Set\(\)](#) is used. If General-Purpose mode is chosen, then [EPIConfigGPMODE\(\)](#) is used.

After the mode has been selected and configured, then the device can be accessed by reading and writing to the memory or peripheral address space that was programmed with [EPIAddressMapSet\(\)](#).

There are more sophisticated ways to use the read/write interface. When an application is writing to the mapped memory or peripheral space, the writes stall the processor until the write to the external interface is completed. However, the EPI contains an internal transaction FIFO and can buffer up to 4 pending writes without stalling the processor. Prior to writing, the application can test to see if the EPI can take more write operations without stalling the processor by using the function `EPINonBlockingWriteCount()`, which returns the number of non-blocking writes that can be made.

For efficient reads from the external device, the EPI contains a programmable read FIFO. After setting a starting address and a count, data from sequential reads from the device can be stored in the FIFO. The application can then periodically drain the FIFO by polling or by interrupts, optionally using the uDMA controller. A non-blocking read is configured by using the function `EPINonBlockingReadConfigure()`. The read operation is started with `EPINonBlockingReadStart()` and can be stopped by calling `EPINonBlockingReadStop()`. The function `EPINonBlockingReadCount()` can be used to determine the number of items remaining to be read, while the function `EPINonBlockingReadAvail()` returns the number of items in the FIFO that can be read immediately without stalling. There are 3 functions available for reading data from the FIFO and into a buffer provided by the application. These functions are `EPINonBlockingReadGet32()`, `EPINonBlockingReadGet16()`, `EPINonBlockingReadGet8()`, to read the data from the FIFO as 32-bit, 16-bit, or 8-bit data items.

The read FIFO and write transaction FIFO can be configured with the function `EPIFIFOConfig()`. This function is used to set the FIFO trigger levels and to enable error interrupts to be generated when a read or write is stalled.

Interrupts are enabled or disabled with the functions `EPIIntEnable()` and `EPIIntDisable()`. The interrupt status can be read by calling `EPIIntStatus()`. If there is an error interrupt pending, the cause of the error can be determined with the function `EPIIntErrorStatus()`. The error can then be cleared with `EPIIntErrorClear()`.

If dynamic interrupt registration is being used by the application, then an EPI interrupt handler can be registered by calling `EPIIntRegister()`. This function loads the interrupt handler's address into the vector table. The handler can be removed with `EPIIntUnregister()`.

11.2.2 Function Documentation

11.2.2.1 EPIAddressMapSet

Configures the address map for the external interface.

Prototype:

```
void
EPIAddressMapSet (uint32_t ui32Base,
                  uint32_t ui32Map)
```

Parameters:

ui32Base is the EPI module base address.

ui32Map is the address mapping configuration.

Description:

This function is used to configure the address mapping for the external interface, which then determines the base address of the external memory or device within the processor peripheral and/or memory space.

The parameter *ui32Map* is the logical OR of the following:

- Peripheral address space size, select one of:
 - **EPI_ADDR_PER_SIZE_256B** sets the peripheral address space to 256 bytes.
 - **EPI_ADDR_PER_SIZE_64KB** sets the peripheral address space to 64 Kbytes.
 - **EPI_ADDR_PER_SIZE_16MB** sets the peripheral address space to 16 Mbytes.
 - **EPI_ADDR_PER_SIZE_256MB** sets the peripheral address space to 256 Mbytes.
- Peripheral base address, select one of:
 - **EPI_ADDR_PER_BASE_NONE** sets the peripheral base address to none.
 - **EPI_ADDR_PER_BASE_A** sets the peripheral base address to 0xA0000000.
 - **EPI_ADDR_PER_BASE_C** sets the peripheral base address to 0xC0000000.
- RAM address space, select one of:
 - **EPI_ADDR_RAM_SIZE_256B** sets the RAM address space to 256 bytes.
 - **EPI_ADDR_RAM_SIZE_64KB** sets the RAM address space to 64 Kbytes.
 - **EPI_ADDR_RAM_SIZE_16MB** sets the RAM address space to 16 Mbytes.
 - **EPI_ADDR_RAM_SIZE_256MB** sets the RAM address space to 256 Mbytes.
- RAM base address, select one of:
 - **EPI_ADDR_RAM_BASE_NONE** sets the RAM space address to none.
 - **EPI_ADDR_RAM_BASE_6** sets the RAM space address to 0x60000000.
 - **EPI_ADDR_RAM_BASE_8** sets the RAM space address to 0x80000000.
- **EPI_ADDR_RAM_QUAD_MODE** maps CS0n to 0x60000000, CS1n to 0x80000000, CS2n to 0xA0000000, and CS3n to 0xC0000000.
- **EPI_ADDR_CODE_SIZE_256B** sets an external code size of 256 bytes, range 0x00 to 0xFF.
- **EPI_ADDR_CODE_SIZE_64KB** sets an external code size of 64 Kbytes, range 0x0000 to 0xFFFF.
- **EPI_ADDR_CODE_SIZE_16MB** sets an external code size of 16 Mbytes, range 0x000000 to 0xFFFFFFF.
- **EPI_ADDR_CODE_SIZE_256MB** sets an external code size of 256 Mbytes, range 0x00000000 to 0xFFFFFFFF.
- **EPI_ADDR_CODE_BASE_NONE** sets external code base to not mapped.
- **EPI_ADDR_CODE_BASE_1** sets external code base to 0x10000000.

Note:

The availability of **EPI_ADDR_RAM_QUAD_MODE** and **EPI_ADDR_CODE_*** varies based on the Tiva part in use. Please consult the data sheet to determine if these features are available.

Returns:

None.

11.2.2.2 EPIConfigGPModeSet

Configures the interface for general-purpose mode operation.

Prototype:

```
void  
EPIConfigGPModeSet (uint32_t ui32Base,  
                     uint32_t ui32Config,  
                     uint32_t ui32FrameCount,  
                     uint32_t ui32MaxWait)
```

Parameters:

ui32Base is the EPI module base address.

ui32Config is the interface configuration.

ui32FrameCount is the frame size in clocks, if the frame signal is used (0-15).

ui32MaxWait is currently not used.

Description:

This function is used to configure the interface when used in general-purpose operation as chosen with the function [EPIModeSet\(\)](#). The parameter ***ui32Config*** is the logical OR of the following:

- **EPI_GPMODE_CLKPIN** interface clock as output on a pin.
- **EPI_GPMODE_CLKGATE** clock is stopped when there is no transaction, otherwise it is free-running.
- **EPI_GPMODE_FRAME50** framing signal is 50/50 duty cycle, otherwise it is a pulse.
- **EPI_GPMODE_WRITE2CYCLE** a two-cycle write is used, otherwise a single-cycle write is used.
- Address bus size, select one of:
 - **EPI_GPMODE_ASIZE_NONE** sets no address bus.
 - **EPI_GPMODE_ASIZE_4** sets an address bus size of 4 bits.
 - **EPI_GPMODE_ASIZE_12** sets an address bus size of 12 bits.
 - **EPI_GPMODE_ASIZE_20** sets an address bus size of 20 bits.
- Data bus size, select one of:
 - **EPI_GPMODE_DSIZ_8** sets a data bus size of 8 bits.
 - **EPI_GPMODE_DSIZ_16** sets a data bus size of 16 bits.
 - **EPI_GPMODE_DSIZ_24** sets a data bus size of 24 bits.
 - **EPI_GPMODE_DSIZ_32** sets a data bus size of 32 bits.

The parameter ***ui32FrameCount*** is the number of clocks used to form the framing signal, if the framing signal is used. The behavior depends on whether the frame signal is a pulse or a 50/50 duty cycle.

Returns:

None.

11.2.2.3 EPICConfigHB16CSSet

Sets the individual chip select configuration for the Host-bus 16 interface.

Prototype:

```
void
EPICConfigHB16CSSet (uint32_t ui32Base,
                      uint32_t ui32CS,
                      uint32_t ui32Config)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select value to configure.

ui32Config is the configuration settings.

Description:

This function is used to configure individual chip select settings for the Host-bus 16 interface mode. [EPIConfigHB16Set\(\)](#) must have been set up with the **EPI_HB16_CSBAUD** flag for the individual chip select configuration option to be available.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The parameter *ui32Config* is the logical OR the following:

- Host-bus 16 submode, select one of:
 - **EPI_HB16_MODE_ADMUX** sets data and address muxed, AD[15:0].
 - **EPI_HB16_MODE_ADDEMUX** sets up data and address separate, D[15:0].
 - **EPI_HB16_MODE_SRAM** same as **EPI_HB8_MODE_ADDEMUX**, but uses address switch for multiple reads instead of OEn strobing, D[15:0].
 - **EPI_HB16_MODE_FIFO** adds XFIFO with sense of XFIFO full and XFIFO empty, D[15:0]. This feature is only available on CS0n and CS1n.
- **EPI_HB16_WRHIGH** sets active high write strobe, otherwise it is active low.
- **EPI_HB16_RDHIGH** sets active high read strobe, otherwise it is active low.
- Write wait state when **EPI_HB16_BAUD** is used, select one of:
 - **EPI_HB16_WRWAIT_0** sets write wait state to 2 EPI clocks (default).
 - **EPI_HB16_WRWAIT_1** sets write wait state to 4 EPI clocks.
 - **EPI_HB16_WRWAIT_2** sets write wait state to 6 EPI clocks.
 - **EPI_HB16_WRWAIT_3** sets write wait state to 8 EPI clocks.
- Read wait state when **EPI_HB16_BAUD** is used, select one of:
 - **EPI_HB16_RDWAIT_0** sets read wait state to 2 EPI clocks (default).
 - **EPI_HB16_RDWAIT_1** sets read wait state to 4 EPI clocks.
 - **EPI_HB16_RDWAIT_2** sets read wait state to 6 EPI clocks.
 - **EPI_HB16_RDWAIT_3** sets read wait state to 8 EPI clocks.
- **EPI_HB16_ALE_HIGH** sets the address latch active high (default).
- **EPI_HB16_ALE_LOW** sets address latch active low.
- **EPI_HB16_BURST_TRAFFIC** enables burst traffic. Only valid with **EPI_HB16_MODE_ADMUX** and a chip select configuration that utilizes an ALE.

Note:

The availability of the unique chip select configuration within the Host-bus 16 interface mode varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.4 EPIConfigHB16Set

Configures the interface for Host-bus 16 operation.

Prototype:

```
void
EPIConfigHB16Set(uint32_t ui32Base,
                  uint32_t ui32Config,
                  uint32_t ui32MaxWait)
```

Parameters:

ui32Base is the EPI module base address.

ui32Config is the interface configuration.

ui32MaxWait is the maximum number of external clocks to wait if a FIFO ready signal is holding off the transaction.

Description:

This function is used to configure the interface when used in Host-bus 16 operation as chosen with the function **EPIModeSet()**. The parameter ***ui32Config*** is the logical OR of the following:

- Host-bus 16 submode, select one of:
 - **EPI_HB16_MODE_ADMUX** sets data and address muxed, AD[15:0].
 - **EPI_HB16_MODE_ADDEMUX** sets up data and address as separate, D[15:0].
 - **EPI_HB16_MODE_SRAM** sets as **EPI_HB16_MODE_ADDEMUX** but uses address switch for multiple reads instead of OEn strobing, D[15:0].
 - **EPI_HB16_MODE_FIFO** adds XFIFO controls with sense of XFIFO full and XFIFO empty, D[15:0]. This submode uses no address or ALE.
- **EPI_HB16_USE_TXEMPTY** enables TXEMPTY signal with FIFO.
- **EPI_HB16_USE_RXFULL** enables RXFULL signal with FIFO.
- **EPI_HB16_WRHIGH** use active high write strobe, otherwise it is active low.
- **EPI_HB16_RDHIGH** use active high read strobe, otherwise it is active low.
- Write wait state, select one of:
 - **EPI_HB16_WRWAIT_0** sets write wait state to 2 EPI clocks.
 - **EPI_HB16_WRWAIT_1** sets write wait state to 4 EPI clocks.
 - **EPI_HB16_WRWAIT_2** sets write wait state to 6 EPI clocks.
 - **EPI_HB16_WRWAIT_3** sets write wait state to 8 EPI clocks.
- Read wait state, select one of:
 - **EPI_HB16_RDWAIT_0** sets read wait state to 2 EPI clocks.
 - **EPI_HB16_RDWAIT_1** sets read wait state to 4 EPI clocks.
 - **EPI_HB16_RDWAIT_2** sets read wait state to 6 EPI clocks.
 - **EPI_HB16_RDWAIT_3** sets read wait state to 8 EPI clocks.
- **EPI_HB16_WORD_ACCESS** use Word Access mode to route bytes to the correct byte lanes allowing data to be stored in bits [31:16]. If absent, all data transfers use bits [15:0].

Note:

EPI_HB16_WORD_ACCESS is not available on all parts. Please consult the data sheet to determine if this feature is available.

- **EPI_HB16_CLOCK_GATE_IDLE** holds the EPI clock low when no data is available to read or write.
- **EPI_HB16_CLOCK_INVERT** inverts the EPI clock.
- **EPI_HB16_IN_READY_EN** sets EPIS032 as a ready/stall signal, active high.
- **EPI_HB16_IN_READY_EN_INVERTED** sets EPIS032 as ready/stall signal, active low.
- Address latch logic, select one of:
 - **EPI_HB16_ALE_HIGH** sets the address latch active high (default).
 - **EPI_HB16_ALE_LOW** sets address latch active low.

- **EPI_HB16_BURST_TRAFFIC** enables burst traffic. Only valid with **EPI_HB16_MODE_ADMUX** and a chip select configuration that utilizes an ALE.
- **EPI_HB16_BSEL** enables byte selects. In this mode, two EPI signals operate as byte selects allowing 8-bit transfers. If this flag is not specified, data must be read and written using only 16-bit transfers.
- **EPI_HB16_CSBAUD** use different baud rates when accessing devices on each chip select. CS0n uses the baud rate specified by the lower 16 bits of the divider passed to [EPIDividerSet\(\)](#) and CS1n uses the divider passed in the upper 16 bits. If this option is absent, both chip selects use the baud rate resulting from the divider in the lower 16 bits of the parameter passed to [EPIDividerSet\(\)](#).

In addition, some parts support CS2n and CS3n for a total of 4 chip selects. If **EPI_HB16_CSBAUD** is configured, [EPIDividerCSSet\(\)](#) should be used to to configure the divider for CS2n and CS3n. They both also use the lower 16 bits passed to [EPIDividerSet\(\)](#) if this option is absent.

The use of **EPI_HB16_CSBAUD** also allows for unique chip select configurations. CS0n, CS1n, CS2n, and CS3n can each be configured by calling [EPIConfigHB16CSSet\(\)](#) if **EPI_HB16_CSBAUD** is used. Otherwise, the configuration provided in *ui32Config* is used for all chip selects.

- Chip select configuration, select one of:
 - **EPI_HB16_CSCFG_CS** sets EPIS030 to operate as a chip select signal.
 - **EPI_HB16_CSCFG_ALE** sets EPIS030 to operate as an address latch (ALE).
 - **EPI_HB16_CSCFG_DUAL_CS** sets EPIS030 to operate as CS0n and EPIS027 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.
 - **EPI_HB16_CSCFG_ALE_DUAL_CS** sets EPIS030 as an address latch (ALE), EPIS027 as CS0n and EPIS026 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.
 - **EPI_HB16_CSCFG_ALE_SINGLE_CS** sets EPIS030 to operate as an address latch (ALE) and EPIS027 is used as a chip select.
 - **EPI_HB16_CSCFG_QUAD_CS** sets EPIS030 as CS0n, EPIS027 as CS1n, EPIS034 as CS2n and EPIS033 as CS3n.
 - **EPI_HB16_CSCFG_ALE_QUAD_CS** sets EPIS030 as an address latch (ALE), EPIS026 as CS0n, EPIS027 as CS1n, EPIS034 as CS2n and EPIS033 as CS3n.

Note:

Dual or quad chip select configurations cannot be used with **EPI_HB16_MODE_SRAM**.

The parameter *ui32MaxWait* is used if the FIFO mode is chosen. If a FIFO is used along with RXFULL or TXEMPTY ready signals, then this parameter determines the maximum number of clocks to wait when the transaction is being held off by by the FIFO using one of these ready signals. A value of 0 means to wait forever.

Note:

Availability of configuration options varies based on the Tiva part in use. Please consult the data sheet to determine if the features desired are available.

Returns:

None.

11.2.2.5 EPIConfigHB16TimingSet

Sets the individual chip select timing settings for the Host-bus 16 interface.

Prototype:

```
void
EPIConfigHB16TimingSet(uint32_t ui32Base,
                      uint32_t ui32CS,
                      uint32_t ui32Config)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select value to configure.

ui32Config is the configuration settings.

Description:

This function is used to set individual chip select timings for the Host-bus 16 interface mode.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The parameter *ui32Config* is the logical OR of the following:

- Input ready stall delay, select one of:
 - **EPI_HB16_IN_READY_DELAY_1** sets the stall on input ready (EPIS032) to start 1 EPI clock after signaled.
 - **EPI_HB16_IN_READY_DELAY_2** sets the stall on input ready (EPIS032) to start 2 EPI clocks after signaled.
 - **EPI_HB16_IN_READY_DELAY_3** sets the stall on input ready (EPIS032) to start 3 EPI clocks after signaled.
- PSRAM size limitation, select one of:
 - **EPI_HB16_PSRAM_NO_LIMIT** defines no row size limitation.
 - **EPI_HB16_PSRAM_128** defines the PSRAM row size to 128 bytes.
 - **EPI_HB16_PSRAM_256** defines the PSRAM row size to 256 bytes.
 - **EPI_HB16_PSRAM_512** defines the PSRAM row size to 512 bytes.
 - **EPI_HB16_PSRAM_1024** defines the PSRAM row size to 1024 bytes.
 - **EPI_HB16_PSRAM_2048** defines the PSRAM row size to 2048 bytes.
 - **EPI_HB16_PSRAM_4096** defines the PSRAM row size to 4096 bytes.
 - **EPI_HB16_PSRAM_8192** defines the PSRAM row size to 8192 bytes.
- Host bus transfer delay, select one of:
 - **EPI_HB16_CAP_WIDTH_1** defines the inter-transfer capture width to create a delay of 1 EPI clock
 - **EPI_HB16_CAP_WIDTH_2** defines the inter-transfer capture width to create a delay of 2 EPI clocks.
- Write wait state timing reduction, select one of:
 - **EPI_HB16_WRWAIT_MINUS_DISABLE** disables the additional write wait state reduction.
 - **EPI_HB16_WRWAIT_MINUS_ENABLE** enables a 1 EPI clock write wait state reduction.

- Read wait state timing reduction, select one of:
 - **EPI_HB16_RDWAIT_MINUS_DISABLE** disables the additional read wait state reduction.
 - **EPI_HB16_RDWAIT_MINUS_ENABLE** enables a 1 EPI clock read wait state reduction.

Note:

The availability of unique chip select timings within Host-bus 16 interface mode varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.6 EPICConfigHB8CSSet

Sets the individual chip select configuration for the Host-bus 8 interface.

Prototype:

```
void
EPICConfigHB8CSSet(uint32_t ui32Base,
                    uint32_t ui32CS,
                    uint32_t ui32Config)
```

Parameters:

ui32Base is the EPI module base address.
ui32CS is the chip select value to configure.
ui32Config is the configuration settings.

Description:

This function is used to configure individual chip select settings for the Host-bus 8 interface mode. [EPICConfigHB8Set\(\)](#) must have been setup with the **EPI_HB8_CSBAUD** flag for the individual chip select configuration option to be available.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The parameter *ui32Config* is the logical OR of the following:

- Host-bus 8 submode, select one of:
 - **EPI_HB8_MODE_ADMUX** sets data and address muxed, AD[7:0].
 - **EPI_HB8_MODE_ADDEMUX** sets up data and address separate, D[7:0].
 - **EPI_HB8_MODE_SRAM** as **EPI_HB8_MODE_ADDEMUX**, but uses address switch for multiple reads instead of OEn strobing, D[7:0].
 - **EPI_HB8_MODE_FIFO** adds XFIFO with sense of XFIFO full and XFIFO empty, D[7:0]. This is only available for CS0n and CS1n.
- **EPI_HB8_WRHIGH** sets active high write strobe, otherwise it is active low.
- **EPI_HB8_RDHIGH** sets active high read strobe, otherwise it is active low.
- Write wait state when **EPI_HB8_BAUD** is used, select one of:
 - **EPI_HB8_WRWAIT_0** sets write wait state to 2 EPI clocks (default).
 - **EPI_HB8_WRWAIT_1** sets write wait state to 4 EPI clocks.
 - **EPI_HB8_WRWAIT_2** sets write wait state to 6 EPI clocks.

- **EPI_HB8_WRWAIT_3** sets write wait state to 8 EPI clocks.
- Read wait state when **EPI_HB8_BAUD** is used, select one of:
 - **EPI_HB8_RDWAIT_0** sets read wait state to 2 EPI clocks (default).
 - **EPI_HB8_RDWAIT_1** sets read wait state to 4 EPI clocks.
 - **EPI_HB8_RDWAIT_2** sets read wait state to 6 EPI clocks.
 - **EPI_HB8_RDWAIT_3** sets read wait state to 8 EPI clocks.
- **EPI_HB8_ALE_HIGH** sets the address latch active high (default).
- **EPI_HB8_ALE_LOW** sets address latch active low.

Note:

The availability of a unique chip select configuration within Host-bus 8 interface mode varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.7 EPIConfigHB8Set

Configures the interface for Host-bus 8 operation.

Prototype:

```
void
EPIConfigHB8Set(uint32_t ui32Base,
                 uint32_t ui32Config,
                 uint32_t ui32MaxWait)
```

Parameters:

ui32Base is the EPI module base address.

ui32Config is the interface configuration.

ui32MaxWait is the maximum number of external clocks to wait if a FIFO ready signal is holding off the transaction, 0-255.

Description:

This function is used to configure the interface when used in host-bus 8 operation as chosen with the function [EPIModeSet\(\)](#). The parameter *ui32Config* is the logical OR of the following:

- Host-bus 8 submode, select one of:
 - **EPI_HB8_MODE_ADMUX** sets data and address muxed, AD[7:0]
 - **EPI_HB8_MODE_ADDEMUX** sets up data and address separate, D[7:0]
 - **EPI_HB8_MODE_SRAM** as **EPI_HB8_MODE_ADDEMUX**, but uses address switch for multiple reads instead of OEn strobing, D[7:0]
 - **EPI_HB8_MODE_FIFO** adds XFIFO with sense of XFIFO full and XFIFO empty, D[7:0]
- **EPI_HB8_USE_TXEMPTY** enables TXEMPTY signal with FIFO
- **EPI_HB8_USE_RXFULL** enables RXFULL signal with FIFO
- **EPI_HB8_WRHIGH** sets active high write strobe, otherwise it is active low
- **EPI_HB8_RDHIGH** sets active high read strobe, otherwise it is active low
- Write wait state when **EPI_HB8_BAUD** is used, select one of:

- **EPI_HB8_WRWAIT_0** sets write wait state to 2 EPI clocks (default)
 - **EPI_HB8_WRWAIT_1** sets write wait state to 4 EPI clocks
 - **EPI_HB8_WRWAIT_2** sets write wait state to 6 EPI clocks
 - **EPI_HB8_WRWAIT_3** sets write wait state to 8 EPI clocks
- Read wait state when **EPI_HB8_BAUD** is used, select one of:
- **EPI_HB8_RDWAIT_0** sets read wait state to 2 EPI clocks (default)
 - **EPI_HB8_RDWAIT_1** sets read wait state to 4 EPI clocks
 - **EPI_HB8_RDWAIT_2** sets read wait state to 6 EPI clocks
 - **EPI_HB8_RDWAIT_3** sets read wait state to 8 EPI clocks
- **EPI_HB8_WORD_ACCESS** - use Word Access mode to route bytes to the correct byte lanes allowing data to be stored in bits [31:8]. If absent, all data transfers use bits [7:0].
- **EPI_HB8_CLOCK_GATE_IDLE** sets the EPI clock to be held low when no data is available to read or write
- **EPI_HB8_CLOCK_INVERT** inverts the EPI clock
- **EPI_HB8_IN_READY_EN** sets EPIS032 as a ready/stall signal, active high
- **EPI_HB8_IN_READY_EN_INVERT** sets EPIS032 as ready/stall signal, active low
- **EPI_HB8_ALE_HIGH** sets the address latch active high (default)
- **EPI_HB8_ALE_LOW** sets address latch active low
- **EPI_HB8_CSBAUD** use different baud rates when accessing devices on each chip select. CS0n uses the baud rate specified by the lower 16 bits of the divider passed to [EPIDividerSet\(\)](#) and CS1n uses the divider passed in the upper 16 bits. If this option is absent, both chip selects use the baud rate resulting from the divider in the lower 16 bits of the parameter passed to [EPIDividerSet\(\)](#).

In addition, some parts support CS2n and CS3n for a total of 4 chip selects. If **EPI_HB8_CSBAUD** is configured, [EPIDividerCSSet\(\)](#) should be used to to configure the divider for CS2n and CS3n. They both also use the lower 16 bits passed to [EPIDividerSet\(\)](#) if this option is absent.

The use of **EPI_HB8_CSBAUD** also allows for unique chip select configurations. CS0n, CS1n, CS2n, and CS3n can each be configured by calling [EPICfgHB8CSSet\(\)](#) if **EPI_HB8_CSBAUD** is used. Otherwise, the configuration provided in *ui32Config* is used for all chip selects enabled.

- Chip select configuration, select one of:
- **EPI_HB8_CSCFG_CS** sets EPIS030 to operate as a chip select signal.
 - **EPI_HB8_CSCFG_ALE** sets EPIS030 to operate as an address latch (ALE).
 - **EPI_HB8_CSCFG_DUAL_CS** sets EPIS030 to operate as CS0n and EPIS027 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.
 - **EPI_HB8_CSCFG_ALE_DUAL_CS** sets EPIS030 as an address latch (ALE), EPIS027 as CS0n and EPIS026 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.
 - **EPI_HB8_CSCFG_ALE_SINGLE_CS** sets EPIS030 to operate as an address latch (ALE) and EPIS027 is used as a chip select.
 - **EPI_HB8_CSCFG_QUAD_CS** sets EPIS030 as CS0n, EPIS027 as CS1n, EPIS034 as CS2n and EPIS033 as CS3n.
 - **EPI_HB8_CSCFG_ALE_QUAD_CS** sets EPIS030 as an address latch (ALE), EPIS026 as CS0n, EPIS027 as CS1n, EPIS034 as CS2n and EPIS033 as CS3n.

Note:

Dual or quad chip select configurations cannot be used with EPI_HB8_MODE_SRAM.

The parameter *ui32MaxWait* is used if the FIFO mode is chosen. If a FIFO is used with RXFULL or TXEMPTY ready signals, then this parameter determines the maximum number of clocks to wait when the transaction is being held off by the FIFO using one of these ready signals. A value of 0 means to wait forever.

Note:

Availability of configuration options varies based on the Tiva part in use. Please consult the data sheet to determine if the features desired are available.

Returns:

None.

11.2.2.8 EPICConfigHB8TimingSet

Sets the individual chip select timing settings for the Host-bus 8 interface.

Prototype:

```
void
EPICConfigHB8TimingSet(uint32_t ui32Base,
                      uint32_t ui32CS,
                      uint32_t ui32Config)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select value to configure.

ui32Config is the configuration settings.

Description:

This function is used to set individual chip select timings for the Host-bus 8 interface mode.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The parameter *ui32Config* is the logical OR of the following:

- Input ready stall delay, select one of:
 - **EPI_HB8_IN_READY_DELAY_1** sets the stall on input ready (EPIS032) to start 1 EPI clock after signaled.
 - **EPI_HB8_IN_READY_DELAY_2** sets the stall on input ready (EPIS032) to start 2 EPI clocks after signaled.
 - **EPI_HB8_IN_READY_DELAY_3** sets the stall on input ready (EPIS032) to start 3 EPI clocks after signaled.
- Host bus transfer delay, select one of:
 - **EPI_HB8_CAP_WIDTH_1** defines the inter-transfer capture width to create a delay of 1 EPI clock.
 - **EPI_HB8_CAP_WIDTH_2** defines the inter-transfer capture width to create a delay of 2 EPI clocks.
- **EPI_HB8_WRWAIT_MINUS_DISABLE** disables the additional write wait state reduction.

- **EPI_HB8_WRWAIT_MINUS_ENABLE** enables a 1 EPI clock write wait state reduction.
- **EPI_HB8_RDWAIT_MINUS_DISABLE** disables the additional read wait state reduction.
- **EPI_HB8_RDWAIT_MINUS_ENABLE** enables a 1 EPI clock read wait state reduction.

Note:

The availability of unique chip select timings within Host-bus 8 interface mode varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.9 EPICConfigSDRAMSet

Configures the SDRAM mode of operation.

Prototype:

```
void
EPICConfigSDRAMSet (uint32_t ui32Base,
                     uint32_t ui32Config,
                     uint32_t ui32Refresh)
```

Parameters:

- ui32Base** is the EPI module base address.
ui32Config is the SDRAM interface configuration.
ui32Refresh is the refresh count in core clocks (0-2047).

Description:

This function is used to configure the SDRAM interface, when the SDRAM mode is chosen with the function [EPIModeSet\(\)](#). The parameter **ui32Config** is the logical OR of several sets of choices:

The processor core frequency must be specified with one of the following:

- **EPI_SDRAM_CORE_FREQ_0_15** defines core clock as 0 MHz < clk <= 15 MHz
- **EPI_SDRAM_CORE_FREQ_15_30** defines core clock as 15 MHz < clk <= 30 MHz
- **EPI_SDRAM_CORE_FREQ_30_50** defines core clock as 30 MHz < clk <= 50 MHz
- **EPI_SDRAM_CORE_FREQ_50_100** defines core clock as 50 MHz < clk <= 100 MHz

The low power mode is specified with one of the following:

- **EPI_SDRAM_LOW_POWER** enter low power, self-refresh state.
- **EPI_SDRAM_FULL_POWER** normal operating state.

The SDRAM device size is specified with one of the following:

- **EPI_SDRAM_SIZE_64MBIT** size is a 64 Mbit device (8 MB).
- **EPI_SDRAM_SIZE_128MBIT** size is a 128 Mbit device (16 MB).
- **EPI_SDRAM_SIZE_256MBIT** size is a 256 Mbit device (32 MB).
- **EPI_SDRAM_SIZE_512MBIT** size is a 512 Mbit device (64 MB).

The parameter **ui16Refresh** sets the refresh counter in units of core clock ticks. It is an 11-bit value with a range of 0 - 2047 counts.

Returns:

None.

11.2.2.10 EPIDividerCSSet

Sets the clock divider for the specified CS in the EPI module.

Prototype:

```
void
EPIDividerCSSet(uint32_t ui32Base,
                 uint32_t ui32CS,
                 uint32_t ui32Divider)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select to modify and has a valid range of 0-3.

ui32Divider is the value of the clock divider to be applied to the external interface (0-65535).

Description:

This function sets the clock divider(s) for the specified CS that is used to determine the clock rate of the external interface. The *ui32Divider* value is used to derive the EPI clock rate from the system clock based on the following formula.

$$\text{EPIClk} = (\text{Divider} == 0) ? \text{SysClk} : (\text{SysClk} / ((\text{Divider} / 2) + 1) * 2)$$

For example, a divider value of 1 results in an EPI clock rate of half the system clock, value of 2 or 3 yields one quarter of the system clock and a value of 4 results in one sixth of the system clock rate.

Note:

The availability of CS2n and CS3n varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.11 EPIDividerSet

Sets the clock divider for the EPI module's CS0n/CS1n.

Prototype:

```
void
EPIDividerSet(uint32_t ui32Base,
               uint32_t ui32Divider)
```

Parameters:

ui32Base is the EPI module base address.

ui32Divider is the value of the clock divider to be applied to the external interface (0-65535).

Description:

This function sets the clock divider(s) that is used to determine the clock rate of the external interface. The *ui32Divider* value is used to derive the EPI clock rate from the system clock based on the following formula.

$$\text{EPIClk} = (\text{Divider} == 0) ? \text{SysClk} : (\text{SysClk} / ((\text{Divider} / 2) + 1) * 2)$$

For example, a divider value of 1 results in an EPI clock rate of half the system clock, value of 2 or 3 yields one quarter of the system clock and a value of 4 results in one sixth of the system clock rate.

In cases where a dual chip select mode is in use and different clock rates are required for each chip select, the *ui32Divider* parameter must contain two dividers. The lower 16 bits define the divider to be used with CS0n and the upper 16 bits define the divider for CS1n.

Returns:

None.

11.2.2.12 EPIDMATxCount

Sets the transfer count for uDMA transmit operations on EPI.

Prototype:

```
void  
EPIDMATxCount (uint32_t ui32Base,  
                uint32_t ui32Count)
```

Parameters:

ui32Base is the EPI module base address.

ui32Count is the number of units to transmit by uDMA to WRFIFO.

Description:

This function is used to help configure the EPI uDMA transmit operations. A non-zero transmit count in combination with a FIFO threshold trigger asserts an EPI uDMA transmit.

Note that, although the EPI peripheral can handle counts of up to 65535, a single uDMA transfer has a maximum length of 1024 units so *ui32Count* should be set to values less than or equal to 1024.

Note:

The availability of the EPI DMA TX count varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.13 EPIFIFOConfig

Configures the read FIFO.

Prototype:

```
void  
EPIFIFOConfig (uint32_t ui32Base,  
                uint32_t ui32Config)
```

Parameters:

ui32Base is the EPI module base address.

ui32Config is the FIFO configuration.

Description:

This function configures the FIFO trigger levels and error generation. The parameter *ui32Config* is the logical OR of the following:

- **EPI_FIFO_CONFIG_WTFULLERR** enables an error interrupt when a write is attempted and the write FIFO is full
- **EPI_FIFO_CONFIG_RSTALLERR** enables an error interrupt when a read is stalled due to an interleaved write or other reason
- FIFO TX trigger level, select one of:
 - **EPI_FIFO_CONFIG_TX_EMPTY** sets the FIFO TX trigger level to empty.
 - **EPI_FIFO_CONFIG_TX_1_4** sets the FIFO TX trigger level to 1/4.
 - **EPI_FIFO_CONFIG_TX_1_2** sets the FIFO TX trigger level to 1/2.
 - **EPI_FIFO_CONFIG_TX_3_4** sets the FIFO TX trigger level to 3/4.
- FIFO RX trigger level, select one of:
 - **EPI_FIFO_CONFIG_RX_1_8** sets the FIFO RX trigger level to 1/8.
 - **EPI_FIFO_CONFIG_RX_1_4** sets the FIFO RX trigger level to 1/4.
 - **EPI_FIFO_CONFIG_RX_1_2** sets the FIFO RX trigger level to 1/2.
 - **EPI_FIFO_CONFIG_RX_3_4** sets the FIFO RX trigger level to 3/4.
 - **EPI_FIFO_CONFIG_RX_7_8** sets the FIFO RX trigger level to 7/8.
 - **EPI_FIFO_CONFIG_RX_FULL** sets the FIFO RX trigger level to full.

Returns:

None.

11.2.2.14 EPIIntDisable

Disables EPI interrupt sources.

Prototype:

```
void
EPIIntDisable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the EPI module base address.

ui32IntFlags is a bit mask of the interrupt sources to be disabled.

Description:

This function disables the specified EPI sources for interrupt generation. The *ui32IntFlags* parameter can be the logical OR of any of the following values:

- **EPI_INT_TXREQ** interrupt when transmit FIFO is below the trigger level.
- **EPI_INT_RXREQ** interrupt when read FIFO is above the trigger level.
- **EPI_INT_ERR** interrupt when an error condition occurs.
- **EPI_INT_DMA_TX_DONE** interrupt when the transmit DMA completes.
- **EPI_INT_DMA_RX_DONE** interrupt when the read DMA completes.

Returns:

Returns None.

11.2.2.15 EPIIntEnable

Enables EPI interrupt sources.

Prototype:

```
void  
EPIIntEnable(uint32_t ui32Base,  
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the EPI module base address.

ui32IntFlags is a bit mask of the interrupt sources to be enabled.

Description:

This function enables the specified EPI sources to generate interrupts. The *ui32IntFlags* parameter can be the logical OR of any of the following values:

- **EPI_INT_TXREQ** interrupt when transmit FIFO is below the trigger level.
- **EPI_INT_RXREQ** interrupt when read FIFO is above the trigger level.
- **EPI_INT_ERR** interrupt when an error condition occurs.
- **EPI_INT_DMA_TX_DONE** interrupt when the transmit DMA completes.
- **EPI_INT_DMA_RX_DONE** interrupt when the read DMA completes.

Returns:

Returns None.

11.2.2.16 EPIIntErrorClear

Clears pending EPI error sources.

Prototype:

```
void  
EPIIntErrorClear(uint32_t ui32Base,  
                  uint32_t ui32ErrFlags)
```

Parameters:

ui32Base is the EPI module base address.

ui32ErrFlags is a bit mask of the error sources to be cleared.

Description:

This function clears the specified pending EPI errors. The *ui32ErrFlags* parameter can be the logical OR of any of the following values:

- **EPI_INT_ERR_DMAWRIC** clears the EPI_INT_DMA_TX_DONE as an interrupt source
- **EPI_INT_ERR_DMARDIC** clears the EPI_INT_DMA_RX_DONE as an interrupt source
- **EPI_INT_ERR_WTFULL** occurs when a write stalled when the transaction FIFO was full
- **EPI_INT_ERR_RSTALL** occurs when a read stalled
- **EPI_INT_ERR_TIMEOUT** occurs when the external clock enable held off a transaction longer than the configured maximum wait time

Returns:

Returns None.

11.2.2.17 EPIIntErrorStatus

Gets the EPI error interrupt status.

Prototype:

```
uint32_t
EPIIntErrorStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the EPI module base address.

Description:

This function returns the error status of the EPI. If the return value of the function [EPIIntStatus\(\)](#) has the flag **EPI_INT_ERR** set, then this function can be used to determine the cause of the error.

Returns:

Returns a bit mask of error flags, which can be the logical OR of any of the following:

- **EPI_INT_ERR_WTFULL** occurs when a write stalled when the transaction FIFO was full
- **EPI_INT_ERR_RSTALL** occurs when a read stalled
- **EPI_INT_ERR_TIMEOUT** occurs when the external clock enable held off a transaction longer than the configured maximum wait time

11.2.2.18 EPIIntRegister

Registers an interrupt handler for the EPI module.

Prototype:

```
void
EPIIntRegister(uint32_t ui32Base,
               void (*pfnHandler)(void))
```

Parameters:

ui32Base is the EPI module base address.

pfnHandler is a pointer to the function to be called when the interrupt is activated.

Description:

This sets and enables the handler to be called when the EPI module generates an interrupt. Specific EPI interrupts must still be enabled with the [EPIIntEnable\(\)](#) function.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

11.2.2.19 EPIIntStatus

Gets the EPI interrupt status.

Prototype:

```
uint32_t  
EPIIntStatus(uint32_t ui32Base,  
             bool bMasked)
```

Parameters:

ui32Base is the EPI module base address.

bMasked is set **true** to get the masked interrupt status, or **false** to get the raw interrupt status.

Description:

This function returns the EPI interrupt status. It can return either the raw or masked interrupt status.

Returns:

Returns the masked or raw EPI interrupt status, as a bit field of any of the following values:

- **EPI_INT_TXREQ** interrupt when transmit FIFO is below the trigger level.
- **EPI_INT_RXREQ** interrupt when read FIFO is above the trigger level.
- **EPI_INT_ERR** interrupt when an error condition occurs.
- **EPI_INT_DMA_TX_DONE** interrupt when the transmit DMA completes.
- **EPI_INT_DMA_RX_DONE** interrupt when the read DMA completes.

11.2.2.20 EPIIntUnregister

Removes a registered interrupt handler for the EPI module.

Prototype:

```
void  
EPIIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the EPI module base address.

Description:

This function disables and clears the handler to be called when the EPI interrupt occurs.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

11.2.2.21 EPIModeSet

Sets the usage mode of the EPI module.

Prototype:

```
void
EPIModeSet(uint32_t ui32Base,
           uint32_t ui32Mode)
```

Parameters:

ui32Base is the EPI module base address.

ui32Mode is the usage mode of the EPI module.

Description:

This function sets the operating mode of the EPI module. The parameter *ui32Mode* must be one of the following:

- **EPI_MODE_GENERAL** - use for general-purpose mode operation
- **EPI_MODE_SDRAM** - use with SDRAM device
- **EPI_MODE_HB8** - use with host-bus 8-bit interface
- **EPI_MODE_HB16** - use with host-bus 16-bit interface
- **EPI_MODE_DISABLE** - disable the EPI module

Selection of any of the above modes enables the EPI module, except for **EPI_MODE_DISABLE**, which is used to disable the module.

Returns:

None.

11.2.2.22 EPINonBlockingReadAvail

Get the count of items available in the read FIFO.

Prototype:

```
uint32_t
EPINonBlockingReadAvail(uint32_t ui32Base)
```

Parameters:

ui32Base is the EPI module base address.

Description:

This function gets the number of items that are available to read in the read FIFO. The read FIFO is filled by a non-blocking read transaction which is configured by the functions [EPINon-BlockingReadConfigure\(\)](#) and [EPINonBlockingReadStart\(\)](#).

Returns:

The number of items available to read in the read FIFO.

11.2.2.23 EPINonBlockingReadConfigure

Configures a non-blocking read transaction.

Prototype:

```
void  
EPINonBlockingReadConfigure(uint32_t ui32Base,  
                           uint32_t ui32Channel,  
                           uint32_t ui32DataSize,  
                           uint32_t ui32Address)
```

Parameters:

ui32Base is the EPI module base address.
ui32Channel is the read channel (0 or 1).
ui32DataSize is the size of the data items to read.
ui32Address is the starting address to read.

Description:

This function is used to configure a non-blocking read channel for a transaction. Two channels are available that can be used in a ping-pong method for continuous reading. It is not necessary to use both channels to perform a non-blocking read.

The parameter **ui8DataSize** is one of **EPI_NBCONFIG_SIZE_8**, **EPI_NBCONFIG_SIZE_16**, or **EPI_NBCONFIG_SIZE_32** for 8-bit, 16-bit, or 32-bit sized data transfers.

The parameter **ui32Address** is the starting address for the read, relative to the external device. The start of the device is address 0.

Once configured, the non-blocking read is started by calling [EPINonBlockingReadStart\(\)](#). If the addresses to be read from the device are in a sequence, it is not necessary to call this function multiple times. Until it is changed, the EPI module stores the last address that was used for a non-blocking read (per channel).

Returns:

None.

11.2.2.24 EPINonBlockingReadCount

Get the count remaining for a non-blocking transaction.

Prototype:

```
uint32_t  
EPINonBlockingReadCount(uint32_t ui32Base,  
                       uint32_t ui32Channel)
```

Parameters:

ui32Base is the EPI module base address.
ui32Channel is the read channel (0 or 1).

Description:

This function gets the remaining count of items for a non-blocking read transaction.

Returns:

The number of items remaining in the non-blocking read transaction.

11.2.2.25 EPINonBlockingReadGet16

Read available data from the read FIFO, as 16-bit data items.

Prototype:

```
uint32_t
EPINonBlockingReadGet16(uint32_t ui32Base,
                        uint32_t ui32Count,
                        uint16_t *pui16Buf)
```

Parameters:

ui32Base is the EPI module base address.

ui32Count is the maximum count of items to read.

pui16Buf is the caller-supplied buffer where the read data is stored.

Description:

This function reads 16-bit data items from the read FIFO and stores the values in a caller-supplied buffer. The function reads and stores data from the FIFO until there is no more data in the FIFO or the maximum count is reached as specified in the parameter *ui32Count*. The actual count of items is returned.

Returns:

The number of items read from the FIFO.

11.2.2.26 EPINonBlockingReadGet32

Read available data from the read FIFO, as 32-bit data items.

Prototype:

```
uint32_t
EPINonBlockingReadGet32(uint32_t ui32Base,
                        uint32_t ui32Count,
                        uint32_t *pui32Buf)
```

Parameters:

ui32Base is the EPI module base address.

ui32Count is the maximum count of items to read.

pui32Buf is the caller supplied buffer where the read data is stored.

Description:

This function reads 32-bit data items from the read FIFO and stores the values in a caller-supplied buffer. The function reads and stores data from the FIFO until there is no more data in the FIFO or the maximum count is reached as specified in the parameter *ui32Count*. The actual count of items is returned.

Returns:

The number of items read from the FIFO.

11.2.2.27 EPINonBlockingReadGet8

Read available data from the read FIFO, as 8-bit data items.

Prototype:

```
uint32_t  
EPINonBlockingReadGet8(uint32_t ui32Base,  
                      uint32_t ui32Count,  
                      uint8_t *pui8Buf)
```

Parameters:

ui32Base is the EPI module base address.

ui32Count is the maximum count of items to read.

pui8Buf is the caller-supplied buffer where the read data is stored.

Description:

This function reads 8-bit data items from the read FIFO and stores the values in a caller-supplied buffer. The function reads and stores data from the FIFO until there is no more data in the FIFO or the maximum count is reached as specified in the parameter **ui32Count**. The actual count of items is returned.

Returns:

The number of items read from the FIFO.

11.2.2.28 EPINonBlockingReadStart

Starts a non-blocking read transaction.

Prototype:

```
void  
EPINonBlockingReadStart(uint32_t ui32Base,  
                       uint32_t ui32Channel,  
                       uint32_t ui32Count)
```

Parameters:

ui32Base is the EPI module base address.

ui32Channel is the read channel (0 or 1).

ui32Count is the number of items to read (1-4095).

Description:

This function starts a non-blocking read that was previously configured with the function [EPINonBlockingReadConfigure\(\)](#). Once this function is called, the EPI module begins reading data from the external device into the read FIFO. The EPI stops reading when the FIFO fills up and resumes reading when the application drains the FIFO, until the total specified count of data items has been read.

Once a read transaction is completed and the FIFO drained, another transaction can be started from the next address by calling this function again.

Returns:

None.

11.2.2.29 EPINonBlockingReadStop

Stops a non-blocking read transaction.

Prototype:

```
void
EPINonBlockingReadStop(uint32_t ui32Base,
                      uint32_t ui32Channel)
```

Parameters:

ui32Base is the EPI module base address.

ui32Channel is the read channel (0 or 1).

Description:

This function cancels a non-blocking read transaction that is already in progress.

Returns:

None.

11.2.2.30 EPIPSRAMConfigRegGet

Retrieves the contents of the EPI PSRAM configuration register.

Prototype:

```
uint32_t
EPIPSRAMConfigRegGet(uint32_t ui32Base,
                      uint32_t ui32CS)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select target.

Description:

This function retrieves the EPI PSRAM configuration register. The register is read once the EPI PSRAM configuration register read enable signal is de-asserted.

The Host-bus 16 interface mode should be set up and [EPIPSRAMConfigRegRead\(\)](#) should be called prior to calling this function.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3.

Note:

The availability of PSRAM support varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

none.

11.2.2.31 EPIPSRAMConfigRegGetNonBlocking

Retrieves the contents of the EPI PSRAM configuration register.

Prototype:

```
bool  
EPIPSRAMConfigRegGetNonBlocking(uint32_t ui32Base,  
                                 uint32_t ui32CS,  
                                 uint32_t *pui32CR)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select target.

pui32CR is the provided storage used to hold the register value.

Description:

This function copies the contents of the EPI PSRAM configuration register to the provided storage if the PSRAM read configuration register enable is no longer asserted. Otherwise the provided storage is not modified.

The Host-bus 16 interface mode should be set up and [EPIPSRAMConfigRegRead\(\)](#) should be called prior to calling this function.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The *pui32CR* parameter is a pointer to provided storage used to hold the register value.

Note:

The availability of PSRAM support varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

true if the value was copied to the provided storage and **false** if it was not.

11.2.2.32 EPIPSRAMConfigRegRead

Requests a configuration register read from the PSRAM.

Prototype:

```
void  
EPIPSRAMConfigRegRead(uint32_t ui32Base,  
                      uint32_t ui32CS)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select target.

Description:

This function requests a read of the PSRAM's configuration register. The Host-bus 16 interface mode should be configured prior to calling this function. The [EPIPSRAMConfigRegGet\(\)](#) and [EPIPSRAMConfigRegGetNonBlocking\(\)](#) can be used to retrieve the configuration register value.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3.

Note:

The availability of PSRAM support varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

none.

11.2.2.33 EPIPSRAMConfigRegSet

Sets the PSRAM configuration register.

Prototype:

```
void
EPIPSRAMConfigRegSet(uint32_t ui32Base,
                      uint32_t ui32CS,
                      uint32_t ui32CR)
```

Parameters:

ui32Base is the EPI module base address.

ui32CS is the chip select target.

ui32CR is the PSRAM configuration register value.

Description:

This function sets the PSRAM's configuration register by using the PSRAM configuration register enable signal. The Host-bus 16 interface mode should be configured prior to calling this function.

The *ui32Base* parameter is the base address for the EPI hardware module. The *ui32CS* parameter specifies the chip select to configure and has a valid range of 0-3. The parameter *ui32CR* value is determined by consulting the PSRAM's data sheet.

Note:

The availability of PSRAM support varies based on the Tiva part in use. Please consult the data sheet to determine if this feature is available.

Returns:

None.

11.2.2.34 EPIWorkaroundByteRead

Safely reads a byte from the EPI 0x10000000 address space.

Prototype:

```
uint8_t
EPIWorkaroundByteRead(uint8_t *pui8Addr)
```

Parameters:

pui8Addr is the address which is to be read.

Description:

This function must be used when reading bytes from EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

The 8-bit byte stored at address *pui8Addr*.

11.2.2.35 EPIWorkaroundByteWrite

Safely writes a byte to the EPI 0x10000000 address space.

Prototype:

```
void  
EPIWorkaroundByteWrite(uint8_t *pui8Addr,  
                      uint8_t ui8Value)
```

Parameters:

pui8Addr is the address which is to be written.

ui8Value is the 8-bit byte to write.

Description:

This function must be used when writing bytes to EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

None.

11.2.2.36 EPIWorkaroundHWordRead

Safely reads a half-word from the EPI 0x10000000 address space.

Prototype:

```
uint16_t  
EPIWorkaroundHWordRead(uint16_t *pui16Addr)
```

Parameters:

pui16Addr is the address which is to be read.

Description:

This function must be used when reading half-words from EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

The 16-bit word stored at address *pui16Addr*.

11.2.2.37 EPIWorkaroundHWordWrite

Safely writes a half-word to the EPI 0x10000000 address space.

Prototype:

```
void
EPIWorkaroundHWordWrite(uint16_t *pui16Addr,
                        uint16_t ui16Value)
```

Parameters:

pui16Addr is the address which is to be written.

ui16Value is the 16-bit half-word to write.

Description:

This function must be used when writing half-words to EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

None.

11.2.2.38 EPIWorkaroundWordRead

Safely reads a word from the EPI 0x10000000 address space.

Prototype:

```
uint32_t  
EPIWorkaroundWordRead(uint32_t *pui32Addr)
```

Parameters:

pui32Addr is the address which is to be read.

Description:

This function must be used when reading words from EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

The 32-bit word stored at address *pui32Addr*.

11.2.2.39 EPIWorkaroundWordWrite

Safely writes a word to the EPI 0x10000000 address space.

Prototype:

```
void  
EPIWorkaroundWordWrite(uint32_t *pui32Addr,  
                      uint32_t ui32Value)
```

Parameters:

pui32Addr is the address which is to be written.

ui32Value is the 32-bit word to write.

Description:

This function must be used when writing words to EPI-attached memory configured to use the address space at 0x10000000 on devices affected by the EPI#01 erratum. Direct access to memory in these cases can cause data corruption depending upon memory accesses immediately before or after the EPI access but using this function will allow EPI accesses to complete correctly. The function is defined as “inline” in epi.h.

Use of this function on a device not affected by the erratum is safe but will impact performance due to an additional overhead of at least 2 cycles per access. This erratum affects only the 0x10000000 address space typically used to store the LCD controller frame buffer. The 0x60000000 address space is not affected and applications using this address mapping need not use this function.

Returns:

None.

11.2.2.40 EPIWriteFIFOCountGet

Reads the number of empty slots in the write transaction FIFO.

Prototype:

```
uint32_t
EPIWriteFIFOCountGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the EPI module base address.

Description:

This function returns the number of slots available in the transaction FIFO. It can be used in a polling method to avoid attempting a write that would stall.

Returns:

The number of empty slots in the transaction FIFO.

11.3 Programming Example

This example illustrates the setup steps required to initialize the EPI to access an SDRAM when the system clock is running at 50MHz.

```
//
// Set the EPI divider.
//
EPIDividerSet(EPIO_BASE, 0);

//
// Select SDRAM mode.
//
EPIModeSet(EPIO_BASE, EPI_MODE_SDRAM);

//
// Configure SDRAM mode.
//
EPIConfigSDRAMSet(EPIO_BASE, (EPI_SDRAM_CORE_FREQ_50_100 |
                                EPI_SDRAM_FULL_POWER | EPI_SDRAM_SIZE_64MBIT), 1024);

//
// Set the address map.
//
EPIAddressMapSet(EPIO_BASE, EPI_ADDR_RAM_SIZE_256MB | EPI_ADDR_RAM_BASE_6);

//
// Wait for the EPI initialization to complete.
//
while(HWREG(EPIO_BASE + EPI_O_STAT) & EPI_STAT_INITSEQ)
{
    //
    // Wait for SDRAM initialization to complete.
    //
}

//
// At this point, the SDRAM is accessible and available for use.
//
```


12 Fan Controller

Introduction	233
API Functions	233
Programming Example	235

12.1 Introduction

The fan controller API provides functions to the use the fan controller peripheral available in the Tiva microcontroller. The fan controller provides multiple channels of fan PWM control. Features include:

- automatic or manual speed control
- filtering of speed readings to smooth fluctuations
- speed reading in RPM
- stall detection
- auto-restart on stall
- fast start to bring fan up to speed quickly
- interrupt notification of fan events

The fan controller allows an application to set the desired cooling fan speed, and the speed is maintained without further intervention from the application software. The application can also choose to be notified by an interrupt when certain events occur such as fan stall or fan reaching a commanded speed.

A FAN channel can also be manually controlled by directly specifying the PWM duty cycle.

This driver is contained in `driverlib/fan.c`, with `driverlib/fan.h` containing the API declarations for use by applications.

12.2 API Functions

In order to function, a FAN channel must first be enabled using the function `FanChannelEnable()`. A channel can be disabled with `FanChannelDisable()`.

A FAN channel can be configured for manual or automatic mode. In manual mode, the application sets the PWM duty cycle directly and can monitor the RPM. In automatic mode, the application sets the desired RPM, and the Fan Controller adjusts the PWM duty cycle to achieve the commanded RPM. A FAN channel must be configured for either automatic or manual mode using `FanChannelConfigAuto()` or `FanChannelConfigManual()`.

Once a FAN channel is configured, the application can update the speed of the cooling fan by using `FanChannelRPMSet()` if in automatic mode or `FanChannelDutySet()` if in manual mode. The actual RPM can be queried for both manual and automatic mode by using `FanChannelRPMGet()`. The duty cycle can be determined by calling `FanChannelDutyGet()`. If the channel is configured for manual mode, the duty cycle value that is returned is the same value that was commanded. But if the channel is in automatic mode, then the duty cycle value is the value that has been calculated by the FAN channel automatic speed control algorithm.

The fan controller can be configured to notify an application of various events using interrupts. The interrupt handler can be registered at run-time using the function `FanIntRegister()`. The function `FanIntUnregister()` can be used to remove the interrupt handler when it is no longer needed. These functions are not needed if static, build-time interrupt registration is used.

Specific interrupt events can be enabled for each channel using the `FanIntEnable()` function. Similarly, interrupts can be disabled with `FanIntDisable()`. The interrupt status can be checked with `FanIntStatus()` and any pending interrupts cleared with `FanIntClear()`.

Fan Channel Configuration Options

The fan controller has several options to control the behavior of a FAN channel. These options are configured using `FanChannelConfigAuto()` or `FanChannelConfigManual()`.

The following options are available in automatic mode:

- **Automatic restart:** a FAN channel can be configured to restart automatically if it stalls. Use the flags `FAN_CONFIG_RESTART` or `FAN_CONFIG_NORESTART` to configure this behavior.
- **Acceleration rate:** a FAN channel can be configured to change speed using a fast or slow acceleration ramp. This ramp is the rate of change that is used when the FAN is hunting for the commanded speed. The acceleration rate is configured using one of the flags `FAN_CONFIG_ACCEL_SLOW` or `FAN_CONFIG_ACCEL_FAST`.
- **Startup setting:** a FAN channel can be configured to start using the calculated value for the PWM duty cycle, or by applying a fixed PWM duty cycle for a period of time in order to quickly bring the cooling fan up to speed. It may also be useful to briefly apply a higher PWM value to a cooling fan intended to run at low speed, in order to overcome static friction and get it started. If an initial fixed PWM duty cycle is not needed to start the cooling fan, then use the configuration flag `FAN_CONFIG_START_DUTY_OFF`. However if a fixed PWM duty cycle is needed to start the cooling fan, then choose one of the flags `FAN_CONFIG_START_DUTY_50`, `_75`, or `_100` to use 50%, 75% or 100% duty cycle to start the cooling fan. If a starting duty cycle is used, then the startup period must also be specified using `FAN_CONFIG_START_2`, `FAN_CONFIG_START_4`, etc. This setting chooses the number of tachometer edges to determine the amount of time that the starting duty cycle is applied.
- **Speed adjustment rate:** the rate at which the FAN makes changes can be adjusted using the flags `FAN_CONFIG_HYST_1`, `FAN_CONFIG_HYST_2`, etc. This setting chooses the number of tachometer pulses to delay between speed changes. Using a larger value can smooth out the changes of cooling fan speed.
- **Speed averaging:** the speed measurement can be averaged over several samples in order to smooth out the speed reading. This parameter can be configured using the configuration flags `FAN_CONFIG_AVG_NONE` for no speed averaging, or `FAN_CONFIG_AVG_2`, etc. to select the number of speed samples to use for averaging.
- **Tachometer rate:** different cooling fans may have a different number of tachometer pulses per revolution. This parameter can be configured using the configuration flags `FAN_TACH_1`, `FAN_TACH_2`, etc.

The following options are available in manual mode:

- **Speed averaging:** the speed measurement can be averaged over several samples in order to smooth out the speed reading. This parameter can be configured using the configuration

flags **FAN_CONFIG_AVG_NONE** for no speed averaging, or **FAN_CONFIG_AVG_2**, etc. to select the number of speed samples to use for averaging.

- **Tachometer rate:** different cooling fans may have a different number of tachometer pulses per revolution. This parameter can be configured using the configuration flags **FAN_TACH_1**, **FAN_TACH_2**, etc.

12.3 Programming Example

```

//  

// Enable the Fan peripheral  

//  

SysCtlPeripheralEnable(SYSCTL_PERIPH_FAN0);  

//  

// Configure Fan channel 0 for automatic mode.  The following  

// configuration choices are used:  

// - automatic restart  

// - fast acceleration  

// - 50% startup duty cycle  

// - start period of 64 tachometer pulse edges  

// - hysteresis smoothing of 16 tachometer edges  

// - speed averaging over 4 samples  

// - 4 pulses per revolution tachometer rate  

//  

FanChannelConfigAuto(FAN0_BASE, 0, FAN_CONFIG_RESTART |  

    FAN_CONFIG_ACCEL_FAST | FAN_CONFIG_HYST_16 |  

    FAN_CONFIG_START_DUTY_50 | FAN_CONFIG_START_64 |  

    FAN_CONFIG_AVG_4 | FAN_CONFIG_TACH_4);  

//  

// Enable fan channel 0 for operation  

//  

FanChannelEnable(FAN0_BASE, 0);  

//  

// Set the fan to run at 1000 RPM  

//  

FanChannelRPMSet(FAN0_BASE, 0, 1000);

```


13 Flash

Introduction	237
API Functions	237
Programming Example	245

13.1 Introduction

The flash API provides a set of functions for dealing with the on-chip flash. Functions are provided to program and erase the flash, configure the flash protection, and handle the flash interrupt.

The flash is organized as a set of blocks that can be individually erased. See the device data sheet to determine the size of the flash blocks on an MCU. Erasing a block causes the entire contents of the block to be reset to all ones. The blocks can be marked as read-only or execute-only, providing differing levels of code protection. Read-only blocks cannot be erased or programmed, protecting the contents of those blocks from being modified. Execute-only blocks cannot be erased or programmed, and can only be read by the processor instruction fetch mechanism, protecting the contents of those blocks from being read by either the processor or by debuggers. Refer to the device data sheet to determine the size of flash blocks that can be configured as read-only or execute-only.

The flash can be programmed on a word-by-word basis. Programming causes 1 bits to become 0 bits (where appropriate); because of this, a word can be repeatedly programmed so long as each programming operation only requires changing 1 bits to 0 bits.

The timing for the flash is automatically handled by the flash controller. On some devices, flash timing depends on the PLL frequency that is specified. For these devices, the [SysCtlClockFreqSet\(\)](#) function properly configures the flash timing.

The flash controller has the ability to generate an interrupt when an invalid access is attempted (such as reading from execute-only flash). This capability can be used to validate the operation of a program as the interrupt ensures that invalid accesses are not silently ignored, hiding potential bugs. The flash protection can be applied without being permanently enabled, which allows the program to be debugged before the flash protection is permanently applied to the device (which is a non-reversible operation on some devices). An interrupt can also be generated when an erase or programming operation has completed.

Depending upon the member of the Tiva family used, the amount of available flash is 8 KB, 16 KB, 32 KB, 64 KB, 96 KB, 128 KB, 256 KB, 512 KB, or 1 MB.

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

13.2 API Functions

Functions

- `int32_t FlashErase (uint32_t ui32Address)`
- `void FlashIntClear (uint32_t ui32IntFlags)`
- `void FlashIntDisable (uint32_t ui32IntFlags)`

- void [FlashIntEnable](#) (uint32_t ui32IntFlags)
- void [FlashIntRegister](#) (void (*pfnHandler)(void))
- uint32_t [FlashIntStatus](#) (bool bMasked)
- void [FlashIntUnregister](#) (void)
- int32_t [FlashProgram](#) (uint32_t *pui32Data, uint32_t ui32Address, uint32_t ui32Count)
- tFlashProtection [FlashProtectGet](#) (uint32_t ui32Address)
- int32_t [FlashProtectSave](#) (void)
- int32_t [FlashProtectSet](#) (uint32_t ui32Address, tFlashProtection eProtect)
- int32_t [FlashUserGet](#) (uint32_t *pui32User0, uint32_t *pui32User1)
- int32_t [FlashUserSave](#) (void)
- int32_t [FlashUserSet](#) (uint32_t ui32User0, uint32_t ui32User1)

13.2.1 Detailed Description

The flash API is broken into three groups of functions: those that deal with programming the flash, those that deal with flash protection, and those that deal with interrupt handling.

Flash programming is managed with [FlashErase\(\)](#), [FlashProgram\(\)](#), [FlashUsecGet\(\)](#), and [FlashUsecSet\(\)](#).

Flash protection is managed with [FlashProtectGet\(\)](#), [FlashProtectSet\(\)](#), and [FlashProtectSave\(\)](#).

Interrupt handling is managed with [FlashIntRegister\(\)](#), [FlashIntUnregister\(\)](#), [FlashIntEnable\(\)](#), [FlashIntDisable\(\)](#), [FlashIntGetStatus\(\)](#), and [FlashIntClear\(\)](#).

13.2.2 Function Documentation

13.2.2.1 FlashErase

Erases a block of flash.

Prototype:

```
int32_t  
FlashErase(uint32_t ui32Address)
```

Parameters:

ui32Address is the start address of the flash block to be erased.

Description:

This function erases a block of the on-chip flash. After erasing, the block is filled with 0xFF bytes. Read-only and execute-only blocks cannot be erased.

The flash block size is device-class dependent. All TM4C123x devices use 1-KB blocks but TM4C129x devices use 16-KB blocks. Please consult the device datasheet to determine the block size in use.

This function does not return until the block has been erased.

Returns:

Returns 0 on success, or -1 if an invalid block address was specified or the block is write-protected.

13.2.2.2 FlashIntClear

Clears flash controller interrupt sources.

Prototype:

```
void
FlashIntClear(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupt sources to be cleared.

Description:

The specified flash controller interrupt sources are cleared, so that they no longer assert. The *ui32IntFlags* parameter can be the logical OR of any of the following values:

- **FLASH_INT_ACCESS** occurs when a program or erase action was attempted on a block of flash that is marked as read-only or execute-only.
- **FLASH_INT_PROGRAM** occurs when a programming or erase cycle completes.
- **FLASH_INT_EEPROM** occurs when an EEPROM interrupt occurs. The source of the EEPROM interrupt can be determined by reading the EEDONE register.
- **FLASH_INT_VOLTAGE_ERR** occurs when the voltage was out of spec during the flash operation and the operation was terminated.
- **FLASH_INT_DATA_ERR** occurs when an operation attempts to program a bit that contains a 0 to a 1.
- **FLASH_INT_ERASE_ERR** occurs when an erase operation fails.
- **FLASH_INT_PROGRAM_ERR** occurs when a program operation fails.

This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

13.2.2.3 FlashIntDisable

Disables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntDisable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is a bit mask of the interrupt sources to be disabled. The *ui32IntFlags* parameter can be the logical OR of any of the following values:

- **FLASH_INT_ACCESS** occurs when a program or erase action was attempted on a block of flash that is marked as read-only or execute-only.
- **FLASH_INT_PROGRAM** occurs when a programming or erase cycle completes.
- **FLASH_INT_EEPROM** occurs when an EEPROM interrupt occurs. The source of the EEPROM interrupt can be determined by reading the EEDONE register.
- **FLASH_INT_VOLTAGE_ERR** occurs when the voltage was out of spec during the flash operation and the operation was terminated.
- **FLASH_INT_DATA_ERR** occurs when an operation attempts to program a bit that contains a 0 to a 1.
- **FLASH_INT_ERASE_ERR** occurs when an erase operation fails.
- **FLASH_INT_PROGRAM_ERR** occurs when a program operation fails.

This function disables the indicated flash controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

13.2.2.4 void FlashIntEnable (uint32_t *ui32IntFlags*)

Enables individual flash controller interrupt sources.

Parameters:

ui32IntFlags is a bit mask of the interrupt sources to be enabled. The *ui32IntFlags* parameter can be the logical OR of any of the following values:

- **FLASH_INT_ACCESS** occurs when a program or erase action was attempted on a block of flash that is marked as read-only or execute-only.
- **FLASH_INT_PROGRAM** occurs when a programming or erase cycle completes.
- **FLASH_INT_EEPROM** occurs when an EEPROM interrupt occurs. The source of the EEPROM interrupt can be determined by reading the EEDONE register.
- **FLASH_INT_VOLTAGE_ERR** occurs when the voltage was out of spec during the flash operation and the operation was terminated.
- **FLASH_INT_DATA_ERR** occurs when an operation attempts to program a bit that contains a 0 to a 1.
- **FLASH_INT_ERASE_ERR** occurs when an erase operation fails.
- **FLASH_INT_PROGRAM_ERR** occurs when a program operation fails.

This function enables the indicated flash controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

13.2.2.5 void FlashIntRegister (void(*)(void) *pfnHandler*)

Registers an interrupt handler for the flash interrupt.

Parameters:

pfnHandler is a pointer to the function to be called when the flash interrupt occurs.

Description:

This function sets the handler to be called when the flash interrupt occurs. The flash controller can generate an interrupt when an invalid flash access occurs, such as trying to program or erase a read-only block, or trying to read from an execute-only block. It can also generate an interrupt when a program or erase operation has completed. The interrupt is automatically enabled when the handler is registered.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

13.2.2.6 FlashIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t
FlashIntStatus (bool bMasked)
```

Parameters:

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the flash controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of **FLASH_INT_ACCESS**, **FLASH_INT_PROGRAM**, **FLASH_INT_EEPROM**, **FLASH_INT_VOLTAGE_ERR**, **FLASH_INT_DATA_ERR**, **FLASH_INT_ERASE_ERR**, and **FLASH_INT_PROGRAM_ERR**.

13.2.2.7 FlashIntUnregister

Unregisters the interrupt handler for the flash interrupt.

Prototype:

```
void
FlashIntUnregister (void)
```

Description:

This function clears the handler to be called when the flash interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler is no longer called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

13.2.2.8 FlashProgram

Programs flash.

Prototype:

```
int32_t  
FlashProgram(uint32_t *pui32Data,  
            uint32_t ui32Address,  
            uint32_t ui32Count)
```

Parameters:

pui32Data is a pointer to the data to be programmed.

ui32Address is the starting address in flash to be programmed. Must be a multiple of four.

ui32Count is the number of bytes to be programmed. Must be a multiple of four.

Description:

This function programs a sequence of words into the on-chip flash. Because the flash is programmed one word at a time, the starting address and byte count must both be multiples of four. It is up to the caller to verify the programmed contents, if such verification is required.

This function does not return until the data has been programmed.

Returns:

Returns 0 on success, or -1 if a programming error is encountered.

13.2.2.9 FlashProtectGet

Gets the protection setting for a block of flash.

Prototype:

```
tFlashProtection  
FlashProtectGet(uint32_t ui32Address)
```

Parameters:

ui32Address is the start address of the flash block to be queried.

Description:

This function gets the current protection for the specified block of flash. Refer to the device data sheet to determine the granularity for each protection option. A block can be read/write, read-only, or execute-only. Read/write blocks can be read, executed, erased, and programmed. Read-only blocks can be read and executed. Execute-only blocks can only be executed; processor and debugger data reads are not allowed.

Returns:

Returns the protection setting for this block. See [FlashProtectSet\(\)](#) for possible values.

13.2.2.10 FlashProtectSave

Saves the flash protection settings.

Prototype:

```
int32_t  
FlashProtectSave(void)
```

Description:

This function makes the currently programmed flash protection settings permanent. This operation is non-reversible; a chip reset or power cycle does not change the flash protection.

This function does not return until the protection has been saved.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

13.2.2.11 FlashProtectSet

Sets the protection setting for a block of flash.

Prototype:

```
int32_t  
FlashProtectSet(uint32_t ui32Address,  
                tFlashProtection eProtect)
```

Parameters:

ui32Address is the start address of the flash block to be protected.

eProtect is the protection to be applied to the block. Can be one of **FlashReadWrite**, **FlashReadOnly**, or **FlashExecuteOnly**.

Description:

This function sets the protection for the specified block of flash. Refer to the device data sheet to determine the granularity for each protection option. Blocks that are read/write can be made read-only or execute-only. Blocks that are read-only can be made execute-only. Blocks that are execute-only cannot have their protection modified. Attempts to make the block protection less stringent (that is, read-only to read/write) result in a failure (and are prevented by the hardware).

Changes to the flash protection are maintained only until the next reset. This protocol allows the application to be executed in the desired flash protection environment to check for inappropriate flash access (via the flash interrupt). To make the flash protection permanent, use the [FlashProtectSave\(\)](#) function.

Returns:

Returns 0 on success, or -1 if an invalid address or an invalid protection was specified.

13.2.2.12 FlashUserGet

Gets the user registers.

Prototype:

```
int32_t  
FlashUserGet(uint32_t *pui32User0,  
             uint32_t *pui32User1)
```

Parameters:

pui32User0 is a pointer to the location to store USER Register 0.
pui32User1 is a pointer to the location to store USER Register 1.

Description:

This function reads the contents of user registers 0 and 1, and stores them in the specified locations.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

13.2.2.13 FlashUserSave

Saves the user registers.

Prototype:

```
int32_t  
FlashUserSave(void)
```

Description:

This function makes the currently programmed user register 0 and 1 settings permanent. This operation is non-reversible; a chip reset or power cycle does not change the flash protection.

This function does not return until the protection has been saved.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

13.2.2.14 FlashUserSet

Sets the user registers.

Prototype:

```
int32_t  
FlashUserSet(uint32_t ui32User0,  
             uint32_t ui32User1)
```

Parameters:

ui32User0 is the value to store in USER Register 0.
ui32User1 is the value to store in USER Register 1.

Description:

This function sets the contents of the user registers 0 and 1 to the specified values.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

13.3 Programming Example

The following example shows how to use the flash API to erase a block of the flash and program a few words.

```
uint32_t pui32Data[2];

//
// Erase a block of the flash.
//
FlashErase(0x800);

//
// Program some data into the newly erased block of the flash.
//
pui32Data[0] = 0x12345678;
pui32Data[1] = 0x56789abc;
FlashProgram(pui32Data, 0x800, sizeof(pui32Data));
```


14 Floating-Point Unit (FPU)

Introduction	247
API Functions	248
Programming Example	252

14.1 Introduction

The floating-point unit (FPU) driver provides methods for manipulating the behavior of the floating-point unit in the Cortex-M processor. By default, the floating-point is disabled and must be enabled prior to the execution of any floating-point instructions. If a floating-point instruction is executed when the floating-point unit is disabled, a NOCP usage fault is generated. This feature can be used by an RTOS, for example, to keep track of which tasks actually use the floating-point unit, and therefore only perform floating-point context save/restore on task switches that involve those tasks.

There are three methods of handling the floating-point context when the processor executes an interrupt handler: it can do nothing with the floating-point context, it can always save the floating-point context, or it can perform a lazy save/restore of the floating-point context. If nothing is done with the floating-point context, the interrupt stack frame is identical to a Cortex-M processor that does not have a floating-point unit, containing only the volatile registers of the integer unit. This method is useful for applications where the floating-point unit is used by the main thread of execution, but not in any of the interrupt handlers. By not saving the floating-point context, stack usage is reduced and interrupt latency is kept to a minimum.

Alternatively, the floating-point context can always be saved onto the stack. This method allows floating-point operations to be performed inside interrupt handlers without any special precautions, at the expense of increased stack usage (for the floating-point context) and increased interrupt latency (due to the additional writes to the stack). The advantage to this method is that the stack frame always contains the floating-point context when inside an interrupt handler.

The default handling of the floating-point context is to perform a lazy save/restore. When an interrupt is taken, space is reserved on the stack for the floating-point context but the context is not written. This method keeps the interrupt latency to a minimum because only the integer state is written to the stack. Then, if a floating-point instruction is executed from within the interrupt handler, the floating-point context is written to the stack prior to the execution of the floating-point instruction. Finally, upon return from the interrupt, the floating-point context is restored from the stack only if it was written. Using lazy save/restore provides a blend between fast interrupt response and the ability to use floating-point instructions in the interrupt handler.

The floating-point unit can generate an interrupt when one of several exceptions occur. The exceptions are underflow, overflow, divide by zero, invalid operation, input denormal, and inexact exception. The application can optionally choose to enable one or more of these interrupts and use the interrupt handler to decide upon a course of action to be taken in each case.

The behavior of the floating-point unit can also be adjusted, specifying the format of half-precision floating-point values, the handle of NaN values, the flush-to-zero mode (which sacrifices full IEEE compliance for execution speed), and the rounding mode for results.

This driver is contained in `driverlib/fpu.c`, with `driverlib/fpu.h` containing the API declarations for use by applications.

14.2 API Functions

Functions

- void [FPUDisable](#) (void)
- void [FPUEnable](#) (void)
- void [FPUFlushToZeroModeSet](#) (uint32_t ui32Mode)
- void [FPUHalfPrecisionModeSet](#) (uint32_t ui32Mode)
- void [FPULazyStackingEnable](#) (void)
- void [FPUNaNModeSet](#) (uint32_t ui32Mode)
- void [FPURoundingModeSet](#) (uint32_t ui32Mode)
- void [FPUStackingDisable](#) (void)
- void [FPUStackingEnable](#) (void)

14.2.1 Detailed Description

The FPU API provides functions for enabling and disabling the floating-point unit ([FPUEnable\(\)](#) and [FPUDisable\(\)](#)), for controlling how the floating-point state is stored on the stack when interrupts occur ([FPUStackingEnable\(\)](#), [FPULazyStackingEnable\(\)](#), and [FPUStackingDisable\(\)](#)), for handling the floating-point interrupt ([FPUIntRegister\(\)](#), [FPUIntUnregister\(\)](#), [FPUIntEnable\(\)](#), [FPUIntDisable\(\)](#), [FPUIntStatus\(\)](#), and [FPUIntClear\(\)](#)), and for adjusting the operation of the floating-point unit ([FPUHalfPrecisionModeSet\(\)](#), [FPUNaNModeSet\(\)](#), [FPUFlushToZeroModeSet\(\)](#), and [FPURoundIngModeSet\(\)](#)).

14.2.2 Function Documentation

14.2.2.1 FPUDisable

Disables the floating-point unit.

Prototype:

```
void  
FPUDisable(void)
```

Description:

This function disables the floating-point unit, preventing floating-point instructions from executing (generating a NOCP usage fault instead).

Returns:

None.

14.2.2.2 FPUEnable

Enables the floating-point unit.

Prototype:

```
void
FPUEnable(void)
```

Description:

This function enables the floating-point unit, allowing the floating-point instructions to be executed. This function must be called prior to performing any hardware floating-point operations; failure to do so results in a NOCP usage fault.

Returns:

None.

14.2.2.3 FPUFlushToZeroModeSet

Selects the flush-to-zero mode.

Prototype:

```
void
FPUFlushToZeroModeSet(uint32_t ui32Mode)
```

Parameters:

ui32Mode is the flush-to-zero mode; which is either **FPU_FLUSH_TO_ZERO_DIS** or **FPU_FLUSH_TO_ZERO_EN**.

Description:

This function enables or disables the flush-to-zero mode of the floating-point unit. When disabled (the default), the floating-point unit is fully IEEE compliant. When enabled, values close to zero are treated as zero, greatly improving the execution speed at the expense of some accuracy (as well as IEEE compliance).

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

14.2.2.4 FPUHalfPrecisionModeSet

Selects the format of half-precision floating-point values.

Prototype:

```
void
FPUHalfPrecisionModeSet(uint32_t ui32Mode)
```

Parameters:

ui32Mode is the format for half-precision floating-point value, which is either **FPU_HALF_IEEE** or **FPU_HALF_ALTERNATE**.

Description:

This function selects between the IEEE half-precision floating-point representation and the Cortex-M processor alternative representation. The alternative representation has a larger

range but does not have a way to encode infinity (positive or negative) or NaN (quiet or signaling). The default setting is the IEEE format.

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

14.2.2.5 FPULazyStackingEnable

Enables the lazy stacking of floating-point registers.

Prototype:

```
void  
FPULazyStackingEnable(void)
```

Description:

This function enables the lazy stacking of floating-point registers s0-s15 when an interrupt is handled. When lazy stacking is enabled, space is reserved on the stack for the floating-point context, but the floating-point state is not saved. If a floating-point instruction is executed from within the interrupt context, the floating-point context is first saved into the space reserved on the stack. On completion of the interrupt handler, the floating-point context is only restored if it was saved (as the result of executing a floating-point instruction).

This method provides a compromise between fast interrupt response (because the floating-point state is not saved on interrupt entry) and the ability to use floating-point in interrupt handlers (because the floating-point state is saved if floating-point instructions are used).

Returns:

None.

14.2.2.6 FPUNaNModeSet

Selects the NaN mode.

Prototype:

```
void  
FPUNaNModeSet(uint32_t ui32Mode)
```

Parameters:

ui32Mode is the mode for NaN results; which is either **FPU_NAN_PROPAGATE** or **FPU_NAN_DEFAULT**.

Description:

This function selects the handling of NaN results during floating-point computations. NaNs can either propagate (the default), or they can return the default NaN.

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

14.2.2.7 FPURoundingModeSet

Selects the rounding mode for floating-point results.

Prototype:

```
void
FPURoundingModeSet(uint32_t ui32Mode)
```

Parameters:

ui32Mode is the rounding mode.

Description:

This function selects the rounding mode for floating-point results. After a floating-point operation, the result is rounded toward the specified value. The default mode is **FPU_ROUND_NEAREST**.

The following rounding modes are available (as specified by *ui32Mode*):

- **FPU_ROUND_NEAREST** - round toward the nearest value
- **FPU_ROUND_POS_INF** - round toward positive infinity
- **FPU_ROUND_NEG_INF** - round toward negative infinity
- **FPU_ROUND_ZERO** - round toward zero

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

14.2.2.8 FPUSTackingDisable

Disables the stacking of floating-point registers.

Prototype:

```
void
FPUSTackingDisable(void)
```

Description:

This function disables the stacking of floating-point registers s0-s15 when an interrupt is handled. When floating-point context stacking is disabled, floating-point operations performed in an interrupt handler destroy the floating-point context of the main thread of execution.

Returns:

None.

14.2.2.9 FPUStrackingEnable

Enables the stacking of floating-point registers.

Prototype:

```
void  
FPUStrackingEnable (void)
```

Description:

This function enables the stacking of floating-point registers s0-s15 when an interrupt is handled. When enabled, space is reserved on the stack for the floating-point context and the floating-point state is saved into this stack space. Upon return from the interrupt, the floating-point context is restored.

If the floating-point registers are not stacked, floating-point instructions cannot be safely executed in an interrupt handler because the values of s0-s15 are not likely to be preserved for the interrupted code. On the other hand, stacking the floating-point registers increases the stacking operation from 8 words to 26 words, also increasing the interrupt response latency.

Returns:

None.

14.3 Programming Example

The following example shows how to use the FPU API to enable the floating-point unit and configure the stacking of floating-point context.

```
//  
// Enable the floating-point unit.  
//  
FPUEnable();  
  
//  
// Configure the floating-point unit to perform lazy stacking of the  
// floating-point state.  
//  
FPULazyStackingEnable();
```

15 GPIO

Introduction	253
API Functions	254
Programming Example	286

15.1 Introduction

The GPIO module provides control for up to eight independent GPIO pins (the actual number present depend upon the GPIO port and part number). Each pin has the following capabilities:

- Can be configured as an input or an output. On reset, GPIOs default to being inputs.
- In input mode, can generate interrupts on high level, low level, rising edge, falling edge, or both edges.
- In output mode, can be configured for 2-mA, 4-mA, or 8-mA drive strength. The 8-mA drive strength configuration has optional slew rate control to limit the rise and fall times of the signal. On reset, GPIOs default to 2-mA drive strength.
- Optional weak pull-up or pull-down resistors. On reset, GPIOs default to no pull-up or pull-down resistors.
- Optional open-drain operation. On reset, GPIOs default to standard push/pull operation.
- Can be configured to be a GPIO or a peripheral pin. On reset, the default is GPIO. Note that not all pins on all parts have peripheral functions, in which case the pin is only useful as a GPIO.

Most of the GPIO functions can operate on more than one GPIO pin (within a single module) at a time. The *ucPins* parameter to these functions is used to specify the pins that are affected; only the GPIO pins corresponding to the bits in this parameter that are set are affected (where pin 0 is bit 0, pin 1 in bit 1, and so on). For example, if *ucPins* is 0x09, then pins 0 and 3 are affected by the function.

This protocol is most useful for the [GPIOPinRead\(\)](#) and [GPIOPinWrite\(\)](#) functions; a read returns only the values of the requested pins (with the other pin values masked out) and a write only affects the requested pins simultaneously (that is, the state of multiple GPIO pins can be changed at the same time). This data masking for the GPIO pin state occurs in the hardware; a single read or write is issued to the hardware, which interprets some of the address bits as an indication of the GPIO pins to operate on (and therefore the ones to not affect). See the part data sheet for details of the GPIO data register address-based bit masking.

For functions that have a *ucPin* (singular) parameter, only a single pin is affected by the function. In this case, the value specifies the pin number (that is, 0 through 7).

NOTE: A subset of GPIO pins on many Tiva devices are protected by a locking mechanism to prevent inadvertent reconfiguration. The actual pins vary by device but typically include any pin that is part of the JTAG or SWD interface, and any pin which may be configured as an NMI input. On a TM4C129XNCZAD part, for example, this affects pins PC[3:0], PD7 and PE7. Locked pins may not be reconfigured without first unlocking them using the mechanism described under “Commit Control” in the GPIO chapter of your device’s datasheet. This mechanism is also illustrated in the TivaWare “gpio_jtag” example application included for all target evaluation and development kits.

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

15.2 API Functions

Functions

- void `GPIOADCTriggerDisable` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOADCTriggerEnable` (uint32_t ui32Port, uint8_t ui8Pins)
- uint32_t `GPIODirModeGet` (uint32_t ui32Port, uint8_t ui8Pin)
- void `GPIODirModeSet` (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32PinIO)
- void `GPIODMATriggerDisable` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIODMATriggerEnable` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOIntClear` (uint32_t ui32Port, uint32_t ui32IntFlags)
- void `GPIOIntDisable` (uint32_t ui32Port, uint32_t ui32IntFlags)
- void `GPIOIntEnable` (uint32_t ui32Port, uint32_t ui32IntFlags)
- void `GPIOIntRegister` (uint32_t ui32Port, void (*pfnIntHandler)(void))
- uint32_t `GPIOIntStatus` (uint32_t ui32Port, bool bMasked)
- uint32_t `GPIOIntTypeGet` (uint32_t ui32Port, uint8_t ui8Pin)
- void `GPIOIntTypeSet` (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32IntType)
- void `GPIOIntUnregister` (uint32_t ui32Port)
- void `GPIOPadConfigGet` (uint32_t ui32Port, uint8_t ui8Pin, uint32_t *pui32Strength, uint32_t *pui32PinType)
- void `GPIOPadConfigSet` (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32Strength, uint32_t ui32PinType)
- void `GPIOPinConfigure` (uint32_t ui32PinConfig)
- int32_t `GPIOPinRead` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeADC` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeCAN` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeCIR` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeComparator` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeEPI` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeEthernetLED` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeEthernetMII` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeGPIOInput` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeGPIOOutput` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeGPIOOutputOD` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeI2C` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeI2CSCL` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeKBColumn` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeKBRow` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeLCD` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeLEDSeq` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeLPC` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypePECIRx` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypePECITx` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypePWM` (uint32_t ui32Port, uint8_t ui8Pins)
- void `GPIOPinTypeQEI` (uint32_t ui32Port, uint8_t ui8Pins)

- void [GPIOPinTypeSSI](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeTimer](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeUART](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeUSBAnalog](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeUSBDigital](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeWakeHigh](#) (uint32_t ui32Port, uint8_t ui8Pins)
- void [GPIOPinTypeWakeLow](#) (uint32_t ui32Port, uint8_t ui8Pins)
- uint32_t [GPIOPinWakeStatus](#) (uint32_t ui32Port)
- void [GPIOPinWrite](#) (uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val)

15.2.1 Detailed Description

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 [GPIODirModeSet\(\)](#), [GPIOPadConfigSet\(\)](#), and [GPIOPinConfigure\(\)](#). The configuration can be read back with [GPIODirModeGet\(\)](#) and [GPIOPadConfigGet\(\)](#).

The GPIO pin state is accessed with [GPIOPinRead\(\)](#) and [GPIOPinWrite\(\)](#).

The GPIO interrupts are handled with [GPIOIntTypeSet\(\)](#), [GPIOIntTypeGet\(\)](#), [GPIOIntEnable\(\)](#), [GPIOIntDisable\(\)](#), [GPIOIntStatus\(\)](#), [GPIOIntClear\(\)](#), [GPIOIntRegister\(\)](#), and [GPIOIntUnregister\(\)](#).

15.2.2 GPIO Pin Configuration

Many of the GPIO pins on the TM4C123 and TM4C129 devices have other peripheral functions that can also use the GPIO pins for peripheral pins. The Peripheral Driver Library provides a set of convenience functions to configure the pins in the required or recommended input/output configuration for a particular peripheral; these are [GPIOPinTypeADC\(\)](#), [GPIOPinTypeCAN\(\)](#), [GPIOPinTypeComparator\(\)](#), [GPIOPinTypeEPI\(\)](#), [GPIOPinTypeEthernetLED\(\)](#), [GPIOPinTypeEthernetMII\(\)](#), [GPIOPinTypeGPIOInput\(\)](#), [GPIOPinTypeGPIOOutput\(\)](#), [GPIOPinTypeGPIOOutputOD\(\)](#), [GPIOPinTypeI2C\(\)](#), [GPIOPinTypeI2CSCL\(\)](#), [GPIOPinTypeLCD\(\)](#), [GPIOPinTypePWM\(\)](#), [GPIOPinTypeQEI\(\)](#), [GPIOPinTypeSSI\(\)](#), [GPIOPinTypeTimer\(\)](#), [GPIOPinTypeUART\(\)](#), [GPIOPinTypeUSBAnalog\(\)](#), [GPIOPinTypeUSBDigital\(\)](#), [GPIOPinTypeWakeHigh\(\)](#), [GPIOPinTypeWakeLow\(\)](#), [GPIOPinWakeStatus\(\)](#), [GPIODMATriggerEnable\(\)](#), [GPIODMATriggerDisable\(\)](#), [GPIOADCTriggerEnable\(\)](#), and [GPIOADCTriggerDisable\(\)](#). In order to complete the pin configuration, the [GPIOPinConfigure\(\)](#) function must also be called to enable the desired peripheral function on the given GPIO pin. The [GPIOPinConfigure\(\)](#) function uses the pin definitions located in the `driverlib/pin_map.h` file. These definitions follow the **GPIO_P<port><pin>_<peripheral_function>** naming scheme. The available pin mappings are supplied on a per-device basis and are selected using the **PART_<partno>** defines to enable only the pin definitions that are valid for the given device. For example, on the TM4C129XNCZAD device the UART1 RX function can be enabled on one of two pins. The UART1 RX is found on GPIO port B pin 0(**GPIO_PB0_U1RX**) or it can also be found on GPIO port Q pin 4(**GPIO_PQ4_U1RX**). The application must define the **PART_TM4C129XNCZAD** in order to get the correct pin mappings for the TM4C129XNCZAD device.

Note:

The **PART_<partno>** macros also control the mapping of interrupt names to interrupt numbers. See the [Interrupt Mapping](#) section of this document for more details on how these defines are used to determine interrupt mapping.

A locked GPIO pin must be unlocked prior to making calls to [GPIODirModeSet\(\)](#), [GPIOPadConfigSet\(\)](#) or any of the [GPIOPinType](#) functions.

15.2.3 Function Documentation

15.2.3.1 GPIOADCTriggerDisable

Disable a GPIO pin as a trigger to start an ADC capture.

Prototype:

```
void  
GPIOADCTriggerDisable(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

This function disables a GPIO pin to be used as a trigger to start an ADC sequence. This function can be used to disable this feature if it was enabled via a call to [GPIOADCTriggerEnable\(\)](#).

Returns:

None.

15.2.3.2 GPIOADCTriggerEnable

Enables a GPIO pin as a trigger to start an ADC capture.

Prototype:

```
void  
GPIOADCTriggerEnable(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

This function enables a GPIO pin to be used as a trigger to start an ADC sequence. Any GPIO pin can be configured to be an external trigger for an ADC sequence. The GPIO pin still generates interrupts if the interrupt is enabled for the selected pin. To enable the use of a GPIO pin to trigger the ADC module, the [ADCSequenceConfigure\(\)](#) function must be called with the **ADC_TRIGGER_EXTERNAL** parameter.

Returns:

None.

15.2.3.3 GPIODirModeGet

Gets the direction and mode of a pin.

Prototype:

```
uint32_t
GPIODirModeGet(uint32_t ui32Port,
               uint8_t ui8Pin)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pin is the pin number.

Description:

This function gets the direction and control mode for a specified pin on the selected GPIO port. The pin can be configured as either an input or output under software control, or it can be under hardware control. The type of control and direction are returned as an enumerated data type.

Returns:

Returns one of the enumerated data types described for [GPIODirModeSet\(\)](#).

15.2.3.4 GPIODirModeSet

Sets the direction and mode of the specified pin(s).

Prototype:

```
void
GPIODirModeSet(uint32_t ui32Port,
                uint8_t ui8Pins,
                uint32_t ui32PinIO)
```

Parameters:

ui32Port is the base address of the GPIO port

ui8Pins is the bit-packed representation of the pin(s).

ui32PinIO is the pin direction and/or mode.

Description:

This function configures the specified pin(s) on the selected GPIO port as either input or output under software control, or it configures the pin to be under hardware control.

The parameter *ui32PinIO* is an enumerated data type that can be one of the following values:

- **GPIO_DIR_MODE_IN**
- **GPIO_DIR_MODE_OUT**
- **GPIO_DIR_MODE_HW**

where **GPIO_DIR_MODE_IN** specifies that the pin is programmed as a software controlled input, **GPIO_DIR_MODE_OUT** specifies that the pin is programmed as a software controlled output, and **GPIO_DIR_MODE_HW** specifies that the pin is placed under hardware control.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

[GPIOPadConfigSet\(\)](#) must also be used to configure the corresponding pad(s) in order for them to propagate the signal to/from the GPIO.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.5 GPIODMATriggerDisable

Disables a GPIO pin as a trigger to start a DMA transaction.

Prototype:

```
void  
GPIODMATriggerDisable(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

This function disables a GPIO pin from being used as a trigger to start a uDMA transaction. This function can be used to disable this feature if it was enabled via a call to [GPIODMATriggerEnable\(\)](#).

Returns:

None.

15.2.3.6 GPIODMATriggerEnable

Enables a GPIO pin as a trigger to start a DMA transaction.

Prototype:

```
void  
GPIODMATriggerEnable(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

This function enables a GPIO pin to be used as a trigger to start a uDMA transaction. Any GPIO pin can be configured to be an external trigger for the uDMA. The GPIO pin still generates interrupts if the interrupt is enabled for the selected pin.

Returns:

None.

15.2.3.7 GPIOIntClear

Clears the specified interrupt sources.

Prototype:

```
void
GPIOIntClear(uint32_t ui32Port,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

Clears the interrupt for the specified interrupt source(s).

The *ui32IntFlags* parameter is the logical OR of the **GPIO_INT_*** values.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

15.2.3.8 GPIOIntDisable

Disables the specified GPIO interrupts.

Prototype:

```
void
GPIOIntDisable(uint32_t ui32Port,
               uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

This function disables the indicated GPIO interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **GPIO_INT_PIN_0** - interrupt due to activity on Pin 0.

- **GPIO_INT_PIN_1** - interrupt due to activity on Pin 1.
- **GPIO_INT_PIN_2** - interrupt due to activity on Pin 2.
- **GPIO_INT_PIN_3** - interrupt due to activity on Pin 3.
- **GPIO_INT_PIN_4** - interrupt due to activity on Pin 4.
- **GPIO_INT_PIN_5** - interrupt due to activity on Pin 5.
- **GPIO_INT_PIN_6** - interrupt due to activity on Pin 6.
- **GPIO_INT_PIN_7** - interrupt due to activity on Pin 7.
- **GPIO_INT_DMA** - interrupt due to DMA activity on this GPIO module.

Returns:

None.

15.2.3.9 GPIOIntEnable

Enables the specified GPIO interrupts.

Prototype:

```
void  
GPIOIntEnable(uint32_t ui32Port,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui32IntFlags is the bit mask of the interrupt sources to enable.

Description:

This function enables the indicated GPIO interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The **ui32IntFlags** parameter is the logical OR of any of the following:

- **GPIO_INT_PIN_0** - interrupt due to activity on Pin 0.
- **GPIO_INT_PIN_1** - interrupt due to activity on Pin 1.
- **GPIO_INT_PIN_2** - interrupt due to activity on Pin 2.
- **GPIO_INT_PIN_3** - interrupt due to activity on Pin 3.
- **GPIO_INT_PIN_4** - interrupt due to activity on Pin 4.
- **GPIO_INT_PIN_5** - interrupt due to activity on Pin 5.
- **GPIO_INT_PIN_6** - interrupt due to activity on Pin 6.
- **GPIO_INT_PIN_7** - interrupt due to activity on Pin 7.
- **GPIO_INT_DMA** - interrupt due to DMA activity on this GPIO module.

Note:

If this call is being used to enable summary interrupts on GPIO port P or Q ([GPIOIntTypeSet\(\)](#) with **GPIO_DISCRETE_INT** not enabled), then all individual interrupts for these ports must be enabled in the GPIO module using [GPIOIntEnable\(\)](#) and all but the interrupt for pin 0 must be disabled in the NVIC using the [IntDisable\(\)](#) function. The summary interrupts for the ports are routed to the INT_GPIOP0 or INT_GPIOQ0 which must be enabled to handle the interrupt. If this is not done then any individual GPIO pin interrupts that are left enabled also trigger the individual interrupts.

Returns:

None.

15.2.3.10 GPIOIntRegister

Registers an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOIntRegister(uint32_t ui32Port,
                void (*pfnIntHandler) (void))
```

Parameters:

ui32Port is the base address of the GPIO port.

pfnIntHandler is a pointer to the GPIO port interrupt handling function.

Description:

This function ensures that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected from the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with [GPIOIntEnable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

15.2.3.11 GPIOIntStatus

Gets interrupt status for the specified GPIO port.

Prototype:

```
uint32_t
GPIOIntStatus(uint32_t ui32Port,
              bool bMasked)
```

Parameters:

ui32Port is the base address of the GPIO port.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

Returns the current interrupt status for the specified GPIO module. The value returned is the logical OR of the **GPIO_INT_*** values that are currently active.

15.2.3.12 GPIOIntTypeGet

Gets the interrupt type for a pin.

Prototype:

```
uint32_t  
GPIOIntTypeGet(uint32_t ui32Port,  
                uint8_t ui8Pin)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pin is the pin number.

Description:

This function gets the interrupt type for a specified pin on the selected GPIO port. The pin can be configured as a falling-edge, rising-edge, or both-edges detected interrupt, or it can be configured as a low-level or high-level detected interrupt. The type of interrupt detection mechanism is returned and can include the **GPIO_DISCRETE_INT** flag.

Returns:

Returns one of the flags described for [GPIOIntTypeSet\(\)](#).

15.2.3.13 GPIOIntTypeSet

Sets the interrupt type for the specified pin(s).

Prototype:

```
void  
GPIOIntTypeSet(uint32_t ui32Port,  
                uint8_t ui8Pins,  
                uint32_t ui32IntType)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

ui32IntType specifies the type of interrupt trigger mechanism.

Description:

This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

One of the following flags can be used to define the *ui32IntType* parameter:

- **GPIO_FALLING_EDGE** sets detection to edge and trigger to falling
- **GPIO_RISING_EDGE** sets detection to edge and trigger to rising
- **GPIO_BOTH_EDGES** sets detection to both edges
- **GPIO_LOW_LEVEL** sets detection to low level
- **GPIO_HIGH_LEVEL** sets detection to high level

In addition to the above flags, the following flag can be OR'd in to the *ui32IntType* parameter:

- **GPIO_DISCRETE_INT** sets discrete interrupts for each pin on a GPIO port.

The **GPIO_DISCRETE_INT** is not available on all devices or all GPIO ports, consult the data sheet to ensure that the device and the GPIO port supports discrete interrupts.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

In order to avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

Returns:

None.

15.2.3.14 GPIOIntUnregister

Removes an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOIntUnregister(uint32_t ui32Port)
```

Parameters:

ui32Port is the base address of the GPIO port.

Description:

This function unregisters the interrupt handler for the specified GPIO port. This function also disables the corresponding GPIO port interrupt in the interrupt controller; individual GPIO interrupts and interrupt sources must be disabled with [GPIOIntDisable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

15.2.3.15 GPIOPadConfigGet

Gets the pad configuration for a pin.

Prototype:

```
void
GPIOPadConfigGet(uint32_t ui32Port,
                  uint8_t ui8Pin,
                  uint32_t *pui32Strength,
                  uint32_t *pui32PinType)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pin is the pin number.

pui32Strength is a pointer to storage for the output drive strength.

pui32PinType is a pointer to storage for the output drive type.

Description:

This function gets the pad configuration for a specified pin on the selected GPIO port. The values returned in *pui32Strength* and *pui32PinType* correspond to the values used in [GPIOPadConfigSet\(\)](#). This function also works for pin(s) configured as input pin(s); however, the only meaningful data returned is whether the pin is terminated with a pull-up or down resistor.

Returns:

None

15.2.3.16 GPIOPadConfigSet

Sets the pad configuration for the specified pin(s).

Prototype:

```
void  
GPIOPadConfigSet(uint32_t ui32Port,  
                  uint8_t ui8Pins,  
                  uint32_t ui32Strength,  
                  uint32_t ui32PinType)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

ui32Strength specifies the output drive strength.

ui32PinType specifies the pin type.

Description:

This function sets the drive strength and type for the specified pin(s) on the selected GPIO port. For pin(s) configured as input ports, the pad is configured as requested, but the only real effect on the input is the configuration of the pull-up or pull-down termination.

The parameter **ui32Strength** can be one of the following values:

- **GPIO_STRENGTH_2MA**
- **GPIO_STRENGTH_4MA**
- **GPIO_STRENGTH_8MA**
- **GPIO_STRENGTH_8MA_SC**
- **GPIO_STRENGTH_6MA**
- **GPIO_STRENGTH_10MA**
- **GPIO_STRENGTH_12MA**

where **GPIO_STRENGTH_xMA** specifies either 2, 4, or 8 mA output drive strength, and **GPIO_OUT_STRENGTH_8MA_SC** specifies 8 mA output drive with slew control.

Some Tiva devices also support output drive strengths of 6, 10, and 12 mA.

The parameter **ui32PinType** can be one of the following values:

- **GPIO_PIN_TYPE_STD**
- **GPIO_PIN_TYPE_STD_WPU**
- **GPIO_PIN_TYPE_STD_WPD**
- **GPIO_PIN_TYPE_OD**
- **GPIO_PIN_TYPE_ANALOG**
- **GPIO_PIN_TYPE_WAKE_HIGH**
- **GPIO_PIN_TYPE_WAKE_LOW**

where **GPIO_PIN_TYPE_STD*** specifies a push-pull pin, **GPIO_PIN_TYPE_OD*** specifies an open-drain pin, ***_WPU** specifies a weak pull-up, ***_WPD** specifies a weak pull-down, and **GPIO_PIN_TYPE_ANALOG** specifies an analog input.

The **GPIO_PIN_TYPE_WAKE_*** settings specify the pin to be used as a hibernation wake source. The pin sense level can be high or low. These settings are only available on some Tiva devices.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.17 GPIOPinConfigure

Configures the alternate function of a GPIO pin.

Prototype:

```
void
GPIOPinConfigure(uint32_t ui32PinConfig)
```

Parameters:

ui32PinConfig is the pin configuration value, specified as only one of the **GPIO_P??_???** values.

Description:

This function configures the pin mux that selects the peripheral function associated with a particular GPIO pin. Only one peripheral function at a time can be associated with a GPIO pin, and each peripheral function should only be associated with a single GPIO pin at a time (despite the fact that many of them can be associated with more than one GPIO pin). To fully configure a pin, a **GPIOPinType()** function should also be called.

The available mappings are supplied on a per-device basis in `pin_map.h`. The **PART_<partno>** defines controls which set of defines are included so that they match the device that is being used. For example, **PART_TM4C129XNCZAD** must be defined in order to get the correct pin mappings for the TM4C129XNCZAD device.

Note:

If the same signal is assigned to two different GPIO port pins, the signal is assigned to the port with the lowest letter and the assignment to the higher letter port is ignored.

Returns:

None.

15.2.3.18 GPIOPinRead

Reads the values present of the specified pin(s).

Prototype:

```
int32_t  
GPIOPinRead(uint32_t ui32Port,  
            uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The values at the specified pin(s) are read, as specified by ***ui8Pins***. Values are returned for both input and output pin(s), and the value for pin(s) that are not specified by ***ui8Pins*** are set to 0.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

Returns a bit-packed byte providing the state of the specified pin, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Any bit that is not specified by ***ui8Pins*** is returned as a 0. Bits 31:8 should be ignored.

15.2.3.19 GPIOPinTypeADC

Configures pin(s) for use as analog-to-digital converter inputs.

Prototype:

```
void  
GPIOPinTypeADC(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The analog-to-digital converter input pins must be properly configured for the analog-to-digital peripheral to function correctly. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an ADC input; it only configures an ADC input pin for proper operation.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration.

These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.20 GPIOPinTypeCAN

Configures pin(s) for use as a CAN device.

Prototype:

```
void
GPIOPinTypeCAN(uint32_t ui32Port,
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The CAN pins must be properly configured for the CAN peripherals to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a CAN pin; it only configures a CAN pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the CAN function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.21 GPIOPinTypeCIR

Configures pin(s) for use as Consumer Infrared inputs or outputs.

Prototype:

```
void
GPIOPinTypeCIR(uint32_t ui32Port,
                 uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as Consumer Infrared pins. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a CIR pin; it only configures a CIR pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the Consumer Infrared function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.22 GPIOPinTypeComparator

Configures pin(s) for use as an analog comparator input.

Prototype:

```
void
GPIOPinTypeComparator(uint32_t ui32Port,
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The analog comparator input pins must be properly configured for the analog comparator to function correctly. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an analog comparator input; it only configures an analog comparator pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the analog comparator function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration.

These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.23 GPIOPinTypeEPI

Configures pin(s) for use by the external peripheral interface.

Prototype:

```
void
GPIOPinTypeEPI(uint32_t ui32Port,
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The external peripheral interface pins must be properly configured for the external peripheral interface to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an external peripheral interface pin; it only configures an external peripheral interface pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the external peripheral interface function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.24 GPIOPinTypeEthernetLED

Configures pin(s) for use by the Ethernet peripheral as LED signals.

Prototype:

```
void  
GPIOPinTypeEthernetLED(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The Ethernet peripheral provides four signals that can be used to drive an LED (for example, for link status/activity). This function provides a typical configuration for the pins.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an Ethernet LED pin; it only configures an Ethernet LED pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure the pin for the Ethernet LED function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_itag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.25 GPIOPinTypeEthernetMII

Configures pin(s) for use by the Ethernet peripheral as MII signals.

Prototype:

```
void  
GPIOPinTypeEthernetMII(uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The Ethernet peripheral on some parts provides a set of MII signals that are used to connect to an external PHY. This function provides a typical configuration for the pins.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an Ethernet MII pin; it only configures an Ethernet MII pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure the pin for the Ethernet MII function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.26 GPIOPinTypeGPIOInput

Configures pin(s) for use as GPIO inputs.

Prototype:

```
void
GPIOPinTypeGPIOInput(uint32_t ui32Port,
                     uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as GPIO inputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.27 GPIOPinTypeGPIOOutput

Configures pin(s) for use as GPIO outputs.

Prototype:

```
void
GPIOPinTypeGPIOOutput(uint32_t ui32Port,
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as GPIO outputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.28 GPIOPinTypeGPIOOutputOD

Configures pin(s) for use as GPIO open drain outputs.

Prototype:

```
void
GPIOPinTypeGPIOOutputOD(uint32_t ui32Port,
                       uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as GPIO outputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and

GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.29 GPIOPinTypeI2C

Configures pin for use as SDA by the I2C peripheral.

Prototype:

```
void
GPIOPinTypeI2C(uint32_t ui32Port,
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin.

Description:

The I2C pins must be properly configured for the I2C peripheral to function correctly. This function provides the proper configuration for the SDA pin.

The pin is specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an I2C SDA pin; it only configures an I2C SDA pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the I2C SDA function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.30 GPIOPinTypeI2CSCL

Configures pin for use as SCL by the I2C peripheral.

Prototype:

```
void
GPIOPinTypeI2CSCL(uint32_t ui32Port,
                    uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin.

Description:

The I2C pins must be properly configured for the I2C peripheral to function correctly. This function provides the proper configuration for the SCL pin.

The pin is specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an I2C SCL pin; it only configures an I2C SCL pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the I2C SCL function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.31 GPIOPinTypeKBColumn

Configures pin(s) for use as scan matrix keyboard columns (inputs).

Prototype:

```
void  
GPIOPinTypeKBColumn(uint32_t ui32Port,  
                     uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as scan matrix keyboard inputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a scan matrix keyboard column pin; it only configures a scan matrix keyboard column pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the scan matrix keyboard function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.32 GPIOPinTypeKBRow

Configures pin(s) for use as scan matrix keyboard rows (outputs).

Prototype:

```
void
GPIOPinTypeKBRow(uint32_t ui32Port,
                  uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as scan matrix keyboard outputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a scan matrix keyboard row pin; it only configures a scan matrix keyboard row pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the scan matrix keyboard function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.33 GPIOPinTypeLCD

Configures pin(s) for use by the LCD Controller.

Prototype:

```
void  
GPIOPinTypeLCD(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The LCD controller pins must be properly configured for the LCD controller to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an LCD pin; it only configures an LCD pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the LCD controller function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.34 GPIOPinTypeLEDSeq

Configures pin(s) for use as an LED sequencer output.

Prototype:

```
void  
GPIOPinTypeLEDSeq(uint32_t ui32Port,  
                   uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as LED sequencers. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an LED sequencer output pin; it only configures an LED sequencer output pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the LED sequencer function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.35 GPIOPinTypeLPC

Configures pin(s) for use by the LPC module.

Prototype:

```
void
GPIOPinTypeLPC(uint32_t ui32Port,
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The LPC pins must be properly configured for the LPC module to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into an LPC pin; it only configures an LPC pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the LPC function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.36 GPIOPinTypePECIRx

Configures a pin for receive use by the PECL module.

Prototype:

```
void
GPIOPinTypePECIRx(uint32_t ui32Port,
                   uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The PECL receive pin must be properly configured for the PECL module to function correctly. This function provides a typical configuration for that pin.

The pin is specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a PECL receive pin; it only configures a PECL receive pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the PECL receive function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.37 GPIOPinTypePECITx

Configures a pin for transmit use by the PECL module.

Prototype:

```
void
GPIOPinTypePECITx(uint32_t ui32Port,
                   uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The PECL transmit pin must be properly configured for the PECL module to function correctly. This function provides a typical configuration for that pin.

The pin is specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a PECI transmit pin; it only configures a PECI transmit pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure the pin for the PECI transmit function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.38 GPIOPinTypePWM

Configures pin(s) for use by the PWM peripheral.

Prototype:

```
void
GPIOPinTypePWM(uint32_t ui32Port,
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The PWM pins must be properly configured for the PWM peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a PWM pin; it only configures a PWM pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the PWM function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.39 GPIOPinTypeQEI

Configures pin(s) for use by the QEI peripheral.

Prototype:

```
void  
GPIOPinTypeQEI(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The QEI pins must be properly configured for the QEI peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, not using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a QEI pin; it only configures a QEI pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the QEI function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.40 GPIOPinTypeSSI

Configures pin(s) for use by the SSI peripheral.

Prototype:

```
void  
GPIOPinTypeSSI(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The SSI pins must be properly configured for the SSI peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a SSI pin; it only configures a SSI pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the SSI function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.41 GPIOPinTypeTimer

Configures pin(s) for use by the Timer peripheral.

Prototype:

```
void
GPIOPinTypeTimer(uint32_t ui32Port,
                  uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The CCP pins must be properly configured for the timer peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a timer pin; it only configures a timer pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the CCP function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.42 GPIOPinTypeUART

Configures pin(s) for use by the UART peripheral.

Prototype:

```
void  
GPIOPinTypeUART(uint32_t ui32Port,  
                 uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The UART pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a UART pin; it only configures a UART pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the UART function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.43 GPIOPinTypeUSBAnalog

Configures pin(s) for use by the USB peripheral.

Prototype:

```
void
GPIOPinTypeUSBAnalog(uint32_t ui32Port,
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

USB analog pins must be properly configured for the USB peripheral to function correctly. This function provides the proper configuration for any USB analog pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a USB pin; it only configures a USB pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the USB function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_itag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.44 GPIOPinTypeUSBDigital

Configures pin(s) for use by the USB peripheral.

Prototype:

```
void
GPIOPinTypeUSBDigital(uint32_t ui32Port,
                      uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

USB digital pins must be properly configured for the USB peripheral to function correctly. This function provides a typical configuration for the digital USB pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

This function should only be used with EPEN and PFAULT pins as all other USB pins are analog in nature or are not used in devices without OTG functionality.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This function cannot be used to turn any pin into a USB pin; it only configures a USB pin for proper operation. Note that a [GPIOPinConfigure\(\)](#) function call is also required to properly configure a pin for the USB function.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.45 GPIOPinTypeWakeHigh

Configures pin(s) for use as a hibernate wake-on-high source.

Prototype:

```
void  
GPIOPinTypeWakeHigh(uint32_t ui32Port,  
                     uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as hibernate wake-high inputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.46 GPIOPinTypeWakeLow

Configures pin(s) for use as a hibernate wake-on-low source.

Prototype:

```
void
GPIOPinTypeWakeLow(uint32_t ui32Port,
                   uint8_t ui8Pins)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as hibernate wake-low inputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration. These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

None.

15.2.3.47 GPIOPinWakeStatus

Retrieves the wake pins status.

Prototype:

```
uint32_t
GPIOPinWakeStatus(uint32_t ui32Port)
```

Parameters:

ui32Port is the base address of the GPIO port.

Description:

This function returns the GPIO wake pin status values. The returned bitfield shows low or high pin state via a value of 0 or 1.

Note:

This function is not available on all devices, consult the data sheet to ensure that the device you are using supports GPIO wake pins.

A subset of GPIO pins on Tiva devices, notably those used by the JTAG/SWD interface and any pin capable of acting as an NMI input, are locked against inadvertent reconfiguration.

These pins must be unlocked using direct register writes to the relevant GPIO_O_LOCK and GPIO_O_CR registers before this function can be called. Please see the “gpio_jtag” example application for the mechanism required and consult your part datasheet for information on affected pins.

Returns:

Returns the wake pin status.

15.2.3.48 GPIOPinWrite

Writes a value to the specified pin(s).

Prototype:

```
void  
GPIOPinWrite(uint32_t ui32Port,  
             uint8_t ui8Pins,  
             uint8_t ui8Val)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin(s).

ui8Val is the value to write to the pin(s).

Description:

Writes the corresponding bit values to the output pin(s) specified by **ui8Pins**. Writing to a pin configured as an input pin has no effect.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

15.3 Programming Example

The following example shows how to use the GPIO API to initialize the GPIO, enable interrupts, read data from pins, and write data to pins.

Example: Configure pins for use as input, interrupt, and output GPIOs.

```
int32_t i32Val;  
  
//  
// Register the port-level interrupt handler. This handler is the first  
// level interrupt handler for all the pin interrupts.  
//  
GPIOIntRegister(GPIO_PORTA_BASE, PortAIntHandler);  
  
//  
// Initialize the GPIO pin configuration.  
//
```

```

// Set pins 2, 4, and 5 as input, SW controlled.
//
GPIOPinTypeGPIOInput(GPIO_PORTA_BASE,
                      GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

//
// Set pins 0 and 3 as output, SW controlled.
//
GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_3);

//
// Make pins 2 and 4 rising edge triggered interrupts.
//
GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4, GPIO_RISING_EDGE);

//
// Make pin 5 high level triggered interrupts.
//
GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_5, GPIO_HIGH_LEVEL);

//
// Read some pins.
//
i32Val = GPIOPinRead(GPIO_PORTA_BASE,
                      (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
                       GPIO_PIN_4 | GPIO_PIN_5));

//
// Write some pins. Even though pins 2, 4, and 5 are specified, those pins
// are unaffected by this write because they are configured as inputs. At
// the end of this write, pin 0 is low, and pin 3 is high.
//
GPIOPinWrite(GPIO_PORTA_BASE,
              (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
               GPIO_PIN_4 | GPIO_PIN_5),
              (GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 |
               GPIO_PIN_7));

//
// Enable the pin interrupts.
//
GPIOIntEnable(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

```

Example: Configure Port B Pins for use as a UART 1.

```

//
// Configure GPIO Port B pins 0 and 1 to be used as UART1.
//
GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Enable UART1 functionality on GPIO Port B pins 0 and 1.
//
GPIOPinConfigure(GPIO_PB0_U1RX);
GPIOPinConfigure(GPIO_PB1_U1TX);

```


16 Hibernation Module

Introduction	289
API Functions	289
Programming Example	316

16.1 Introduction

The Hibernate API provides a set of functions for using the Hibernation module on the Tiva microcontroller. The Hibernation module allows the software application to remove power from the microcontroller, and then be powered on later based on specific time or when the external **WAKE** pin is asserted. The API provides functions to configure wake conditions, manage interrupts, read status, save and restore program state information, and request hibernation mode.

Some of the features of the Hibernation module are:

- 32-bit real time clock, with 15-bit subseconds counter on some devices
- Internal low frequency oscillator
- Calendar mode for the hibernation counter
- Tamper detection and response
- Trim register for fine tuning the RTC rate
- One RTC match registers for generating RTC events
- External **WAKE** pin to initiate a wake-up
- External **RST** pin and/or four GPIO port pins as alternate wake-up sources.
- Maintain GPIO state during hibernation.
- Low-battery detection
- 16 32-bit words of battery-backed memory
- Programmable interrupts for hibernation events Check the device data sheet to determine if a selected MCU supports each of these features.

This driver is contained in `driverlib/hibernate.c`, with `driverlib/hibernate.h` containing the API declarations for use by applications.

16.2 API Functions

Functions

- `uint32_t HibernateBatCheckDone (void)`
- `void HibernateBatCheckStart (void)`
- `int HibernateCalendarGet (struct tm *psTime)`
- `void HibernateCalendarMatchGet (uint32_t ui32Index, struct tm *psTime)`
- `void HibernateCalendarMatchSet (uint32_t ui32Index, struct tm *psTime)`
- `void HibernateCalendarSet (struct tm *psTime)`

- void **HibernateClockConfig** (uint32_t ui32Config)
- void **HibernateCounterMode** (uint32_t ui32Config)
- void **HibernateDataGet** (uint32_t *pui32Data, uint32_t ui32Count)
- void **HibernateDataSet** (uint32_t *pui32Data, uint32_t ui32Count)
- void **HibernateDisable** (void)
- void **HibernateEnableExpClk** (uint32_t ui32HibClk)
- void **HibernateGPIORetentionDisable** (void)
- void **HibernateGPIORetentionEnable** (void)
- bool **HibernateGPIORetentionGet** (void)
- void **HibernateIntClear** (uint32_t ui32IntFlags)
- void **HibernateIntDisable** (uint32_t ui32IntFlags)
- void **HibernateIntEnable** (uint32_t ui32IntFlags)
- void **HibernateIntRegister** (void (*pfnHandler)(void))
- uint32_t **HibernateIntStatus** (bool bMasked)
- void **HibernateIntUnregister** (void)
- uint32_t **HibernateIsActive** (void)
- uint32_t **HibernateLowBatGet** (void)
- void **HibernateLowBatSet** (uint32_t ui32LowBatFlags)
- void **HibernateRequest** (void)
- void **HibernateRTCDisable** (void)
- void **HibernateRTCEnable** (void)
- uint32_t **HibernateRTCGet** (void)
- uint32_t **HibernateRTCMatchGet** (uint32_t ui32Match)
- void **HibernateRTCMatchSet** (uint32_t ui32Match, uint32_t ui32Value)
- void **HibernateRTCSet** (uint32_t ui32RTCValue)
- uint32_t **HibernateRTCSSGet** (void)
- uint32_t **HibernateRTCSSMatchGet** (uint32_t ui32Match)
- void **HibernateRTCSSMatchSet** (uint32_t ui32Match, uint32_t ui32Value)
- uint32_t **HibernateRTCTrimGet** (void)
- void **HibernateRTCTrimSet** (uint32_t ui32Trim)
- void **HibernateTamperDisable** (void)
- void **HibernateTamperEnable** (void)
- void **HibernateTamperEventsClear** (void)
- void **HibernateTamperEventsClearNoLock** (void)
- void **HibernateTamperEventsConfig** (uint32_t ui32Config)
- bool **HibernateTamperEventsGet** (uint32_t ui32Index, uint32_t *pui32RTC, uint32_t *pui32Event)
- void **HibernateTamperExtOscRecover** (void)
- bool **HibernateTamperExtOscValid** (void)
- void **HibernateTamperIODisable** (uint32_t ui32Input)
- void **HibernateTamperIOEnable** (uint32_t ui32Input, uint32_t ui32Config)
- void **HibernateTamperLock** (void)
- uint32_t **HibernateTamperStatusGet** (void)
- void **HibernateTamperUnLock** (void)
- uint32_t **HibernateWakeGet** (void)
- void **HibernateWakeSet** (uint32_t ui32WakeFlags)

16.2.1 Detailed Description

The Hibernation module must be enabled before it can be used. Use the [HibernateEnableExpClk\(\)](#) function to enable it. If a crystal is used for the clock source, then the initializing code must allow time for the crystal to stabilize after calling the [HibernateEnableExpClk\(\)](#) function. Refer to the device data sheet for information about crystal stabilization time. If an oscillator is used, then no delay is necessary. After the module is enabled, the clock source must be configured by calling [HibernateClockConfig\(\)](#).

In order to use the RTC feature of the Hibernation module, the RTC must be enabled by calling [HibernateRTCEnable\(\)](#). It can be later disabled by calling [HibernateRTCDisable\(\)](#). These functions can be called at any time to start and stop the RTC. The RTC value can be read or set by using the [HibernateRTCGet\(\)](#) and [HibernateRTCSet\(\)](#) functions. The match register can be read and set by using the [HibernateRTCMatchGet\(\)](#), and [HibernateRTCMatchSet\(\)](#), functions. The real-time clock rate can be adjusted by using the trim register. Use the [HibernateRTCTrimGet\(\)](#) and [HibernateRTCTrimSet\(\)](#) functions for this purpose. The value of the subseconds counter can be read using [HibernateRTCSSGet\(\)](#). The match value of the subseconds counter can be set and read using the [HibernateRTCSSMatchSet\(\)](#) and [HibernateRTCSSMatchSet\(\)](#) functions.

Some devices provide a calendar mode of operation for the RTC. The value of the RTC can be read or set in calendar mode with the [HibernateCalendarGet\(\)](#) and [HibernateCalendarSet\(\)](#) functions. The match register can also be read and set in calendar mode with the [HibernateCalendarMatchGet\(\)](#) and [HibernateCalendarMatchSet\(\)](#) functions.

The tamper feature provides mechanisms to detect, respond to, and log system tamper events. A tamper event is detected by state transitions on select GPIOs or the failure of the external oscillator if used as a clock source. Note that the tamper GPIOs do not require special configuration to be used for the tamper function. See the device datasheet to determine which GPIOs support the tamper function.

The tamper GPIOs are configured to use with [HibernateTamperIOEnable\(\)](#) and [HibernateTamperIODisable\(\)](#). None of the GPIO API functions are needed to configure the tamper GPIOs. The tamper GPIOs configured by using these functions override any configuration by GPIO APIs. The external oscillator state can be retrieved with [HibernateTamperExtOscValid\(\)](#). If an external oscillator failure is detected, a recovery attempt can be triggered with [HibernateTamperExtOscRecover\(\)](#).

The module always responds to a tamper event by generating a tamper event signal to the System Control module. The tamper feature can be also be configured to respond to a tamper event by clearing all or part of the hibernate memory and/or waking from hibernate via [HibernateTamperEventsConfig\(\)](#). The detected events are logged with a real-time clock time stamp to allow investigation. The logged events can be managed with [HibernateTamperEventsGet\(\)](#) and [HibernateTamperEventsClear\(\)](#).

The overall status of tamper retrieved with [HibernateTamperStatusGet\(\)](#). The tamper feature can be enabled and disabled with [HibernateTamperEnable\(\)](#) and [HibernateTamperDisable\(\)](#).

Application state information can be stored in the battery-backed memory of the Hibernation module when the processor is powered off. Use the [HibernateDataSet\(\)](#) and [HibernateDataGet\(\)](#) functions to access the battery-backed memory area.

The module can be configured to wake when the external **WAKE** pin is asserted, when an RTC match occurs, and when the battery level has reached a set level. On some devices, the module can also be configured to wake when a GPIO pin is asserted or when the RESET pin is asserted. Finally on devices that support tamper detection, the module can also be configured to wake on a tamper related event. Use the [HibernateWakeSet\(\)](#) function to configure the wake conditions. The current configuration can be read by calling [HibernateWakeGet\(\)](#).

The Hibernation module can detect a low battery and signal the processor. It can also be configured to abort a hibernation request if the battery voltage is too low. Use the [HibernateLowBatSet\(\)](#) and [HibernateLowBatGet\(\)](#) functions to configure this feature. The battery level can be measured using the [HibernateBatCheckStart\(\)](#) and [HibernateBatCheckDone\(\)](#) functions.

Several functions are provided for managing interrupts. Use the [HibernateIntRegister\(\)](#) and [HibernateIntUnregister\(\)](#) functions to install or uninstall an interrupt handler into the vector table. Refer to the [IntRegister\(\)](#) function for notes about using the interrupt vector table. The module can generate several different interrupts. Use the [HibernateIntEnable\(\)](#) and [HibernateIntDisable\(\)](#) functions to enable and disable specific interrupt sources. The present interrupt status can be found by calling [HibernateIntStatus\(\)](#). In the interrupt handler, all pending interrupts must be cleared. Use the [HibernateIntClear\(\)](#) function to clear pending interrupts.

Finally, once the module is appropriately configured, the state saved, and the software application is ready to hibernate, call the [HibernateRequest\(\)](#) function. This function initiates the sequence to remove power from the processor. At a power-on reset, the software application can use the [HibernateIsActive\(\)](#) function to determine if the Hibernation module is already active and therefore does not need to be enabled. This function can provide a hint to the software that the processor is waking from hibernation instead of a cold start. The software can then use the [HibernateIntStatus\(\)](#) and [HibernateDataGet\(\)](#) functions to discover the cause of the wake and to get the saved system state.

The [HibernateEnable\(\)](#) API from previous versions of the peripheral driver library has been replaced by the [HibernateEnableExpClk\(\)](#) API. A macro has been provided in `hibernate.h` to map the old API to the new API, allowing existing applications to link and run with the new API. It is recommended that new applications use the new API in favor of the old one.

16.2.2 Function Documentation

16.2.2.1 HibernateBatCheckDone

Determines whether or not a forced battery check has completed.

Prototype:

```
uint32_t  
HibernateBatCheckDone (void)
```

Description:

This function determines whether the forced battery check initiated by a call to the [HibernateBatCheckStart\(\)](#) function has completed. This function returns a non-zero value until the battery level check has completed. Once this function returns a value of zero, the Hibernation module has completed the battery check and the [HibernateIntStatus\(\)](#) function can be used to check if the battery was low by checking if the value returned has the **HIBERNATE_INT_LOW_BAT** set.

Returns:

The value is zero when the battery level check has completed or non-zero if the check is still in process.

16.2.2.2 HibernateBatCheckStart

Forces the Hibernation module to initiate a check of the battery voltage.

Prototype:

```
void
HibernateBatCheckStart(void)
```

Description:

This function forces the Hibernation module to initiate a check of the battery voltage immediately rather than waiting for the next check interval to pass. After calling this function, the application should call the [HibernateBatCheckDone\(\)](#) function and wait for the function to return a zero value before calling the [HibernateIntStatus\(\)](#) to check if the return code has the **HIBERNATE_INT_LOW_BAT** set. If **HIBERNATE_INT_LOW_BAT** is set, the battery level is low. The application can also enable the **HIBERNATE_INT_LOW_BAT** interrupt and wait for an interrupt to indicate that the battery level is low.

Note:

A hibernation request is held off if a battery check is in progress.

Returns:

None.

16.2.2.3 HibernateCalendarGet

Returns the Hibernation module's date and time in calendar mode.

Prototype:

```
int
HibernateCalendarGet(struct tm *psTime)
```

Parameters:

psTime is the structure that is filled with the current date and time.

Description:

This function returns the current date and time in the structure provided by the *psTime* parameter. Regardless of the calendar mode, the *psTime* parameter uses a 24-hour representation of the time. This function can only be called when the Hibernation module is configured in calendar mode using the [HibernateCounterMode\(\)](#) function with one of the calendar modes.

The only case where this function fails and returns a non-zero value is when the function detects that the counter is passing from the last second of the day to the first second of the next day. This exception must be handled in the application by waiting at least one second before calling again to get the updated calendar information.

Note:

The hibernate calendar mode is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

Returns zero if the time and date were read successfully and returns a non-zero value if the *psTime* structure was not updated.

16.2.2.4 HibernateCalendarMatchGet

Returns the Hibernation module's date and time match value in calendar mode.

Prototype:

```
void  
HibernateCalendarMatchGet (uint32_t ui32Index,  
                           struct tm *psTime)
```

Parameters:

ui32Index indicates which match register to access.

psTime is the structure to fill with the current date and time match value.

Description:

This function returns the current date and time match value in the structure provided by the *psTime* parameter. Regardless of the mode, the *psTime* parameter uses a 24-hour clock representation of time. This function can only be called when the Hibernation module is configured in calendar mode using the [HibernateCounterMode\(\)](#) function. The *ui32Index* value is reserved for future use and should always be zero.

Note:

The hibernate calendar mode is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

Returns zero if the time and date match value were read successfully and returns a non-zero value if the *psTime* structure was not updated.

16.2.2.5 HibernateCalendarMatchSet

Sets the Hibernation module's date and time match value in calendar mode.

Prototype:

```
void  
HibernateCalendarMatchSet (uint32_t ui32Index,  
                           struct tm *psTime)
```

Parameters:

ui32Index indicates which match register to access.

psTime is the structure that holds all of the information to set the current date and time match values.

Description:

This function uses the *psTime* parameter to set the current date and time match value in the Hibernation module's calendar. Regardless of the mode, the *psTime* parameter uses a 24-hour clock representation of time. This function can only be called when the Hibernation module is configured in calendar mode using the [HibernateCounterMode\(\)](#) function. The *ui32Index* value is reserved for future use and should always be zero. Calendar match can be enabled for every day, every hour, every minute or every second, setting any of these fields to 0xFF causes a match for that field. For example, setting the day of month field to 0xFF results in a calendar match daily at the same time.

Note:

The hibernate calendar mode is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

None.

16.2.2.6 HibernateCalendarSet

Sets the Hibernation module's date and time in calendar mode.

Prototype:

```
void
HibernateCalendarSet(struct tm *psTime)
```

Parameters:

psTime is the structure that holds the information for the current date and time.

Description:

This function uses the *psTime* parameter to set the current date and time when the Hibernation module is in calendar mode. Regardless of whether 24-hour or 12-hour mode is in use, the *psTime* structure uses a 24-hour representation of the time. This function can only be called when the hibernate counter is configured in calendar mode using the [HibernateCounterMode\(\)](#) function with one of the calendar modes.

Note:

The hibernate calendar mode is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

None.

16.2.2.7 HibernateClockConfig

Configures the clock input for the Hibernation module.

Prototype:

```
void
HibernateClockConfig(uint32_t ui32Config)
```

Parameters:

ui32Config is one of the possible configuration options for the clock input listed below.

Description:

This function is used to configure the clock input for the Hibernation module. The *ui32Config* parameter can be one of the following values:

- **HIBERNATE_OSC_DISABLE** specifies that the internal oscillator is powered off. This option is used when an externally supplied oscillator is connected to the XOSC0 pin or to save power when the LFIOSC is used in devices that have an LFIOSC in the Hibernation module.
- **HIBERNATE_OSC_HIGHDRIVE** specifies a higher drive strength when a 24-pF filter capacitor is used with a crystal.
- **HIBERNATE_OSC_LOWDRAVE** specifies a lower drive strength when a 12-pF filter capacitor is used with a crystal.

On some devices, there is an option to use an internal low frequency oscillator (LFIOSC) as the clock source for the Hibernation module. Because of the low accuracy of this oscillator, this option should not be used when the system requires a real time counter. Adding the **HIBERNATE_OSC_LFIOSC** value enables the LFIOSC as the clock source to the Hibernation module.

- **HIBERNATE_OSC_LFIOSC** enables the Hibernation module's internal low frequency oscillator as the clock to the Hibernation module.

This *ui32Config* also configures how the clock output from the hibernation is used to clock other peripherals in the system. The ALT clock settings allow clocking a subset of the peripherals. See the hibernate section in the datasheet to determine which peripherals can be clocked by the ALT clock outputs from the Hibernation module. The *ui32Config* parameter can have any combination of the following values:

- **HIBERNATE_OUT_SYSCLK** enables the hibernate clock output to the system clock.

The **HIBERNATE_OSC_DISABLE** option is used to disable and power down the internal oscillator if an external clock source or no clock source is used instead of a 32.768-kHz crystal. In the case where an external crystal is used, either the **HIBERNATE_OSC_HIGHDRIVE** or **HIBERNATE_OSC_LOWDRAVE** is used. These settings optimizes the oscillator drive strength to match the size of the filter capacitor that is used with the external crystal circuit.

Returns:

None.

16.2.2.8 HibernateCounterMode

Configures the Hibernation module's internal counter mode.

Prototype:

```
void  
HibernateCounterMode(uint32_t ui32Config)
```

Parameters:

ui32Config is the configuration to use for the Hibernation module's counter.

Description:

This function configures the Hibernation module's counter mode to operate as a standard RTC counter or to operate in a calendar mode. The *ui32Config* parameter is used to provide the configuration for the counter and must include only one of the following values:

- **HIBERNATE_COUNTER_24HR** specifies 24-hour calendar mode.
- **HIBERNATE_COUNTER_12HR** specifies 12-hour AM/PM calendar mode.
- **HIBERNATE_COUNTER_RTC** specifies RTC counter mode.

The `HibernateCalendar` functions can only be called when either **HIBERNATE_COUNTER_24HR** or **HIBERNATE_COUNTER_12HR** is specified.

Example: Configure hibernate counter to 24-hour calendar mode.

```
//  
// Configure the hibernate module counter to 24-hour calendar mode.  
//  
HibernateCounterMode(HIBERNATE_COUNTER_24HR);
```

Note:

The hibernate calendar mode is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

None.

16.2.2.9 HibernateDataGet

Reads a set of data from the battery-backed memory of the Hibernation module.

Prototype:

```
void  
HibernateDataGet(uint32_t *pui32Data,  
                  uint32_t ui32Count)
```

Parameters:

pui32Data points to a location where the data that is read from the Hibernation module is stored.

ui32Count is the count of 32-bit words to read.

Description:

This function retrieves a set of data from the Hibernation module battery-backed memory that was previously stored with the [HibernateDataSet\(\)](#) function. The caller must ensure that *pui32Data* points to a large enough memory block to hold all the data that is read from the battery-backed memory.

Returns:

None.

16.2.2.10 HibernateDataSet

Stores data in the battery-backed memory of the Hibernation module.

Prototype:

```
void  
HibernateDataSet(uint32_t *pui32Data,  
                  uint32_t ui32Count)
```

Parameters:

pui32Data points to the data that the caller wants to store in the memory of the Hibernation module.

ui32Count is the count of 32-bit words to store.

Description:

Stores a set of data in the Hibernation module battery-backed memory. This memory is preserved when the power to the processor is turned off and can be used to store application state information that is needed when the processor wakes. Up to 16 32-bit words can be stored in the battery-backed memory. The data can be restored by calling the [HibernateDataGet\(\)](#) function.

Returns:

None.

16.2.2.11 HibernateDisable

Disables the Hibernation module for operation.

Prototype:

```
void  
HibernateDisable(void)
```

Description:

This function disables the Hibernation module. After this function is called, none of the Hibernation module features are available.

Returns:

None.

16.2.2.12 HibernateEnableExpClk

Enables the Hibernation module for operation.

Prototype:

```
void  
HibernateEnableExpClk(uint32_t ui32HibClk)
```

Parameters:

ui32HibClk is the rate of the clock supplied to the Hibernation module.

Description:

This function enables the Hibernation module for operation. This function should be called before any of the Hibernation module features are used.

The peripheral clock is the same as the processor clock. This value is returned by [SysCtlClockGet\(\)](#), or it can be explicitly hard-coded if it is constant and known (to save the code/execution overhead of a call to [SysCtlClockGet\(\)](#)).

Returns:

None.

16.2.2.13 HibernateGPIORetentionDisable

Disables GPIO retention after wake from hibernation.

Prototype:

```
void  
HibernateGPIORetentionDisable(void)
```

Description:

This function disables the retention of the GPIO pin state during hibernation and allows the GPIO pins to be controlled by the system. If the [HibernateGPIORetentionEnable\(\)](#) function is called before entering hibernation, this function must be called after returning from hibernation to allow the GPIO pins to be controlled by GPIO module.

Note:

The hibernate GPIO retention setting is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

None.

16.2.2.14 HibernateGPIORetentionEnable

Enables GPIO retention after wake from hibernation.

Prototype:

```
void
HibernateGPIORetentionEnable(void)
```

Description:

This function enables the GPIO pin state to be maintained during hibernation and remain active even when waking from hibernation. The GPIO module itself is reset upon entering hibernation and no longer controls the output pins. To maintain the current output level after waking from hibernation, the GPIO module must be reconfigured and then the [HibernateGPIORetentionDisable\(\)](#) function must be called to return control of the GPIO pin to the GPIO module.

Note:

The hibernation GPIO retention setting is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

None.

16.2.2.15 HibernateGPIORetentionGet

Returns the current setting for GPIO retention.

Prototype:

```
bool
HibernateGPIORetentionGet(void)
```

Description:

This function returns the current setting for GPIO retention in the hibernate module.

Note:

The hibernation GPIO retention setting is not available on all Tiva devices. Please consult the data sheet to determine if the device you are using supports this feature in the Hibernation module.

Returns:

Returns true if GPIO retention is enabled and false if GPIO retention is disabled.

16.2.2.16 HibernateIntClear

Clears pending interrupts from the Hibernation module.

Prototype:

```
void
HibernateIntClear(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupts to be cleared.

Description:

This function clears the specified interrupt sources. This function must be called within the interrupt handler or else the handler is called again upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to the [HibernateIntEnable\(\)](#) function.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

16.2.2.17 HibernateIntDisable

Disables interrupts for the Hibernation module.

Prototype:

```
void  
HibernateIntDisable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupts to be disabled.

Description:

This function disables the specified interrupt sources from the Hibernation module.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to the [HibernateIntEnable\(\)](#) function.

Returns:

None.

16.2.2.18 HibernateIntEnable

Enables interrupts for the Hibernation module.

Prototype:

```
void  
HibernateIntEnable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupts to be enabled.

Description:

This function enables the specified interrupt sources from the Hibernation module.

The *ui32IntFlags* parameter must be the logical OR of any combination of the following:

- **HIBERNATE_INT_WR_COMPLETE** - write complete interrupt
- **HIBERNATE_INT_PIN_WAKE** - wake from pin interrupt
- **HIBERNATE_INT_LOW_BAT** - low-battery interrupt
- **HIBERNATE_INT_RTC_MATCH_0** - RTC match 0 interrupt
- **HIBERNATE_INT_VDDFAIL** - supply failure interrupt.
- **HIBERNATE_INT_RESET_WAKE** - wake from reset pin interrupt
- **HIBERNATE_INT_GPIO_WAKE** - wake from GPIO pin or reset pin interrupt.

Note:

The **HIBERNATE_INT_RESET_WAKE**, **HIBERNATE_INT_GPIO_WAKE**, and **HIBERNATE_INT_VDDFAIL** settings are not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if these interrupt sources are available.

Returns:

None.

16.2.2.19 HibernateIntRegister

Registers an interrupt handler for the Hibernation module interrupt.

Prototype:

```
void
HibernateIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler points to the function to be called when a hibernation interrupt occurs.

Description:

This function registers the interrupt handler in the system interrupt controller. The interrupt is enabled at the global level, but individual interrupt sources must still be enabled with a call to [HibernateIntEnable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

16.2.2.20 HibernateIntStatus

Gets the current interrupt status of the Hibernation module.

Prototype:

```
uint32_t
HibernateIntStatus(bool bMasked)
```

Parameters:

bMasked is false to retrieve the raw interrupt status, and true to retrieve the masked interrupt status.

Description:

This function returns the interrupt status of the Hibernation module. The caller can use this function to determine the cause of a hibernation interrupt. Either the masked or raw interrupt status can be returned.

Note:

A wake from reset pin also signals a wake from GPIO pin with the value returned being HIBERNATE_INT_GPIO_WAKE | HIBERNATE_INT_RESET_WAKE. Hence a wake from reset pin should take priority over wake from GPIO pin.

Returns:

Returns the interrupt status as a bit field with the values as described in the [HibernateIntEnable\(\)](#) function.

16.2.2.21 HibernateIntUnregister

Unregisters an interrupt handler for the Hibernation module interrupt.

Prototype:

```
void  
HibernateIntUnregister(void)
```

Description:

This function unregisters the interrupt handler in the system interrupt controller. The interrupt is disabled at the global level, and the interrupt handler is no longer called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

16.2.2.22 HibernateIsActive

Checks to see if the Hibernation module is already powered up.

Prototype:

```
uint32_t  
HibernateIsActive(void)
```

Description:

This function queries the control register to determine if the module is already active. This function can be called at a power-on reset to help determine if the reset is due to a wake from hibernation or a cold start. If the Hibernation module is already active, then it does not need to be re-enabled, and its status can be queried immediately.

The software application should also use the [HibernateIntStatus\(\)](#) function to read the raw interrupt status to determine the cause of the wake. The [HibernateDataGet\(\)](#) function can be used to restore state. These combinations of functions can be used by the software to determine if the processor is waking from hibernation and the appropriate action to take as a result.

Returns:

Returns **true** if the module is already active, and **false** if not.

16.2.2.23 HibernateLowBatGet

Gets the currently configured low-battery detection behavior.

Prototype:

```
uint32_t
HibernateLowBatGet (void)
```

Description:

This function returns a value representing the currently configured low battery detection behavior.

The return value is a combination of the values described in the [HibernateLowBatSet\(\)](#) function.

Returns:

Returns a value indicating the configured low-battery detection.

16.2.2.24 HibernateLowBatSet

Configures the low-battery detection.

Prototype:

```
void
HibernateLowBatSet (uint32_t ui32LowBatFlags)
```

Parameters:

ui32LowBatFlags specifies behavior of low-battery detection.

Description:

This function enables the low-battery detection and whether hibernation is allowed if a low battery is detected. If low-battery detection is enabled, then a low-battery condition is indicated in the raw interrupt status register, which can be enabled to trigger an interrupt. Optionally, hibernation can be aborted if a low battery condition is detected.

The *ui32LowBatFlags* parameter is one of the following values:

- **HIBERNATE_LOW_BAT_DETECT** - detect a low-battery condition
- **HIBERNATE_LOW_BAT_ABORT** - detect a low-battery condition and abort hibernation if low-battery is detected

The other setting in the *ui32LowBatFlags* allows the caller to set one of the following voltage level trigger values :

- **HIBERNATE_LOW_BAT_1_9V** - voltage low level is 1.9 V
- **HIBERNATE_LOW_BAT_2_1V** - voltage low level is 2.1 V
- **HIBERNATE_LOW_BAT_2_3V** - voltage low level is 2.3 V
- **HIBERNATE_LOW_BAT_2_5V** - voltage low level is 2.5 V

Example: Abort hibernate if the voltage level is below 2.1 V.

```
HibernateLowBatSet(HIBERNATE_LOW_BAT_ABORT | HIBERNATE_LOW_BAT_2_1V);
```

Returns:

None.

16.2.2.25 HibernateRequest

Requests hibernation mode.

Prototype:

```
void  
HibernateRequest(void)
```

Description:

This function requests the Hibernation module to disable the external regulator, thus removing power from the processor and all peripherals. The Hibernation module remains powered from the battery or auxiliary power supply.

The Hibernation module re-enables the external regulator when one of the configured wake conditions occurs (such as RTC match or external **WAKE** pin). When the power is restored, the processor goes through a power-on reset although the Hibernation module is not reset. The processor can retrieve saved state information with the [HibernateDataGet\(\)](#) function. Prior to calling the function to request hibernation mode, the conditions for waking must have already been set by using the [HibernateWakeSet\(\)](#) function.

Note that this function may return because some time may elapse before the power is actually removed, or it may not be removed at all. For this reason, the processor continues to execute instructions for some time, and the caller should be prepared for this function to return. There are various reasons why the power may not be removed. For example, if the [HibernateLowBat-Set\(\)](#) function was used to configure an abort if low battery is detected, then the power is not removed if the battery voltage is too low. There may be other reasons related to the external circuit design, that a request for hibernation may not actually occur.

For all these reasons, the caller must be prepared for this function to return. The simplest way to handle it is to just enter an infinite loop and wait for the power to be removed.

Returns:

None.

16.2.2.26 HibernateRTCDisable

Disables the RTC feature of the Hibernation module.

Prototype:

```
void  
HibernateRTCDisable(void)
```

Description:

This function disables the RTC in the Hibernation module. After calling this function, the RTC features of the Hibernation module are not available.

Returns:

None.

16.2.2.27 HibernateRTCEnable

Enables the RTC feature of the Hibernation module.

Prototype:

```
void  
HibernateRTCEnable(void)
```

Description:

This function enables the RTC in the Hibernation module. The RTC can be used to wake the processor from hibernation at a certain time, or to generate interrupts at certain times. This function must be called before using any of the RTC features of the Hibernation module.

Returns:

None.

16.2.2.28 HibernateRTCGet

Gets the value of the real time clock (RTC) counter.

Prototype:

```
uint32_t  
HibernateRTCGet(void)
```

Description:

This function gets the value of the RTC and returns it to the caller.

Returns:

Returns the value of the RTC counter in seconds.

16.2.2.29 HibernateRTCMatchGet

Gets the value of the requested RTC match register.

Prototype:

```
uint32_t  
HibernateRTCMatchGet(uint32_t ui32Match)
```

Parameters:

ui32Match is the index of the match register.

Description:

This function gets the value of the match register for the RTC. The only value that can be used with the *ui32Match* parameter is zero, other values are reserved for future use.

Returns:

Returns the value of the requested match register.

16.2.2.30 HibernateRTCMatchSet

Sets the value of the RTC match register.

Prototype:

```
void  
HibernateRTCMatchSet(uint32_t ui32Match,  
                      uint32_t ui32Value)
```

Parameters:

ui32Match is the index of the match register.

ui32Value is the value for the match register.

Description:

This function sets a match register for the RTC. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match register.

Returns:

None.

16.2.2.31 HibernateRTCSet

Sets the value of the real time clock (RTC) counter.

Prototype:

```
void  
HibernateRTCSet(uint32_t ui32RTCValue)
```

Parameters:

ui32RTCValue is the new value for the RTC.

Description:

This function sets the value of the RTC. The RTC counter contains the count in seconds when a 32.768kHz clock source is in use. The RTC must be enabled by calling [HibernateRTCEnable\(\)](#) before calling this function.

Returns:

None.

16.2.2.32 HibernateRTCSSGet

Returns the current value of the RTC sub second count.

Prototype:

```
uint32_t  
HibernateRTCSSGet(void)
```

Description:

This function returns the current value of the sub second count for the RTC in 1/32768 of a second increments. The only value that can be used with the **ui32Match** parameter is zero, other values are reserved for future use.

Returns:

The current RTC sub second count in 1/32768 seconds.

16.2.2.33 HibernateRTCSSMatchGet

Returns the value of the requested RTC sub second match register.

Prototype:

```
uint32_t
HibernateRTCSSMatchGet (uint32_t ui32Match)
```

Parameters:

ui32Match is the index of the match register.

Description:

This function returns the current value of the sub second match register for the RTC. The value returned is in 1/32768 second increments. The only value that can be used with the *ui32Match* parameter is zero, other values are reserved for future use.

Returns:

Returns the value of the requested sub section match register.

16.2.2.34 HibernateRTCSSMatchSet

Sets the value of the RTC sub second match register.

Prototype:

```
void
HibernateRTCSSMatchSet (uint32_t ui32Match,
                        uint32_t ui32Value)
```

Parameters:

ui32Match is the index of the match register.

ui32Value is the value for the sub second match register.

Description:

This function sets the sub second match register for the RTC in 1/32768 of a second increments. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match combined with the sub second match register. The only value that can be used with the *ui32Match* parameter is zero, other values are reserved for future use.

Returns:

None.

16.2.2.35 HibernateRTCTrimGet

Gets the value of the RTC pre-divider trim register.

Prototype:

```
uint32_t  
HibernateRTCTrimGet (void)
```

Description:

This function gets the value of the pre-divider trim register. This function can be used to get the current value of the trim register prior to making an adjustment by using the [HibernateRTCTrimSet\(\)](#) function.

Returns:

None.

16.2.2.36 HibernateRTCTrimSet

Sets the value of the RTC pre-divider trim register.

Prototype:

```
void  
HibernateRTCTrimSet (uint32_t ui32Trim)
```

Parameters:

ui32Trim is the new value for the pre-divider trim register.

Description:

This function sets the value of the pre-divider trim register. The input time source is divided by the pre-divider to achieve a one-second clock rate. Once every 64 seconds, the value of the pre-divider trim register is applied to the pre-divider to allow fine-tuning of the RTC rate, in order to make corrections to the rate. The software application can make adjustments to the pre-divider trim register to account for variations in the accuracy of the input time source. The nominal value is 0x7FFF, and it can be adjusted up or down in order to fine-tune the RTC rate.

Returns:

None.

16.2.2.37 HibernateTamperDisable

Disables the tamper feature.

Prototype:

```
void  
HibernateTamperDisable (void)
```

Description:

This function is used to disable the tamper feature functionality. All other configuration settings are left unmodified, allowing a call to [HibernateTamperEnable\(\)](#) to quickly enable the tamper feature with its previous configuration.

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.38 HibernateTamperEnable

Enables the tamper feature.

Prototype:

```
void
HibernateTamperEnable(void)
```

Description:

This function is used to enable the tamper feature functionality. This function should only be called after the global configuration is set with a call to [HibernateTamperEventsConfig\(\)](#) and the tamper inputs have been configured with a call to [HibernateTamperIOEnable\(\)](#).

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.39 HibernateTamperEventsClear

Clears the tamper feature events.

Prototype:

```
void
HibernateTamperEventsClear(void)
```

Description:

This function is used to clear all tamper events. This function always clears the tamper feature event state indicator along with all tamper log entries. Logged event data should be retrieved with [HibernateTamperEventsGet\(\)](#) prior to requesting a event clear.

[HibernateTamperEventsClear\(\)](#) should be called prior to clearing the system control NMI that resulted from the tamper event.

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.40 HibernateTamperEventsClearNoLock

Clears the tamper feature events without Unlock and Lock.

Prototype:

```
void
HibernateTamperEventsClearNoLock(void)
```

Description:

This function is used to clear all tamper events without unlock/locking the tamper control registers, so API [HibernateTamperUnLock\(\)](#) should be called before this function, and API [HibernateTamperLock\(\)](#) should be called after to ensure that tamper control registers are locked.

This function doesn't block until the write is complete. Therefore, care must be taken to ensure the next immediate write will occur only after the write complete bit is set.

This function is used to implement a software workaround in NMI interrupt handler to fix an issue when a new tamper event could be missed during the clear of current tamper event.

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.41 HibernateTamperEventsConfig

Configures the tamper feature event response.

Prototype:

```
void  
HibernateTamperEventsConfig(uint32_t ui32Config)
```

Parameters:

ui32Config specifies the configuration options for tamper events.

Description:

This function is used to configure the event response options for the tamper feature. The *ui32Config* parameter provides a combination of the **HIBERNATE_TAMPER_EVENTS_*** features to set these options. The application should choose from the following set of defines to determine what happens to the system when a tamper event occurs:

- **HIBERNATE_TAMPER_EVENTS_ERASE_ALL_HIB_MEM** all of the Hibernation module's battery-backed RAM is cleared due to a tamper event
- **HIBERNATE_TAMPER_EVENTS_ERASE_HIGH_HIB_MEM** the upper half of the Hibernation module's battery-backed RAM is cleared due to a tamper event
- **HIBERNATE_TAMPER_EVENTS_ERASE_LOW_HIB_MEM** the lower half of the Hibernation module's battery-backed RAM is cleared due to a tamper event
- **HIBERNATE_TAMPER_EVENTS_ERASE_NO_HIB_MEM** the Hibernation module's battery-backed RAM is not changed due to a tamper event
- **HIBERNATE_TAMPER_EVENTS_HIB_WAKE** a tamper event wakes the MCU from hibernation
- **HIBERNATE_TAMPER_EVENTS_NO_HIB_WAKE** a tamper event does not wake the MCU from hibernation

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.42 HibernateTamperEventsGet

Returns a tamper log entry.

Prototype:

```
bool
HibernateTamperEventsGet(uint32_t ui32Index,
                           uint32_t *pui32RTC,
                           uint32_t *pui32Event)
```

Parameters:

ui32Index is the index of the log entry to return.

pui32RTC is a pointer to the memory to store the logged RTC data.

pui32Event is a pointer to the memory to store the logged tamper event.

Description:

This function is used to return a tamper log entry from the hibernate feature. The *ui32Index* specifies the zero-based index of the log entry to query and has a valid range of 0-3.

When this function returns, the *pui32RTC* value contains the time value and *pui32Event* parameter contains the tamper I/O event that triggered this log.

The format of the returned *pui32RTC* data is dependent on the configuration of the RTC within the Hibernation module. If the RTC is configured for counter mode, the returned data contains counted seconds from the RTC enable. If the RTC is configured for calendar mode, the data returned is formatted as follows:

31:26	25:22	21:17	16:12	11:6	5:0
year	month	day of month	hours	minutes	seconds

The data returned in the *pui32Events* parameter could include any of the following flags:

- **HIBERNATE_TAMPER_EVENT_0** indicates a tamper event was triggered on I/O signal 0
- **HIBERNATE_TAMPER_EVENT_1** indicates a tamper event was triggered on I/O signal 1
- **HIBERNATE_TAMPER_EVENT_2** indicates a tamper event was triggered on I/O signal 2
- **HIBERNATE_TAMPER_EVENT_3** indicates a tamper event was triggered on I/O signal 3
- **HIBERNATE_TAMPER_EVENT_XOSC** indicates an external oscillator failure triggered the tamper event

Note:

Tamper event logs are not consumed when read and remain available until cleared. Events are only logged if unused log space is available.

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

Returns **true** if the *pui32RTC* and *pui32Events* were updated successfully and returns **false** if the values were not updated.

16.2.2.43 HibernateTamperExtOscRecover

Attempts to recover the external oscillator.

Prototype:

```
void  
HibernateTamperExtOscRecover(void)
```

Description:

This function is used to attempt to recover the external oscillator after a **HIBERNATE_TAMPER_STATUS_EXT_OSC_FAILED** status is reported. This function must not be called if the external oscillator is not used as the hibernation clock input. [HibernateTamperExtOscValid\(\)](#) should be called before calling this function.

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.44 HibernateTamperExtOscValid

Reports if the external oscillator signal is active and stable.

Prototype:

```
bool  
HibernateTamperExtOscValid(void)
```

Description:

This function should be used to verify the external oscillator is active and valid before attempting to recover from a **HIBERNATE_TAMPER_STATUS_EXT_OSC_FAILED** status by calling [HibernateTamperExtOscRecover\(\)](#).

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

Returns **true** if the external oscillator is both active and stable, otherwise a **false** indicator is returned.

16.2.2.45 HibernateTamperIODisable

Disables an input to the tamper feature.

Prototype:

```
void  
HibernateTamperIODisable(uint32_t ui32Input)
```

Parameters:

ui32Input is the tamper input to disable.

Description:

This function is used to disable an input to the tamper feature. The *ui32Input* parameter specifies the tamper signal to disable and has a valid range of 0-3.

Note:

None of the GPIO API functions are needed to configure the tamper pins. The tamper pins configured by using this function overrides any configuration by GPIO APIs.

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.46 HibernateTamperIOEnable

Configures an input to the tamper feature.

Prototype:

```
void
HibernateTamperIOEnable(uint32_t ui32Input,
                         uint32_t ui32Config)
```

Parameters:

ui32Input is the tamper input to configure.

ui32Config holds the configuration options for a given input to the tamper feature.

Description:

This function is used to configure an input to the tamper feature. The *ui32Input* parameter specifies the tamper signal to configure and has a valid range of 0-3. The *ui32Config* parameter provides the set of tamper features in the **HIBERNATE_TAMPER_IO_*** values. The values that are valid in the *ui32Config* parameter are:

- **HIBERNATE_TAMPER_IO_MATCH_SHORT** configures the trigger to match after 2 hibernation clocks
- **HIBERNATE_TAMPER_IO_MATCH_LONG** configures the trigger to match after 3071 hibernation clocks
- **HIBERNATE_TAMPER_IO_WPU_ENABLED** turns on an internal weak pull up
- **HIBERNATE_TAMPER_IO_WPU_DISABLED** turns off an internal weak pull up
- **HIBERNATE_TAMPER_IO_TRIGGER_HIGH** sets the tamper event to active high
- **HIBERNATE_TAMPER_IO_TRIGGER_LOW** sets the tamper event to active low

Note:

None of the GPIO API functions are needed to configure the tamper pins. The tamper pins configured by using this function overrides any configuration by GPIO APIs.

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.47 HibernateTamperLock

Lock temper registers.

Prototype:

```
void  
HibernateTamperLock(void)
```

Description:

This function is used to lock the temper control registers. This function should be used after calling API [HibernateTamperEventsClearNoLock\(\)](#).

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.48 HibernateTamperStatusGet

Returns the current tamper feature status.

Prototype:

```
uint32_t  
HibernateTamperStatusGet(void)
```

Description:

This function is used to return the tamper feature status. This function returns one of the values from this group of options:

- **HIBERNATE_TAMPER_STATUS_INACTIVE** indicates tamper detection is disabled
- **HIBERNATE_TAMPER_STATUS_ACTIVE** indicates tamper detection is enabled and ready
- **HIBERNATE_TAMPER_STATUS_EVENT** indicates tamper event was detected

In addition, one of the values is included from this group:

- **HIBERNATE_TAMPER_STATUS_EXT_OSC_INACTIVE** indicates the external oscillator is not active
- **HIBERNATE_TAMPER_STATUS_EXT_OSC_ACTIVE** indicates the external oscillator is active

And one of the values is included from this group:

- **HIBERNATE_TAMPER_STATUS_EXT_OSC_FAILED** indicates the external oscillator signal has transitioned from valid to invalid
- **HIBERNATE_TAMPER_STATUS_EXT_OSC_VALID** indicates the external oscillator is providing a valid signal

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

Returns a combination of the **HIBERNATE_TAMPER_STATUS_*** values.

16.2.2.49 HibernateTamperUnLock

Unlock temper registers.

Prototype:

```
void
HibernateTamperUnLock (void)
```

Description:

This function is used to unlock the temper control registers. This function should be only used before calling API [HibernateTamperEventsClearNoLock\(\)](#).

Note:

The hibernate tamper feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

16.2.2.50 HibernateWakeGet

Gets the currently configured wake conditions for the Hibernation module.

Prototype:

```
uint32_t
HibernateWakeGet (void)
```

Description:

This function returns the flags representing the wake configuration for the Hibernation module. The return value is a combination of the following flags:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted
- **HIBERNATE_WAKE_RTC** - wake when the RTC matches occurs
- **HIBERNATE_WAKE_LOW_BAT** - wake from hibernation due to a low-battery level being detected
- **HIBERNATE_WAKE_GPIO** - wake when a GPIO pin is asserted
- **HIBERNATE_WAKE_RESET** - wake when a reset pin is asserted

Note:

The **HIBERNATE_WAKE_LOW_BAT**, **HIBERNATE_WAKE_GPIO**, and **HIBERNATE_WAKE_RESET** parameters are only available on some Tiva devices.

On some Tiva devices a tamper event acts as a wake source for the Hibernation module. Refer the function [HibernateTamperEventsConfig\(\)](#) to wake from hibernation on a tamper event.

Returns:

Returns flags indicating the configured wake conditions.

16.2.2.51 HibernateWakeSet

Configures the wake conditions for the Hibernation module.

Prototype:

```
void  
HibernateWakeSet(uint32_t ui32WakeFlags)
```

Parameters:

ui32WakeFlags specifies which conditions should be used for waking.

Description:

This function enables the conditions under which the Hibernation module wakes. The *ui32WakeFlags* parameter is the logical OR of any combination of the following:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted.
- **HIBERNATE_WAKE_RTC** - wake when the RTC match occurs.
- **HIBERNATE_WAKE_LOW_BAT** - wake from hibernate due to a low-battery level being detected.
- **HIBERNATE_WAKE_GPIO** - wake when a GPIO pin is asserted.
- **HIBERNATE_WAKE_RESET** - wake when a reset pin is asserted.

If the **HIBERNATE_WAKE_GPIO** flag is set, then one of the GPIO configuration functions [GPIOPinTypeWakeHigh\(\)](#) or [GPIOPinTypeWakeLow\(\)](#) must be called to properly configure and enable a GPIO as a wake source for hibernation.

Note:

The **HIBERNATE_WAKE_GPIO** and **HIBERNATE_WAKE_RESET** parameters are only available on some Tiva devices.

On some Tiva devices a tamper event acts as a wake source for the Hibernation module. Refer the function [HibernateTamperEventsConfig\(\)](#) to wake from hibernation on a tamper event.

Returns:

None.

16.3 Programming Example

The following example shows how to determine if the processor reset is due to a wake from hibernation and to restore saved state:

```
uint32_t ui32Status;  
uint32_t pui32NVData[64];  
  
//  
// Need to enable the hibernation peripheral after wake/reset, before using  
// it.  
//  
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);  
  
//  
// Determine if the Hibernation module is active.  
//  
if(HibernateIsActive())
```

```

{
    //
    // Read the status to determine cause of wake.
    //
    ui32Status = HibernateIntStatus(false);

    //
    // Test the status bits to see the cause.
    //
    if(ui32Status & HIBERNATE_INT_PIN_WAKE)
    {
        //
        // Wake up was due to WAKE pin assertion.
        //
    }
    if(ui32Status & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // Wake up was due to RTC match register.
        //
    }

    //
    // Restore program state information that was saved prior to
    // hibernation.
    //
    HibernateDataGet(pui32NVData, 64);

    //
    // Now that wake up cause has been determined and state has been
    // restored, the program can proceed with normal processor and
    // peripheral initialization.
    //
}

//
// Hibernation module was not active, so this is a cold power-up/reset.
//
else
{
    //
    // Perform normal power-on initialization.
    //
}

```

The following example shows how to set up the Hibernation module and initiate hibernation with wake up at a future time:

```

uint32_t ui32Status;
uint32_t pui32NVData[64];

//
// Enable the hibernation peripheral before using it.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);

//
// Enable clocking to the Hibernation module.
//
HibernateEnableExpClk(SysCtlClockGet());

//
// User-implemented delay here to allow crystal to power up and stabilize.
//

```

```
//  
// Configure the clock source for Hibernation module and enable the RTC  
// feature.  
//  
HibernateClockConfig(HIBERNATE_OSC_LOWDRAVE);  
HibernateRTCEnable();  
  
//  
// Set the RTC to 0 or an initial value. The RTC can be set once when the  
// system is initialized after the cold startup and then left to run. Or  
// it can be initialized before every hibernate.  
//  
HibernateRTCSet(0);  
  
//  
// Set the match 0 register for 30 seconds from now.  
//  
HibernateRTCMatchSet(0, HibernateRTCGet() + 30);  
  
//  
// Clear any pending status.  
//  
ui32Status = HibernateIntStatus(0);  
HibernateIntClear(ui32Status);  
  
//  
// Save the program state information. The state information is stored in  
// the pui32NVData[] array. It is not necessary to save the full 16 words  
// of data, only as much as is actually needed by the program.  
//  
HibernateDataSet(pui32NVData, 16);  
  
//  
// Configure to wake on RTC match.  
//  
HibernateWakeSet(HIBERNATE_WAKE_RTC);  
  
//  
// Request hibernation. The following call may return because it takes a  
// finite amount of time for power to be removed.  
//  
HibernateRequest();  
  
//  
// Need a loop here to wait for the power to be removed. Power is  
// removed while executing in this loop.  
//  
for(;;)  
{  
}
```

The following example shows how to use the Hibernation module RTC to generate an interrupt at a certain time:

```
//  
// Handler for hibernate interrupts.  
//  
void  
HibernateHandler(void)  
{  
    uint32_t ui32Status;  
  
    //  
    // Get the interrupt status and clear any pending interrupts.  
    //
```

```

ui32Status = HibernateIntStatus(1);
HibernateIntClear(ui32Status);

//
// Process the RTC match 0 interrupt.
//
if(ui32Status & HIBERNATE_INT_RTC_MATCH_0)
{
    //
    // RTC match 0 interrupt actions go here.
    //
}

//
// Main function.
//
int
main(void)
{
    //
    // System initialization code ...
    //

    //
    // Enable the Hibernation module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
    HibernateEnableExpClk(SysCtlClockGet());

    //
    // Wait an amount of time for the module to power up.
    //

    //
    // Configure the clock source for Hibernation module and enable the
    // RTC feature.
    //
    HibernateClockConfig(HIBERNATE_OSC_LOWDRAVE);
    HibernateRTCEnable();

    //
    // Set the RTC to an initial value.
    //
    HibernateRTCSet(0);

    //
    // Set Match 0 for 30 seconds from now.
    //
    HibernateRTCMatchSet(0, HibernateRTCGet() + 30);

    //
    // Set up interrupts on the Hibernation module to enable the RTC match
    // 0 interrupt. Clear all pending interrupts and register the
    // interrupt handler.
    //
    HibernateIntEnable(HIBERNATE_INT_RTC_MATCH_0);
    HibernateIntClear(HIBERNATE_INT_PIN_WAKE | HIBERNATE_INT_LOW_BAT |
                      HIBERNATE_INT_RTC_MATCH_0);
    HibernateIntRegister(HibernateHandler);

    //
    // Hibernate handler (above) is invoked in 30 seconds.
    //

    // ...
}

```


17 Inter-Integrated Circuit (I2C)

Introduction	321
API Functions	322
Programming Example	349

17.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the Tiva I2C master and slave 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 master and slave modules 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 Tiva 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 Tiva I2C modules can operate at the following speeds: Standard (100 kbps), Fast (400 kbps), Fast plus (1 Mbps) and High Speed (3.33 Mbps).

Both the master and slave I2C modules can generate interrupts. The I2C master module generates interrupts when a transmit or receive operation is completed (or aborted due to an error); and on some devices when a clock low timeout has occurred. The I2C slave module generates interrupts when data has been sent or requested by a master; and on some devices, when a START or STOP condition is present.

17.1.1 Master Operations

When using this API to drive the I2C master module, the user must first initialize the I2C master module with a call to [I2CMasterInitExpClk\(\)](#). That function sets the bus speed and enables the master module.

The user may transmit or receive data after the successful initialization of the I2C master module. Data is transferred by first setting the slave address using [I2CMasterSlaveAddrSet\(\)](#). That function is also used to define whether the transfer is a send (a write to the slave from the master) or a receive (a read from the slave by the master). Then, if connected to an I2C bus that has multiple masters, the Tiva I2C master must first call [I2CMasterBusBusy\(\)](#) before attempting to initiate the desired transaction. After determining that the bus is not busy, if trying to send data, the user must call the [I2CMasterDataPut\(\)](#) function. The transaction can then be initiated on the bus by calling the [I2CMasterControl\(\)](#) function with any of the following commands:

- [I2C_MASTER_CMD_SINGLE_SEND](#)
- [I2C_MASTER_CMD_SINGLE_RECEIVE](#)
- [I2C_MASTER_CMD_BURST_SEND_START](#)
- [I2C_MASTER_CMD_BURST_RECEIVE_START](#)

Any of those commands results in the master arbitrating for the bus, driving the start sequence onto the bus, and sending the slave address and direction bit across the bus. The remainder of the transaction can then be driven using either a polling or interrupt-driven method.

For the single send and receive cases, the polling method involves looping on the return from `I2CMasterBusy()`. Once that function indicates that the I2C master is no longer busy, the bus transaction has been completed and can be checked for errors using `I2CMasterErr()`. If there are no errors, then the data has been sent or is ready to be read using `I2CMasterDataGet()`. For the burst send and receive cases, the polling method also involves calling the `I2CMasterControl()` function for each byte transmitted or received (using either the `I2C_MASTER_CMD_BURST_SEND_CONT` or `I2C_MASTER_CMD_BURST_RECEIVE_CONT` commands), and for the last byte sent or received (using either the `I2C_MASTER_CMD_BURST_SEND_FINISH` or `I2C_MASTER_CMD_BURST_RECEIVE_FINISH` commands). If any error is detected during the burst transfer, the `I2CMasterControl()` function should be called using the appropriate stop command (`I2C_MASTER_CMD_BURST_SEND_ERROR_STOP` or `I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP`).

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C master interrupt; the interrupt occurs when the master is no longer busy.

17.1.2 Slave Operations

When using this API to drive the I2C slave module, the user must first initialize the I2C slave module with a call to `I2CSlaveInit()`. This function enables the I2C slave module and initializes the slave's own address. After the initialization is complete, the user may poll the slave status using `I2CSlaveStatus()` to determine if a master requested a send or receive operation. Depending on the type of operation requested, the user can call `I2CSlaveDataPut()` or `I2CSlaveDataGet()` to complete the transaction. Alternatively, the I2C slave can handle transactions using an interrupt handler registered with `I2CIntRegister()`, and by enabling the I2C slave interrupt.

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

17.2 API Functions

Functions

- `uint32_t I2CFIFODataGet (uint32_t ui32Base)`
- `uint32_t I2CFIFODataGetNonBlocking (uint32_t ui32Base, uint8_t *pui8Data)`
- `void I2CFIFODataPut (uint32_t ui32Base, uint8_t ui8Data)`
- `uint32_t I2CFIFODataPutNonBlocking (uint32_t ui32Base, uint8_t ui8Data)`
- `uint32_t I2CFIFOStatus (uint32_t ui32Base)`
- `void I2CIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))`
- `void I2CIntUnregister (uint32_t ui32Base)`
- `uint32_t I2CMasterBurstCountGet (uint32_t ui32Base)`
- `void I2CMasterBurstLengthSet (uint32_t ui32Base, uint8_t ui8Length)`
- `bool I2CMasterBusBusy (uint32_t ui32Base)`
- `bool I2CMasterBusy (uint32_t ui32Base)`
- `void I2CMasterControl (uint32_t ui32Base, uint32_t ui32Cmd)`
- `uint32_t I2CMasterDataGet (uint32_t ui32Base)`
- `void I2CMasterDataPut (uint32_t ui32Base, uint8_t ui8Data)`

- void `I2CMasterDisable` (uint32_t ui32Base)
- void `I2CMasterEnable` (uint32_t ui32Base)
- uint32_t `I2CMasterErr` (uint32_t ui32Base)
- void `I2CMasterGlitchFilterConfigSet` (uint32_t ui32Base, uint32_t ui32Config)
- void `I2CMasterInitExpClk` (uint32_t ui32Base, uint32_t ui32I2CClk, bool bFast)
- void `I2CMasterIntClear` (uint32_t ui32Base)
- void `I2CMasterIntClearEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `I2CMasterIntDisable` (uint32_t ui32Base)
- void `I2CMasterIntDisableEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `I2CMasterIntEnable` (uint32_t ui32Base)
- void `I2CMasterIntEnableEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- bool `I2CMasterIntStatus` (uint32_t ui32Base, bool bMasked)
- uint32_t `I2CMasterIntStatusEx` (uint32_t ui32Base, bool bMasked)
- uint32_t `I2CMasterLineStateGet` (uint32_t ui32Base)
- void `I2CMasterSlaveAddrSet` (uint32_t ui32Base, uint8_t ui8SlaveAddr, bool bReceive)
- void `I2CMasterTimeoutSet` (uint32_t ui32Base, uint32_t ui32Value)
- void `I2CRxFIFOConfigSet` (uint32_t ui32Base, uint32_t ui32Config)
- void `I2CRxFIFOFlush` (uint32_t ui32Base)
- void `I2CSlaveACKOverride` (uint32_t ui32Base, bool bEnable)
- void `I2CSlaveACKValueSet` (uint32_t ui32Base, bool bACK)
- void `I2CSlaveAddressSet` (uint32_t ui32Base, uint8_t ui8AddrNum, uint8_t ui8SlaveAddr)
- uint32_t `I2CSlaveDataGet` (uint32_t ui32Base)
- void `I2CSlaveDataPut` (uint32_t ui32Base, uint8_t ui8Data)
- void `I2CSlaveDisable` (uint32_t ui32Base)
- void `I2CSlaveEnable` (uint32_t ui32Base)
- void `I2CSlaveFIFODisable` (uint32_t ui32Base)
- void `I2CSlaveFIFOEnable` (uint32_t ui32Base, uint32_t ui32Config)
- void `I2CSlaveInit` (uint32_t ui32Base, uint8_t ui8SlaveAddr)
- void `I2CSlaveIntClear` (uint32_t ui32Base)
- void `I2CSlaveIntClearEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `I2CSlaveIntDisable` (uint32_t ui32Base)
- void `I2CSlaveIntDisableEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `I2CSlaveIntEnable` (uint32_t ui32Base)
- void `I2CSlaveIntEnableEx` (uint32_t ui32Base, uint32_t ui32IntFlags)
- bool `I2CSlaveIntStatus` (uint32_t ui32Base, bool bMasked)
- uint32_t `I2CSlaveIntStatusEx` (uint32_t ui32Base, bool bMasked)
- uint32_t `I2CSlaveStatus` (uint32_t ui32Base)
- void `I2CTxFIFOConfigSet` (uint32_t ui32Base, uint32_t ui32Config)
- void `I2CTxFIFOFlush` (uint32_t ui32Base)

17.2.1 Detailed Description

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 the [I2CIntRegister\(\)](#), [I2CIntUnregister\(\)](#), [I2CMasterIntEnable\(\)](#), [I2CMasterIntDisable\(\)](#), [I2CMasterIntClear\(\)](#), [I2CMasterIntStatus\(\)](#), [I2CSlaveIntEnable\(\)](#), [I2CSlaveIntDisable\(\)](#), [I2CSlaveIntClear\(\)](#), [I2CSlaveIntStatus\(\)](#), [I2CSlaveIntEnableEx\(\)](#), [I2CSlaveIntDisableEx\(\)](#), [I2CSlaveIntClearEx\(\)](#), and [I2CSlaveIntStatusEx\(\)](#) functions.

Status and initialization functions for the I2C modules are [I2CMasterInitExpClk\(\)](#), [I2CMasterEnable\(\)](#), [I2CMasterDisable\(\)](#), [I2CMasterBusBusy\(\)](#), [I2CMasterBusy\(\)](#), [I2CMasterErr\(\)](#), [I2CSlaveInit\(\)](#), [I2CSlaveEnable\(\)](#), [I2CSlaveDisable\(\)](#), and [I2CSlaveStatus\(\)](#).

Sending and receiving data from the I2C modules are handled by the [I2CMasterSlaveAddrSet\(\)](#), [I2CMasterControl\(\)](#), [I2CMasterDataGet\(\)](#), [I2CMasterDataPut\(\)](#), [I2CSlaveDataGet\(\)](#), and [I2CSlaveDataPut\(\)](#) functions.

The [I2CMasterInit\(\)](#) API from previous versions of the peripheral driver library has been replaced by the [I2CMasterInitExpClk\(\)](#) API. A macro has been provided in `i2c.h` to map the old API to the new API, allowing existing applications to link and run with the new API. It is recommended that new applications utilize the new API in favor of the old one.

17.2.2 Function Documentation

17.2.2.1 I2CFIFODataGet

Reads a byte from the I2C receive FIFO.

Prototype:

```
uint32_t  
I2CFIFODataGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function reads a byte of data from I2C receive FIFO and places it in the location specified by the *pui8Data* parameter. If there is no data available, this function waits until data is received before returning.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

The data byte.

17.2.2.2 I2CFIFODataGetNonBlocking

Reads a byte from the I2C receive FIFO.

Prototype:

```
uint32_t  
I2CFIFODataGetNonBlocking(uint32_t ui32Base,  
                           uint8_t *pui8Data)
```

Parameters:

ui32Base is the base address of the I2C module.

pui8Data is a pointer where the read data is stored.

Description:

This function reads a byte of data from I2C receive FIFO and places it in the location specified by the **pui8Data** parameter. If there is no data available, this function returns 0.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

The number of elements read from the I2C receive FIFO.

17.2.2.3 I2CFIFODataPut

Writes a data byte to the I2C transmit FIFO.

Prototype:

```
void
I2CFIFODataPut(uint32_t ui32Base,
                uint8_t ui8Data)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8Data is the data to be placed into the transmit FIFO.

Description:

This function adds a byte of data to the I2C transmit FIFO. If there is no space available in the FIFO, this function waits for space to become available before returning.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.4 I2CFIFODataPutNonBlocking

Writes a data byte to the I2C transmit FIFO.

Prototype:

```
uint32_t
I2CFIFODataPutNonBlocking(uint32_t ui32Base,
                           uint8_t ui8Data)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8Data is the data to be placed into the transmit FIFO.

Description:

This function adds a byte of data to the I2C transmit FIFO. If there is no space available in the FIFO, this function returns a zero.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

The number of elements added to the I2C transmit FIFO.

17.2.2.5 I2CFIFOStatus

Gets the current FIFO status.

Prototype:

```
uint32_t  
I2CFIFOStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function retrieves the status for both the transmit (TX) and receive (RX) FIFOs. The trigger level for the transmit FIFO is set using [I2CTxFIFOConfigSet\(\)](#) and for the receive FIFO using [I2CRxFIFOConfigSet\(\)](#).

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

Returns the FIFO status, enumerated as a bit field containing **I2C_FIFO_RX_BELOW_TRIG_LEVEL**, **I2C_FIFO_RX_FULL**, **I2C_FIFO_RX_EMPTY**, **I2C_FIFO_TX_BELOW_TRIG_LEVEL**, **I2C_FIFO_TX_FULL**, and **I2C_FIFO_TX_EMPTY**.

17.2.2.6 I2CIntRegister

Registers an interrupt handler for the I2C module.

Prototype:

```
void  
I2CIntRegister(uint32_t ui32Base,  
                void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the I2C module.

pfnHandler is a pointer to the function to be called when the I2C interrupt occurs.

Description:

This function sets the handler to be called when an I2C interrupt occurs. This function enables the global interrupt in the interrupt controller; specific I2C interrupts must be enabled via [I2CMasterIntEnable\(\)](#) and [I2CSlaveIntEnable\(\)](#). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via [I2CMasterIntClear\(\)](#) and [I2CSlaveIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

17.2.2.7 I2CIntUnregister

Unregisters an interrupt handler for the I2C module.

Prototype:

```
void
I2CIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function clears the handler to be called when an I2C interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

17.2.2.8 I2CMasterBurstCountGet

Returns the current value of the burst transfer counter.

Prototype:

```
uint32_t
I2CMasterBurstCountGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function returns the current value of the burst transfer counter that is used by the FIFO mechanism. Software can use this value to determine how many bytes remain in a transfer, or where in the transfer the burst operation was if an error has occurred.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.9 I2CMasterBurstLengthSet

Set the burst length for a I2C master FIFO operation.

Prototype:

```
void  
I2CMasterBurstLengthSet(uint32_t ui32Base,  
                        uint8_t ui8Length)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8Length is the length of the burst transfer.

Description:

This function configures the burst length for a I2C Master FIFO operation. The burst field is limited to 8 bits or 256 bytes. The burst length applies to a single I2CMCS BURST operation meaning that it specifies the burst length for only the current operation (can be TX or RX). Each burst operation must configure the burst length prior to writing the BURST bit in the I2CMCS using [I2CMasterControl\(\)](#).

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.10 I2CMasterBusBusy

Indicates whether or not the I2C bus is busy.

Prototype:

```
bool  
I2CMasterBusBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function returns an indication of whether or not the I2C bus is busy. This function can be used in a multi-master environment to determine if another master is currently using the bus.

Returns:

Returns **true** if the I2C bus is busy; otherwise, returns **false**.

17.2.2.11 I2CMasterBusy

Indicates whether or not the I2C Master is busy.

Prototype:

```
bool
I2CMasterBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function returns an indication of whether or not the I2C Master is busy transmitting or receiving data.

Returns:

Returns **true** if the I2C Master is busy; otherwise, returns **false**.

17.2.2.12 I2CMasterControl

Controls the state of the I2C Master.

Prototype:

```
void
I2CMasterControl(uint32_t ui32Base,
                  uint32_t ui32Cmd)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Cmd command to be issued to the I2C Master.

Description:

This function is used to control the state of the Master send and receive operations. The **ui8Cmd** parameter can be one of the following values:

- **I2C_MASTER_CMD_SINGLE_SEND**
- **I2C_MASTER_CMD_SINGLE_RECEIVE**
- **I2C_MASTER_CMD_BURST_SEND_START**
- **I2C_MASTER_CMD_BURST_SEND_CONT**
- **I2C_MASTER_CMD_BURST_SEND_FINISH**
- **I2C_MASTER_CMD_BURST_SEND_ERROR_STOP**
- **I2C_MASTER_CMD_BURST_RECEIVE_START**
- **I2C_MASTER_CMD_BURST_RECEIVE_CONT**
- **I2C_MASTER_CMD_BURST_RECEIVE_FINISH**
- **I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP**
- **I2C_MASTER_CMD_QUICK_COMMAND**
- **I2C_MASTER_CMD_HS_MASTER_CODE_SEND**
- **I2C_MASTER_CMD_FIFO_SINGLE_SEND**
- **I2C_MASTER_CMD_FIFO_SINGLE_RECEIVE**
- **I2C_MASTER_CMD_FIFO_BURST_SEND_START**

- I2C_MASTER_CMD_FIFO_BURST_SEND_CONT
- I2C_MASTER_CMD_FIFO_BURST_SEND_FINISH
- I2C_MASTER_CMD_FIFO_BURST_SEND_ERROR_STOP
- I2C_MASTER_CMD_FIFO_BURST_RECEIVE_START
- I2C_MASTER_CMD_FIFO_BURST_RECEIVE_CONT
- I2C_MASTER_CMD_FIFO_BURST_RECEIVE_FINISH
- I2C_MASTER_CMD_FIFO_BURST_RECEIVE_ERROR_STOP

Note:

Not all Tiva devices have an I2C FIFO and support the FIFO commands. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.13 I2CMasterDataGet

Receives a byte that has been sent to the I2C Master.

Prototype:

```
uint32_t  
I2CMasterDataGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function reads a byte of data from the I2C Master Data Register.

Returns:

Returns the byte received from by the I2C Master, cast as an uint32_t.

17.2.2.14 I2CMasterDataPut

Transmits a byte from the I2C Master.

Prototype:

```
void  
I2CMasterDataPut (uint32_t ui32Base,  
                  uint8_t ui8Data)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8Data data to be transmitted from the I2C Master.

Description:

This function places the supplied data into I2C Master Data Register.

Returns:

None.

17.2.2.15 I2CMasterDisable

Disables the I2C master block.

Prototype:

```
void
I2CMasterDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function disables operation of the I2C master block.

Returns:

None.

17.2.2.16 I2CMasterEnable

Enables the I2C Master block.

Prototype:

```
void
I2CMasterEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function enables operation of the I2C Master block.

Returns:

None.

17.2.2.17 I2CMasterErr

Gets the error status of the I2C Master.

Prototype:

```
uint32_t
I2CMasterErr(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function is used to obtain the error status of the Master send and receive operations.

Returns:

Returns the error status, as one of **I2C_MASTER_ERR_NONE**,
I2C_MASTER_ERR_ADDR_ACK, **I2C_MASTER_ERR_DATA_ACK**, or
I2C_MASTER_ERR_ARB_LOST.

17.2.2.18 I2CMasterGlitchFilterConfigSet

Configures the I2C Master glitch filter.

Prototype:

```
void  
I2CMasterGlitchFilterConfigSet(uint32_t ui32Base,  
                                uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Config is the glitch filter configuration.

Description:

This function configures the I2C Master glitch filter. The value passed in to *ui32Config* determines the sampling range of the glitch filter, which is configurable between 1 and 32 system clock cycles. The default configuration of the glitch filter is 0 system clock cycles, which means that it's disabled.

The *ui32Config* field should be any of the following values:

- **I2C_MASTER_GLITCH_FILTER_DISABLED**
- **I2C_MASTER_GLITCH_FILTER_1**
- **I2C_MASTER_GLITCH_FILTER_2**
- **I2C_MASTER_GLITCH_FILTER_3**
- **I2C_MASTER_GLITCH_FILTER_4**
- **I2C_MASTER_GLITCH_FILTER_8**
- **I2C_MASTER_GLITCH_FILTER_16**
- **I2C_MASTER_GLITCH_FILTER_32**

Note:

Not all Tiva devices support this function. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.19 I2CMasterInitExpClk

Initializes the I2C Master block.

Prototype:

```
void  
I2CMasterInitExpClk(uint32_t ui32Base,  
                     uint32_t ui32I2CClk,  
                     bool bFast)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32I2CClk is the rate of the clock supplied to the I2C module.

bFast set up for fast data transfers.

Description:

This function initializes operation of the I2C Master block by configuring the bus speed for the master and enabling the I2C Master block.

If the parameter *bFast* is **true**, then the master block is set up to transfer data at 400 Kbps; otherwise, it is set up to transfer data at 100 Kbps. If Fast Mode Plus (1 Mbps) is desired, software should manually write the I2CMTPR after calling this function. For High Speed (3.4 Mbps) mode, a specific command is used to switch to the faster clocks after the initial communication with the slave is done at either 100 Kbps or 400 Kbps.

The peripheral clock is the same as the processor clock. This value is returned by [SysCtlClockGet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtlClockGet\(\)](#)).

Returns:

None.

17.2.2.20 I2CMasterIntClear

Clears I2C Master interrupt sources.

Prototype:

```
void
I2CMasterIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

The I2C Master interrupt source is cleared, so that it no longer asserts. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

17.2.2.21 I2CMasterIntClearEx

Clears I2C Master interrupt sources.

Prototype:

```
void
I2CMasterIntClearEx(uint32_t ui32Base,
                     uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified I2C Master interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CMasterIntEnableEx\(\)](#).

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

17.2.2.22 I2CMasterIntDisable

Disables the I2C Master interrupt.

Prototype:

```
void  
I2CMasterIntDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function disables the I2C Master interrupt source.

Returns:

None.

17.2.2.23 I2CMasterIntDisableEx

Disables individual I2C Master interrupt sources.

Prototype:

```
void  
I2CMasterIntDisableEx(uint32_t ui32Base,  
                      uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated I2C Master interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CMasterIntEnableEx\(\)](#).

Returns:

None.

17.2.2.24 I2CMasterIntEnable

Enables the I2C Master interrupt.

Prototype:

```
void
I2CMasterIntEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function enables the I2C Master interrupt source.

Returns:

None.

17.2.2.25 I2CMasterIntEnableEx

Enables individual I2C Master interrupt sources.

Prototype:

```
void
I2CMasterIntEnableEx(uint32_t ui32Base,
                      uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated I2C Master interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **I2C_MASTER_INT_RX_FIFO_FULL** - RX FIFO Full interrupt
- **I2C_MASTER_INT_TX_FIFO_EMPTY** - TX FIFO Empty interrupt

- **I2C_MASTER_INT_RX_FIFO_REQ** - RX FIFO Request interrupt
- **I2C_MASTER_INT_TX_FIFO_REQ** - TX FIFO Request interrupt
- **I2C_MASTER_INT_ARB_LOST** - Arbitration Lost interrupt
- **I2C_MASTER_INT_STOP** - Stop Condition interrupt
- **I2C_MASTER_INT_START** - Start Condition interrupt
- **I2C_MASTER_INT_NACK** - Address/Data NACK interrupt
- **I2C_MASTER_INT_TX_DMA_DONE** - TX DMA Complete interrupt
- **I2C_MASTER_INT_RX_DMA_DONE** - RX DMA Complete interrupt
- **I2C_MASTER_INT_TIMEOUT** - Clock Timeout interrupt
- **I2C_MASTER_INT_DATA** - Data interrupt

Note:

Not all Tiva devices support all of the listed interrupt sources. Please consult the device data sheet to determine if these features are supported.

Returns:

None.

17.2.2.26 I2CMasterIntStatus

Gets the current I2C Master interrupt status.

Prototype:

```
bool  
I2CMasterIntStatus(uint32_t ui32Base,  
                    bool bMasked)
```

Parameters:

ui32Base is the base address of the I2C module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This function returns the interrupt status for the I2C module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, returned as **true** if active or **false** if not active.

17.2.2.27 I2CMasterIntStatusEx

Gets the current I2C Master interrupt status.

Prototype:

```
uint32_t  
I2CMasterIntStatusEx(uint32_t ui32Base,  
                      bool bMasked)
```

Parameters:

ui32Base is the base address of the I2C module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This function returns the interrupt status for the I2C module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in [I2CMasterIntEnableEx\(\)](#).

17.2.2.28 I2CMasterLineStateGet

Reads the state of the SDA and SCL pins.

Prototype:

```
uint32_t
I2CMasterLineStateGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function returns the state of the I2C bus by providing the real time values of the SDA and SCL pins.

Note:

Not all Tiva devices support this function. Please consult the device data sheet to determine if this feature is supported.

Returns:

Returns the state of the bus with SDA in bit position 1 and SCL in bit position 0.

17.2.2.29 I2CMasterSlaveAddrSet

Sets the address that the I2C Master places on the bus.

Prototype:

```
void
I2CMasterSlaveAddrSet (uint32_t ui32Base,
                      uint8_t ui8SlaveAddr,
                      bool bReceive)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8SlaveAddr 7-bit slave address

bReceive flag indicating the type of communication with the slave

Description:

This function configures the address that the I2C Master places on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address indicates that the I2C

Master is initiating a read from the slave; otherwise the address indicates that the I2C Master is initiating a write to the slave.

Returns:

None.

17.2.2.30 I2CMasterTimeoutSet

Sets the Master clock timeout value.

Prototype:

```
void  
I2CMasterTimeoutSet(uint32_t ui32Base,  
                     uint32_t ui32Value)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Value is the number of I2C clocks before the timeout is asserted.

Description:

This function enables and configures the clock low timeout feature in the I2C peripheral. This feature is implemented as a 12-bit counter, with the upper 8-bits being programmable. For example, to program a timeout of 20ms with a 100-kHz SCL frequency, *ui32Value* is 0x7d.

Note:

Not all Tiva devices support this function. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.31 I2CRxFIFOConfigSet

Configures the I2C receive (RX) FIFO.

Prototype:

```
void  
I2CRxFIFOConfigSet(uint32_t ui32Base,  
                     uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Config is the configuration of the FIFO using specified macros.

Description:

This configures the I2C peripheral's receive FIFO. The receive FIFO can be used by the master or slave, but not both. The following macros are used to configure the RX FIFO behavior for master or slave, with or without DMA:

I2C_FIFO_CFG_RX_MASTER, **I2C_FIFO_CFG_RX_SLAVE**, **I2C_FIFO_CFG_RX_MASTER_DMA**,
I2C_FIFO_CFG_RX_SLAVE_DMA

To select the trigger level, one of the following macros should be used:

**I2C_FIFO_CFG_RX_TRIG_1, I2C_FIFO_CFG_RX_TRIG_2, I2C_FIFO_CFG_RX_TRIG_3,
I2C_FIFO_CFG_RX_TRIG_4, I2C_FIFO_CFG_RX_TRIG_5, I2C_FIFO_CFG_RX_TRIG_6,
I2C_FIFO_CFG_RX_TRIG_7, I2C_FIFO_CFG_RX_TRIG_8**

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.32 I2CRxFIFOFlush

Flushes the receive (RX) FIFO.

Prototype:

```
void
I2CRxFIFOFlush(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function flushes the I2C receive FIFO.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.33 I2CSlaveACKOverride

Configures ACK override behavior of the I2C Slave.

Prototype:

```
void
I2CSlaveACKOverride(uint32_t ui32Base,
                     bool bEnable)
```

Parameters:

ui32Base is the base address of the I2C module.

bEnable enables or disables ACK override.

Description:

This function enables or disables ACK override, allowing the user application to drive the value on SDA during the ACK cycle.

Note:

Not all Tiva devices support this function. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.34 I2CSlaveACKValueSet

Writes the ACK value.

Prototype:

```
void  
I2CSlaveACKValueSet(uint32_t ui32Base,  
                      bool bACK)
```

Parameters:

ui32Base is the base address of the I2C module.

bACK chooses whether to ACK (true) or NACK (false) the transfer.

Description:

This function puts the desired ACK value on SDA during the ACK cycle. The value written is only valid when ACK override is enabled using [I2CSlaveACKOverride\(\)](#).

Returns:

None.

17.2.2.35 I2CSlaveAddressSet

Sets the I2C slave address.

Prototype:

```
void  
I2CSlaveAddressSet(uint32_t ui32Base,  
                    uint8_t ui8AddrNum,  
                    uint8_t ui8SlaveAddr)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8AddrNum determines which slave address is set.

ui8SlaveAddr is the 7-bit slave address

Description:

This function writes the specified slave address. The **ui32AddrNum** field dictates which slave address is configured. For example, a value of 0 configures the primary address and a value of 1 configures the secondary.

Note:

Not all Tiva devices support a secondary address. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.36 I2CSlaveDataGet

Receives a byte that has been sent to the I2C Slave.

Prototype:

```
uint32_t
I2CSlaveDataGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function reads a byte of data from the I2C Slave Data Register.

Returns:

Returns the byte received from by the I2C Slave, cast as an uint32_t.

17.2.2.37 I2CSlaveDataPut

Transmits a byte from the I2C Slave.

Prototype:

```
void
I2CSlaveDataPut(uint32_t ui32Base,
                 uint8_t ui8Data)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8Data is the data to be transmitted from the I2C Slave

Description:

This function places the supplied data into I2C Slave Data Register.

Returns:

None.

17.2.2.38 I2CSlaveDisable

Disables the I2C slave block.

Prototype:

```
void
I2CSlaveDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function disables operation of the I2C slave block.

Returns:

None.

17.2.2.39 I2CSlaveEnable

Enables the I2C Slave block.

Prototype:

```
void  
I2CSlaveEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function enables operation of the I2C Slave block.

Returns:

None.

17.2.2.40 I2CSlaveFIFODisable

Disable FIFO usage for the I2C Slave.

Prototype:

```
void  
I2CSlaveFIFODisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function disables the FIFOs for the I2C Slave. After calling this function, the FIFOs are disabled, but the Slave remains active.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.41 I2CSlaveFIFOEnable

Enables FIFO usage for the I2C Slave.

Prototype:

```
void  
I2CSlaveFIFOEnable(uint32_t ui32Base,  
                    uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Config is the desired FIFO configuration of the I2C Slave.

Description:

This function configures the I2C Slave to use the FIFO(s). This function should be used in combination with [I2CTxFIFOConfigSet\(\)](#) and/or [I2CRxFIFOConfigSet\(\)](#), which configure the FIFO trigger level and tell the FIFO hardware whether to interact with the I2C Master or Slave. The application appropriate combination of **I2C_SLAVE_TX_FIFO_ENABLE** and **I2C_SLAVE_RX_FIFO_ENABLE** should be passed in to the *ui32Config* field.

The Slave I2CSCSR register is write-only, so any call to [I2CSlaveEnable\(\)](#), [I2CSlaveDisable](#) or [I2CSlaveFIFOEnable\(\)](#) overwrites the slave configuration. Therefore, application software should call [I2CSlaveEnable\(\)](#) followed by [I2CSlaveFIFOEnable\(\)](#) with the desired FIFO configuration.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.42 I2CSlaveInit

Initializes the I2C Slave block.

Prototype:

```
void
I2CSlaveInit(uint32_t ui32Base,
              uint8_t ui8SlaveAddr)
```

Parameters:

ui32Base is the base address of the I2C module.

ui8SlaveAddr 7-bit slave address

Description:

This function initializes operation of the I2C Slave block by configuring the slave address and enabling the I2C Slave block.

The parameter *ui8SlaveAddr* is the value that is compared against the slave address sent by an I2C master.

Returns:

None.

17.2.2.43 I2CSlaveIntClear

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

The I2C Slave interrupt source is cleared, so that it no longer asserts. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

17.2.2.44 I2CSlaveIntClearEx

Clears I2C Slave interrupt sources.

Prototype:

```
void  
I2CSlaveIntClearEx(uint32_t ui32Base,  
                    uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified I2C Slave interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CSlaveIntEnableEx\(\)](#).

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

17.2.2.45 I2CSlaveIntDisable

Disables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function disables the I2C Slave interrupt source.

Returns:

None.

17.2.2.46 I2CSlaveIntDisableEx

Disables individual I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntDisableEx(uint32_t ui32Base,
                     uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated I2C Slave interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CSlaveIntEnableEx\(\)](#).

Returns:

None.

17.2.2.47 I2CSlaveIntEnable

Enables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function enables the I2C Slave interrupt source.

Returns:

None.

17.2.2.48 I2CSlaveIntEnableEx

Enables individual I2C Slave interrupt sources.

Prototype:

```
void  
I2CSlaveIntEnableEx(uint32_t ui32Base,  
                     uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated I2C Slave interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The **ui32IntFlags** parameter is the logical OR of any of the following:

- **I2C_SLAVE_INT_RX_FIFO_FULL** - RX FIFO Full interrupt
- **I2C_SLAVE_INT_TX_FIFO_EMPTY** - TX FIFO Empty interrupt
- **I2C_SLAVE_INT_RX_FIFO_REQ** - RX FIFO Request interrupt
- **I2C_SLAVE_INT_TX_FIFO_REQ** - TX FIFO Request interrupt
- **I2C_SLAVE_INT_TX_DMA_DONE** - TX DMA Complete interrupt
- **I2C_SLAVE_INT_RX_DMA_DONE** - RX DMA Complete interrupt
- **I2C_SLAVE_INT_STOP** - Stop condition detected interrupt
- **I2C_SLAVE_INT_START** - Start condition detected interrupt
- **I2C_SLAVE_INT_DATA** - Data interrupt

Note:

Not all Tiva devices support the all of the listed interrupts. Please consult the device data sheet to determine if these features are supported.

Returns:

None.

17.2.2.49 I2CSlaveIntStatus

Gets the current I2C Slave interrupt status.

Prototype:

```
bool  
I2CSlaveIntStatus(uint32_t ui32Base,  
                   bool bMasked)
```

Parameters:

ui32Base is the base address of the I2C module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This function returns the interrupt status for the I2C Slave. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, returned as **true** if active or **false** if not active.

17.2.2.50 I2CSlaveIntStatusEx

Gets the current I2C Slave interrupt status.

Prototype:

```
uint32_t
I2CSlaveIntStatusEx(uint32_t ui32Base,
                     bool bMasked)
```

Parameters:

ui32Base is the base address of the I2C module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This function returns the interrupt status for the I2C Slave. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in [I2CSlaveIntEnableEx\(\)](#).

17.2.2.51 I2CSlaveStatus

Gets the I2C Slave status

Prototype:

```
uint32_t
I2CSlaveStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function returns the action requested from a master, if any. Possible values are:

- **I2C_SLAVE_ACT_NONE**
- **I2C_SLAVE_ACT_RREQ**
- **I2C_SLAVE_ACT_TREQ**
- **I2C_SLAVE_ACT_RREQ_FBR**
- **I2C_SLAVE_ACT_OWN2SEL**
- **I2C_SLAVE_ACT_QCMD**
- **I2C_SLAVE_ACT_QCMD_DATA**

Note:

Not all Tiva devices support the second I2C slave's own address or the quick command function. Please consult the device data sheet to determine if these features are supported.

Returns:

Returns **I2C_SLAVE_ACT_NONE** to indicate that no action has been requested of the I2C Slave, **I2C_SLAVE_ACT_RREQ** to indicate that an I2C master has sent data to the I2C Slave, **I2C_SLAVE_ACT_TREQ** to indicate that an I2C master has requested that the I2C Slave send data, **I2C_SLAVE_ACT_RREQ_FBR** to indicate that an I2C master has sent data to the I2C slave and the first byte following the slave's own address has been received, **I2C_SLAVE_ACT_OWN2SEL** to indicate that the second I2C slave address was matched, **I2C_SLAVE_ACT_QCMD** to indicate that a quick command was received, and **I2C_SLAVE_ACT_QCMD_DATA** to indicate that the data bit was set when the quick command was received.

17.2.2.52 I2CTxFIFOConfigSet

Configures the I2C transmit (TX) FIFO.

Prototype:

```
void
I2CTxFIFOConfigSet(uint32_t ui32Base,
                    uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the I2C module.

ui32Config is the configuration of the FIFO using specified macros.

Description:

This configures the I2C peripheral's transmit FIFO. The transmit FIFO can be used by the master or slave, but not both. The following macros are used to configure the TX FIFO behavior for master or slave, with or without DMA:

I2C_FIFO_CFG_TX_MASTER, **I2C_FIFO_CFG_TX_SLAVE**, **I2C_FIFO_CFG_TX_MASTER_DMA**,
I2C_FIFO_CFG_TX_SLAVE_DMA

To select the trigger level, one of the following macros should be used:

I2C_FIFO_CFG_TX_TRIG_1, **I2C_FIFO_CFG_TX_TRIG_2**, **I2C_FIFO_CFG_TX_TRIG_3**,
I2C_FIFO_CFG_TX_TRIG_4, **I2C_FIFO_CFG_TX_TRIG_5**, **I2C_FIFO_CFG_TX_TRIG_6**,
I2C_FIFO_CFG_TX_TRIG_7, **I2C_FIFO_CFG_TX_TRIG_8**

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.2.2.53 I2CTxFIFOFlush

Flushes the transmit (TX) FIFO.

Prototype:

```
void
I2CTxFIFOFlush(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C module.

Description:

This function flushes the I2C transmit FIFO.

Note:

Not all Tiva devices have an I2C FIFO. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

17.3 Programming Example

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

```
//
// Initialize Master and Slave
//
I2CMasterInitExpClk(I2C0_BASE, SysCtlClockGet(), true);

//
// Specify slave address
//
I2CMasterSlaveAddrSet(I2C0_BASE, 0x3B, false);

//
// Place the character to be sent in the data register
//
I2CMasterDataPut(I2C0_BASE, 'Q');

//
// Initiate send of character from Master to Slave
//
I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_SEND);

//
// Delay until transmission completes
//
while(I2CMasterBusBusy(I2C0_BASE))
{
}
```


18 Interrupt Controller (NVIC)

Introduction	351
API Functions	352
Programming Example	362

18.1 Introduction

The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, register interrupt handlers, and set the priority of interrupts.

The NVIC provides global interrupt masking, prioritization, and handler dispatching. Devices within the Tiva family support up to 154 interrupt sources and eight priority levels. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The NVIC is tightly coupled with the Cortex-M microprocessor. When the processor responds to an interrupt, the NVIC supplies the address of the function to handle the interrupt directly to the processor. This action eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, reducing interrupt response time.

The interrupt prioritization in the NVIC allows higher priority interrupts to be handled before lower priority interrupts, as well as allowing preemption of lower priority interrupt handlers by higher priority interrupts. Again, this helps reduce interrupt response time (for example, a 1 ms system control interrupt is not held off by the execution of a lower priority 1 second housekeeping interrupt handler).

Sub-prioritization is also possible; instead of having N bits of preemptable prioritization, the NVIC can be configured (via software) for N - M bits of preemptable prioritization and M bits of sub-priority. In this scheme, two interrupts with the same preemptable prioritization but different sub-priorities do not cause a preemption; tail chaining is used instead to process the two interrupts back-to-back.

If two interrupts with the same priority (and sub-priority if so configured) are asserted at the same time, the one with the lower interrupt number is processed first. The NVIC keeps track of the nesting of interrupt handlers, allowing the processor to return from interrupt context only once all nested and pending interrupts have been handled.

Interrupt handlers can be configured in one of two ways; statically at compile time or dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the application's startup code. When statically configured, the interrupts must be explicitly enabled in the NVIC via [IntEnable\(\)](#) before the processor can respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself). Statically configuring the interrupt table provides the fastest interrupt response time because the stacking operation (a write to SRAM) can be performed in parallel with the interrupt handler table fetch (a read from Flash), as well as the prefetch of the interrupt handler itself (assuming it is also in Flash).

Alternatively, interrupts can be configured at run-time using [IntRegister\(\)](#) (or the analog in each individual driver). When using [IntRegister\(\)](#), the interrupt must also be enabled as before; when using the analogue in each individual driver, [IntEnable\(\)](#) is called by the driver and does not need to be called by the application. Run-time configuration of interrupts adds a small latency to the interrupt response time because the stacking operation (a write to SRAM) and the interrupt handler table fetch (a read from SRAM) must be performed sequentially.

Run-time configuration of interrupt handlers requires that the interrupt handler table be placed on a 1-kB boundary in SRAM (typically this is at the beginning of SRAM). Failure to do so results in an incorrect vector address being fetched in response to an interrupt. The vector table is in a section called “vtable” and must be placed appropriately with a linker script.

This driver is contained in `driverlib/interrupt.c`, with `driverlib/interrupt.h` containing the API declarations for use by applications.

18.2 API Functions

Functions

- `void IntDisable (uint32_t ui32Interrupt)`
- `void IntEnable (uint32_t ui32Interrupt)`
- `uint32_t IntIsEnabled (uint32_t ui32Interrupt)`
- `bool IntMasterDisable (void)`
- `bool IntMasterEnable (void)`
- `void IntPendClear (uint32_t ui32Interrupt)`
- `void IntPendSet (uint32_t ui32Interrupt)`
- `int32_t IntPriorityGet (uint32_t ui32Interrupt)`
- `uint32_t IntPriorityGroupingGet (void)`
- `void IntPriorityGroupingSet (uint32_t ui32Bits)`
- `uint32_t IntPriorityMaskGet (void)`
- `void IntPriorityMaskSet (uint32_t ui32PriorityMask)`
- `void IntPrioritySet (uint32_t ui32Interrupt, uint8_t ui8Priority)`
- `void IntRegister (uint32_t ui32Interrupt, void (*pfnHandler)(void))`
- `void IntTrigger (uint32_t ui32Interrupt)`
- `void IntUnregister (uint32_t ui32Interrupt)`

18.2.1 Detailed Description

The primary function of the interrupt controller API is to manage the interrupt vector table used by the NVIC to dispatch interrupt requests. Registering an interrupt handler is a simple matter of inserting the handler address into the table. By default, the table is filled with pointers to an internal handler that loops forever; it is an error for an interrupt to occur when there is no interrupt handler registered to process it. Therefore, interrupt sources must not be enabled before a handler has been registered, and interrupt sources must be disabled before a handler is unregistered. Interrupt handlers are managed with `IntRegister()` and `IntUnregister()`.

Each interrupt source can be individually enabled and disabled via `IntEnable()` and `IntDisable()`. The processor interrupt can be enabled and disabled via `IntMasterEnable()` and `IntMasterDisable()`; this does not affect the individual interrupt enable states. Masking of the processor interrupt can be used as a simple critical section (only an NMI can interrupt the processor while the processor interrupt is disabled), although masking the processor interrupt can have adverse effects on the interrupt response time.

The priority of each interrupt source can be set and examined via `IntPrioritySet()` and `IntPriorityGet()`. The priority assignments are defined by the hardware; the upper N bits of the 8-bit priority

are examined to determine the priority of an interrupt (for the Tiva family, N is 3). This protocol allows priorities to be defined without knowledge of the exact number of supported priorities; moving to a device with more or fewer priority bits is made easier as the interrupt source continues to have a similar level of priority. Smaller priority numbers correspond to higher interrupt priority, so 0 is the highest priority.

18.2.2 Interrupt Mapping

The TM4C123 and TM4C129 devices have different interrupt mapping for the same peripheral interrupts. This requires that the application have a way to control the mapping of interrupts so that the correct interrupt number is used. For example the same interrupt name **INT_USB0** has interrupt number 60 on TM4C123 devices and 58 on TM4C129 devices. All of the interrupt number macros start with **INT_*** and there are two defines that an application uses to allow the correct interrupt number to be mapped to these **INT_*** macros. The first set of macros that are the **TARGET_IS_TM4C*** which are used to define the class of part in use by the application. The second option is to specify the exact part that the application is using by defining one of the **PART_<partno>** values. For example, for a board using a TM4C129XNCZAD device the application would define the **PART_TM4C129XNCZAD** and/or one of the **TARGET_IS_TM4C129_*** macros depending on the revision of the device that is in use by the application. This conditional mapping of the interrupts allows applications to use a common name for the interrupt numbers without having to look up the actual interrupt number.

Note:

The **TARGET_IS_TM4C*** and **PART_<partno>** macros also control ROM and pin mapping functions as well. See the [Direct ROM Calls](#) and the [GPIO Pin Configuration](#) sections of this document for more details on how these defines are used by these modules.

The valid interrupt for the interrupt API functions are the following: **INT_ADC0SS0**, **INT_ADC0SS1**, **INT_ADC0SS2**, **INT_ADC0SS3**, **INT_ADC1SS0**, **INT_ADC1SS1**, **INT_ADC1SS2**, **INT_ADC1SS3**, **INT_AES0**, **INT_CAN0**, **INT_CAN1**, **INT_COMP0**, **INT_COMP1**, **INT_COMP2**, **INT_DES0**, **INT_EMAC0**, **INT_EPIO**, **INT_FLASH**, **INT_GPIOA**, **INT_GPIOB**, **INT_GPIOC**, **INT_GPIOD**, **INT_GPIOE**, **INT_GPIOF**, **INT_GPIOG**, **INT_GPIOH**, **INT_GPIOJ**, **INT_GPIOK**, **INT_GPIOL**, **INT_GPIOM**, **INT_GPION**, **INT_GPIOP0**, **INT_GPIOP1**, **INT_GPIOP2**, **INT_GPIOP3**, **INT_GPIOP4**, **INT_GPIOP5**, **INT_GPIOP6**, **INT_GPIOP7**, **INT_GPIOQ0**, **INT_GPIOQ1**, **INT_GPIOQ2**, **INT_GPIOQ3**, **INT_GPIOQ4**, **INT_GPIOQ5**, **INT_GPIOQ6**, **INT_GPIOQ7**, **INT_GPIOR**, **INT_GPIO_S**, **INT_GPIO_T**, **INT_HIBERNATE**, **INT_I2C0**, **INT_I2C1**, **INT_I2C2**, **INT_I2C3**, **INT_I2C4**, **INT_I2C5**, **INT_I2C6**, **INT_I2C7**, **INT_I2C8**, **INT_I2C9**, **INT_LCD0**, **INT_ONEWIRE0**, **INT_PWM0_0**, **INT_PWM0_1**, **INT_PWM0_2**, **INT_PWM0_3**, **INT_PWM0_FAULT**, **INT_PWM1_0**, **INT_PWM1_1**, **INT_PWM1_2**, **INT_PWM1_3**, **INT_PWM1_FAULT**, **INT_QEI0**, **INT_QEI1**, **INT_SHA0**, **INT_SSI0**, **INT_SSI1**, **INT_SSI2**, **INT_SSI3**, **INT_SYSCTL**, **INT_SYSEXC**, **INT_TAMPER0**, **INT_TIMER0A**, **INT_TIMER0B**, **INT_TIMER1A**, **INT_TIMER1B**, **INT_TIMER2A**, **INT_TIMER2B**, **INT_TIMER3A**, **INT_TIMER3B**, **INT_TIMER4A**, **INT_TIMER4B**, **INT_TIMER5A**, **INT_TIMER5B**, **INT_TIMER6A**, **INT_TIMER6B**, **INT_TIMER7A**, **INT_TIMER7B**, **INT_UART0**, **INT_UART1**, **INT_UART2**, **INT_UART3**, **INT_UART4**, **INT_UART5**, **INT_UART6**, **INT_UART7**, **INT_UDMA**, **INT_UDMAERR**, **INT_USB0**, **INT_WATCHDOG**, **INT_WTIMER0A**, **INT_WTIMER0B**, **INT_WTIMER1A**, **INT_WTIMER1B**, **INT_WTIMER2A**, **INT_WTIMER2B**, **INT_WTIMER3A**, **INT_WTIMER3B**, **INT_WTIMER4A**, **INT_WTIMER4B**, **INT_WTIMER5A**, **INT_WTIMER5B**

18.2.3 Function Documentation

18.2.3.1 IntDisable

Disables an interrupt.

Prototype:

```
void  
IntDisable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be disabled.

Description:

The specified interrupt is disabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Example: Disable the UART 0 interrupt.

```
//  
// Disable the UART 0 interrupt in the interrupt controller.  
//  
IntDisable(INT_UART0);
```

Returns:

None.

18.2.3.2 IntEnable

Enables an interrupt.

Prototype:

```
void  
IntEnable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be enabled.

Description:

The specified interrupt is enabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Example: Enable the UART 0 interrupt.

```
//  
// Enable the UART 0 interrupt in the interrupt controller.  
//  
IntEnable(INT_UART0);
```

Returns:

None.

18.2.3.3 IntIsEnabled

Returns if a peripheral interrupt is enabled.

Prototype:

```
uint32_t
IntIsEnabled(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to check.

Description:

This function checks if the specified interrupt is enabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file.

Example: Disable the UART 0 interrupt if it is enabled.

```
//  
// Disable the UART 0 interrupt if it is enabled.  
//  
if(IntIsEnabled(INT_UART0))  
{  
    IntDisable(INT_UART0);  
}
```

Returns:

A non-zero value if the interrupt is enabled.

18.2.3.4 IntMasterDisable

Disables the processor interrupt.

Prototype:

```
bool
IntMasterDisable(void)
```

Description:

This function prevents the processor from receiving interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Note:

Previously, this function had no return value. As such, it was possible to include `interrupt.h` and call this function without having included `hw_types.h`. Now that the return is a `bool`, a compiler error occurs in this case. The solution is to include `hw_types.h` before including `interrupt.h`.

Example: Disable interrupts to the processor.

```
//  
// Disable interrupts to the processor.  
//  
IntMasterDisable();
```

Returns:

Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

18.2.3.5 IntMasterEnable

Enables the processor interrupt.

Prototype:

```
bool  
IntMasterEnable(void)
```

Description:

This function allows the processor to respond to interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Example: Enable interrupts to the processor.

```
//  
// Enable interrupts to the processor.  
//  
IntMasterEnable();
```

Returns:

Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

18.2.3.6 IntPendClear

Un-pends an interrupt.

Prototype:

```
void  
IntPendClear(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be un-pended. The **ui32Interrupt** parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file.

Description:

The specified interrupt is un-pended in the interrupt controller. This causes any previously generated interrupts that have not been handled yet (due to higher priority interrupts or the interrupt not having been enabled yet) to be discarded.

Example: Un-pend a UART 0 interrupt.

```
//  
// Un-pend a UART 0 interrupt.  
//  
IntPendClear(INT_UART0);
```

Returns:

None.

18.2.3.7 IntPendSet

Pends an interrupt.

Prototype:

```
void
IntPendSet (uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be pended.

Description:

The specified interrupt is pended in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Pending an interrupt causes the interrupt controller to execute the corresponding interrupt handler at the next available time, based on the current interrupt state priorities. For example, if called by a higher priority interrupt handler, the specified interrupt handler is not called until after the current interrupt handler has completed execution. The interrupt must have been enabled for it to be called.

Example: Pend a UART 0 interrupt.

```
//
// Pend a UART 0 interrupt.
//
IntPendSet (INT_UART0);
```

Returns:

None.

18.2.3.8 IntPriorityGet

Gets the priority of an interrupt.

Prototype:

```
int32_t
IntPriorityGet (uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt in question.

Description:

This function gets the priority of an interrupt. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. See [IntPrioritySet\(\)](#) for a full definition of the priority value.

Example: Get the current UART 0 interrupt priority.

```
//
// Get the current UART 0 interrupt priority.
//
IntPriorityGet (INT_UART0);
```

Returns:

Returns the interrupt priority for the given interrupt.

18.2.3.9 IntPriorityGroupingGet

Gets the priority grouping of the interrupt controller.

Prototype:

```
uint32_t  
IntPriorityGroupingGet (void)
```

Description:

This function returns the split between preemptable priority levels and sub-priority levels in the interrupt priority specification.

Example: Get the priority grouping for the interrupt controller.

```
//  
// Get the priority grouping for the interrupt controller.  
//  
IntPriorityGroupingGet();
```

Returns:

The number of bits of preemptable priority.

18.2.3.10 IntPriorityGroupingSet

Sets the priority grouping of the interrupt controller.

Prototype:

```
void  
IntPriorityGroupingSet (uint32_t ui32Bits)
```

Parameters:

ui32Bits specifies the number of bits of preemptable priority.

Description:

This function specifies the split between preemptable priority levels and sub-priority levels in the interrupt priority specification. The range of the grouping values are dependent upon the hardware implementation; on the Tiva C and E Series family, three bits are available for hardware interrupt prioritization and therefore priority grouping values of three through seven have the same effect.

Example: Set the priority grouping for the interrupt controller.

```
//  
// Set the priority grouping for the interrupt controller to 2 bits.  
//  
IntPriorityGroupingSet(2);
```

Returns:

None.

18.2.3.11 IntPriorityMaskGet

Gets the priority masking level

Prototype:

```
uint32_t
IntPriorityMaskGet (void)
```

Description:

This function gets the current setting of the interrupt priority masking level. The value returned is the priority level such that all interrupts of that and lesser priority are masked. A value of 0 means that priority masking is disabled.

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater are blocked.

The hardware priority mechanism only looks at the upper 3 bits of the priority level, so any prioritization must be performed in those bits.

Example: Get the current interrupt priority mask.

```
//
// Get the current interrupt priority mask.
//
IntPriorityMaskGet();
```

Returns:

Returns the value of the interrupt priority level mask.

18.2.3.12 IntPriorityMaskSet

Sets the priority masking level

Prototype:

```
void
IntPriorityMaskSet (uint32_t ui32PriorityMask)
```

Parameters:

ui32PriorityMask is the priority level that is masked.

Description:

This function sets the interrupt priority masking level so that all interrupts at the specified or lesser priority level are masked. Masking interrupts can be used to globally disable a set of interrupts with priority below a predetermined threshold. A value of 0 disables priority masking.

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater are blocked.

Note:

The hardware priority mechanism only looks at the upper 3 bits of the priority level, so any prioritization must be performed in those bits.

Example: Mask of interrupt priorities greater than or equal to 0x80.

```
//
// Mask of interrupt priorities greater than or equal to 0x80.
//
IntPriorityMaskSet (0x80);
```

Returns:

None.

18.2.3.13 IntPrioritySet

Sets the priority of an interrupt.

Prototype:

```
void  
IntPrioritySet (uint32_t ui32Interrupt,  
                uint8_t ui8Priority)
```

Parameters:

ui32Interrupt specifies the interrupt in question.

ui8Priority specifies the priority of the interrupt.

Description:

This function is used to set the priority of an interrupt. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. The *ui8Priority* parameter specifies the interrupts hardware priority level of the interrupt in the interrupt controller. When multiple interrupts are asserted simultaneously, the ones with the highest priority are processed before the lower priority interrupts. Smaller numbers correspond to higher interrupt priorities; priority 0 is the highest interrupt priority.

Note:

The hardware priority mechanism only looks at the upper 3 bits of the priority level, so any prioritization must be performed in those bits. The remaining bits can be used to sub-prioritize the interrupt sources, and may be used by the hardware priority mechanism on a future part. This arrangement allows priorities to migrate to different NVIC implementations without changing the gross prioritization of the interrupts.

Example: Set priorities for UART 0 and USB interrupts.

```
//  
// Set the UART 0 interrupt priority to the lowest priority.  
//  
IntPrioritySet (INT_UART0, 0xE0);  
  
//  
// Set the USB 0 interrupt priority to the highest priority.  
//  
IntPrioritySet (INT_USB0, 0);
```

Returns:

None.

18.2.3.14 IntRegister

Registers a function to be called when an interrupt occurs.

Prototype:

```
void
IntRegister(uint32_t ui32Interrupt,
            void (*pfnHandler) (void))
```

Parameters:

ui32Interrupt specifies the interrupt in question.
pfnHandler is a pointer to the function to be called.

Description:

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. When the interrupt occurs, if it is enabled (via [IntEnable\(\)](#)), the handler function is called in interrupt context. Because the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Note:

The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise the NVIC does not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter.

Example: Set the UART 0 interrupt handler.

```
//
// UART 0 interrupt handler.
//
void
UART0Handler(void)
{
    //
    // Handle interrupt.
    //
}

//
// Set the UART 0 interrupt handler.
//
IntRegister(INT_UART0,    UART0Handler);
```

Returns:

None.

18.2.3.15 IntTrigger

Triggers an interrupt.

Prototype:

```
void
IntTrigger(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be triggered.

Description:

This function performs a software trigger of an interrupt. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. The interrupt controller behaves as if the corresponding interrupt line was asserted, and the interrupt is handled in the same manner (meaning that it must be enabled in order to be processed, and the processing is based on its priority with respect to other unhandled interrupts).

Returns:

None.

18.2.3.16 IntUnregister

Unregisters the function to be called when an interrupt occurs.

Prototype:

```
void  
IntUnregister(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt in question.

Description:

This function is used to indicate that no handler is called when the given interrupt is asserted to the processor. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. The interrupt source is automatically disabled (via [IntDisable\(\)](#)) if necessary.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Example: Reset the UART 0 interrupt handler to the default handler.

```
//  
// Reset the UART 0 interrupt handler to the default handler.  
//  
IntUnregister(INT_UART0);
```

Returns:

None.

18.3 Programming Example

The following example shows how to use the Interrupt Controller API to register an interrupt handler for UART 0 and enable the interrupt.

```
//  
// The interrupt handler function.  
//  
extern void IntHandler(void);  
  
//  
// Register the interrupt handler function for UART 0.  
//  
IntRegister(INT_UART0, IntHandler);  
  
//  
// Enable the interrupt for UART 0.  
//  
IntEnable(INT_UART0);  
  
//  
// Enable UART 0.  
//  
IntMasterEnable();
```


19 LCD Controller (LCD)

Introduction	365
API Functions	365
Programming Example	393

19.1 Introduction

The LCD Controller allows a variety of different character and graphic displays to be connected to and driven by the microcontroller. The LCD module contains two independent controllers, one supporting LCD Interface Display Driver (LIDD) mode command and data transactions to character displays as well as displays containing an integrated controller with a packet-based interface, and the other driving clock, syncs and data suitable for RGB raster displays. Up to two simultaneous LIDD displays or a single RGB raster mode display may be driven.

The LCD API provides functions to configure the interface type and timing for the attached display or displays. For LIDD mode displays, functions allow an application to send commands or data to the display or read back status or data. For raster displays, functions allow the pixel clock, HSYNC, VSYNC and ACTIVE timings to be set. Additional functions allow the frame buffer memory to be configured and the color palette to be set.

This driver is contained in `driverlib/lcd.c`, with `driverlib/lcd.h` containing the API declarations for use by applications.

19.2 API Functions

Data Structures

- [tLCDIDDTiming](#)
- [tLCDRasterTiming](#)

Defines

- [CYCLES_FROM_TIME_NS\(ui32ClockFreq, ui32Time_nS\)](#)
- [CYCLES_FROM_TIME_US\(ui32ClockFreq, ui32Time_uS\)](#)
- [PAL_FROM_RGB\(ui32RGBColor\)](#)

Functions

- void [LCDClockReset](#) (uint32_t ui32Base, uint32_t ui32Clocks)
- void [LCDDMAConfigSet](#) (uint32_t ui32Base, uint32_t ui32Config)
- void [LCDIDDCmdWrite](#) (uint32_t ui32Base, uint32_t ui32CS, uint16_t ui16Cmd)
- void [LCDIDDCfgSet](#) (uint32_t ui32Base, uint32_t ui32Config)
- uint16_t [LCDIDDDataRead](#) (uint32_t ui32Base, uint32_t ui32CS)

- void [LCDIDDDDataWrite](#) (uint32_t ui32Base, uint32_t ui32CS, uint16_t ui16Data)
- void [LCDIDDDMADisable](#) (uint32_t ui32Base)
- void [LCDIDDDMAWrite](#) (uint32_t ui32Base, uint32_t ui32CS, const uint32_t *pu32Data, uint32_t ui32Count)
- uint16_t [LCDIDDIndexedRead](#) (uint32_t ui32Base, uint32_t ui32CS, uint16_t ui16Addr)
- void [LCDIDDIndexedWrite](#) (uint32_t ui32Base, uint32_t ui32CS, uint16_t ui16Addr, uint16_t ui16Data)
- uint16_t [LCDIDDStatusRead](#) (uint32_t ui32Base, uint32_t ui32CS)
- void [LCDIDDTimingSet](#) (uint32_t ui32Base, uint32_t ui32CS, const [tLCDIDDTiming](#) *pTiming)
- void [LCDIntClear](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [LCDIntDisable](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [LCDIntEnable](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [LCDIntRegister](#) (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t [LCDIntStatus](#) (uint32_t ui32Base, bool bMasked)
- void [LCDIntUnregister](#) (uint32_t ui32Base)
- uint32_t [LCDModeSet](#) (uint32_t ui32Base, uint8_t ui8Mode, uint32_t ui32PixClk, uint32_t ui32SysClk)
- void [LCDRasterACBiasIntCountSet](#) (uint32_t ui32Base, uint8_t ui8Count)
- void [LCDRasterConfigSet](#) (uint32_t ui32Base, uint32_t ui32Config, uint8_t ui8PalLoadDelay)
- void [LCDRasterDisable](#) (uint32_t ui32Base)
- void [LCDRasterEnable](#) (uint32_t ui32Base)
- bool [LCDRasterEnabled](#) (uint32_t ui32Base)
- void [LCDRasterFrameBufferSet](#) (uint32_t ui32Base, uint8_t ui8Buffer, uint32_t *pu32Addr, uint32_t ui32NumBytes)
- void [LCDRasterPaletteSet](#) (uint32_t ui32Base, uint32_t ui32Type, uint32_t *pu32Addr, const uint32_t *pu32SrcColors, uint32_t ui32Start, uint32_t ui32Count)
- void [LCDRasterSubPanelConfigSet](#) (uint32_t ui32Base, uint32_t ui32Flags, uint32_t ui32BottomLines, uint32_t ui32DefaultPixel)
- void [LCDRasterSubPanelDisable](#) (uint32_t ui32Base)
- void [LCDRasterSubPanelEnable](#) (uint32_t ui32Base)
- void [LCDRasterTimingSet](#) (uint32_t ui32Base, const [tLCDRasterTiming](#) *pTiming)

19.2.1 Detailed Description

The LCD Controller API is broken into 4 groups of functions: those that deal with configuration, those relating to control when in LCD Interface Display Driver (LIDD) mode, those relating to control when in Raster mode, and those that manage interrupts.

The configuration of the LCD Controller module is managed by the [LCDModeSet\(\)](#), [LCDIDDDConfigSet\(\)](#), [LCDIDDTimingSet\(\)](#), [LCDRasterConfigSet\(\)](#), [LCDRasterTimingSet\(\)](#), [LCDRasterSubPanelConfigSet\(\)](#), [LCDDDMAConfigSet\(\)](#), [LCDRasterPaletteSet\(\)](#), and [LCDRasterFrameBufferSet\(\)](#) functions.

When in LIDD mode, data transfer to and from the display is controlled by the [LCDIDDDCommandWrite\(\)](#), [LCDIDDDDataWrite\(\)](#), [LCDIDDIndexedWrite\(\)](#), [LCDIDDStatusRead\(\)](#), [LCDIDDDDataRead\(\)](#), [LCDIDDIndexedRead\(\)](#), [LCDIDDDMAWrite\(\)](#) and [LCDIDDDMADisable\(\)](#) functions.

When in raster mode, communication with the display is controlled by the [LCDRasterEnable\(\)](#), [LCDRasterDisable\(\)](#), [LCDRasterSubPanelEnable\(\)](#), [LCDRasterSubPanelDisable\(\)](#) and [LCDRasterACBiasIntCountSet\(\)](#) functions.

Interrupts from the LCD Controller module are managed using the [LCDIntStatus\(\)](#), [LCDIntClear\(\)](#), [LCDIntDisable\(\)](#), [LCDIntEnable\(\)](#), [LCDIntRegister\(\)](#) and [LCDIntUnregister\(\)](#) functions.

19.2.2 LCD Interface Display Driver (LIDD) Mode

The LIDD mode controller allows connection of displays via a synchronous or asynchronous interface using Chip Select (CS), Write Enable (WE), Output Enable (OE) and Address Latch Enable (ALE) signals along with a parallel data bus of 8 to 16 bits.

Several different bus signaling modes are supported to allow connection of devices making use of Hitachi, Motorola or Intel bus conventions. For timing diagrams showing the operation of each of these modes, please consult the datasheet for your particular Tiva part.

LIDD mode would typically be used with displays where updates are made via a packet-based command and data protocol rather than by direct access to a local frame buffer. These will generally be lower resolution RGB panels or character-mode displays.

Interface signaling, timing and basic mode are set using three API functions. To select LIDD mode, [LCDModeSet\(\)](#) is called with the ui8Mode parameter set to **LCD_MODE_LIDD** and the desired bit clock rate. [LCDIDDDConfigSet\(\)](#) is then called to set the basic operating mode of the LIDD interface (synchronous or asynchronous Motorola or Intel mode, or asynchronous Hitachi mode) and configure the polarities of the various interface control signals. Finally [LCDIDDTimingSet\(\)](#) is called to set the timings associated with the interface strobes. When using the asynchronous Intel or Motorola interface modes, two independent chip select (CS) signals are available and timings may be set for these individually allowing two different LIDD panels to be attached simultaneously.

Data may be transferred to or from the panel either one item (8-bit byte or 16-bit word depending upon the panel) at a time or in blocks using DMA. In basic operation, the API provides two sets of functions which allow reading and writing. The choice of function is dictated by the specification of the display in use and the hardware interface it uses. Functions [LCDIDDDDataRead\(\)](#), [LCDIDDDDataWrite\(\)](#), [LCDIDDCommandWrite\(\)](#) and [LCDIDDStatusRead\(\)](#) can be used with panels which support a Data/Control (DC) signal to control command or data accesses. Command writes and status reads are performed with the DC signal (on the ALE pin) active whereas data operations occur with the DC signal inactive.

For displays using an external address latch and configured in one of the Intel or Motorola modes, [LCDIDDDIndexedRead\(\)](#) and [LCDIDDDIndexedWrite\(\)](#) may be used to read or write indexed registers in the display.

To transfer large blocks of data to the display, DMA may be used via the [LCDIDDDDMAWrite\(\)](#) function. This function enables the DMA engine in the LCD controller before transferring the required block of data to the display. The DMA engine transfers data 16 bits at a time so data must be padded if a display with an 8 bit interface is used. The completion of a DMA transfer is indicated via the **LCD_INT_DMA_DONE** interrupt.

Care must be taken when mixing DMA and non-DMA accesses to the display. The application is responsible for ensuring that any previous DMA operation has completed before another is scheduled. Similarly, the application must ensure that it disables DMA using [LCDIDDDMADisable\(\)](#) before making a call to [LCDIDDCommandWrite\(\)](#), [LCDIDDStatusRead\(\)](#), [LCDIDDDDataWrite\(\)](#), [LCDIDDDDataRead\(\)](#), [LCDIDDDIndexedWrite\(\)](#) or [LCDIDDDIndexedRead\(\)](#).

19.2.3 Raster Mode

Raster mode connects both passive- and active-matrix displays using a traditional video-style, synchronous interface based on VSYNC (LCDFP), HSYNC (LCDLP), VALID (LCDAC), CLK (LCPCP) and DATA (LCDDATA) signals. Unlike LIDD displays which contain their own frame buffer, the display image for a raster display is stored in internal or EPI-attached memory and is scanned to the display using the LCD controller hardware and appropriate refresh rate and line timings provided by the application.

The function [LCDModeSet\(\)](#) with parameter *ui8Mode* set to **LCD_MODE_RASTER** will select the raster controller mode. This function also sets the required pixel clock frequency. Note that this must be an integer factor of the system clock frequency so applications must ensure that they choose an appropriate system clock frequency to allow the required pixel clock to be set. In cases where the current system clock setting is such that the exact requested pixel clock cannot be set, [LCDModeSet\(\)](#) will set the closest lower pixel clock that can be derived given the current system clock frequency and will return that frequency to the caller.

Properties of the display being driven can be set using a call to the function [LCDRasterConfigSet\(\)](#). This configures active- or passive-matrix display, how the color palette is used and various parameters relating to the packing order of pixels in memory. Raster timings and signal polarities are configured using a call to [LCDRasterTimingSet\(\)](#).

The frame buffer, whose layout is described in the following section, is configured using a call to [LCDRasterFrameBufferSet\(\)](#) which accepts parameters indicating the address of the start of the buffer in memory and its size. Because the frame buffer also contains the pixel format identifier and color palette in addition to the pixel data, [LCDRasterPaletteSet\(\)](#) must be called to initialize the palette and format header before the raster is enabled. This is required for all pixel formats, even those which do not require a color lookup table, to ensure correct display.

Once all controller and frame buffer initialization is complete, the display raster can be enabled by calling [LCDRasterEnable\(\)](#). If the display is to be shut down at any point, [LCDRasterDisable\(\)](#) may be used.

The LCD controller also supports a subpanel mode which may be helpful in memory constrained systems. This allows the active area on the display to be set to use a number of lines less than the native height. Lines above or below the active area are filled with a default color. The subpanel may be configured using [LCDRasterSubPanelConfigSet\(\)](#) which defines the split point between active image and default color and also determines whether the active image area is above or below the split line. When a subpanel is configured, [LCDRasterFrameBufferSet\(\)](#) can be called with a frame buffer sized for the number of lines in the subpanel rather than the whole screen. The subpanel may be enabled using [LCDRasterSubPanelEnable\(\)](#) and disabled using [LCDRasterSubPanelDisable\(\)](#).

Subpanel Example: 480 Line Display

```
ui32Flags = LCD_SUBPANEL_AT_TOP
ui32BottomLines = 160
ui32DefaultPixel = 0x001F (16bpp 565)
```



320 lines (480 - 160) of frame buffer content at top of display.

160 lines of default color at bottom of display.

19.2.4 Frame Buffer and Palette Formats

When using raster mode, the frame buffer is stored in local SRAM or EPI- connected SDRAM and the application is responsible for ensuring that it is formatted correctly for the LCD controller's current configuration.

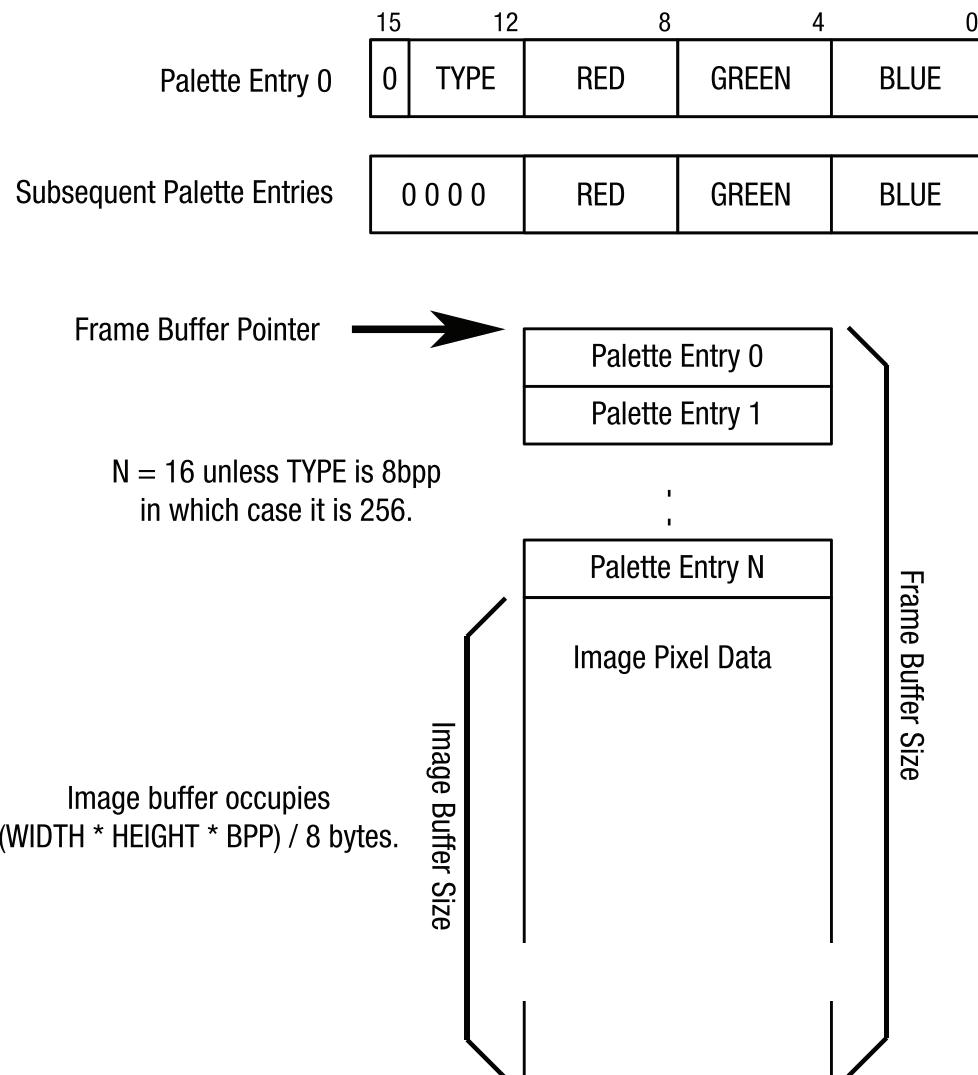
The frame buffer contains two sections:

- A header containing the pixel format identifier and palette (color lookup table). This contains either 16 or 128 16-bit entries.
- An array of pixels comprising the image to display.

The frame buffer header is 8 words (or 16 half-word palette entries) long when displaying any pixel format other than 8 bits-per-pixel in which case it is 128 words (256 half-word palette entries) in length. Note that the header cannot be removed even when using pixel formats which do not require a palette.

Each entry in the header comprises a 16-bit half-word containing a 12-bit RGB444 color in the bottom 12 bits. The top 4 bits of each entry other than the first are reserved and must be set to 0. The top 4 bits of the first entry contains an identifier informing the LCD controller of the color depth (1-, 2-, 4- or 8-bpp palettized, or direct color 12-, 16- or 24-bpp) in use for the following image data.

The frame buffer type identifier may be set by calling [LCDRasterPaletteSet\(\)](#) or by directly writing the first half-word of the frame buffer header with **LCD_PALETTE_TYPE_1BPP**, **LCD_PALETTE_TYPE_2BPP**, **LCD_PALETTE_TYPE_4BPP**, **LCD_PALETTE_TYPE_8BPP** or **LCD_PALETTE_TYPE_DIRECT**.



19.2.5 Data Structure Documentation

19.2.5.1 tLCDIDDTiming

Definition:

```
typedef struct
{
    uint8_t ui8WSSetup;
    uint8_t ui8WSDuration;
    uint8_t ui8WSHold;
    uint8_t ui8RSSetup;
    uint8_t ui8RSDuration;
    uint8_t ui8RSHold;
```

```

        uint8_t ui8DelayCycles;
    }
tLCDIDDTiming

```

Members:

ui8WSSetup Write Strobe Set-Up cycles. When performing a write access, this field defines the number of MCLK cycles that Data Bus/Pad Output Enable, ALE, the Direction bit, and Chip Select have to be ready before the Write Strobe is asserted. Valid values are from 0 to 31.

ui8WSDuration Write Strobe Duration cycles. Field value defines the number of MCLK cycles for which the Write Strobe is held active when performing a write access. Valid values are from 1 to 63.

ui8WSHold Write Strobe Hold cycles. Field value defines the number of MCLK cycles for which Data Bus/Pad Output Enable, ALE, the Direction bit, and Chip Select are held after the Write Strobe is deasserted when performing a write access. Valid values are from 1 to 15.

ui8RSSetup Read Strobe Set-Up cycles. When performing a read access, this field defines the number of MCLK cycles that Data Bus/Pad Output Enable, ALE, the Direction bit, and Chip Select have to be ready before the Read Strobe is asserted. Valid values are from 0 to 31.

ui8RSDuration Read Strobe Duration cycles. Field value defines the number of MCLK cycles for which the Read Strobe is held active when performing a read access. Valid values are from 1 to 63.

ui8RSHold Read Strobe Hold cycles. Field value defines the number of MCLK cycles for which Data Bus/Pad Output Enable, ALE, the Direction bit, and Chip Select are held after the Read Strobe is deasserted when performing a read access. Valid values are from 1 to 15.

ui8DelayCycles Field value defines the number of MCLK cycles between the end of one device access and the start of another device access using the same Chip Select unless the two accesses are both Reads. In this case, this delay is not incurred. Valid values are from 1 to 4.

Description:

A structure containing timing parameters for the LIDD (LCD Interface Display Driver) interface. This is used with the LCDIDDTimingSet function.

19.2.5.2 tLCDRasterTiming

Definition:

```

typedef struct
{
    uint32_t ui32Flags;
    uint16_t ui16PanelWidth;
    uint16_t ui16PanelHeight;
    uint16_t ui16HFrontPorch;
    uint16_t ui16HBackPorch;
    uint16_t ui16HSyncWidth;
    uint8_t ui8VFrontPorch;
    uint8_t ui8VBackPorch;
    uint8_t ui8VSyncWidth;
    uint8_t ui8ACBiasLineCount;
}

```

```
}
```

tLCDRasterTiming

Members:

- ui32Flags*** Flags configuring the polarity and active edges of the various signals in the raster interface. This field is comprised of a logical OR of the labels with prefix “RASTER_TIMING_”.
- ui16PanelWidth*** The number of pixels contained within each line on the LCD display. Valid values are multiple of 16 less than or equal to 2048.
- ui16PanelHeight*** The number of lines on the LCD display. Valid values are from 1 to 2048.
- ui16HFrontPorch*** A value from 1 to 1024 that specifies the number of pixel clock periods to add to the end of each line after active video has ended.
- ui16HBackPorch*** A value from 1 to 1024 that specifies the number of pixel clock periods to add to the beginning of a line before active video is asserted.
- ui16HSyncWidth*** A value from 1 to 1024 that specifies the number of pixel clock periods to pulse the line clock at the end of each line.
- ui8VFrontPorch*** A value from 0 to 255 that specifies the number of line clock periods to add to the end of each frame after the last active line.
- ui8VBackPorch*** A value from 0 to 255 that specifies the number of line clock periods to add to the beginning of a frame before the first active line is output to the display.
- ui8VSyncWidth*** In active mode, a value from 1 to 64 that specifies the number of line clock periods to set the lcd_fp pin active at the end of each frame after the vertical front porch period elapses. The number of The frame clock is used as the VSYNC signal in active mode.
In passive mode, a value from 1 to 64 that specifies the number of extra line clock periods to insert after the vertical front porch period has elapsed. Note that the width of lcd_fp is not affected by this value in passive mode.
- ui8ACBiasLineCount*** A value from 0 to 255 that specifies the number of line clocks to count before transitioning the AC Bias pin. This pin is used to periodically invert the polarity of the power supply to prevent DC charge build-up within the display.

Description:

A structure containing timing parameters for the raster interface. This is used with the LCDRasterTimingSet function.

19.2.6 Define Documentation

19.2.6.1 CYCLES_FROM_TIME_NS

This macro can be used to convert from time in nanoseconds to periods of the supplied clock in Hertz as required when setting up the LIDD and raster timing structures. The calculation will round such that the number of cycles returned represents no longer a time than specified in the ui32Time_nS parameter. Values of ui32Time_nS less than or equal to 35791394 (35.79 milliseconds) are supported by the macro. Larger values will cause arithmetic overflow and yield incorrect values. It is further assumed that ui32ClockFreq is a non-zero multiple of 1000000 (1MHz).

19.2.6.2 #define CYCLES_FROM_TIME_US(ui32ClockFreq, ui32Time_uS)

Definition:

```
#define CYCLES_FROM_TIME_NS(ui32ClockFreq,  
                           ui32Time_nS)
```

Description:

This macro can be used to convert from time in microseconds to periods of the supplied clock in Hertz as required when setting up the LIDD and raster timing structures. The calculation will round such that the number of cycles returned represents no longer a time than specified in the ui32Time_uS parameter. Values of ui32Time_uS less than or equal to 4294967uS (4.29 seconds) are supported by the macro. Larger values will cause arithmetic overflow and yield incorrect values. It is further assumed that ui32ClockFreq is a non-zero multiple of 1000000 (1MHz).

19.2.6.3 #define PAL_FROM_RGB(ui32RGBColor)

This macro can be used to convert a 24-bit RGB color value as used by the TivaWare Graphics Library into a 12-bit LCD controller color palette entry.

19.2.7 Function Documentation

19.2.7.1 LCDClockReset

Resets one or more of the LCD controller clock domains.

Prototype:

```
void  
LCDClockReset(uint32_t ui32Base,  
              uint32_t ui32Clocks)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32Clocks defines the subset of clock domains to be reset.

Description:

This function allows sub-modules of the LCD controller to be reset under software control. The **ui32Clocks** parameter is the logical OR of the following clocks:

- **LCD_CLOCK_MAIN** causes the entire LCD controller module to be reset.
- **LCD_CLOCK_DMA** causes the DMA controller submodule to be reset.
- **LCD_CLOCK_LIDD** causes the LIDD submodule to be reset.
- **LCD_CLOCK_CORE** causes the core module, including the raster logic to be reset.

In all cases, LCD controller register values are preserved across these resets.

Returns:

None.

19.2.7.2 LCDDMAConfigSet

Configures the LCD controller's internal DMA engine.

Prototype:

```
void  
LCDDMAConfigSet(uint32_t ui32Base,  
                  uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the controller.

ui32Config provides flags defining the desired DMA parameters.

Description:

This function is used to configure the DMA engine within the LCD controller. This engine is responsible for performing bulk data transfers to the display when in LIDD mode or for transferring palette and pixel data from SRAM to the display panel when in raster mode.

The *ui32Config* parameter is a logical OR of various flags. It must contain one value from each of the following groups.

The first group of flags set the number of words that have to be in the FIFO before it signals that it is ready:

- **LCD_DMA_FIFORDY_8_WORDS**
- **LCD_DMA_FIFORDY_16_WORDS**
- **LCD_DMA_FIFORDY_32_WORDS**
- **LCD_DMA_FIFORDY_64_WORDS**
- **LCD_DMA_FIFORDY_128_WORDS**
- **LCD_DMA_FIFORDY_256_WORDS**
- **LCD_DMA_FIFORDY_512_WORDS**

The second group of flags set the number of 32-bit words in each DMA burst transfer:

- **LCD_DMA_BURST_1**
- **LCD_DMA_BURST_2**
- **LCD_DMA_BURST_4**
- **LCD_DMA_BURST_8**
- **LCD_DMA_BURST_16**

The final group of flags set internal byte lane controls and allow byte swapping within the DMA engine. The label represents the output byte order for an input 32-bit word ordered "0123".

- **LCD_DMA_BYTE_ORDER_0123**
- **LCD_DMA_BYTE_ORDER_1023**
- **LCD_DMA_BYTE_ORDER_3210**
- **LCD_DMA_BYTE_ORDER_2301**

Additionally, **LCD_DMA_PING_PONG** may be specified. This flag configures the controller to operate in double-buffered mode. When data is scanned out from the first frame buffer, the DMA engine immediately moves to the second frame buffer and scans from there before moving back to the first. If this flag is clear, the DMA engine uses a single frame buffer, restarting the scan from the beginning of the buffer each time it completes a frame.

Note:

DMA burst size **LCD_DMA_BURST_16** should be set when using frame buffers in external, EPI-connected memory. Using a smaller burst size in this case is likely to result in occasional FIFO underflows and associated display glitches.

Returns:

None.

19.2.7.3 LCDIDDCommandWrite

Writes a command to the display when the LCD controller is in LIDD mode.

Prototype:

```
void
LCDIDDCommandWrite(uint32_t ui32Base,
                     uint32_t ui32CS,
                     uint16_t ui16Cmd)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

ui16Cmd is the 16-bit command word to write.

Description:

This function writes a 16-bit command word to the display when the LCD controller is in LIDD mode. A command write occurs with the ALE signal active.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None.

19.2.7.4 LCDIDDCConfigSet

Sets the LCD controller communication parameters when in LIDD mode.

Prototype:

```
void
LCDIDDCConfigSet(uint32_t ui32Base,
                  uint32_t ui32Config)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32Config defines the display interface configuration.

Description:

This function is used when the LCD controller is configured in LIDD mode and specifies the configuration of the interface between the controller and the display panel. The *ui32Config* parameter is comprised of one of the following modes:

- **LIDD_CONFIG_SYNC_MP68** selects Sync MPU68 mode. LCDCP = EN, LCDLP = DIR, LCDFP = ALE, LCDAC = CS0, LCDMCLK = MCLK.
- **LIDD_CONFIG_ASYNC_MP68** selects Async MPU68 mode. LCDCP = EN, LCDLP = DIR, LCDFP = ALE, LCDAC = CS0, LCDMCLK = CS1.
- **LIDD_CONFIG_SYNC_MP80** selects Sync MPU80 mode. LCDCP = RS, LCDLP = WS, LCDFP = ALE, LCDAC = CS0, LCDMCLK = MCLK.
- **LIDD_CONFIG_ASYNC_MP80** selects Async MPU80 mode. LCDCP = RS, LCDLP = WS, LCDFP = ALE, LCDAC = CS0, LCDMCLK = CS1.
- **LIDD_CONFIG_ASYNC_HITACHI** selects Hitachi (async) mode. LCDCP = N/C, LCDLP = DIR, LCDFP = ALE, LCDAC = E0, LCDMCLK = E1.

Additional flags may be ORed into *ui32Config* to control the polarities of various control signals:

- **LIDD_CONFIG_INVERT_ALE** - Address Latch Enable (ALE) polarity control. By default, ALE is active low. If this flag is set, it becomes active high.
- **LIDD_CONFIG_INVERT_RS_EN** - Read Strobe/Enable polarity control. By default, RS is active low and Enable is active high. If this flag is set, RS becomes active high and Enable active low.
- **LIDD_CONFIG_INVERT_WS_DIR** - Write Strobe/Direction polarity control. By default, WS is active low and Direction write low/read high. If this flag is set, WS becomes active high and Direction becomes write high/read low.
- **LIDD_CONFIG_INVERT_CS0** - Chip Select 0/Enable 0 polarity control. By default, CS0 and E0 are active high. If this flag is set, they become active low.
- **LIDD_CONFIG_INVERT_CS1** - Chip Select 1/Enable 1 polarity control. By default, CS1 and E1 are active high. If this flag is set, they become active low.

Returns:

None.

19.2.7.5 LCDIDDDDataRead

Reads a data word from the display when the LCD controller is in LIDD mode.

Prototype:

```
uint16_t
LCDIDDDDataRead(uint32_t ui32Base,
                 uint32_t ui32CS)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

Description:

This function reads the 16-bit data word from the display when the LCD controller is in LIDD mode. A data read occurs with the ALE signal inactive.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

Returns the status word read from the display panel.

19.2.7.6 LCDIDDDDataWrite

Writes a data value to the display when the LCD controller is in LIDD mode.

Prototype:

```
void
LCDIDDDDataWrite(uint32_t ui32Base,
                  uint32_t ui32CS,
                  uint16_t ui16Data)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

ui16Data is the 16-bit data word to write.

Description:

This function writes a 16-bit data word to the display when the LCD controller is in LIDD mode. A data write occurs with the ALE signal inactive.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None.

19.2.7.7 LCDIDDDMADisable

Disables internal DMA operation when the LCD controller is in LIDD mode.

Prototype:

```
void
LCDIDDDMADisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the LCD controller module base address.

Description:

When the LCD controller is operating in LCD Interface Display Driver mode, this function must be called after completion of a DMA transaction and before calling [LCDIDDDCommandWrite\(\)](#), [LCDIDDDDataWrite\(\)](#), [LCDIDDDStatusRead\(\)](#), [LCDIDDDIndexedWrite\(\)](#), [LCDIDDDIndexedRead\(\)](#) or [LCDIDDDDataRead\(\)](#) to disable DMA mode and allow CPU-initiated transactions to the display.

Note:

LIDD DMA mode is enabled automatically when [LCDIDDDDMAWrite\(\)](#) is called.

Returns:

None.

19.2.7.8 LCDIDDDDMAWrite

Writes a block of data to the display using DMA when the LCD controller is in LIDD mode.

Prototype:

```
void
LCDIDDDDMAWrite(uint32_t ui32Base,
                  uint32_t ui32CS,
                  const uint32_t *pui32Data,
                  uint32_t ui32Count)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

pui32Data is the address of the first 16-bit word to write. This address must be aligned on a 32-bit word boundary.

ui32Count is the number of 16-bit words to write. This value must be a multiple of 2.

Description:

This function writes a block of 16-bit data words to the display using DMA. It is only valid when the LCD controller is in LIDD mode. Completion of the DMA transfer is signaled by the [LCD_INT_DMA_DONE](#) interrupt.

This function enables DMA mode prior to starting the transfer. The caller is responsible for ensuring that any earlier DMA transfer has completed before initiating another transfer.

During the time that DMA is enabled, none of the other LCD LIDD data transfer functions may be called. When the DMA transfer is complete and the application wishes to use the CPU to communicate with the display, [LCDIDDDMADisable\(\)](#) must be called to disable DMA access prior to calling [LCDIDDDCommandWrite\(\)](#), [LCDIDDDDataWrite\(\)](#), [LCDIDDDStatusRead\(\)](#), [LCDIDDDIndexedWrite\(\)](#), [LCDIDDDIndexedRead\(\)](#) or [LCDIDDDDataRead\(\)](#).

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None.

19.2.7.9 LCDIDDIIndexedRead

Reads a given display register when the LCD controller is in LIDD mode.

Prototype:

```
uint16_t
LCDIDDIIndexedRead(uint32_t ui32Base,
                    uint32_t ui32CS,
                    uint16_t ui16Addr)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

ui16Addr is the address of the display register to read.

Description:

This function reads a 16-bit word from a register in the display when the LCD controller is in LIDD mode and configured to use either the Motorola (**LIDD_CONFIG_SYNC_MP68** or **LIDD_CONFIG_ASYNC_MP68**) or Intel (**LIDD_CONFIG_SYNC_MP80** or **LIDD_CONFIG_ASYNC_MP80**) modes that employ an external address latch.

When configured in Hitachi mode (**LIDD_CONFIG_ASYNC_HITACHI**), this function should not be used. In this case, the functions [LCDIDDDStatusRead\(\)](#) and [LCDIDDDDataRead\(\)](#) may be used to read status and data bytes from the panel.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None.

19.2.7.10 LCDIDDIIndexedWrite

Writes data to a given display register when the LCD controller is in LIDD mode.

Prototype:

```
void
LCDIDDIIndexedWrite(uint32_t ui32Base,
                     uint32_t ui32CS,
                     uint16_t ui16Addr,
                     uint16_t ui16Data)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

ui16Addr is the address of the display register to write.

ui16Data is the data to write.

Description:

This function writes a 16-bit data word to a register in the display when the LCD controller is in LIDD mode and configured to use either the Motorola (**LIDD_CONFIG_SYNC_MP68** or **LIDD_CONFIG_ASYNC_MP68**) or Intel (**LIDD_CONFIG_SYNC_MP80** or **LIDD_CONFIG_ASYNC_MP80**) modes that employ an external address latch.

When configured in Hitachi mode (**LIDD_CONFIG_ASYNC_HITACHI**), this function should not be used. In this case the functions [LCDIDDDCommandWrite\(\)](#) and [LCDIDDDDataWrite\(\)](#) may be used to transfer command and data bytes to the panel.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None.

19.2.7.11 LCDIDDDStatusRead

Reads a status word from the display when the LCD controller is in LIDD mode.

Prototype:

```
uint16_t  
LCDIDDDStatusRead(uint32_t ui32Base,  
                  uint32_t ui32CS)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select to use. Valid values are 0 and 1.

Description:

This function reads the 16-bit status word from the display when the LCD controller is in LIDD mode. A status read occurs with the ALE signal active. If the interface is configured in Hitachi mode (**LIDD_CONFIG_ASYNC_HITACHI**), this operation corresponds to a command mode read.

This function must not be called if the LIDD interface is currently configured to expect DMA transactions. If DMA was previously used to write to the panel, [LCDIDDDDMADisable\(\)](#) must be called before this function can be used.

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

Returns the status word read from the display panel.

19.2.7.12 LCDIDDTimingSet

Sets the LCD controller interface timing when in LIDD mode.

Prototype:

```
void
LCDIDDTimingSet(uint32_t ui32Base,
                 uint32_t ui32CS,
                 const tLCDIDDTiming *pTiming)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32CS specifies the chip select whose timings are to be set.

pTiming points to a structure containing the desired timing parameters.

Description:

This function is used in LIDD mode to set the setup, strobe and hold times for the various interface control signals. Independent timings are stored for each of the two supported chip selects offered by the LCD controller.

For a definition of the timing parameters required, see the definition of [tLCDIDDTiming](#).

Note:

CS1 is not available when operating in Sync MPU68 or Sync MPU80 modes.

Returns:

None

19.2.7.13 LCDIntClear

Clears LCD controller interrupt sources.

Prototype:

```
void
LCDIntClear(uint32_t ui32Base,
            uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the controller.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified LCD controller interrupt sources are cleared so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The **ui32IntFlags** parameter is the logical OR of any of the following:

- **LCD_INT_DMA_DONE** - indicates that a LIDD DMA transfer is complete.
- **LCD_INT_RASTER_FRAME_DONE** - indicates that a raster-mode frame is complete.
- **LCD_INT_SYNC_LOST** - indicates that frame synchronization was lost.
- **LCD_INT_AC_BIAS_CNT** - indicates that the AC bias transition counter has decremented to zero and is valid for passive matrix panels only. The counter, set by a call to [LCDRasterACBiasIntCountSet\(\)](#), is reloaded but remains disabled until this interrupt is cleared.
- **LCD_INT_UNDERFLOW** - indicates that a data underflow occurred. The internal FIFO was empty when the output logic attempted to read data to send to the display.

- **LCD_INT_PAL_LOAD** - indicates that the color palette has been loaded.
- **LCD_INT_EOF0** - indicates that the raw End-of-Frame 0 has been signaled.
- **LCD_INT_EOF2** - indicates that the raw End-of-Frame 1 has been signaled.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

19.2.7.14 LCDIntDisable

Disables individual LCD controller interrupt sources.

Prototype:

```
void  
LCDIntDisable(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the controller.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated LCD controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **LCD_INT_DMA_DONE** - indicates that a LIDD DMA transfer is complete.
- **LCD_INT_RASTER_FRAME_DONE** - indicates that a raster-mode frame is complete.
- **LCD_INT_SYNC_LOST** - indicates that frame synchronization was lost.
- **LCD_INT_AC_BIAS_CNT** - indicates that that AC bias transition counter has decremented to zero and is valid for passive matrix panels only. The counter, set by a call to [LCDRasterACBiasIntCountSet\(\)](#), is reloaded but remains disabled until this interrupt is cleared.
- **LCD_INT_UNDERFLOW** - indicates that a data underflow occurred. The internal FIFO was empty when the output logic attempted to read data to send to the display.
- **LCD_INT_PAL_LOAD** - indicates that the color palette has been loaded.
- **LCD_INT_EOF0** - indicates that the raw End-of-Frame 0 has been signaled.
- **LCD_INT_EOF2** - indicates that the raw End-of-Frame 1 has been signaled.

Returns:

None.

19.2.7.15 LCDIntEnable

Enables individual LCD controller interrupt sources.

Prototype:

```
void
LCDIntEnable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated LCD controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **LCD_INT_DMA_DONE** - indicates that a LIDD DMA transfer is complete.
- **LCD_INT_RASTER_FRAME_DONE** - indicates that a raster-mode frame is complete.
- **LCD_INT_SYNC_LOST** - indicates that frame synchronization was lost.
- **LCD_INT_AC_BIAS_CNT** - indicates that that AC bias transition counter has decremented to zero and is valid for passive matrix panels only. The counter, set by a call to [LCDRasterACBiasIntCountSet\(\)](#), is reloaded but remains disabled until this interrupt is cleared.
- **LCD_INT_UNDERFLOW** - indicates that a data underflow occurred. The internal FIFO was empty when the output logic attempted to read data to send to the display.
- **LCD_INT_PAL_LOAD** - indicates that the color palette has been loaded.
- **LCD_INT_EOF0** - indicates that the raw End-of-Frame 0 has been signaled.
- **LCD_INT_EOF2** - indicates that the raw End-of-Frame 1 has been signaled.

Returns:

None.

19.2.7.16 LCDIntRegister

Registers an interrupt handler for the LCD controller module.

Prototype:

```
void
LCDIntRegister(uint32_t ui32Base,
                void (*pfnHandler) (void))
```

Parameters:

ui32Base specifies the LCD controller module base address.

pfhHandler is a pointer to the function to be called when the LCD controller interrupt occurs.

Description:

This function registers the handler to be called when the LCD controller module interrupt occurs.

Note:

This function need not be called if the appropriate interrupt vector is statically linked into the vector table in the application startup code.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

19.2.7.17 LCDIntStatus

Gets the current LCD controller interrupt status.

Prototype:

```
uint32_t  
LCDIntStatus(uint32_t ui32Base,  
             bool bMasked)
```

Parameters:

ui32Base is the base address of the controller.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the LCD controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status as the logical OR of any of the following:

- **LCD_INT_DMA_DONE** - indicates that a LIDD DMA transfer is complete.
- **LCD_INT_RASTER_FRAME_DONE** - indicates that a raster-mode frame is complete.
- **LCD_INT_SYNC_LOST** - indicates that frame synchronization was lost.
- **LCD_INT_AC_BIAS_CNT** - indicates that that AC bias transition counter has decremented to zero and is valid for passive matrix panels only. The counter, set by a call to [LCDRasterACBiasIntCountSet\(\)](#), is reloaded but remains disabled until this interrupt is cleared.
- **LCD_INT_UNDERFLOW** - indicates that a data underflow occurred. The internal FIFO was empty when the output logic attempted to read data to send to the display.
- **LCD_INT_PAL_LOAD** - indicates that the color palette has been loaded.
- **LCD_INT_EOF0** - indicates that the raw End-of-Frame 0 has been signaled.
- **LCD_INT_EOF2** - indicates that the raw End-of-Frame 1 has been signaled.

19.2.7.18 LCDIntUnregister

Unregisters the interrupt handler for the LCD controller module.

Prototype:

```
void
LCDIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the LCD controller module base address.

Description:

This function unregisters the interrupt handler and disables the global LCD controller interrupt in the interrupt controller.

Note:

This function need not be called if the appropriate interrupt vector is statically linked into the vector table in the application startup code.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

19.2.7.19 LCDModeSet

Configures the basic operating mode and clock rate for the LCD controller.

Prototype:

```
uint32_t
LCDModeSet(uint32_t ui32Base,
           uint8_t ui8Mode,
           uint32_t ui32PixClk,
           uint32_t ui32SysClk)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui8Mode specifies the basic operating mode to be used.

ui32PixClk specifies the desired LCD controller pixel or master clock rate in Hz.

ui32SysClk specifies the current system clock rate in Hz.

Description:

This function sets the basic operating mode of the LCD controller and also its master clock. The **ui8Mode** parameter may be set to either **LCD_MODE_LIDD** or **LCD_MODE_RASTER**. **LCD_MODE_LIDD** is used to select LCD Interface Display Driver mode for character panels connected via an asynchronous interface (CS, WE, OE, ALE, data) and **LCD_MODE_RASTER** is used to communicate with panels via a synchronous video interface using data and sync signals. Additionally, **LCD_MODE_AUTO_UFLOW_RESTART** may be ORed with either of these modes to indicate that the hardware should restart automatically if a data underflow occurs.

The **ui32PixClk** parameter specifies the desired master clock for the the LCD controller. In LIDD mode, this value controls the MCLK used in communication with the display and valid values are between **ui32SysClk** and **ui32SysClk/255**. In raster mode, **ui32PixClk** specifies the pixel clock rate for the raster interface and valid values are between **ui32SysClk/2** and **ui32SysClk/255**. The actual clock rate set may differ slightly from the desired rate due to the

fact that only integer dividers are supported. The rate set will, however, be no higher than the requested value.

The *ui32SysClk* parameter provides the current system clock rate and is used to allow the LCD controller clock rate divisor to be correctly set to give the desired *ui32PixClk* rate.

Returns:

Returns the actual LCD controller pixel clock or MCLK rate set.

19.2.7.20 LCDRasterACBiasIntCountSet

Sets the number of AC bias pin transitions per interrupt.

Prototype:

```
void  
LCDRasterACBiasIntCountSet (uint32_t ui32Base,  
                           uint8_t ui8Count)
```

Parameters:

ui32Base is the base address of the controller.

ui8Count is the number of AC bias pin transitions to count before the AC bias count interrupt is asserted. Valid values are from 0 to 15.

Description:

This function is used to set the number of AC bias transitions between each AC bias count interrupt (**LCD_INT_AC_BIAS_CNT**). If *ui8Count* is 0, no AC bias count interrupt is generated.

Returns:

None.

19.2.7.21 LCDRasterConfigSet

Sets the LCD controller interface timing when in raster mode.

Prototype:

```
void  
LCDRasterConfigSet (uint32_t ui32Base,  
                     uint32_t ui32Config,  
                     uint8_t ui8PalLoadDelay)
```

Parameters:

ui32Base specifies the LCD controller module base address.

ui32Config specifies properties of the raster interface and the attached display panel.

ui8PalLoadDelay specifies the number of system clocks to wait between each 16 halfword (16-bit) burst when loading the palette from SRAM into the internal palette RAM of the controller.

Description:

This function configures the basic operating mode of the raster interface and specifies the type of panel that the controller is to drive.

The *ui32Config* parameter must be defined as one of the following to select the required target panel type and output pixel format:

- **RASTER_FMT_ACTIVE_24BPP_PACKED** selects an active matrix display and uses a packed 24-bit per pixel packet frame buffer where 4 pixels are described within 3 consecutive 32-bit words.
- **RASTER_FMT_ACTIVE_24BPP_UNPACKED** selects an active matrix display and uses an unpacked 24-bit per pixel packet frame buffer where each 32-bit word contains a single pixel and 8 bits of padding.
- **RASTER_FMT_ACTIVE_16BPP** selects an active matrix display and uses a 16-bit per pixel frame buffer with 2 pixels in each 32-bit word.
- **RASTER_FMT_ACTIVE_PALETTIZED_12BIT** selects an active matrix display and uses a 1, 2, 4 or 8bpp frame buffer with palette lookup. Output color data is described in 12-bit format using bits 11:0 of the data bus. The frame buffer pixel format is defined by the value passed in the *ui32Type* parameter to [LCDRasterPaletteSet\(\)](#).
- **RASTER_FMT_ACTIVE_PALETTIZED_16BIT** selects an active matrix display and uses a 1, 2, 4 or 8bpp frame buffer with palette lookup. Output color data is described in 16-bit 5:6:5 format. The frame buffer pixel format is defined by the value passed in the *ui32Type* parameter to [LCDRasterPaletteSet\(\)](#).
- **RASTER_FMT_PASSIVE_MONO_4PIX** selects a monochrome, passive matrix display that outputs 4 pixels on each pixel clock.
- **RASTER_FMT_PASSIVE_MONO_8PIX** selects a monochrome, passive matrix display that outputs 8 pixels on each pixel clock.
- **RASTER_FMT_PASSIVE_COLOR_12BIT** selects a passive matrix display and uses a 12bpp frame buffer. The palette is bypassed and 12-bit pixel data is sent to the grayscaler for the display.
- **RASTER_FMT_PASSIVE_COLOR_16BIT** selects a passive matrix display and uses a 16bpp frame buffer with pixels in 5:6:5 format. Only the 4 most significant bits of each color component are sent to the grayscaler for the display.

Additionally, the following flags may be ORed into *ui32Config*:

- **RASTER_ACTVID_DURING_BLANK** sets Actvid to toggle during vertical blanking.
- **RASTER_NIBBLE_MODE_ENABLED** enables nibble mode. This parameter works with **RASTER_READ_ORDER_REVERSED** to determine how 1, 2 and 4bpp pixels are extracted from words read from the frame buffer. If specified, words read from the frame buffer are byte swapped prior to individual pixels being parsed from them.
- **RASTER_LOAD_DATA_ONLY** tells the controller to read only pixel data from the frame buffer and to use the last palette read. No palette load is performed.
- **RASTER_LOAD_PALETTE_ONLY** tells the controller to read only the palette data from the frame buffer.
- **RASTER_READ_ORDER_REVERSED** when using 1, 2, 4 and 8bpp frame buffers, this option reverses the order in which frame buffer words are parsed. When this option is specified, the leftmost pixel in a word is taken from the most significant bits. When absent, the leftmost pixel is parsed from the least significant bits.

If the LCD controller's raster engine is enabled when this function is called, it is disabled as a result of the call.

Returns:

None.

19.2.7.22 LCDRasterDisable

Disables the raster output.

Prototype:

```
void  
LCDRasterDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function disables the LCD controller raster output and stops driving the attached display.

Note:

Once disabled, the raster engine continues to scan data until the end of the current frame. If the display is to be re-enabled, wait until after the final **LCD_INT_RASTER_FRAME_DONE** has been received, indicating that the raster engine has stopped.

Returns:

None.

19.2.7.23 LCDRasterEnable

Enables the raster output.

Prototype:

```
void  
LCDRasterEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function enables the LCD controller raster output and starts displaying the content of the current frame buffer on the attached panel. Prior to enabling the raster output, [LCDModeSet\(\)](#), [LCDRasterConfigSet\(\)](#), [LCDDMAConfigSet\(\)](#), [LCDRasterTimingSet\(\)](#), [LCDRasterPaletteSet\(\)](#) and [LCDRasterFrameBufferSet\(\)](#) must have been called.

Returns:

None.

19.2.7.24 LCDRasterEnabled

Determines whether or not the raster output is currently enabled.

Prototype:

```
bool  
LCDRasterEnabled(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function may be used to query whether or not the raster output is currently enabled.

Returns:

Returns **true** if the raster is enabled or **false** if it is disabled.

19.2.7.25 LCDRasterFrameBufferSet

Sets the LCD controller frame buffer start address and size in raster mode.

Prototype:

```
void
LCDRasterFrameBufferSet(uint32_t ui32Base,
                        uint8_t ui8Buffer,
                        uint32_t *pui32Addr,
                        uint32_t ui32NumBytes)
```

Parameters:

ui32Base is the base address of the controller.

ui8Buffer specifies which frame buffer to configure. Valid values are 0 and 1.

pui32Addr points to the first byte of the frame buffer. This pointer must be aligned on a 32-bit (word) boundary.

ui32NumBytes specifies the size of the frame buffer in bytes. This value must be a multiple of 4.

Description:

This function is used to configure the position and size of one of the two supported frame buffers while in raster mode. The second frame buffer (configured when **ui8Buffer** is set to 1) is only used if the controller is set to operate in ping-pong mode (by specifying the **LCD_DMA_PING_PONG** configuration flag on a call to [LCDDMAConfigSet\(\)](#)).

The format of the frame buffer depends on the image type in use and the current raster configuration settings. If **RASTER_LOAD_DATA_ONLY** was specified in a previous call to [LCDRasterConfigSet\(\)](#), the frame buffer contains only packed pixel data in the required bit depth and format. In other cases, the frame buffer comprises a palette of either 8 or 128 32-bit words followed by the packed pixel data. The palette size is 8 words (16 16-bit entries) for all pixel formats other than 8bpp which uses a palette of 128 words (256 16-bit entries). Note that the 8 word palette is still present even for 12, 16 and 24-bit formats, which do not use the lookup table.

The frame buffer size, specified using the **ui32NumBytes** parameter, must be the palette size (if any) plus the size of the image bitmap required for the currently configured display resolution.

$$ui32NumBytes = (\text{Palette Size}) + ((\text{Width} * \text{Height}) * \text{BPP}) / 8$$

If **RASTER_LOAD_DATA_ONLY** is not specified, frame buffers passed to this function must be initialized using a call to [LCDRasterPaletteSet\(\)](#) prior to enabling the raster output. If this is not done, the pixel format identifier and color table required by the hardware is not present and the results are unpredictable.

Returns:

None.

19.2.7.26 LCDRasterPaletteSet

Initializes the color palette in a frame buffer.

Prototype:

```
void
LCDRasterPaletteSet(uint32_t ui32Base,
```

```
    uint32_t ui32Type,
    uint32_t *pui32Addr,
    const uint32_t *pui32SrcColors,
    uint32_t ui32Start,
    uint32_t ui32Count)
```

Parameters:

ui32Base is the base address of the controller.

ui32Type specifies the type of pixel data to be held in the frame buffer and also the format of the source color values passed.

pui32Addr points to the start of the frame buffer into which the palette information is to be written.

pui32SrcColors points to the first color value that is to be written into the frame buffer palette.

ui32Start specifies the index of the first color in the palette to update.

ui32Count specifies the number of source colors to be copied into the frame buffer palette.

Description:

This function is used to initialize the color palette stored at the beginning of a frame buffer. It writes the relevant pixel type into the first entry of the frame buffer and copies the requested number of colors from a source buffer into the palette starting at the required index, optionally converting them from 24-bit color format into the 12-bit format used by the LCD controller.

ui32Type must be set to one of the following values to indicate the type of frame buffer for which the palette is being initialized:

- **LCD_PALETTE_TYPE_1BPP** indicates a 1 bit per pixel (monochrome) frame buffer. This format requires a 2 entry palette.
- **LCD_PALETTE_TYPE_2BPP** indicates a 2 bit per pixel frame buffer. This format requires a 4 entry palette.
- **LCD_PALETTE_TYPE_4BPP** indicates a 4 bit per pixel frame buffer. This format requires a 4 entry palette.
- **LCD_PALETTE_TYPE_8BPP** indicates an 8 bit per pixel frame buffer. This format requires a 256 entry palette.
- **LCD_PALETTE_TYPE_DIRECT** indicates a direct color (12, 16 or 24 bit per pixel). The color palette is not used in these modes, but the frame buffer type must still be initialized to ensure that the hardware uses the correct pixel type. When this value is used, the format of the pixels in the frame buffer is defined by the *ui32Config* parameter previously passed to [LCDRasterConfigSet\(\)](#).

Optionally, the **LCD_PALETTE_SRC_24BIT** flag may be ORed into *ui32Type* to indicate that the supplied colors in the *pui32SrcColors* array are in the 24-bit format as used by the TivaWare Graphics Library with one color stored in each 32-bit word. In this case, the colors read from the source array are converted to the 12-bit format used by the LCD controller before being written into the frame buffer palette.

If **LCD_PALETTE_SRC_24BIT** is not present, it is assumed that the *pui32SrcColors* array contains 12-bit colors in the format required by the LCD controller with 2 colors stored in each 32-bit word. In this case, the values are copied directly into the frame buffer palette without any reformatting.

Returns:

None.

19.2.7.27 LCDRasterSubPanelConfigSet

Sets the position and size of the subpanel on the raster display.

Prototype:

```
void
LCDRasterSubPanelConfigSet(uint32_t ui32Base,
                            uint32_t ui32Flags,
                            uint32_t ui32BottomLines,
                            uint32_t ui32DefaultPixel)
```

Parameters:

ui32Base is the base address of the controller.

ui32Flags may be either **LCD_SUBPANEL_AT_TOP** to show frame buffer image data in the top portion of the display and default color in the bottom portion, or **LCD_SUBPANEL_AT_BOTTOM** to show image data at the bottom of the display and default color at the top.

ui32BottomLines defines the number of lines comprising the bottom portion of the display. If **LCD_SUBPANEL_AT_TOP** is set in **ui32Flags**, these lines contain the default pixel color when the subpanel is enabled, otherwise they contain image data.

ui32DefaultPixel is the 24-bit RGB color to show in the portion of the display not configured to show image data.

Description:

The LCD controller provides a feature that allows a portion of the display to be filled with a default color rather than image data from the frame buffer. This feature reduces SRAM bandwidth requirements because no data is fetched for lines containing the default color. This feature is only available when the LCD controller is in raster mode and configured to drive an active matrix display.

The subpanel area containing image data from the frame buffer may be positioned either at the top or bottom of the display as controlled by the value of **ui32Flags**. The height of the bottom portion of the display is defined by **ui32BottomLines**.

When a subpanel is configured, the application must also reconfigure the frame buffer to ensure that it contains the correct number of lines for the subpanel size in use. This configuration can be achieved by calling [LCDRasterFrameBufferSet\(\)](#) with the **ui32NumBytes** parameter set appropriately to describe the required number of active video lines in the subpanel area.

The subpanel display mode is not enabled using this function. To enable the subpanel once it has been configured, call [LCDRasterSubPanelEnable\(\)](#).

Returns:

None.

19.2.7.28 LCDRasterSubPanelDisable

Disables subpanel display mode.

Prototype:

```
void
LCDRasterSubPanelDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function disables subpanel display mode and reverts to showing the entire frame buffer image on the display. After the subpanel is disabled, the frame buffer size must be reconfigured to match the full dimensions of the display area by calling [LCDRasterFrameBufferSet\(\)](#) with an appropriate value for the *ui32NumBytes* parameter.

Returns:

None.

19.2.7.29 LCDRasterSubPanelEnable

Enables subpanel display mode.

Prototype:

```
void  
LCDRasterSubPanelEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the controller.

Description:

This function enables subpanel display mode and displays a default color rather than image data in the number of lines and at the position specified by a previous call to [LCDRasterSubPanelConfigSet\(\)](#). Prior to calling [LCDRasterSubPanelEnable\(\)](#), the frame buffer should have been reconfigured to match the desired subpanel size using a call to [LCDRasterFrameBufferSet\(\)](#).

Subpanel display is only possible when the LCD controller is in raster mode and is configured to drive an active matrix display.

Returns:

None.

19.2.7.30 LCDRasterTimingSet

Sets the LCD controller interface timing when in raster mode.

Prototype:

```
void  
LCDRasterTimingSet(uint32_t ui32Base,  
                   const tLCDRasterTiming *pTiming)
```

Parameters:

ui32Base specifies the LCD controller module base address.

pTiming points to a structure containing the desired timing parameters.

Description:

This function is used in raster mode to set the panel size and sync timing parameters.

For a definition of the timing parameters required, see the definition of [tLCDRasterTiming](#).

Returns:

None

19.3 Programming Example

The following example shows how to use the LCD Controller API to configure an 800x480 resolution raster panel and start it displaying from an 8bpp frame buffer. Note that raster timing configuration varies from display to display. Consult your display's datasheet and modify [tLCDRasterTiming](#) values as required for your display.

```
/*
// Define the frame buffer dimensions and format.
//
// Define labels containing the size of the image bitmap and palette.
//
#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 480
#define SCREEN_BPP 8

// Define labels containing the size of the image bitmap and palette.
//
#define SIZE_IMAGE ((SCREEN_WIDTH * SCREEN_HEIGHT * SCREEN_BPP) / 8)
#define SIZE_PALETTE ((SCREEN_BPP == 8) ? (256 * 2) : (16 * 2))

// Position the frame buffer at an appropriate point in memory, most likely in
// EPI-connected SDRAM at 0x10000000. The frame buffer will be (SIZE_IMAGE +
// SIZE_PALETTE) bytes long.
//
uint32_t *g_pui32DisplayBuffer = (uint32_t *)LCD_FRAME_BUFFER_ADDR;

// Calculate pointers to the frame buffer palette and the first byte of the
// actual image bitmap.
//
uint16_t *g_pui16Palette = (uint16_t *)LCD_FRAME_BUFFER_ADDR;
uint8_t *g_pBitmap = (uint8_t *)(LCD_FRAME_BUFFER_ADDR + SIZE_PALETTE);

// Initialize our source color palette. These colors are defined in RGB888
// format as used by the TivaWare Graphics Library.
//
const uint32_t g_pulSrcPalette[256] =
{
    ClrBlack,
    ClrWhite,
    ClrRed,
    ClrLightGreen,
    ClrBlue,
    ClrYellow,
    ClrMagenta,
    ClrCyan,
```

```
    ClrOrange,
    //
    // and so on...
    //
};

//*****
// Labels defining the desired pixel clock, PLL VCO frequency and system clock.
// Note that the system clock must be an integer multiple of the pixel clock.
//
//*****
#define PIXEL_CLOCK_FREQ 3000000
#define SYSTEM_VCO_FREQ  SYSCTL_CFG_VCO_480
#define SYSTEM_CLOCK_FREQ 120000000

//*****
// The timings and signal polarities for the various raster interface signals.
// Timing parameters are defined in terms of pixel clocks (for horizontal
// timings) and lines (for vertical timings).
//
//*****
tLCDRasterTiming g_sRasterTimings =
{
    (RASTER_TIMING_ACTIVE_HIGH_PIXCLK |
     RASTER_TIMING_SYNCS_ON_RISING_PIXCLK),
    SCREEN_WIDTH, SCREEN_HEIGHT,
    2, 30, 8,
    10, 10, 8,
    0
};

//*****
// The interrupt handler for the LCD controller. This function merely
// counts the interrupts received.
//
//*****
void
LCDIntHandler(void)
{
    uint32_t ui32Status;

    //
    // Get the current interrupt status and clear any active interrupts.
    //
    ui32Status = LCDIntStatus(LCD0_BASE, true);
    LCDIntClear(LCD0_BASE, ui32Status);

    //
    // Handle any interrupts as required by the application. In normal
    // operation, no action is required at interrupt time to keep the raster
    // scan running.
    //
}

//*****
// Initialize the LCD controller to drive the display in raster mode and
// enable the raster engine.
//
//*****
void
InitDisplay(uint32_t ui32SysClkHsz, uint32_t ui32PixClock,
            tLCDRasterTiming *psTimings)
```

```

{
    uint32_t ui32Loop;

    //
    // Enable the LCD controller peripheral.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_LCD0);

    //
    // Wait to ensure that the LCD controller is enabled.
    //
    SysCtlDelay(2);

    //
    // Configure the LCD controller for raster operation with a pixel clock
    // as close to the requested pixel clock as possible.
    //
    ui32PixClock = LCDModeSet(LCD0_BASE, (LCD_MODE_RASTER |
                                            LCD_MODE_AUTO_UFLOW_RESTART),
                               ui32PixClock, ui32SysClkHz);

    //
    // Set the output format for the raster interface.
    //
    LCDRasterConfigSet(LCD0_BASE, RASTER_FMT_ACTIVE_PALETTIZED_16BIT, 0);

    //
    // Program the raster engine timings according to the display requirements.
    //
    LCDRasterTimingSet(LCD0_BASE, psTimings);

    //
    // Configure LCD DMA-related parameters.
    //
    LCDDMAConfigSet(LCD0_BASE, LCD_DMA_BURST_4);

    //
    // Set up the frame buffer.
    //
    LCDRasterFrameBufferSet(LCD0_BASE, 0, g_pui32DisplayBuffer,
                           SIZE_PALETTE + SIZE_IMAGE);

    //
    // Write the palette to the frame buffer.
    //
    LCDRasterPaletteSet(LCD0_BASE,
                        LCD_PALETTE_SRC_24BIT | LCD_PALETTE_TYPE_8BPP,
                        (uint32_t *)g_pui16Palette, g_pulSrcPalette, 0,
                        (SIZE_PALETTE / 2));

    //
    // Fill the frame buffer with black (pixel value 0 corresponds to black
    // in the palette we just set).
    //
    for(ui32Loop = 0; ui32Loop < (SIZE_IMAGE / sizeof(uint32_t)); ui32Loop++)
    {
        g_pui32DisplayBuffer[ui32Loop] = 0;
    }

    //
    // Enable the LCD interrupts.
    //
    LCDIntEnable(LCD0_BASE, (LCD_INT_DMA_DONE | LCD_INT_RASTER_FRAME_DONE |
                            LCD_INT_SYNC_LOST | LCD_INT_AC_BIAS_CNT | LCD_INT_UNDERFLOW |
                            LCD_INT_PAL_LOAD | LCD_INT_EOF0 | LCD_INT_EOF1));

```

```
    IntEnable(INT_LCDO);

    //
    // Enable the raster output.
    //
    LCDRasterEnable(LCD0_BASE);
}

//*****
// This example demonstrates the use of the LCD Controller in raster mode.
//
//*****
int
main(void)
{
    uint32_t ui32SysClock;

    //
    // Set the system clock to run from the PLL at 120 MHz.
    //
    ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN |
                                         SYSCTL_USE_PLL |
                                         SYSTEM_VCO_FREQ), SYSTEM_CLOCK_FREQ);

    //
    // Configure the device pins.
    //
    PinoutSet();

    //
    // Enable interrupts in the CPU.
    //
    IntMasterEnable();

    //
    // Initialize the display controller and start the raster engine.
    //
    InitDisplay(ui32SysClock, PIXEL_CLOCK_FREQ, &g_sRasterTimings);

    while(1)
    {
        //
        // Other application code...
        //
    }
}
```

20 Memory Protection Unit (MPU)

Introduction	397
API Functions	397
Programming Example	404

20.1 Introduction

The Memory Protection Unit (MPU) API provides functions to configure the MPU. The MPU is tightly coupled to the Cortex-M processor core and provides a means to establish access permissions on regions of memory.

Up to eight memory regions can be defined. Each region has a base address and a size. The size is specified as a power of 2 between 32 bytes and 4 GB, inclusive. The region's base address must be aligned to the size of the region. Each region also has access permissions. Code execution can be allowed or disallowed for a region. A region can be configured for read-only access, read/write access, or no access for both privileged and user modes. Access permissions can be used to create an environment where only kernel or system code can access certain hardware registers or sections of code.

The MPU creates 8 sub-regions within each region. Any sub-region or combination of sub-regions can be disabled, allowing creation of “holes” or complex overlaying regions with different permissions. The sub-regions can also be used to create an unaligned beginning or ending of a region by disabling one or more of the leading or trailing sub-regions.

Once the regions are defined and the MPU is enabled, any access violation of a region causes a memory management fault, and the fault handler is activated.

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

20.2 API Functions

Functions

- void `MPUDisable` (void)
- void `MPUEnable` (uint32_t ui32MPUConfig)
- void `MPUIntRegister` (void (*pfnHandler)(void))
- void `MPUIntUnregister` (void)
- uint32_t `MPURegionCountGet` (void)
- void `MPURegionDisable` (uint32_t ui32Region)
- void `MPURegionEnable` (uint32_t ui32Region)
- void `MPURegionGet` (uint32_t ui32Region, uint32_t *pui32Addr, uint32_t *pui32Flags)
- void `MPURegionSet` (uint32_t ui32Region, uint32_t ui32Addr, uint32_t ui32Flags)

20.2.1 Detailed Description

The MPU APIs provide a means to enable and configure the MPU and memory protection regions.

Generally, the memory protection regions should be defined before enabling the MPU. The regions can be configured by calling [MPURegionSet\(\)](#) once for each region to be configured.

A region that is defined by [MPURegionSet\(\)](#) can be initially enabled or disabled. If the region is not initially enabled, it can be enabled later by calling [MPURegionEnable\(\)](#). An enabled region can be disabled by calling [MPURegionDisable\(\)](#). When a region is disabled, its configuration is preserved as long as it is not overwritten. In this case, it can be enabled again with [MPURegionEnable\(\)](#) without the need to reconfigure the region.

Care must be taken when setting up a protection region using [MPURegionSet\(\)](#). The function writes to multiple registers and is not protected from interrupts. Therefore, it is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to protect against this is to make sure that a region is always disabled before making any changes. Otherwise, it is up to the caller to ensure that [MPURegionSet\(\)](#) is always called from within code that cannot be interrupted, or from code that is not affected if an interrupt occurs while the region attributes are being changed.

The attributes of a region that have already been programmed can be retrieved and saved using the [MPURegionGet\(\)](#) function. This function is intended to save the attributes in a format that can be used later to reload the region using the [MPURegionSet\(\)](#) function. Note that the enable state of the region is saved with the attributes and takes effect when the region is reloaded.

When one or more regions are defined, the MPU can be enabled by calling [MPUEnable\(\)](#). This function turns on the MPU and also defines the behavior in privileged mode and in the Hard Fault and NMI fault handlers. The MPU can be configured so that when in privileged mode and no regions are enabled, a default memory map is applied. If this feature is not enabled, then a memory management fault is generated if the MPU is enabled and no regions are configured and enabled. The MPU can also be set to use a default memory map when in the Hard Fault or NMI handlers, instead of using the configured regions. All of these features are selected when calling [MPUEnable\(\)](#). When the MPU is enabled, it can be disabled by calling [MPUDisable\(\)](#).

Finally, if the application is using run-time interrupt registration (see [IntRegister\(\)](#)), then the function [MPUIntRegister\(\)](#) can be used to install the fault handler which is called whenever a memory protection violation occurs. This function also enables the fault handler. If compile-time interrupt registration is used, then the [IntEnable\(\)](#) function with the parameter **FAULT_MPUMPU** must be used to enable the memory management fault handler. When the memory management fault handler has been installed with [MPUIntRegister\(\)](#), it can be removed by calling [MPUIntUnregister\(\)](#).

20.2.2 Function Documentation

20.2.2.1 MPUDisable

Disables the MPU for use.

Prototype:

```
void  
MPUDisable(void)
```

Description:

This function disables the Cortex-M memory protection unit. When the MPU is disabled, the

default memory map is used and memory management faults are not generated.

Returns:

None.

20.2.2.2 MPUEnable

Enables and configures the MPU for use.

Prototype:

```
void
MPUEnable(uint32_t ui32MPUConfig)
```

Parameters:

ui32MPUConfig is the logical OR of the possible configurations.

Description:

This function enables the Cortex-M memory protection unit. It also configures the default behavior when in privileged mode and while handling a hard fault or NMI. Prior to enabling the MPU, at least one region must be set by calling [MPURegionSet\(\)](#) or else by enabling the default region for privileged mode by passing the **MPU_CONFIG_PRIV_DEFAULT** flag to [MPUEnable\(\)](#). Once the MPU is enabled, a memory management fault is generated for memory access violations.

The *ui32MPUConfig* parameter should be the logical OR of any of the following:

- **MPU_CONFIG_PRIV_DEFAULT** enables the default memory map when in privileged mode and when no other regions are defined. If this option is not enabled, then there must be at least one valid region already defined when the MPU is enabled.
- **MPU_CONFIG_HARDFLT_NMI** enables the MPU while in a hard fault or NMI exception handler. If this option is not enabled, then the MPU is disabled while in one of these exception handlers and the default memory map is applied.
- **MPU_CONFIG_NONE** chooses none of the above options. In this case, no default memory map is provided in privileged mode, and the MPU is not enabled in the fault handlers.

Returns:

None.

20.2.2.3 MPUIntRegister

Registers an interrupt handler for the memory management fault.

Prototype:

```
void
MPUIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the memory management fault occurs.

Description:

This function sets and enables the handler to be called when the MPU generates a memory management fault due to a protection region access violation.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

20.2.2.4 MPUIntUnregister

Unregisters an interrupt handler for the memory management fault.

Prototype:

```
void  
MPUIntUnregister(void)
```

Description:

This function disables and clears the handler to be called when a memory management fault occurs.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

20.2.2.5 MPURegionCountGet

Gets the count of regions supported by the MPU.

Prototype:

```
uint32_t  
MPURegionCountGet(void)
```

Description:

This function is used to get the total number of regions that are supported by the MPU, including regions that are already programmed.

Returns:

The number of memory protection regions that are available for programming using [MPURegionSet\(\)](#).

20.2.2.6 MPURegionDisable

Disables a specific region.

Prototype:

```
void  
MPURegionDisable(uint32_t ui32Region)
```

Parameters:

ui32Region is the region number to disable.

Description:

This function is used to disable a previously enabled memory protection region. The region remains configured if it is not overwritten with another call to [MPURegionSet\(\)](#), and can be enabled again by calling [MPURegionEnable\(\)](#).

Returns:

None.

20.2.2.7 MPURegionEnable

Enables a specific region.

Prototype:

```
void  
MPURegionEnable(uint32_t ui32Region)
```

Parameters:

ui32Region is the region number to enable.

Description:

This function is used to enable a memory protection region. The region should already be configured with the [MPURegionSet\(\)](#) function. Once enabled, the memory protection rules of the region are applied and access violations cause a memory management fault.

Returns:

None.

20.2.2.8 MPURegionGet

Gets the current settings for a specific region.

Prototype:

```
void  
MPURegionGet(uint32_t ui32Region,  
              uint32_t *pui32Addr,  
              uint32_t *pui32Flags)
```

Parameters:

ui32Region is the region number to get.

pui32Addr points to storage for the base address of the region.

pui32Flags points to the attribute flags for the region.

Description:

This function retrieves the configuration of a specific region. The meanings and format of the parameters is the same as that of the [MPURRegionSet\(\)](#) function.

This function can be used to save the configuration of a region for later use with the [MPURRegionSet\(\)](#) function. The region's enable state is preserved in the attributes that are saved.

Returns:

None.

20.2.2.9 MPURRegionSet

Sets up the access rules for a specific region.

Prototype:

```
void  
MPURRegionSet(uint32_t ui32Region,  
              uint32_t ui32Addr,  
              uint32_t ui32Flags)
```

Parameters:

ui32Region is the region number to set up.

ui32Addr is the base address of the region. It must be aligned according to the size of the region specified in *ui32Flags*.

ui32Flags is a set of flags to define the attributes of the region.

Description:

This function sets up the protection rules for a region. The region has a base address and a set of attributes including the size. The base address parameter, *ui32Addr*, must be aligned according to the size, and the size must be a power of 2.

The *ui32Flags* parameter is the logical OR of all of the attributes of the region. It is a combination of choices for region size, execute permission, read/write permissions, disabled sub-regions, and a flag to determine if the region is enabled.

The size flag determines the size of a region and must be one of the following:

- **MPU_RGN_SIZE_32B**
- **MPU_RGN_SIZE_64B**
- **MPU_RGN_SIZE_128B**
- **MPU_RGN_SIZE_256B**
- **MPU_RGN_SIZE_512B**
- **MPU_RGN_SIZE_1K**
- **MPU_RGN_SIZE_2K**
- **MPU_RGN_SIZE_4K**
- **MPU_RGN_SIZE_8K**
- **MPU_RGN_SIZE_16K**
- **MPU_RGN_SIZE_32K**
- **MPU_RGN_SIZE_64K**
- **MPU_RGN_SIZE_128K**
- **MPU_RGN_SIZE_256K**

- **MPU_RGN_SIZE_512K**
- **MPU_RGN_SIZE_1M**
- **MPU_RGN_SIZE_2M**
- **MPU_RGN_SIZE_4M**
- **MPU_RGN_SIZE_8M**
- **MPU_RGN_SIZE_16M**
- **MPU_RGN_SIZE_32M**
- **MPU_RGN_SIZE_64M**
- **MPU_RGN_SIZE_128M**
- **MPU_RGN_SIZE_256M**
- **MPU_RGN_SIZE_512M**
- **MPU_RGN_SIZE_1G**
- **MPU_RGN_SIZE_2G**
- **MPU_RGN_SIZE_4G**

The execute permission flag must be one of the following:

- **MPU_RGN_PERM_EXEC** enables the region for execution of code
- **MPU_RGN_PERM_NOEXEC** disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

- **MPU_RGN_PERM_PRV_NO_USR_NO** - no access in privileged or user mode
- **MPU_RGN_PERM_PRV_RW_USR_NO** - privileged read/write, user no access
- **MPU_RGN_PERM_PRV_RW_USR_RO** - privileged read/write, user read-only
- **MPU_RGN_PERM_PRV_RW_USR_RW** - privileged read/write, user read/write
- **MPU_RGN_PERM_PRV_RO_USR_NO** - privileged read-only, user no access
- **MPU_RGN_PERM_PRV_RO_USR_RO** - privileged read-only, user read-only

The region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled, allowing for creation of “holes” in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

- **MPU_SUB_RGN_DISABLE_0**
- **MPU_SUB_RGN_DISABLE_1**
- **MPU_SUB_RGN_DISABLE_2**
- **MPU_SUB_RGN_DISABLE_3**
- **MPU_SUB_RGN_DISABLE_4**
- **MPU_SUB_RGN_DISABLE_5**
- **MPU_SUB_RGN_DISABLE_6**
- **MPU_SUB_RGN_DISABLE_7**

Finally, the region can be initially enabled or disabled with one of the following flags:

- **MPU_RGN_ENABLE**
- **MPU_RGN_DISABLE**

As an example, to set a region with the following attributes: size of 32 KB, execution enabled, read-only for both privileged and user, one sub-region disabled, and initially enabled; the *ui32Flags* parameter would have the following value:

```
(MPU_RGN_SIZE_32K | MPU_RGN_PERM_EXEC | MPU_RGN_PERM_PRV_RO_USR_RO |
MPU_SUB_RGN_DISABLE_2 | MPU_RGN_ENABLE)
```

Note:

This function writes to multiple registers and is not protected from interrupts. It is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to handle this is to disable a region before changing it. Refer to the discussion of this in the API Detailed Description section.

Returns:

None.

20.3 Programming Example

The following example sets up a basic set of protection regions to provide the following:

- a 28-KB region in flash for read-only code execution
- 32 KB of RAM for read-write access in privileged and user modes
- an additional 8 KB of RAM for use only in privileged mode
- 1 MB of peripheral space for access only in privileged mode, except for a 128-KB hole that is not accessible at all, and another 128-KB region that is accessible from user mode

```
//
// Define a 28-KB region of flash from 0x00000000 to 0x00007000. The
// region is executable, and read-only for both privileged and user
// modes. To set up the region, a 32-KB region (#0) is defined
// starting at address 0, and then a 4 KB hole removed at the end by
// disabling the last sub-region. The region is initially enabled.
//
MPURegionSet(0, 0,
              MPU_RGN_SIZE_32K |
              MPU_RGN_PERM_EXEC |
              MPU_RGN_PERM_PRV_RO_USR_RO |
              MPU_SUB_RGN_DISABLE_7 |
              MPU_RGN_ENABLE);

//
// Define a 32-KB region (#1) of RAM from 0x20000000 to 0x20008000. The
// region is not executable, and is read/write access for
// privileged and user modes.
//
MPURegionSet(1, 0x20000000,
              MPU_RGN_SIZE_32K |
              MPU_RGN_PERM_NOEXEC |
              MPU_RGN_PERM_PRV_RW_USR_RW |
              MPU_RGN_ENABLE);

//
// Define an additional 8-KB region (#2) in RAM from 0x20008000 to
// 0x2000A000 that is read/write accessible only from privileged
// mode. This region is initially disabled, to be enabled later.
//
```

```

MPURegionSet(2, 0x20008000,
              MPU_RGN_SIZE_8K |
              MPU_RGN_PERM_NOEXEC |
              MPU_RGN_PERM_PRV_RW_USR_NO |
              MPU_RGN_DISABLE);

//
// Define a region (#3) in peripheral space from 0x40000000 to 0x40100000
// (1 MB). This region is accessible only in privileged mode. There is
// an area from 0x40020000 to 0x40040000 that has no peripherals and is not
// accessible at all. This inaccessible region is created by disabling the
// second sub-region(1) and creating a hole. Further, there is an area
// from 0x40080000 to 0x400A0000 that should be accessible from user mode
// as well. This area is created by disabling the fifth sub-region (4),
// and overlaying an additional region (#4) in that space with the
// appropriate permissions.
//
MPURegionSet(3, 0x40000000,
              MPU_RGN_SIZE_1M |
              MPU_RGN_PERM_NOEXEC |
              MPU_RGN_PERM_PRV_RW_USR_NO |
              MPU_SUB_RGN_DISABLE_1 | MPU_SUB_RGN_DISABLE_4 |
              MPU_RGN_ENABLE);
MPURegionSet(4, 0x40080000,
              MPU_RGN_SIZE_128K |
              MPU_RGN_PERM_NOEXEC |
              MPU_RGN_PERM_PRV_RW_USR_RW |
              MPU_RGN_ENABLE);

//
// In this example, compile-time registration of interrupts is used, so the
// handler does not have to be registered. However, it must be enabled.
//
IntEnable(FAULT_MPU);

//
// When setting up the regions, region 2 was initially disabled for some
// reason. At some point it must be enabled.
//
MPURegionEnable(2);

//
// Now the MPU is enabled. It is configured so that a default
// map is available in privileged mode if no regions are defined. The MPU
// is not enabled for the hard fault and NMI handlers, meaning that a
// default is not used whenever these handlers are active, effectively
// giving the fault handlers access to all of memory without any
// protection.
//
MPUEnable(MPU_CONFIG_PRIV_DEFAULT);

//
// At this point, the MPU is configured and enabled and if any code causes
// an access violation, the memory management fault occurs.
//

```

The following example shows how to save and restore region configurations.

```

//
// The following arrays provide space for saving the address and
// attributes for 4 region configurations.
//
uint32_t ui32RegionAddr[4];
uint32_t ui32RegionAttr[4];

```

```
...  
  
//  
// At some point in the system code, we want to save the state of 4 regions  
// (0-3).  
//  
for(ui8Idx = 0; ui8Idx < 4; ui8Idx++)  
{  
    MPURegionGet(ui8Idx, &ui32RegionAddr[ui8Idx], &ui32RegionAttr[ui8Idx]);  
}  
  
...  
  
//  
// At some other point, the previously saved regions should be restored.  
//  
for(ui8Idx = 0; ui8Idx < 4; ui8Idx++)  
{  
    MPURegionSet(ui8Idx, ui32RegionAddr[ui8Idx], ui32RegionAttr[ui8Idx]);  
}
```

21 1-Wire Master Module

Introduction	407
API Functions	407
Programming Example	415

21.1 Introduction

The 1-Wire API provides functions to use the 1-Wire Master module in the Tiva microcontroller.

The 1-Wire specification defines a bi-directional serial communication protocol that provides both power and data over a single wire. The 1-Wire Master module can interface with one or more slave devices. Typical slave devices include thermometers, mixed-signal devices, memory, and authentication devices.

Some features of the 1-Wire Master module include:

- Support for standard and overdrive speeds, including a late-sample mechanism
- Data size transfers of 1, 2, 3, or 4 bytes with sub-byte support
- Interrupt capability for transaction pacing and line error

This driver is contained in `driverlib/onewire.c`, with `driverlib/onewire.h` containing the API declarations for use by applications.

21.2 API Functions

Functions

- void `OneWireBusReset` (uint32_t ui32Base)
- uint32_t `OneWireBusStatus` (uint32_t ui32Base)
- void `OneWireDataGet` (uint32_t ui32Base, uint32_t *pui32Data)
- bool `OneWireDataGetNonBlocking` (uint32_t ui32Base, uint32_t *pui32Data)
- void `OneWireDMADisable` (uint32_t ui32Base, uint32_t ui32DMAFlags)
- void `OneWireDMAEnable` (uint32_t ui32Base, uint32_t ui32DMAFlags)
- void `OneWireInit` (uint32_t ui32Base, uint32_t ui32InitFlags)
- void `OneWireIntClear` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `OneWireIntDisable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `OneWireIntEnable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `OneWireIntRegister` (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t `OneWireIntStatus` (uint32_t ui32Base, bool bMasked)
- void `OneWireIntUnregister` (uint32_t ui32Base)
- void `OneWireTransaction` (uint32_t ui32Base, uint32_t ui32OpMode, uint32_t ui32Data, uint32_t ui32BitCnt)

21.2.1 Function Documentation

21.2.1.1 OneWireBusReset

Issues a reset on the 1-Wire bus.

Prototype:

```
void  
OneWireBusReset (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

Description:

This function causes the 1-Wire module to generate a reset signal on the 1-Wire bus.

Returns:

None.

21.2.1.2 OneWireBusStatus

Retrieves the 1-Wire bus condition status.

Prototype:

```
uint32_t  
OneWireBusStatus (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

Description:

This function returns the 1-Wire bus conditions reported by the 1-Wire module. These conditions could be a logical OR of any of the following:

- **ONEWIRE_BUS_STATUS_BUSY** - A read, write, or reset is active.
- **ONEWIRE_BUS_STATUS_NO_SLAVE** - No slave presence pulses detected.
- **ONEWIRE_BUS_STATUS_STUCK** - The bus is being held low by non-master.

Returns:

Returns the 1-Wire bus conditions if detected else zero.

21.2.1.3 OneWireDataGet

Retrieves data from the 1-Wire interface.

Prototype:

```
void  
OneWireDataGet (uint32_t ui32Base,  
                uint32_t *pui32Data)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.
pui32Data is a pointer to storage to hold the read data.

Description:

This function reads data from the 1-Wire module once all active bus operations are completed. By protocol definition, bit data defaults to a 1. Thus if a slave did not signal any 0-bit data, this read returns 0xffffffff.

Returns:

None.

21.2.1.4 OneWireDataGetNonBlocking

Retrieves data from the 1-Wire interface.

Prototype:

```
bool
OneWireDataGetNonBlocking(uint32_t ui32Base,
                          uint32_t *pui32Data)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.
pui32Data is a pointer to storage to hold the read data.

Description:

This function reads data from the 1-Wire module if there are no active operations on the bus. Otherwise it returns without reading the data from the module.

By protocol definition, bit data defaults to a 1. Thus if a slave did not signal any 0-bit data, this read returns 0xffffffff.

Returns:

Returns **true** if a data read was performed, or **false** if the bus was not idle and no data was read.

21.2.1.5 OneWireDMADisable

Disables 1-Wire DMA operations.

Prototype:

```
void
OneWireDMADisable(uint32_t ui32Base,
                  uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the 1-Wire module.
ui32DMAFlags is a bit mask of the DMA features to disable.

Description:

This function is used to disable 1-Wire DMA features that were enabled by [OneWireDMAEnable\(\)](#). The specified 1-Wire DMA features are disabled. The **ui32DMAFlags** parameter is a combination of the following:

- **ONEWIRE_DMA_BUS_RESET** - Issue a 1-Wire bus reset before starting
- **ONEWIRE_DMA_OP_READ** - Read after each module transaction
- **ONEWIRE_DMA_OP_MULTI_WRITE** - Write after each previous write
- **ONEWIRE_DMA_OP_MULTI_READ** - Read after each previous read
- **ONEWIRE_DMA_MODE_SG** - Start DMA on enable then repeat on each completion
- **ONEWIRE_DMA_OP_SZ_8** - Bus read/write of 8 bits
- **ONEWIRE_DMA_OP_SZ_16** - Bus read/write of 16 bits
- **ONEWIRE_DMA_OP_SZ_32** - Bus read/write of 32 bits

Returns:

None.

21.2.1.6 OneWireDMAEnable

Enables 1-Wire DMA operations.

Prototype:

```
void  
OneWireDMAEnable(uint32_t ui32Base,  
                  uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the 1-Wire module.

ui32DMAFlags is a bit mask of the DMA features to enable.

Description:

This function enables the specified 1-Wire DMA features. The 1-Wire module can be configured for write operations, read operations, small write and read operations, and scatter-gather support of mixed operations.

The **ui32DMAFlags** parameter is a combination of the following:

- **ONEWIRE_DMA_BUS_RESET** - Issue a 1-Wire bus reset before starting
- **ONEWIRE_DMA_OP_READ** - Read after each module transaction
- **ONEWIRE_DMA_OP_MULTI_WRITE** - Write after each previous write
- **ONEWIRE_DMA_OP_MULTI_READ** - Read after each previous read
- **ONEWIRE_DMA_MODE_SG** - Start DMA on enable then repeat on each completion
- **ONEWIRE_DMA_OP_SZ_8** - Bus read/write of 8 bits
- **ONEWIRE_DMA_OP_SZ_16** - Bus read/write of 16 bits
- **ONEWIRE_DMA_OP_SZ_32** - Bus read/write of 32 bits

Note:

The uDMA controller must be properly configured before DMA can be used with the 1-Wire module.

Returns:

None.

21.2.1.7 OneWireInit

Initializes the 1-Wire module.

Prototype:

```
void
OneWireInit(uint32_t ui32Base,
            uint32_t ui32InitFlags)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

ui32InitFlags provides the initialization flags.

Description:

This function configures and initializes the 1-Wire interface for use.

The *ui32InitFlags* parameter is a combination of the following:

- **ONEWIRE_INIT_SPD_STD** - standard speed bus timings
- **ONEWIRE_INIT_SPD_OD** - overdrive speed bus timings
- **ONEWIRE_INIT_READ_STD** - standard read sampling timing
- **ONEWIRE_INIT_READ_LATE** - late read sampling timing
- **ONEWIRE_INIT_ATR** - standard answer-to-reset presence detect
- **ONEWIRE_INIT_NO_ATR** - no answer-to-reset presence detect
- **ONEWIRE_INIT_STD_POL** - normal signal polarity
- **ONEWIRE_INIT_ALT_POL** - alternate (reverse) signal polarity
- **ONEWIRE_INIT_1_WIRE_CFG** - standard 1-Wire (1 data pin) setup
- **ONEWIRE_INIT_2_WIRE_CFG** - alternate 2-Wire (2 data pin) setup

Returns:

None.

21.2.1.8 OneWireIntClear

Clears the 1-Wire module interrupt sources.

Prototype:

```
void
OneWireIntClear(uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

This function clears the specified 1-Wire interrupt sources so that they no longer assert. This function must be called in the interrupt handler to keep the interrupts from being triggered again immediately upon exit. The *ui32IntFlags* parameter can be a logical OR of any of the following:

- **ONEWIRE_INT_RESET_DONE** - Bus reset has just completed.

- **ONEWIRE_INT_OP_DONE** - Read or write operation completed. If a combined write and read operation was set up, the interrupt signals the read is done.
- **ONEWIRE_INT_NO_SLAVE** - No presence detect was signaled by a slave.
- **ONEWIRE_INT_STUCK** - Bus is being held low by non-master.
- **ONEWIRE_INT_DMA_DONE** - DMA operation has completed.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

21.2.1.9 OneWireIntDisable

Disables individual 1-Wire module interrupt sources.

Prototype:

```
void  
OneWireIntDisable(uint32_t ui32Base,  
                  uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

ui32IntFlags is a bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated 1-Wire interrupt sources. The *ui32IntFlags* parameter can be a logical OR of any of the following:

- **ONEWIRE_INT_RESET_DONE** - Bus reset has just completed.
- **ONEWIRE_INT_OP_DONE** - Read or write operation completed. If a combined write and read operation was set up, the interrupt signals the read is done.
- **ONEWIRE_INT_NO_SLAVE** - No presence detect was signaled by a slave.
- **ONEWIRE_INT_STUCK** - Bus is being held low by non-master.
- **ONEWIRE_INT_DMA_DONE** - DMA operation has completed

Returns:

None.

21.2.1.10 OneWireIntEnable

Enables individual 1-Wire module interrupt sources.

Prototype:

```
void
OneWireIntEnable(uint32_t ui32Base,
                 uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

ui32IntFlags is a bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated 1-Wire interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The *ui32IntFlags* parameter can be a logical OR of any of the following:

- **ONEWIRE_INT_RESET_DONE** - Bus reset has just completed.
- **ONEWIRE_INT_OP_DONE** - Read or write operation completed. If a combined write and read operation was set up, the interrupt signals the read is done.
- **ONEWIRE_INT_NO_SLAVE** - No presence detect was signaled by a slave.
- **ONEWIRE_INT_STUCK** - Bus is being held low by non-master.
- **ONEWIRE_INT_DMA_DONE** - DMA operation has completed

Returns:

None.

21.2.1.11 OneWireIntRegister

Registers an interrupt handler for the 1-Wire module.

Prototype:

```
void
OneWireIntRegister(uint32_t ui32Base,
                   void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the 1-Wire module.

pfnHandler is a pointer to the function to be called when the 1-Wire interrupt occurs.

Description:

This function sets the handler to be called when a 1-Wire interrupt occurs. This function enables the global interrupt in the interrupt controller; specific 1-Wire interrupts must be enabled via [OneWireIntEnable\(\)](#). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via [OneWireIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

21.2.1.12 OneWireIntStatus

Gets the current 1-Wire interrupt status.

Prototype:

```
uint32_t  
OneWireIntStatus(uint32_t ui32Base,  
                 bool bMasked)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

bMasked is **false** if the raw interrupt status is required or **true** if the masked interrupt status is required.

Description:

This function returns the interrupt status for the 1-Wire module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the masked or raw 1-Wire interrupt status, as a bit field of any of the following values:

- **ONEWIRE_INT_RESET_DONE** - Bus reset has just completed.
- **ONEWIRE_INT_OP_DONE** - Read or write operation completed.
- **ONEWIRE_INT_NO_SLAVE** - No presence detect was signaled by a slave.
- **ONEWIRE_INT_STUCK** - Bus is being held low by non-master.
- **ONEWIRE_INT_DMA_DONE** - DMA operation has completed

21.2.1.13 OneWireIntUnregister

Unregisters an interrupt handler for the 1-Wire module.

Prototype:

```
void  
OneWireIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the 1-Wire module.

Description:

This function clears the handler to be called when an 1-Wire interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

21.2.1.14 OneWireTransaction

Performs a 1-Wire protocol transaction on the bus.

Prototype:

```
void
OneWireTransaction(uint32_t ui32Base,
                   uint32_t ui32OpMode,
                   uint32_t ui32Data,
                   uint32_t ui32BitCnt)
```

Parameters:

ui32Base specifies the base address of the 1-Wire module.

ui32OpMode sets the transaction type.

ui32Data is the data for a write operation.

ui32BitCnt specifies the number of valid bits (1-32) for the operation.

Description:

This function performs a 1-Wire protocol transaction, read and/or write, on the bus. The application should confirm the bus is idle before starting a read or write transaction.

The *ui32OpMode* defines the activity for the bus operations and is a logical OR of the following:

- **ONEWIRE_OP_RESET** - Indicates the operation should be started with a bus reset.
- **ONEWIRE_OP_WRITE** - A write operation
- **ONEWIRE_OP_READ** - A read operation

Note:

If both a read and write operation are requested, the write will be performed prior to the read.

Returns:

None.

21.3 Programming Example

The following example sets up the 1-Wire Master module and sends a command sequence over a single slave device connection. A connection with multiple slave devices would require enumeration and selection of an individual device. This example assumes that the interrupt handler was allocated statically in the vector table and the GPIOs are properly configured.

```
//
// Enable the 1-Wire peripheral
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_ONEWIRE0);

//
// Initialize the module to use standard speed and read distance.
//
OneWireInit(ONEWIRE0_BASE, (ONEWIRE_INIT_READ_STD | ONEWIRE_INIT_SPD_STD));

//
// Clear any pending interrupts.
//
```

```
OneWireIntClear(ONEWIRE0_BASE, (ONEWIRE_INT_STUCK | ONEWIRE_INT_NO_SLAVE |
                                ONEWIRE_INT_RESET_DONE |
                                ONEWIRE_INT_OP_DONE));

//
// Enable the interrupts for operation done and the stuck bus error
// condition.
//
OneWireIntEnable(ONEWIRE0_BASE, (ONEWIRE_INT_OP_DONE |
                                 ONEWIRE_INT_BUS_STUCK));
IntEnable(INT_ONEWIRE0);

//
// Setup a four byte sequence and a bit count.
//
ulData = ((0x01 << 24) | (0x12 << 16) | (0x10 << 8) | 0xcc;
ulBitCount = sizeof(ulData) * 8;

//
// Initiate a Reset and Write operation on the 1-Wire bus.
//
OneWireTransaction(ONEWIRE0_BASE, (ONEWIRE_OP_RESET | ONEWIRE_OP_WRITE),
                   ulData, ulBitCount);
```

22 Pulse Width Modulator (PWM)

Introduction	417
API Functions	417
Programming Example	439

22.1 Introduction

Each instance of a Tiva PWM module provides up to four instances of a PWM generator block, and an output control block. Each generator block has two PWM output signals, which can be operated independently or as a pair of signals with dead band delays inserted. Each generator block also has an interrupt output and a trigger output. The control block determines the polarity of the PWM signals and which signals are passed through to the pins.

Some of the features of the Tiva PWM module are:

- Up to four generator blocks, each containing
 - One 16-bit down or up/down counter
 - Two comparators
 - PWM generator
 - Dead band generator
- Control block
 - PWM output enable
 - Output polarity control
 - Synchronization
 - Fault handling
 - Interrupt status

This driver is contained in `driverlib/pwm.c`, with `driverlib/pwm.h` containing the API declarations for use by applications.

22.2 API Functions

Functions

- `uint32_t PWMClockGet (uint32_t ui32Base)`
- `void PWMClockSet (uint32_t ui32Base, uint32_t ui32Config)`
- `void PWMDeadBandDisable (uint32_t ui32Base, uint32_t ui32Gen)`
- `void PWMDeadBandEnable (uint32_t ui32Base, uint32_t ui32Gen, uint16_t ui16Rise, uint16_t ui16Fall)`
- `void PWMFaultIntClear (uint32_t ui32Base)`
- `void PWMFaultIntClearExt (uint32_t ui32Base, uint32_t ui32FaultInts)`
- `void PWMFaultIntRegister (uint32_t ui32Base, void (*pfnIntHandler)(void))`
- `void PWMFaultIntUnregister (uint32_t ui32Base)`

- void **PWMGenConfigure** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Config)
- void **PWMGenDisable** (uint32_t ui32Base, uint32_t ui32Gen)
- void **PWMGenEnable** (uint32_t ui32Base, uint32_t ui32Gen)
- void **PWMGenFaultClear** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group, uint32_t ui32FaultTriggers)
- void **PWMGenFaultConfigure** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32MinFaultPeriod, uint32_t ui32FaultSenses)
- uint32_t **PWMGenFaultStatus** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group)
- uint32_t **PWMGenFaultTriggerGet** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group)
- void **PWMGenFaultTriggerSet** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Group, uint32_t ui32FaultTriggers)
- void **PWMGenIntClear** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Ints)
- void **PWMGenIntRegister** (uint32_t ui32Base, uint32_t ui32Gen, void (*pfnIntHandler)(void))
- uint32_t **PWMGenIntStatus** (uint32_t ui32Base, uint32_t ui32Gen, bool bMasked)
- void **PWMGenIntTrigDisable** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32IntTrig)
- void **PWMGenIntTrigEnable** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32IntTrig)
- void **PWMGenIntUnregister** (uint32_t ui32Base, uint32_t ui32Gen)
- uint32_t **PWMGenPeriodGet** (uint32_t ui32Base, uint32_t ui32Gen)
- void **PWMGenPeriodSet** (uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Period)
- void **PWMIntDisable** (uint32_t ui32Base, uint32_t ui32GenFault)
- void **PWMIntEnable** (uint32_t ui32Base, uint32_t ui32GenFault)
- uint32_t **PWMIntStatus** (uint32_t ui32Base, bool bMasked)
- void **PWMOutputFault** (uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bFaultSuppress)
- void **PWMOutputFaultLevel** (uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bDriveHigh)
- void **PWMOutputInvert** (uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bInvert)
- void **PWMOutputState** (uint32_t ui32Base, uint32_t ui32PWMOutBits, bool bEnable)
- void **PWMOutputUpdateMode** (uint32_t ui32Base, uint32_t ui32PWMOutBits, uint32_t ui32Mode)
- uint32_t **PWMPulseWidthGet** (uint32_t ui32Base, uint32_t ui32PWMOut)
- void **PWMPulseWidthSet** (uint32_t ui32Base, uint32_t ui32PWMOut, uint32_t ui32Width)
- void **PWMSyncTimeBase** (uint32_t ui32Base, uint32_t ui32GenBits)
- void **PWMSyncUpdate** (uint32_t ui32Base, uint32_t ui32GenBits)

22.2.1 Detailed Description

These functions perform high-level operations on PWM modules.

The following functions provide the user with a way to configure the PWM for the most common operations, such as setting the period, generating left- and center-aligned pulses, modifying the pulse width, and controlling interrupts, triggers, and output characteristics. However, the PWM module is very versatile and can be configured in a number of different ways, many of which are beyond the scope of this API. In order to fully exploit the many features of the PWM module, users are advised to use register access macros.

When discussing the various components of a PWM module, this API uses the following labeling convention:

- The generator blocks are called **Gen0**, **Gen1**, **Gen2** and **Gen3**.

- The two PWM output signals associated with each generator block are called **OutA** and **OutB**.
- The output signals are called **PWM0**, **PWM1**, **PWM2**, **PWM3**, **PWM4**, **PWM5**, **PWM6** and **PWM7**.
- **PWM0** and **PWM1** are associated with **Gen0**, **PWM2** and **PWM3** are associated with **Gen1**, **PWM4** and **PWM5** are associated with **Gen2** and **PWM6** and **PWM7** are associated with **Gen3**.

Also, as a simplifying assumption for this API, comparator A for each generator block is used exclusively to adjust the pulse width of the even numbered PWM outputs (**PWM0**, **PWM2**, **PWM4** and **PWM6**). In addition, comparator B is used exclusively for the odd numbered PWM outputs (**PWM1**, **PWM3**, **PWM5** and **PWM7**).

Note that the number of generators and PWM outputs supported varies depending upon the Tiva part in use. Please consult the datasheet for the part you are using to determine whether it supports 1 or 2 modules with 3 or 4 generators each and 6 or 8 outputs each.

22.2.2 Function Documentation

22.2.2.1 PWMClockGet

Gets the current PWM clock configuration.

Prototype:

```
uint32_t
PWMClockGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the PWM module.

Description:

This function returns the current PWM clock configuration.

Note:

This function should not be used with TM4C123 devices. For TM4C123 devices, the [SysCtlP-WMClockGet\(\)](#) function should be used.

Returns:

Returns the current PWM clock configuration; is one of **PWM_SYSCLK_DIV_1**, **PWM_SYSCLK_DIV_2**, **PWM_SYSCLK_DIV_4**, **PWM_SYSCLK_DIV_8**, **PWM_SYSCLK_DIV_16**, **PWM_SYSCLK_DIV_32**, or **PWM_SYSCLK_DIV_64**.

22.2.2.2 PWMClockSet

Sets the PWM clock configuration.

Prototype:

```
void
PWMClockSet(uint32_t ui32Base,
            uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Config is the configuration for the PWM clock; it must be one of **PWM_SYSCLK_DIV_1**, **PWM_SYSCLK_DIV_2**, **PWM_SYSCLK_DIV_4**, **PWM_SYSCLK_DIV_8**, **PWM_SYSCLK_DIV_16**, **PWM_SYSCLK_DIV_32**, or **PWM_SYSCLK_DIV_64**.

Description:

This function sets the PWM clock divider as the PWM clock source. It also configures the clock frequency to the PWM module as a division of the system clock. This clock is used by the PWM module to generate PWM signals; its rate forms the basis for all PWM signals.

Note:

This function should not be used with TM4C123 devices. For TM4C123 devices, the [SysCtlP-WMClockGet\(\)](#) function should be used.

The clocking of the PWM is dependent upon the system clock rate as configured by [SysCtl-ClockFreqSet\(\)](#).

Returns:

None.

22.2.2.3 PWMDeadBandDisable

Disables the PWM dead band output.

Prototype:

```
void  
PWMDeadBandDisable(uint32_t ui32Base,  
                    uint32_t ui32Gen)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to modify. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function disables the dead band mode for the specified PWM generator. Doing so decouples the **OutA** and **OutB** signals.

Returns:

None.

22.2.2.4 PWMDeadBandEnable

Enables the PWM dead band output and sets the dead band delays.

Prototype:

```
void  
PWMDeadBandEnable(uint32_t ui32Base,  
                   uint32_t ui32Gen,  
                   uint16_t ui16Rise,  
                   uint16_t ui16Fall)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to modify. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui16Rise specifies the width of delay from the rising edge.

ui16Fall specifies the width of delay from the falling edge.

Description:

This function sets the dead bands for the specified PWM generator, where the dead bands are defined as the number of PWM clock ticks from the rising or falling edge of the generator's **OutA** signal. Note that this function causes the coupling of **OutB** to **OutA**.

Returns:

None.

22.2.2.5 PWMFaultIntClear

Clears the fault interrupt for a PWM module.

Prototype:

```
void
PWMFaultIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the PWM module.

Description:

This function clears the fault interrupt by writing to the appropriate bit of the interrupt status register for the selected PWM module.

This function clears only the FAULT0 interrupt and is retained for backwards compatibility. It is recommended that [PWMFaultIntClearExt\(\)](#) be used instead because it supports all fault interrupts supported on devices with and without extended PWM fault handling support.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

22.2.2.6 PWMFaultIntClearExt

Clears the fault interrupt for a PWM module.

Prototype:

```
void  
PWMFaultIntClearExt(uint32_t ui32Base,  
                      uint32_t ui32FaultInts)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32FaultInts specifies the fault interrupts to clear.

Description:

This function clears one or more fault interrupts by writing to the appropriate bit of the PWM interrupt status register. The parameter **ui32FaultInts** must be the logical OR of any of **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

The fault interrupts are derived by performing a logical OR of each of the configured fault trigger signals for a given generator. Therefore, these interrupts are not directly related to the four possible FAULTn inputs to the device but indicate that a fault has been signaled to one of the four possible PWM generators.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

22.2.2.7 PWMFaultIntRegister

Registers an interrupt handler for a fault condition detected in a PWM module.

Prototype:

```
void  
PWMFaultIntRegister(uint32_t ui32Base,  
                     void (*pfnIntHandler)(void))
```

Parameters:

ui32Base is the base address of the PWM module.

pfIntHandler is a pointer to the function to be called when the PWM fault interrupt occurs.

Description:

This function ensures that the interrupt handler specified by **pfIntHandler** is called when a fault interrupt is detected for the selected PWM module. This function also enables the PWM fault interrupt in the NVIC; the PWM fault interrupt must also be enabled at the module level using [PWMIIntEnable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

22.2.2.8 PWMFaultIntUnregister

Removes the PWM fault condition interrupt handler.

Prototype:

```
void
PWMFaultIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the PWM module.

Description:

This function removes the interrupt handler for a PWM fault interrupt from the selected PWM module. This function also disables the PWM fault interrupt in the NVIC; the PWM fault interrupt must also be disabled at the module level using [PWMIntDisable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

22.2.2.9 PWMGenConfigure

Configures a PWM generator.

Prototype:

```
void
PWMGenConfigure(uint32_t ui32Base,
                 uint32_t ui32Gen,
                 uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to configure. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Config is the configuration for the PWM generator.

Description:

This function is used to set the mode of operation for a PWM generator. The counting mode, synchronization mode, and debug behavior are all configured. After configuration, the generator is left in the disabled state.

A PWM generator can count in two different modes: count down mode or count up/down mode. In count down mode, it counts from a value down to zero, and then resets to the preset value, producing left-aligned PWM signals (that is, the rising edge of the two PWM signals produced by the generator occur at the same time). In count up/down mode, it counts up from zero to the preset value, counts back down to zero, and then repeats the process, producing center-aligned PWM signals (that is, the middle of the high/low period of the PWM signals produced by the generator occurs at the same time).

When the PWM generator parameters (period and pulse width) are modified, their effect on the output PWM signals can be delayed. In synchronous mode, the parameter updates are

not applied until a synchronization event occurs. This mode allows multiple parameters to be modified and take effect simultaneously, instead of one at a time. Additionally, parameters to multiple PWM generators in synchronous mode can be updated simultaneously, allowing them to be treated as if they were a unified generator. In non-synchronous mode, the parameter updates are not delayed until a synchronization event. In either mode, the parameter updates only occur when the counter is at zero to help prevent oddly formed PWM signals during the update (that is, a PWM pulse that is too short or too long).

The PWM generator can either pause or continue running when the processor is stopped via the debugger. If configured to pause, it continues to count until it reaches zero, at which point it pauses until the processor is restarted. If configured to continue running, it keeps counting as if nothing had happened.

The *ui32Config* parameter contains the desired configuration. It is the logical OR of the following:

- **PWM_GEN_MODE_DOWN** or **PWM_GEN_MODE_UP_DOWN** to specify the counting mode
- **PWM_GEN_MODE_SYNC** or **PWM_GEN_MODE_NO_SYNC** to specify the counter load and comparator update synchronization mode
- **PWM_GEN_MODE_DBG_RUN** or **PWM_GEN_MODE_DBG_STOP** to specify the debug behavior
- **PWM_GEN_MODE_GEN_NO_SYNC**, **PWM_GEN_MODE_GEN_SYNC_LOCAL**, or **PWM_GEN_MODE_GEN_SYNC_GLOBAL** to specify the update synchronization mode for generator counting mode changes
- **PWM_GEN_MODE_DB_NO_SYNC**, **PWM_GEN_MODE_DB_SYNC_LOCAL**, or **PWM_GEN_MODE_DB_SYNC_GLOBAL** to specify the deadband parameter synchronization mode
- **PWM_GEN_MODE_FAULT_LATCHED** or **PWM_GEN_MODE_FAULT_UNLATCHED** to specify whether fault conditions are latched or not
- **PWM_GEN_MODE_FAULT_MINPER** or **PWM_GEN_MODE_FAULT_NO_MINPER** to specify whether minimum fault period support is required
- **PWM_GEN_MODE_FAULT_EXT** or **PWM_GEN_MODE_FAULT_LEGACY** to specify whether extended fault source selection support is enabled or not

Setting **PWM_GEN_MODE_FAULT_MINPER** allows an application to set the minimum duration of a PWM fault signal. Faults are signaled for at least this time even if the external fault pin deasserts earlier. Care should be taken when using this mode because during the fault signal period, the fault interrupt from the PWM generator remains asserted. The fault interrupt handler may, therefore, reenter immediately if it exits prior to expiration of the fault timer.

Note:

Changes to the counter mode affect the period of the PWM signals produced. [PWMPulseWidthSet\(\)](#) and [PWMPulseWidthSet\(\)](#) should be called after any changes to the counter mode of a generator.

Returns:

None.

22.2.2.10 PWMPulseWidthSet

Disables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGGenDisable(uint32_t ui32Base,
                uint32_t ui32Gen)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to be disabled. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function blocks the PWM clock from driving the timer/counter for the specified generator block.

Returns:

None.

22.2.2.11 PWMGGenEnable

Enables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGGenEnable(uint32_t ui32Base,
               uint32_t ui32Gen)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to be enabled. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function allows the PWM clock to drive the timer/counter for the specified generator block.

Returns:

None.

22.2.2.12 PWMGGenFaultClear

Clears one or more latched fault triggers for a given PWM generator.

Prototype:

```
void
PWMGGenFaultClear(uint32_t ui32Base,
                   uint32_t ui32Gen,
                   uint32_t ui32Group,
                   uint32_t ui32FaultTriggers)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator for which fault trigger states are being queried. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Group indicates the subset of faults that are being queried. This parameter must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

ui32FaultTriggers is the set of fault triggers which are to be cleared.

Description:

This function allows an application to clear the fault triggers for a given PWM generator. This function is only required if [PWMGenConfigure\(\)](#) has previously been called with flag **PWM_GEN_MODEFAULT_LATCHED** in parameter *ui32Config*.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

22.2.2.13 PWMGenFaultConfigure

Configures the minimum fault period and fault pin senses for a given PWM generator.

Prototype:

```
void  
PWMGenFaultConfigure(uint32_t ui32Base,  
                      uint32_t ui32Gen,  
                      uint32_t ui32MinFaultPeriod,  
                      uint32_t ui32FaultSenses)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator for which fault configuration is being set. This function must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32MinFaultPeriod is the minimum fault active period expressed in PWM clock cycles.

ui32FaultSenses indicates which sense of each FAULT input should be considered the “asserted” state. Valid values are logical OR combinations of **PWM_FAULTn_SENSE_HIGH** and **PWM_FAULTn_SENSE_LOW**.

Description:

This function configures the minimum fault period for a given generator along with the sense of each of the 4 possible fault inputs. The minimum fault period is expressed in PWM clock cycles and takes effect only if [PWMGenConfigure\(\)](#) is called with flag **PWM_GEN_MODEFAULT_PER** set in the *ui32Config* parameter. When a fault input is asserted, the minimum fault period timer ensures that it remains asserted for at least the number of clock cycles specified.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

22.2.2.14 PWMGenFaultStatus

Returns the current state of the fault triggers for a given PWM generator.

Prototype:

```
uint32_t
PWMGenFaultStatus(uint32_t ui32Base,
                  uint32_t ui32Gen,
                  uint32_t ui32Group)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator for which fault trigger states are being queried. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Group indicates the subset of faults that are being queried. This parameter must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

Description:

This function allows an application to query the current state of each of the fault trigger inputs to a given PWM generator. The current state of each fault trigger input is returned unless [PWMGenConfigure\(\)](#) has previously been called with flag **PWM_GEN_MODEFAULT_LATCHED** in the *ui32Config* parameter, in which case the returned status is the latched fault trigger status.

If latched faults are configured, the application must call [PWMGenFaultClear\(\)](#) to clear each trigger.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

Returns the current state of the fault triggers for the given PWM generator. A set bit indicates that the associated trigger is active. For **PWM_FAULT_GROUP_0**, the returned value is a logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**. For **PWM_FAULT_GROUP_1**, the return value is the logical OR of **PWM_FAULT_DCMP0**, **PWM_FAULT_DCMP1**, **PWM_FAULT_DCMP2**, **PWM_FAULT_DCMP3**, **PWM_FAULT_DCMP4**, **PWM_FAULT_DCMP5**, **PWM_FAULT_DCMP6**, or **PWM_FAULT_DCMP7**.

22.2.2.15 PWMGenFaultTriggerGet

Returns the set of fault triggers currently configured for a given PWM generator.

Prototype:

```
uint32_t
PWMGenFaultTriggerGet(uint32_t ui32Base,
                      uint32_t ui32Gen,
                      uint32_t ui32Group)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator for which fault triggers are being queried. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Group indicates the subset of faults that are being queried. This parameter must be **PWMFAULT_GROUP_0** or **PWMFAULT_GROUP_1**.

Description:

This function allows an application to query the current set of inputs that contribute to the generation of a fault condition to a given PWM generator.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

Returns the current fault triggers configured for the fault group provided. For **PWMFAULT_GROUP_0**, the returned value is a logical OR of **PWMFAULT_FAULT0**, **PWMFAULT_FAULT1**, **PWMFAULT_FAULT2**, or **PWMFAULT_FAULT3**. For **PWMFAULT_GROUP_1**, the return value is the logical OR of **PWMFAULT_DCMP0**, **PWMFAULT_DCMP1**, **PWMFAULT_DCMP2**, **PWMFAULT_DCMP3**, **PWMFAULT_DCMP4**, **PWMFAULT_DCMP5**, **PWMFAULT_DCMP6**, or **PWMFAULT_DCMP7**.

22.2.2.16 PWMGenFaultTriggerSet

Configures the set of fault triggers for a given PWM generator.

Prototype:

```
void  
PWMGenFaultTriggerSet(uint32_t ui32Base,  
                      uint32_t ui32Gen,  
                      uint32_t ui32Group,  
                      uint32_t ui32FaultTriggers)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator for which fault triggers are being set. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Group indicates the subset of possible faults that are to be configured. This parameter must be **PWMFAULT_GROUP_0** or **PWMFAULT_GROUP_1**.

ui32FaultTriggers defines the set of inputs that are to contribute towards generation of the fault signal to the given PWM generator. For **PWMFAULT_GROUP_0**, this is the logical OR of **PWMFAULT_FAULT0**, **PWMFAULT_FAULT1**, **PWMFAULT_FAULT2**, or **PWMFAULT_FAULT3**. For **PWMFAULT_GROUP_1**, this is the logical OR of **PWMFAULT_DCMP0**, **PWMFAULT_DCMP1**, **PWMFAULT_DCMP2**, **PWMFAULT_DCMP3**, **PWMFAULT_DCMP4**, **PWMFAULT_DCMP5**, **PWMFAULT_DCMP6**, or **PWMFAULT_DCMP7**.

Description:

This function allows selection of the set of fault inputs that is combined to generate a fault condition to a given PWM generator. By default, all generators use only FAULT0 (for backwards compatibility) but if [PWMGenConfigure\(\)](#) is called with flag **PWM_GEN_MODE_FAULT_SRC** in the *ui32Config* parameter, extended fault handling is enabled and this function must be called to configure the fault triggers.

The fault signal to the PWM generator is generated by ORing together each of the signals specified in the *ui32FaultTriggers* parameter after having adjusted the sense of each FAULTn input based on the configuration previously set using a call to [PWMGenFaultConfigure\(\)](#).

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

22.2.2.17 PWMGenIntClear

Clears the specified interrupt(s) for the specified PWM generator block.

Prototype:

```
void
PWMGenIntClear(uint32_t ui32Base,
                uint32_t ui32Gen,
                uint32_t ui32Ints)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to query. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Ints specifies the interrupts to be cleared.

Description:

This function clears the specified interrupt(s) by writing a 1 to the specified bits of the interrupt status register for the specified PWM generator. The *ui32Ints* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, or **PWM_INT_CNT_BD**.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

22.2.2.18 PWMGenIntRegister

Registers an interrupt handler for the specified PWM generator block.

Prototype:

```
void
PWMGenIntRegister(uint32_t ui32Base,
```

```
    uint32_t ui32Gen,  
    void (*pfnIntHandler) (void))
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator in question. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

pfnIntHandler is a pointer to the function to be called when the PWM generator interrupt occurs.

Description:

This function ensures that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected for the specified PWM generator block. This function also enables the corresponding PWM generator interrupt in the interrupt controller; individual generator interrupts and interrupt sources must be enabled with [PWMIntEnable\(\)](#) and [PWMGGenIntTrigEnable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

22.2.2.19 PWMGGenIntStatus

Gets interrupt status for the specified PWM generator block.

Prototype:

```
uint32_t  
PWMGGenIntStatus(uint32_t ui32Base,  
                  uint32_t ui32Gen,  
                  bool bMasked)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to query. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

Returns the contents of the interrupt status register or the contents of the raw interrupt status register for the specified PWM generator.

22.2.2.20 PWMGGenIntTrigDisable

Disables interrupts for the specified PWM generator block.

Prototype:

```
void
PWMGGenIntTrigDisable(uint32_t ui32Base,
                      uint32_t ui32Gen,
                      uint32_t ui32IntTrig)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to have interrupts and triggers disabled. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32IntTrig specifies the interrupts and triggers to be disabled.

Description:

This function masks the specified interrupt(s) and trigger(s) by clearing the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ui32IntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

22.2.2.21 PWMGGenIntTrigEnable

Enables interrupts and triggers for the specified PWM generator block.

Prototype:

```
void
PWMGGenIntTrigEnable(uint32_t ui32Base,
                      uint32_t ui32Gen,
                      uint32_t ui32IntTrig)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to have interrupts and triggers enabled. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32IntTrig specifies the interrupts and triggers to be enabled.

Description:

This function unmasks the specified interrupt(s) and trigger(s) by setting the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ui32IntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

22.2.2.22 PWMGenIntUnregister

Removes an interrupt handler for the specified PWM generator block.

Prototype:

```
void  
PWMGenIntUnregister(uint32_t ui32Base,  
                      uint32_t ui32Gen)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator in question. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function unregisters the interrupt handler for the specified PWM generator block. This function also disables the corresponding PWM generator interrupt in the interrupt controller; individual generator interrupts and interrupt sources must be disabled with [PWMIntDisable\(\)](#) and [PWMGenIntTrigDisable\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

22.2.2.23 PWMGenPeriodGet

Gets the period of a PWM generator block.

Prototype:

```
uint32_t  
PWMGenPeriodGet(uint32_t ui32Base,  
                  uint32_t ui32Gen)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to query. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function gets the period of the specified PWM generator block. The period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

If the update of the counter for the specified PWM generator has yet to be completed, the value returned may not be the active period. The value returned is the programmed period, measured in PWM clock ticks.

Returns:

Returns the programmed period of the specified generator block in PWM clock ticks.

22.2.2.24 PWMGenPeriodSet

Sets the period of a PWM generator.

Prototype:

```
void
PWMGenPeriodSet(uint32_t ui32Base,
                 uint32_t ui32Gen,
                 uint32_t ui32Period)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32Gen is the PWM generator to be modified. This parameter must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ui32Period specifies the period of PWM generator output, measured in clock ticks.

Description:

This function sets the period of the specified PWM generator block, where the period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

Note:

Any subsequent calls made to this function before an update occurs cause the previous values to be overwritten.

Returns:

None.

22.2.2.25 PWMIntDisable

Disables generator and fault interrupts for a PWM module.

Prototype:

```
void
PWMIntDisable(uint32_t ui32Base,
               uint32_t ui32GenFault)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32GenFault contains the interrupts to be disabled. This parameter must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

This function masks the specified interrupt(s) by clearing the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

22.2.2.26 PWMIntEnable

Enables generator and fault interrupts for a PWM module.

Prototype:

```
void  
PWMIntEnable(uint32_t ui32Base,  
             uint32_t ui32GenFault)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32GenFault contains the interrupts to be enabled. This parameter must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

This function unmasks the specified interrupt(s) by setting the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

22.2.2.27 PWMIntStatus

Gets the interrupt status for a PWM module.

Prototype:

```
uint32_t  
PWMIntStatus(uint32_t ui32Base,  
             bool bMasked)
```

Parameters:

ui32Base is the base address of the PWM module.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If **bMasked** is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

The current interrupt status, enumerated as a bit field of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, and **PWM_INT_FAULT3**.

22.2.2.28 PWMOutputFault

Specifies the state of PWM outputs in response to a fault condition.

Prototype:

```
void  
PWMOutputFault(uint32_t ui32Base,
```

```
    uint32_t ui32PWMOutBits,
    bool bFaultSuppress)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PWMOutBits are the PWM outputs to be modified. This parameter must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bFaultSuppress determines if the signal is suppressed or passed through during an active fault condition.

Description:

This function sets the fault handling characteristics of the selected PWM outputs. The outputs are selected using the parameter *ui32PWMOutBits*. The parameter *bFaultSuppress* determines the fault handling characteristics for the selected outputs. If *bFaultSuppress* is **true**, then the selected outputs are made inactive. If *bFaultSuppress* is **false**, then the selected outputs are unaffected by the detected fault.

On devices supporting extended PWM fault handling, the state the affected output pins are driven to can be configured with [PWMOoutputFaultLevel\(\)](#). If not configured, or if the device does not support extended PWM fault handling, affected outputs are driven low on a fault condition.

Returns:

None.

22.2.2.29 PWMOoutputFaultLevel

Specifies the level of PWM outputs suppressed in response to a fault condition.

Prototype:

```
void
PWMOoutputFaultLevel(uint32_t ui32Base,
                      uint32_t ui32PWMOutBits,
                      bool bDriveHigh)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PWMOutBits are the PWM outputs to be modified. This parameter must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bDriveHigh determines if the signal is driven high or low during an active fault condition.

Description:

This function determines whether a PWM output pin that is suppressed in response to a fault condition is driven high or low. The affected outputs are selected using the parameter *ui32PWMOutBits*. The parameter *bDriveHigh* determines the output level for the pins identified by *ui32PWMOutBits*. If *bDriveHigh* is **true** then the selected outputs are driven high when a fault is detected. If it is **false**, the pins are driven low.

In a fault condition, pins which have not been configured to be suppressed via a call to [PWMOoutputFault\(\)](#) are unaffected by this function.

Note:

This function is available only on devices which support extended PWM fault handling.

Returns:

None.

22.2.2.30 PWMOutputInvert

Selects the inversion mode for PWM outputs.

Prototype:

```
void  
PWMOutputInvert(uint32_t ui32Base,  
                 uint32_t ui32PwmOutBits,  
                 bool bInvert)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PwmOutBits are the PWM outputs to be modified. This parameter must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bInvert determines if the signal is inverted or passed through.

Description:

This function is used to select the inversion mode for the selected PWM outputs. The outputs are selected using the parameter *ui32PwmOutBits*. The parameter *bInvert* determines the inversion mode for the selected outputs. If *bInvert* is **true**, this function causes the specified PWM output signals to be inverted or made active low. If *bInvert* is **false**, the specified outputs are passed through as is or made active high.

Returns:

None.

22.2.2.31 PWMOutputState

Enables or disables PWM outputs.

Prototype:

```
void  
PWMOutputState(uint32_t ui32Base,  
                uint32_t ui32PwmOutBits,  
                bool bEnable)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PwmOutBits are the PWM outputs to be modified. This parameter must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bEnable determines if the signal is enabled or disabled.

Description:

This function enables or disables the selected PWM outputs. The outputs are selected using the parameter *ui32PWMOutBits*. The parameter *bEnable* determines the state of the selected outputs. If *bEnable* is **true**, then the selected PWM outputs are enabled, or placed in the active state. If *bEnable* is **false**, then the selected outputs are disabled or placed in the inactive state.

Returns:

None.

22.2.2.32 PWMOutputUpdateMode

Sets the update mode or synchronization mode to the PWM outputs.

Prototype:

```
void
PWMOutputUpdateMode(uint32_t ui32Base,
                    uint32_t ui32PWMOutBits,
                    uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PWMOutBits are the PWM outputs to be modified. This parameter must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

ui32Mode specifies the enable update mode to use when enabling or disabling PWM outputs.

Description:

This function sets one of three possible update modes to enable or disable the requested PWM outputs. The *ui32Mode* parameter controls when changes made via calls to [PWMOutputState\(\)](#) take effect. Possible values are:

- **PWM_OUTPUT_MODE_NO_SYNC**, which enables/disables changes to take effect immediately.
- **PWM_OUTPUT_MODE_SYNC_LOCAL**, which causes changes to take effect when the local PWM generator's count next reaches 0.
- **PWM_OUTPUT_MODE_SYNC_GLOBAL**, which causes changes to take effect when the local PWM generator's count next reaches 0 following a call to [PWMSyncUpdate\(\)](#) which specifies the same generator in its *ui32GenBits* parameter.

Note:

This function is only available on Snowflake class devices.

Returns:

None.

22.2.2.33 PWMPulseWidthGet

Gets the pulse width of a PWM output.

Prototype:

```
uint32_t  
PWMPulseWidthGet(uint32_t ui32Base,  
                  uint32_t ui32PWMOut)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PWMOut is the PWM output to query. This parameter must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

Description:

This function gets the currently programmed pulse width for the specified PWM output. If the update of the comparator for the specified output has yet to be completed, the value returned may not be the active pulse width. The value returned is the programmed pulse width, measured in PWM clock ticks.

Returns:

Returns the width of the pulse in PWM clock ticks.

22.2.2.34 PWMPulseWidthSet

Sets the pulse width for the specified PWM output.

Prototype:

```
void  
PWMPulseWidthSet(uint32_t ui32Base,  
                  uint32_t ui32PWMOut,  
                  uint32_t ui32Width)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32PWMOut is the PWM output to modify. This parameter must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

ui32Width specifies the width of the positive portion of the pulse.

Description:

This function sets the pulse width for the specified PWM output, where the pulse width is defined as the number of PWM clock ticks.

Note:

Any subsequent calls made to this function before an update occurs cause the previous values to be overwritten.

Returns:

None.

22.2.2.35 PWMSyncTimeBase

Synchronizes the counters in one or multiple PWM generator blocks.

Prototype:

```
void
PWMSyncTimeBase(uint32_t ui32Base,
                 uint32_t ui32GenBits)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32GenBits are the PWM generator blocks to be synchronized. This parameter must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM module, this function synchronizes the time base of the generator blocks by causing the specified generator counters to be reset to zero.

Returns:

None.

22.2.2.36 PWMSyncUpdate

Synchronizes all pending updates.

Prototype:

```
void
PWMSyncUpdate(uint32_t ui32Base,
               uint32_t ui32GenBits)
```

Parameters:

ui32Base is the base address of the PWM module.

ui32GenBits are the PWM generator blocks to be updated. This parameter must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM generators, this function causes all queued updates to the period or pulse width to be applied the next time the corresponding counter becomes zero.

Returns:

None.

22.3 Programming Example

The following example shows how to use the PWM API to initialize the PWM0 with a 50 KHz frequency, and with a 25% duty cycle on **PWM0** and a 75% duty cycle on **PWM1**.

```
//
// Configure the PWM generator for count down mode with immediate updates
// to the parameters.
//
PWMGenConfigure(PWM_BASE, PWM_GEN_0,
                 PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
```

```
//  
// Set the period. For a 50 KHz frequency, the period = 1/50,000, or 20  
// microseconds. For a 20 MHz clock, this translates to 400 clock ticks.  
// Use this value to set the period.  
//  
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 100);  
  
//  
// Set the pulse width of PWM0 for a 25% duty cycle.  
//  
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 100);  
  
//  
// Set the pulse width of PWM1 for a 75% duty cycle.  
//  
PWMPulseWidthSet(PWM_BASE, PWM_OUT_1, 300);  
  
//  
// Start the timers in generator 0.  
//  
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 100);  
  
//  
// Enable the outputs.  
//  
PWMPulseWidthSet(PWM_BASE, (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
```

23 Quadrature Encoder (QEI)

Introduction	441
API Functions	441
Programming Example	450

23.1 Introduction

The quadrature encoder API provides a set of functions for dealing with the Quadrature Encoder with Index (QEI). Functions are provided to configure and read the position and velocity captures, register a QEI interrupt handler, and handle QEI interrupt masking/clearing.

The quadrature encoder module provides hardware encoding of the two channels and the index signal from a quadrature encoder device into an absolute or relative position. There is additional hardware for capturing a measure of the encoder velocity, which is simply a count of encoder pulses during a fixed time period; the number of pulses is directly proportional to the encoder speed. Note that the velocity capture can only operate when the position capture is enabled.

The QEI module supports two modes of operation: phase mode and clock/direction mode. In phase mode, the encoder produces two clocks that are 90 degrees out of phase; the edge relationship is used to determine the direction of rotation. In clock/direction mode, the encoder produces a clock signal to indicate steps and a direction signal to indicate the direction of rotation.

When in phase mode, edges on the first channel or edges on both channels can be counted; counting edges on both channels provides higher encoder resolution if required. In either mode, the input signals can be swapped before being processed, allowing wiring mistakes to be corrected without modifying the circuit board.

The index pulse can be used to reset the position counter, allowing the position counter to maintain the absolute encoder position. Otherwise, the position counter maintains the relative position and is never reset.

The velocity capture has a timer to measure equal periods of time. The number of encoder pulses over each time period is accumulated as a measure of the encoder velocity. The running total for the current time period and the final count for the previous time period are available to be read. The final count for the previous time period is usually used as the velocity measure.

The QEI module generates interrupts when the index pulse is detected, when the velocity timer expires, when the encoder direction changes, and when a phase signal error is detected. These interrupt sources can be individually masked so that only the events of interest cause a processor interrupt.

This driver is contained in `driverlib/qei.c`, with `driverlib/qei.h` containing the API declarations for use by applications.

23.2 API Functions

Functions

- void `QEIConfigure` (`uint32_t ui32Base, uint32_t ui32Config, uint32_t ui32MaxPosition`)

- int32_t QEIDirectionGet (uint32_t ui32Base)
- void QEIDisable (uint32_t ui32Base)
- void QEIEnable (uint32_t ui32Base)
- bool QEIErrorGet (uint32_t ui32Base)
- void QEIIIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t QEIIIntStatus (uint32_t ui32Base, bool bMasked)
- void QEIIIntUnregister (uint32_t ui32Base)
- uint32_t QEIPositionGet (uint32_t ui32Base)
- void QEIPositionSet (uint32_t ui32Base, uint32_t ui32Position)
- void QEIVelocityConfigure (uint32_t ui32Base, uint32_t ui32PreDiv, uint32_t ui32Period)
- void QEIVelocityDisable (uint32_t ui32Base)
- void QEIVelocityEnable (uint32_t ui32Base)
- uint32_t QEIVelocityGet (uint32_t ui32Base)

23.2.1 Detailed Description

The quadrature encoder API is broken into three groups of functions: those that deal with position capture, those that deal with velocity capture, and those that deal with interrupt handling.

The position capture is managed with [QEIEnable\(\)](#), [QEIDisable\(\)](#), [QEIConfigure\(\)](#), and [QEIPositionSet\(\)](#). The positional information is retrieved with [QEIPositionGet\(\)](#), [QEIDirectionGet\(\)](#), and [QEIErrorGet\(\)](#).

The velocity capture is managed with [QEIVelocityEnable\(\)](#), [QEIVelocityDisable\(\)](#), and [QEIVelocityConfigure\(\)](#). The computed encoder velocity is retrieved with [QEIVelocityGet\(\)](#).

The interrupt handler for the QEI interrupt is managed with [QEIIIntRegister\(\)](#) and [QEIIIntUnregister\(\)](#). The individual interrupt sources within the QEI module are managed with [QEIIIntEnable\(\)](#), [QEIIIntDisable\(\)](#), [QEIIIntStatus\(\)](#), and [QEIIIntClear\(\)](#).

23.2.2 Function Documentation

23.2.2.1 QEIConfigure

Configures the quadrature encoder.

Prototype:

```
void
QEIConfigure(uint32_t ui32Base,
             uint32_t ui32Config,
             uint32_t ui32MaxPosition)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32Config is the configuration for the quadrature encoder. See below for a description of this parameter.

ui32MaxPosition specifies the maximum position value.

Description:

This function configures the operation of the quadrature encoder. The *ui32Config* parameter provides the configuration of the encoder and is the logical OR of several values:

- **QEI_CONFIG_CAPTURE_A** or **QEI_CONFIG_CAPTURE_A_B** specify if edges on channel A or on both channels A and B should be counted by the position integrator and velocity accumulator.
- **QEI_CONFIG_NO_RESET** or **QEI_CONFIG_RESET_IDX** specify if the position integrator should be reset when the index pulse is detected.
- **QEI_CONFIG_QUADRATURE** or **QEI_CONFIG_CLOCK_DIR** specify if quadrature signals are being provided on ChA and ChB, or if a direction signal and a clock are being provided instead.
- **QEI_CONFIG_NO_SWAP** or **QEI_CONFIG_SWAP** to specify if the signals provided on ChA and ChB should be swapped before being processed.

ui32MaxPosition is the maximum value of the position integrator and is the value used to reset the position capture when in index reset mode and moving in the reverse (negative) direction.

Returns:

None.

23.2.2.2 QEIDirectionGet

Gets the current direction of rotation.

Prototype:

```
int32_t
QEIDirectionGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function returns the current direction of rotation. In this case, current means the most recently detected direction of the encoder; it may not be presently moving but this is the direction it last moved before it stopped.

Returns:

Returns 1 if moving in the forward direction or -1 if moving in the reverse direction.

23.2.2.3 QEIDisable

Disables the quadrature encoder.

Prototype:

```
void
QEIDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function disables operation of the quadrature encoder module.

Returns:

None.

23.2.2.4 QEIEnable

Enables the quadrature encoder.

Prototype:

```
void  
QEIEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function enables operation of the quadrature encoder module. The module must be configured before it is enabled.

See also:

[QEIConfigure\(\)](#)

Returns:

None.

23.2.2.5 QEIErrorGet

Gets the encoder error indicator.

Prototype:

```
bool  
QEIErrorGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function returns the error indicator for the quadrature encoder. It is an error for both of the signals of the quadrature input to change at the same time.

Returns:

Returns **true** if an error has occurred and **false** otherwise.

23.2.2.6 QEIIIntClear

Clears quadrature encoder interrupt sources.

Prototype:

```
void
QEIIIntClear(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32IntFlags is a bit mask of the interrupt sources to be cleared. This parameter can be any of the **QEI_INTERRUPT**, **QEI_INDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:

The specified quadrature encoder interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

23.2.2.7 QEIIIntDisable

Disables individual quadrature encoder interrupt sources.

Prototype:

```
void
QEIIIntDisable(uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32IntFlags is a bit mask of the interrupt sources to be disabled. This parameter can be any of the **QEI_INTERRUPT**, **QEI_INDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:

This function disables the indicated quadrature encoder interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

23.2.2.8 QEIIIntEnable

Enables individual quadrature encoder interrupt sources.

Prototype:

```
void  
QEIIIntEnable(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32IntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **QEI_INTERRUPT**, **QEI_INDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:

This function enables the indicated quadrature encoder interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

23.2.2.9 QEIIIntRegister

Registers an interrupt handler for the quadrature encoder interrupt.

Prototype:

```
void  
QEIIIntRegister(uint32_t ui32Base,  
                 void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

pfnHandler is a pointer to the function to be called when the quadrature encoder interrupt occurs.

Description:

This function registers the handler to be called when a quadrature encoder interrupt occurs. This function enables the global interrupt in the interrupt controller; specific quadrature encoder interrupts must be enabled via [QEIIIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [QEIIIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

23.2.2.10 QEIIIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t
QEIIIntStatus(uint32_t ui32Base,
              bool bMasked)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the quadrature encoder module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of **QEI_INTERRUPT**, **QEI_INDIR**, **QEI_INTTIMER**, and **QEI_INTINDEX**.

23.2.2.11 QEIIIntUnregister

Unregisters an interrupt handler for the quadrature encoder interrupt.

Prototype:

```
void
QEIIIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function unregisters the handler to be called when a quadrature encoder interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

23.2.2.12 QEIPositionGet

Gets the current encoder position.

Prototype:

```
uint32_t
QEIPositionGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function returns the current position of the encoder. Depending upon the configuration of the encoder, and the incident of an index pulse, this value may or may not contain the expected data (that is, if in reset on index mode, if an index pulse has not been encountered, the position counter is not yet aligned with the index pulse).

Returns:

The current position of the encoder.

23.2.2.13 QEIPositionSet

Sets the current encoder position.

Prototype:

```
void  
QEIPositionSet(uint32_t ui32Base,  
                uint32_t ui32Position)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32Position is the new position for the encoder.

Description:

This function sets the current position of the encoder; the encoder position is then measured relative to this value.

Returns:

None.

23.2.2.14 QEIVelocityConfigure

Configures the velocity capture.

Prototype:

```
void  
QEIVelocityConfigure(uint32_t ui32Base,  
                     uint32_t ui32PreDiv,  
                     uint32_t ui32Period)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

ui32PreDiv specifies the predivider applied to the input quadrature signal before it is counted; can be one of **QEI_VELDIV_1**, **QEI_VELDIV_2**, **QEI_VELDIV_4**, **QEI_VELDIV_8**, **QEI_VELDIV_16**, **QEI_VELDIV_32**, **QEI_VELDIV_64**, or **QEI_VELDIV_128**.

ui32Period specifies the number of clock ticks over which to measure the velocity; must be non-zero.

Description:

This function configures the operation of the velocity capture portion of the quadrature encoder. The position increment signal is predivided as specified by *ui32PreDiv* before being accumulated by the velocity capture. The divided signal is accumulated over *ui32Period* system clock before being saved and resetting the accumulator.

Returns:

None.

23.2.2.15 QEIVelocityDisable

Disables the velocity capture.

Prototype:

```
void  
QEIVelocityDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function disables operation of the velocity capture in the quadrature encoder module.

Returns:

None.

23.2.2.16 QEIVelocityEnable

Enables the velocity capture.

Prototype:

```
void  
QEIVelocityEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function enables operation of the velocity capture in the quadrature encoder module. The module must be configured before velocity capture is enabled.

See also:

[QEIVelocityConfigure\(\)](#) and [QEIEnable\(\)](#)

Returns:

None.

23.2.2.17 QEIVelocityGet

Gets the current encoder speed.

Prototype:

```
uint32_t  
QEIVelocityGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the quadrature encoder module.

Description:

This function returns the current speed of the encoder. The value returned is the number of pulses detected in the specified time period; this number can be multiplied by the number of time periods per second and divided by the number of pulses per revolution to obtain the number of revolutions per second.

Returns:

Returns the number of pulses captured in the given time period.

23.3 Programming Example

The following example shows how to use the Quadrature Encoder API to configure the quadrature encoder read back an absolute position.

```
//  
// Configure the quadrature encoder to capture edges on both signals and  
// maintain an absolute position by resetting on index pulses. Using a  
// 1000 line encoder at four edges per line, there are 4000 pulses per  
// revolution; therefore set the maximum position to 3999 as the count  
// is zero based.  
//  
QEIConfigure(QEI_BASE, (QEI_CONFIG_CAPTURE_A_B | QEI_CONFIG_RESET_IDX |  
                         QEI_CONFIG_QUADRATURE | QEI_CONFIG_NO_SWAP), 3999);  
  
//  
// Enable the quadrature encoder.  
//  
QEIEnable(QEI_BASE);  
  
//  
// Delay for some time...  
//  
//  
// Read the encoder position.  
//  
QEIPositionGet(QEI_BASE);
```

24 SHA/MD5

Introduction	451
API Functions	451
Programming Examples	461

24.1 Introduction

The SHA/MD% module driver provides a method for generating hash values and hash-based message authentication codes. The configuration and feature highlights are:

- Supports MD5, SHA-1, SHA-224, and SHA-256 hashing algorithms.
- Allows pre-processing of the HMAC key to speed processing of later messages.

This driver is contained in `driverlib/shamd5.c`, with `driverlib/shamd5.h` containing the API declarations for use by applications.

24.2 API Functions

Functions

- void `SHAMD5ConfigSet` (uint32_t ui32Base, uint32_t ui32Mode)
- void `SHAMD5DataProcess` (uint32_t ui32Base, uint32_t *pui32DataSrc, uint32_t ui32DataLength, uint32_t *pui32HashResult)
- void `SHAMD5DataWrite` (uint32_t ui32Base, uint32_t *pui32Src)
- bool `SHAMD5DataWriteNonBlocking` (uint32_t ui32Base, uint32_t *pui32Src)
- void `SHAMD5DMADisable` (uint32_t ui32Base)
- void `SHAMD5DMAEnable` (uint32_t ui32Base)
- void `SHAMD5HashLengthSet` (uint32_t ui32Base, uint32_t ui32Length)
- void `SHAMD5HMACKeySet` (uint32_t ui32Base, uint32_t *pui32Src)
- void `SHAMD5HMACPPKeyGenerate` (uint32_t ui32Base, uint32_t *pui32Key, uint32_t *pui32PPKey)
- void `SHAMD5HMACPPKeySet` (uint32_t ui32Base, uint32_t *pui32Src)
- void `SHAMD5HMACProcess` (uint32_t ui32Base, uint32_t *pui32DataSrc, uint32_t ui32DataLength, uint32_t *pui32HashResult)
- void `SHAMD5IntClear` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `SHAMD5IntDisable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `SHAMD5IntEnable` (uint32_t ui32Base, uint32_t ui32IntFlags)
- void `SHAMD5IntRegister` (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t `SHAMD5IntStatus` (uint32_t ui32Base, bool bMasked)
- void `SHAMD5IntUnregister` (uint32_t ui32Base)
- void `SHAMD5Reset` (uint32_t ui32Base)
- void `SHAMD5ResultRead` (uint32_t ui32Base, uint32_t *pui32Dest)

24.2.1 Detailed Description

The SHA/MD5 API consists of functions for configuring the SHA/MD5 module, processing data, and reading the resultant hash.

24.2.2 Function Documentation

24.2.2.1 SHAMD5ConfigSet

Writes the mode in the SHA/MD5 module.

Prototype:

```
void  
SHAMD5ConfigSet(uint32_t ui32Base,  
                 uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

ui32Mode is the mode of the SHA/MD5 module.

Description:

This function writes the mode register configuring the SHA/MD5 module.

The ui32Mode parameter is a bit-wise OR of values:

- **SHAMD5_ALGO_MD5** - Regular hash with MD5
- **SHAMD5_ALGO_SHA1** - Regular hash with SHA-1
- **SHAMD5_ALGO_SHA224** - Regular hash with SHA-224
- **SHAMD5_ALGO_SHA256** - Regular hash with SHA-256
- **SHAMD5_ALGO_HMAC_MD5** - HMAC with MD5
- **SHAMD5_ALGO_HMAC_SHA1** - HMAC with SHA-1
- **SHAMD5_ALGO_HMAC_SHA224** - HMAC with SHA-224
- **SHAMD5_ALGO_HMAC_SHA256** - HMAC with SHA-256

Returns:

None

24.2.2.2 SHAMD5DataProcess

Compute a hash using the SHA/MD5 module.

Prototype:

```
void  
SHAMD5DataProcess(uint32_t ui32Base,  
                  uint32_t *pui32DataSrc,  
                  uint32_t ui32DataLength,  
                  uint32_t *pui32HashResult)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32DataSrc is a pointer to an array of data that contains the data that will be hashed.

ui32DataLength specifies the length of the data to be hashed in bytes.

pui32HashResult is a pointer to an array that holds the result of the hashing operation.

Description:

This function computes the hash of an array of data using the SHA/MD5 module.

The length of the hash result is dependent on the algorithm that is in use. The following table shows the correct array size for each algorithm:

	Algorithm	Number of Words in Result			
MD5	4 Words (128 bits)	SHA-1	5 Words (160 bits)	SHA-224	7 Words (224 bits)
SHA-256	8 Words (256 bits)				

Returns:

None

24.2.2.3 SHAMD5DataWrite

Perform a blocking write of 16 words of data to the SHA/MD5 module.

Prototype:

```
void
SHAMD5DataWrite(uint32_t ui32Base,
                uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Src is the pointer to the 16-word array of data that will be written.

Description:

This function does not return until the module is ready to accept data and the data has been written.

Returns:

None.

24.2.2.4 SHAMD5DataWriteNonBlocking

Perform a non-blocking write of 16 words of data to the SHA/MD5 module.

Prototype:

```
bool
SHAMD5DataWriteNonBlocking(uint32_t ui32Base,
                           uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Src is the pointer to the 16-word array of data that will be written.

Description:

This function writes 16 words of data into the data register regardless of whether or not the module is ready to accept the data.

Returns:

This function returns true if the write completed successfully. It returns false if the module was not ready.

24.2.2.5 SHAMD5DMADisable

Disables the uDMA requests in the SHA/MD5 module.

Prototype:

```
void  
SHAMD5DMADisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

Description:

This function configures the DMA options of the SHA/MD5 module.

Returns:

None

24.2.2.6 SHAMD5DMAEnable

Enables the uDMA requests in the SHA/MD5 module.

Prototype:

```
void  
SHAMD5DMAEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

Description:

This function configures the DMA options of the SHA/MD5 module.

Returns:

None

24.2.2.7 SHAMD5HashLengthSet

Write the hash length to the SHA/MD5 module.

Prototype:

```
void  
SHAMD5HashLengthSet(uint32_t ui32Base,  
                     uint32_t ui32Length)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

ui32Length is the hash length in bytes.

Description:

This function writes the length of the hash data of the current operation to the SHA/MD5 module. The value must be a multiple of 64 if the close hash is not set in the mode register.

Note:

When this register is written, hash processing is triggered.

Returns:

None.

24.2.2.8 SHAMD5HMACKeySet

Writes an HMAC key to the digest registers in the SHA/MD5 module.

Prototype:

```
void
SHAMD5HMACKeySet(uint32_t ui32Base,
                  uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Src is the pointer to the 16-word array of the HMAC key.

Description:

This function is used to write HMAC key to the digest registers for key preprocessing. The size of pui32Src must be 512 bytes. If the key is less than 512 bytes, then it must be padded with zeros.

Note:

It is recommended to use the [SHAMD5IntStatus\(\)](#) function to check whether the context is ready before writing the key.

Returns:

None

24.2.2.9 SHAMD5HMACPPKeyGenerate

Process an HMAC key using the SHA/MD5 module.

Prototype:

```
void
SHAMD5HMACPPKeyGenerate(uint32_t ui32Base,
                        uint32_t *pui32Key,
                        uint32_t *pui32PPKey)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Key is a pointer to an array that contains the key to be processed.
pui32PPKey is the pointer to the array that contains the pre-processed key.

Description:

This function processes an HMAC key using the SHA/MD5. The resultant pre-processed key can then be used with later HMAC operations to speed processing time.

The *pui32Key* array must be 16 words (512 bits) long. If the key is less than 512 bits, it must be padded with zeros. The *pui32PPKey* array must each be 16 words (512 bits) long.

Returns:

None

24.2.2.10 SHAMD5HMACPPKeySet

Writes a pre-processed HMAC key to the digest registers in the SHA/MD5 module.

Prototype:

```
void  
SHAMD5HMACPPKeySet (uint32_t ui32Base,  
                      uint32_t *pui32Src)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Src is the pointer to the 16-word array of the HMAC key.

Description:

This function is used to write HMAC key to the digest registers for key preprocessing. The size of pui32Src must be 512 bytes. If the key is less than 512 bytes, then it must be padded with zeros.

Note:

It is recommended to use the [SHAMD5IntStatus\(\)](#) function to check whether the context is ready before writing the key.

Returns:

None

24.2.2.11 SHAMD5HMACProcess

Compute a HMAC with key pre-processing using the SHA/MD5 module.

Prototype:

```
void  
SHAMD5HMACProcess (uint32_t ui32Base,  
                     uint32_t *pui32DataSrc,  
                     uint32_t ui32DataLength,  
                     uint32_t *pui32HashResult)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32DataSrc is a pointer to an array of data that contains the data that is to be hashed.

ui32DataLength specifies the length of the data to be hashed in bytes.

pui32HashResult is a pointer to an array that holds the result of the hashing operation.

Description:

This function computes a HMAC with the given data using the SHA/MD5 module with a pre-processed key.

The length of the hash result is dependent on the algorithm that is selected with the *ui32Algo* argument. The following table shows the correct array size for each algorithm:

	Algorithm	Number of Words in Result			
MD5	4 Words (128 bits)	SHA-1	5 Words (160 bits)	SHA-224	7 Words (224 bits)
SHA-256	8 Words (256 bits)				

Returns:

None

24.2.2.12 SHAMD5IntClear

Clears interrupt sources in the SHA/MD5 module.

Prototype:

```
void
SHAMD5IntClear(uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

ui32IntFlags contains desired interrupts to disable.

Description:

ui32IntFlags must be a logical OR of one or more of the following values:

- **SHAMD5_INT_CONTEXT_READY** - Context input registers are ready.
- **SHAMD5_INT_PARTHASH_READY** - Context output registers are ready after a context switch.
- **SHAMD5_INT_INPUT_READY** - Data FIFO is ready to receive data.
- **SHAMD5_INT_OUTPUT_READY** - Context output registers are ready.

Returns:

None.

24.2.2.13 SHAMD5IntDisable

Disable interrupt sources in the SHA/MD5 module.

Prototype:

```
void
SHAMD5IntDisable(uint32_t ui32Base,
                  uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

ui32IntFlags contains desired interrupts to disable.

Description:

ui32IntFlags must be a logical OR of one or more of the following values:

- **SHAMD5_INT_CONTEXT_READY** - Context input registers are ready.
- **SHAMD5_INT_PARTHASH_READY** - Context output registers are ready after a context switch.
- **SHAMD5_INT_INPUT_READY** - Data FIFO is ready to receive data.
- **SHAMD5_INT_OUTPUT_READY** - Context output registers are ready.

Returns:

None.

24.2.2.14 SHAMD5IntEnable

Enable interrupt sources in the SHA/MD5 module.

Prototype:

```
void  
SHAMD5IntEnable(uint32_t ui32Base,  
                 uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

ui32IntFlags contains desired interrupts to enable.

Description:

This function enables interrupt sources in the SHA/MD5 module. *ui32IntFlags* must be a logical OR of one or more of the following values:

- **SHAMD5_INT_CONTEXT_READY** - Context input registers are ready.
- **SHAMD5_INT_PARTHASH_READY** - Context output registers are ready after a context switch.
- **SHAMD5_INT_INPUT_READY** - Data FIFO is ready to receive data.
- **SHAMD5_INT_OUTPUT_READY** - Context output registers are ready.

Returns:

None.

24.2.2.15 SHAMD5IntRegister

Registers an interrupt handler for the SHA/MD5 module.

Prototype:

```
void  
SHAMD5IntRegister(uint32_t ui32Base,  
                   void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pfnHandler is a pointer to the function to be called when the enabled SHA/MD5 interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables SHA/MD5 interrupts on the interrupt controller; specific SHA/MD5 interrupt sources must be enabled using [SHAMD5IntEnable\(\)](#). The interrupt handler being registered must clear the source of the interrupt using [SHAMD5IntClear\(\)](#).

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, [IntEnable\(\)](#) should be used to enable SHA/MD5 interrupts on the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

24.2.2.16 SHAMD5IntStatus

Get the interrupt status of the SHA/MD5 module.

Prototype:

```
uint32_t
SHAMD5IntStatus(uint32_t ui32Base,
                 bool bMasked)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function returns the current value of the IRQSTATUS register. The value will be a logical OR of the following:

- **SHAMD5_INT_CONTEXT_READY** - Context input registers are ready.
- **SHAMD5_INT_PARTHASH_READY** - Context output registers are ready after a context switch.
- **SHAMD5_INT_INPUT_READY** - Data FIFO is ready to receive data.
- **SHAMD5_INT_OUTPUT_READY** - Context output registers are ready.

Returns:

Interrupt status

24.2.2.17 SHAMD5IntUnregister

Unregisters an interrupt handler for the SHA/MD5 module.

Prototype:

```
void  
SHAMD5IntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

Description:

This function unregisters the previously registered interrupt handler and disables the interrupt in the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

24.2.2.18 SHAMD5Reset

Resets the SHA/MD5 module.

Prototype:

```
void  
SHAMD5Reset(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

Description:

This function performs a soft-reset of the SHA/MD5 module using the SYSCONFIG register.

Returns:

None.

24.2.2.19 SHAMD5ResultRead

Reads the result of a hashing operation.

Prototype:

```
void  
SHAMD5ResultRead(uint32_t ui32Base,  
                  uint32_t *pui32Dest)
```

Parameters:

ui32Base is the base address of the SHA/MD5 module.

pui32Dest is the pointer to the 16-word array of data that will be written.

Description:

This function does not return until the module is ready to accept data and the data has been written.

Returns:

None.

24.3 Hashing Programming Example

The following example generates a hash using the SHA-1 algorithm.

```

// Random data to be hashed.
//
uint32_t g_ui32RandomData[16] =
{
    0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373,
    0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45,
    0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520ala,
    0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6
};

int
main(void)
{
    uint32_t ui32HashResult[5];

    //
    // Enable the CCM module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_EC0);

    //
    // Wait for the CCM module to be ready.
    //
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_EC0))
    {
    }

    //
    // Reset the SHA/MD5 module before use.
    //
    SHAMD5Reset(SHAMD5_BASE);

    //
    // Configure the SHA/MD5 module.
    //
    SHAMD5ConfigSet(SHAMD5_BASE, SHAMD5_ALGO_SHA1);

    //
    // Generate the hash from the data.
    //
    // The resulting hash should be:
    // {0x856210e0, 0xfa2dffe6, 0x94be52d0, 0xca7b2491
    // 0x40d53371}
    //
    SHAMD5ProcessData(SHAMD5_BASE, g_ui32RandomData, 64, ui32HashResult);
}

```

24.4 HMAC Programming Example

The following example generates an HMAC using the SHA-1 algorithm.

```

// Random data to be hashed.
//
uint32_t g_ui32RandomData[16] =

```

```
{  
    0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373,  
    0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45,  
    0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520a1a,  
    0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6  
};  
  
//  
// HMAC key  
//  
uint32_t g_ui32HMACKey[16] =  
{  
    0x8a5f1b22, 0xcb935d29, 0xcc1ac092, 0x5dad8c9e,  
    0x6a83b39f, 0x8607dc60, 0xda0ba4d2, 0xf49b0fa2,  
    0xaf35d524, 0xffa8001d, 0xbcc931e8, 0x4a2c99ef,  
    0x7fa297ab, 0xab943bae, 0x07c61cc4, 0x47c8627d  
};  
  
int  
main(void)  
{  
    uint32_t pui32HashResult[5];  
  
    //  
    // Enable the CCM module.  
    //  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_EC0);  
  
    //  
    // Wait for the CCM module to be ready.  
    //  
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_EC0))  
    {  
    }  
  
    //  
    // Reset the SHA/MD5 module before use.  
    //  
    SHAMD5Reset(SHAMD5_BASE);  
  
    //  
    // Configure the SHA/MD5 module.  
    //  
    SHAMD5ConfigSet(SHAMD5_BASE, SHAMD5_ALGO_HMAC_SHA1);  
  
    //  
    // Set the HMAC key.  
    //  
    SHAMD5HMACKeySet(SHAMD5_BASE, g_ui32HMACKey);  
  
    //  
    // Generate the hash from the data.  
    //  
    // The resulting hash should be:  
    // {0x326e8759, 0xf138cd36, 0xfb44cd58, 0x132b563a  
    // 0x76772c4b}  
    //  
    SHAMD5HMACProcessData(SHAMD5_BASE, g_ui32RandomData, 64, ui32HashResult);  
}
```

25 Synchronous Serial Interface (SSI)

Introduction	463
API Functions	463
Programming Example	476

25.1 Introduction

The Synchronous Serial Interface (SSI) module provides the functionality for synchronous serial communications with peripheral devices, and can be configured to use either the Motorola® SPI™ or the Texas Instruments® synchronous serial interface frame formats. In addition, some devices also can be configured to use the National Semiconductor® Microwire format. The size of the data frame is also configurable, and can be set to be between 4 and 16 bits, inclusive.

The SSI module performs serial-to-parallel data conversion on data received from a peripheral device, and parallel-to-serial conversion on data transmitted to a peripheral device. The TX and RX paths are buffered with internal FIFOs, allowing up to eight 16-bit values to be stored independently.

The SSI module can be configured as either a master or a slave device. As a slave device, the SSI module can also be configured to disable its output, which allows a master device to be coupled with multiple slave devices.

The SSI 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. Some Tiva devices can use the PIOSC as the serial bit clock. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

For parts that include a DMA controller, the SSI module also provides a DMA interface to facilitate data transfer via DMA.

This driver is contained in `driverlib/ssi.c`, with `driverlib/ssi.h` containing the API declarations for use by applications.

25.2 API Functions

Functions

- void [SSIAdvDataPutFrameEnd](#) (uint32_t ui32Base, uint32_t ui32Data)
- int32_t [SSIAdvDataPutFrameEndNonBlocking](#) (uint32_t ui32Base, uint32_t ui32Data)
- void [SSIAdvFrameHoldDisable](#) (uint32_t ui32Base)
- void [SSIAdvFrameHoldEnable](#) (uint32_t ui32Base)
- void [SSIAdvModeSet](#) (uint32_t ui32Base, uint32_t ui32Mode)
- bool [SSIBusy](#) (uint32_t ui32Base)
- uint32_t [SSIClockSourceGet](#) (uint32_t ui32Base)
- void [SSIClockSourceSet](#) (uint32_t ui32Base, uint32_t ui32Source)
- void [SSIConfigSetExpClk](#) (uint32_t ui32Base, uint32_t ui32SSIClk, uint32_t ui32Protocol, uint32_t ui32Mode, uint32_t ui32BitRate, uint32_t ui32DataWidth)
- void [SSIDataGet](#) (uint32_t ui32Base, uint32_t *pui32Data)

- `int32_t SSIDataGetNonBlocking (uint32_t ui32Base, uint32_t *pui32Data)`
- `void SSIDataPut (uint32_t ui32Base, uint32_t ui32Data)`
- `int32_t SSIDataPutNonBlocking (uint32_t ui32Base, uint32_t ui32Data)`
- `void SSIDisable (uint32_t ui32Base)`
- `void SSIDMADisable (uint32_t ui32Base, uint32_t ui32DMAFlags)`
- `void SSIDMAEnable (uint32_t ui32Base, uint32_t ui32DMAFlags)`
- `void SSIEnable (uint32_t ui32Base)`
- `void SSIIIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void SSIIIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void SSIIIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void SSIIIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))`
- `uint32_t SSIIIntStatus (uint32_t ui32Base, bool bMasked)`
- `void SSIIIntUnregister (uint32_t ui32Base)`

25.2.1 Detailed Description

The SSI API is broken into several groups of functions. Each of those groups is addressed below.

The configuration of the SSI module is managed by the `SSIConfigSetExpClk()` function, while state is managed by the `SSIEnable()` and `SSIDisable()` functions. The DMA interface is enabled or disabled by the `SSIDMAEnable()` and `SSIDMADisable()` functions. The SSI baud clock is managed by the `SSIClockSourceGet()` and `SSIClockSourceSet()` functions.

Data handling is performed by the `SSIDataPut()`, `SSIDataPutNonBlocking()`, `SSIDataGet()`, and `SSIDataGetNonBlocking()` functions.

Interrupts from the SSI module are managed using the `SSIIIntClear()`, `SSIIIntDisable()`, `SSIIIntEnable()`, `SSIIIntRegister()`, `SSIIIntStatus()`, and `SSIIIntUnregister()` functions.

The `SSIConfig()`, `SSIDataNonBlockingGet()`, and `SSIDataNonBlockingPut()` APIs from previous versions of the peripheral driver library have been replaced by the `SSIConfigSetExpClk()`, `SSI-DataGetNonBlocking()`, and `SSIDataPutNonBlocking()` APIs. Macros have been provided in `ssi.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

25.2.2 Function Documentation

25.2.2.1 SSIAdvDataPutFrameEnd

Puts a data element into the SSI transmit FIFO as the end of a frame.

Prototype:

```
void  
SSIAdvDataPutFrameEnd(uint32_t ui32Base,  
                      uint32_t ui32Data)
```

Parameters:

`ui32Base` specifies the SSI module base address.

`ui32Data` is the data to be transmitted over the SSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SSI module, marking it as the end of a frame. If there is no space available in the transmit FIFO, this function waits until there is space available before returning. After this byte is transmitted by the SSI module, the FSS signal de-asserts for at least one SSI clock.

Note:

The upper 24 bits of *ui32Data* are discarded by the hardware.

The availability of the advanced mode of SSI operation varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

None.

25.2.2.2 SSIAdvDataPutFrameEndNonBlocking

Puts a data element into the SSI transmit FIFO as the end of a frame.

Prototype:

```
int32_t
SSIAdvDataPutFrameEndNonBlocking(uint32_t ui32Base,
                                  uint32_t ui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32Data is the data to be transmitted over the SSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SSI module, marking it as the end of a frame. After this byte is transmitted by the SSI module, the FSS signal de-asserts for at least one SSI clock. If there is no space in the FIFO, then this function returns a zero.

Note:

The upper 24 bits of *ui32Data* are discarded by the hardware.

The availability of the advanced mode of SSI operation varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

Returns the number of elements written to the SSI transmit FIFO.

25.2.2.3 SSIAdvFrameHoldDisable

Configures the SSI advanced mode to de-assert the SSIFss signal after every byte transfer.

Prototype:

```
void
SSIAdvFrameHoldDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SSI module.

Description:

This function configures the SSI module to de-assert the SSIFss signal for one SSI clock cycle after every byte is transferred using one of the advanced modes (instead of leaving it asserted for the entire transfer). This mode is the default operation.

Note:

The availability of the advanced mode of SSI operation varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

None.

25.2.2.4 SSIAdvFrameHoldEnable

Configures the SSI advanced mode to hold the SSIFss signal during the full transfer.

Prototype:

```
void  
SSIAdvFrameHoldEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SSI module.

Description:

This function configures the SSI module to de-assert the SSIFss signal during the entire data transfer when using one of the advanced modes (instead of briefly de-asserting it after every byte). When using this mode, SSIFss can be directly controlled via [SSIAdvDataPutFrameEnd\(\)](#) and [SSIAdvDataPutFrameEndNonBlocking\(\)](#).

Note:

The availability of the advanced mode of SSI operation varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

None.

25.2.2.5 SSIAdvModeSet

Selects the advanced mode of operation for the SSI module.

Prototype:

```
void  
SSIAdvModeSet(uint32_t ui32Base,  
              uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the SSI module.

ui32Mode is the mode of operation to use.

Description:

This function selects the mode of operation for the SSI module, which is needed when using the advanced operation modes (Bi- or Quad-SPI). One of the following modes can be selected:

- **SSI_ADV_MODE_LEGACY** - Disables the advanced modes of operation, resulting in legacy, or backwards-compatible, operation. When this mode is selected, it is not valid to switch to Bi- or Quad-SPI operation. This mode is the default.
- **SSI_ADV_MODE_WRITE** - The advanced mode of operation where data is only written to the slave; any data clocked in via the **SSIRx** pin is thrown away (instead of being placed into the SSI Rx FIFO).
- **SSI_ADV_MODE_READ_WRITE** - The advanced mode of operation where data is written to and read from the slave; this mode is the same as **SSI_ADV_MODE_LEGACY** but allows transitions to Bi- or Quad-SPI operation.
- **SSI_ADV_MODE_BI_READ** - The advanced mode of operation where data is read from the slave in Bi-SPI mode, with two bits of data read on every SSI clock.
- **SSI_ADV_MODE_BI_WRITE** - The advanced mode of operation where data is written to the slave in Bi-SPI mode, with two bits of data written on every SSI clock.
- **SSI_ADV_MODE_QUAD_READ** - The advanced mode of operation where data is read from the slave in Quad-SPI mode, with four bits of data read on every SSI clock.
- **SSI_ADV_MODE_QUAD_WRITE** - The advanced mode of operation where data is written to the slave in Quad-SPI mode, with four bits of data written on every SSI clock.

The following mode transitions are valid (other transitions produce undefined results):

FROM	TO						
	Legacy	Write	Read Write	Bi Read	Bi Write	Quad Read	Quad Write
Legacy	yes	yes	yes				
Write	yes	yes	yes	yes	yes	yes	yes
Read/Write	yes	yes	yes	yes	yes	yes	yes
Bi Read		yes	yes	yes	yes		
Bi write		yes	yes	yes	yes		
Quad read		yes	yes			yes	yes
Quad write		yes				yes	yes

When using an advanced mode of operation, the SSI module must have been configured for eight data bits and the **SSI_FRF_MOTO_MODE_0** protocol. The advanced mode operation that is selected applies only to data newly written into the FIFO; the data that is already present in the FIFO is handled using the advanced mode of operation in effect when that data was written.

Switching into and out of legacy mode should only occur when the FIFO is empty.

Note:

The availability of the advanced mode of SSI operation varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

None.

25.2.2.6 SSIBusy

Determines whether the SSI transmitter is busy or not.

Prototype:

```
bool
SSIBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SSI module.

Description:

This function allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the hardware shift register.

Returns:

Returns **true** if the SSI is transmitting or **false** if all transmissions are complete.

25.2.2.7 SSIClockSourceGet

Gets the data clock source for the specified SSI peripheral.

Prototype:

```
uint32_t  
SSIClockSourceGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the SSI module.

Description:

This function returns the data clock source for the specified SSI.

Note:

The ability to specify the SSI data clock source varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

Returns the current clock source, which is either **SSI_CLOCK_SYSTEM** or **SSI_CLOCK_PIOSC**.

25.2.2.8 SSIClockSourceSet

Sets the data clock source for the specified SSI peripheral.

Prototype:

```
void  
SSIClockSourceSet (uint32_t ui32Base,  
                   uint32_t ui32Source)
```

Parameters:

ui32Base is the base address of the SSI module.

ui32Source is the baud clock source for the SSI.

Description:

This function allows the baud clock source for the SSI to be selected. The possible clock source are the system clock (**SSI_CLOCK_SYSTEM**) or the precision internal oscillator (**SSI_CLOCK_PIOSC**).

Changing the baud clock source changes the data rate generated by the SSI. Therefore, the data rate should be reconfigured after any change to the SSI clock source.

Note:

The ability to specify the SSI baud clock source varies with the Tiva part and SSI in use. Please consult the data sheet for the part in use to determine whether this support is available.

Returns:

None.

25.2.2.9 SSIConfigSetExpClk

Configures the synchronous serial interface.

Prototype:

```
void
SSIConfigSetExpClk(uint32_t ui32Base,
                    uint32_t ui32SSIClk,
                    uint32_t ui32Protocol,
                    uint32_t ui32Mode,
                    uint32_t ui32BitRate,
                    uint32_t ui32DataWidth)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32SSIClk is the rate of the clock supplied to the SSI module.

ui32Protocol specifies the data transfer protocol.

ui32Mode specifies the mode of operation.

ui32BitRate specifies the clock rate.

ui32DataWidth specifies number of bits transferred per frame.

Description:

This function configures the synchronous serial interface. It sets the SSI protocol, mode of operation, bit rate, and data width.

The **ui32Protocol** parameter defines the data frame format. The **ui32Protocol** parameter can be one of the following values: **SSI_FRF_MOTO_MODE_0**, **SSI_FRF_MOTO_MODE_1**, **SSI_FRF_MOTO_MODE_2**, **SSI_FRF_MOTO_MODE_3**, **SSI_FRF_TI**, or **SSI_FRF_NMW**.

Note that the **SSI_FRF_NMW** option is only available on some devices. Refer to the device data sheet to determine if the Microwire format is supported on a particular device. The Motorola frame formats encode the following polarity and phase configurations:

Polarity	Phase	Mode
0	0	SSI_FRF_MOTO_MODE_0
0	1	SSI_FRF_MOTO_MODE_1
1	0	SSI_FRF_MOTO_MODE_2
1	1	SSI_FRF_MOTO_MODE_3

The **ui32Mode** parameter defines the operating mode of the SSI module. The SSI module can operate as a master or slave; if it is a slave, the SSI can be configured to disable output on its serial output line. The **ui32Mode** parameter can be one of the following values: **SSI_MODE_MASTER**, **SSI_MODE_SLAVE**, or **SSI_MODE_SLAVE_OD**.

The **ui32BitRate** parameter defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria:

- FSSI \geq 2 * bit rate (master mode)
- FSSI \geq 12 * bit rate (slave modes)

where FSSI is the frequency of the clock supplied to the SSI module. Note that there are frequency limits for FSSI that are described in the Bit Rate Generation section of the SSI chapter in the data sheet.

The *ui32DataWidth* parameter defines the width of the data transfers and can be a value between 4 and 16, inclusive.

The peripheral clock is the same as the processor clock. This value is returned by [SysCtlClockGet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtlClockGet\(\)](#)).

Returns:

None.

25.2.2.10 SSIDataGet

Gets a data element from the SSI receive FIFO.

Prototype:

```
void  
SSIDataGet (uint32_t ui32Base,  
            uint32_t *pui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

pui32Data is a pointer to a storage location for data that was received over the SSI interface.

Description:

This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the *pui32Data* parameter. If there is no data available, this function waits until data is received before returning.

Note:

Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SSICfgSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

Returns:

None.

25.2.2.11 SSIDataGetNonBlocking

Gets a data element from the SSI receive FIFO.

Prototype:

```
int32_t  
SSIDataGetNonBlocking (uint32_t ui32Base,  
                      uint32_t *pui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

pui32Data is a pointer to a storage location for data that was received over the SSI interface.

Description:

This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the *pui32Data* parameter. If there is no data in the FIFO, then this function returns a zero.

Note:

Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

Returns:

Returns the number of elements read from the SSI receive FIFO.

25.2.2.12 SSIDataPut

Puts a data element into the SSI transmit FIFO.

Prototype:

```
void
SSIDataPut(uint32_t ui32Base,
           uint32_t ui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32Data is the data to be transmitted over the SSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SSI module. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

Note:

The upper 32 - N bits of *ui32Data* are discarded by the hardware, where N is the data width as configured by [SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

Returns:

None.

25.2.2.13 SSIDataPutNonBlocking

Puts a data element into the SSI transmit FIFO.

Prototype:

```
int32_t
SSIDataPutNonBlocking(uint32_t ui32Base,
                      uint32_t ui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32Data is the data to be transmitted over the SSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function returns a zero.

Note:

The upper 32 - N bits of *ui32Data* are discarded by the hardware, where N is the data width as configured by [SSICfgSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

Returns:

Returns the number of elements written to the SSI transmit FIFO.

25.2.2.14 SSIDisable

Disables the synchronous serial interface.

Prototype:

```
void  
SSIDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the SSI module base address.

Description:

This function disables operation of the synchronous serial interface.

Returns:

None.

25.2.2.15 SSIDMADisable

Disables SSI DMA operation.

Prototype:

```
void  
SSIDMADisable(uint32_t ui32Base,  
               uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the SSI module.

ui32DMAFlags is a bit mask of the DMA features to disable.

Description:

This function is used to disable SSI DMA features that were enabled by [SSIDMAEnable\(\)](#). The specified SSI DMA features are disabled. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - disable DMA for receive
- SSI_DMA_TX - disable DMA for transmit

Returns:

None.

25.2.2.16 SSIDMAEnable

Enables SSI DMA operation.

Prototype:

```
void
SSIDMAEnable(uint32_t ui32Base,
             uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the SSI module.

ui32DMAFlags is a bit mask of the DMA features to enable.

Description:

This function enables the specified SSI DMA features. The SSI can be configured to use DMA for transmit and/or receive data transfers. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - enable DMA for receive
- SSI_DMA_TX - enable DMA for transmit

Note:

The uDMA controller must also be set up before DMA can be used with the SSI.

Returns:

None.

25.2.2.17 SSIEnable

Enables the synchronous serial interface.

Prototype:

```
void
SSIEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the SSI module base address.

Description:

This function enables operation of the synchronous serial interface. The synchronous serial interface must be configured before it is enabled.

Returns:

None.

25.2.2.18 SSIIIntClear

Clears SSI interrupt sources.

Prototype:

```
void  
SSIIIntClear(uint32_t ui32Base,  
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

This function clears the specified SSI interrupt sources so that they no longer assert. This function must be called in the interrupt handler to keep the interrupts from being triggered again immediately upon exit. The *ui32IntFlags* parameter can consist of either or both the **SSI_RXTO** and **SSI_RXOR** values.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

25.2.2.19 SSIIIntDisable

Disables individual SSI interrupt sources.

Prototype:

```
void  
SSIIIntDisable(uint32_t ui32Base,  
               uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32IntFlags is a bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated SSI interrupt sources. The *ui32IntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

25.2.2.20 SSIIIntEnable

Enables individual SSI interrupt sources.

Prototype:

```
void
SSIIIntEnable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base specifies the SSI module base address.

ui32IntFlags is a bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated SSI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The *ui32IntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

25.2.2.21 SSIIIntRegister

Registers an interrupt handler for the synchronous serial interface.

Prototype:

```
void
SSIIIntRegister(uint32_t ui32Base,
                 void (*pfnHandler) (void))
```

Parameters:

ui32Base specifies the SSI module base address.

pfnHandler is a pointer to the function to be called when the synchronous serial interface interrupt occurs.

Description:

This function registers the handler to be called when an SSI interrupt occurs. This function enables the global interrupt in the interrupt controller; specific SSI interrupts must be enabled via [SSIIIntEnable\(\)](#). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via [SSIIIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

25.2.2.22 SSIIIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t  
SSIIIntStatus(uint32_t ui32Base,  
              bool bMasked)
```

Parameters:

ui32Base specifies the SSI module base address.

bMasked is **false** if the raw interrupt status is required or **true** if the masked interrupt status is required.

Description:

This function returns the interrupt status for the SSI module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, and **SSI_RXOR**.

25.2.2.23 SSIIIntUnregister

Unregisters an interrupt handler for the synchronous serial interface.

Prototype:

```
void  
SSIIIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the SSI module base address.

Description:

This function clears the handler to be called when an SSI interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

25.3 Programming Example

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

```
char *pcChars = "SSI Master send data.";
int32_t i32Idx;

//
// Configure the SSI.
//
SSICfgSetExpClk(SSI_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
                 SSI_MODE_MASTER, 2000000, 8);

//
// Enable the SSI module.
//
SSIEnable(SSI_BASE);

//
// Send some data.
//
i32Idx = 0;
while(pcChars[i32Idx])
{
    SSIDataPut(SSI_BASE, pcChars[i32Idx]);
    i32Idx++;
}
```


26 Software CRC Module

Introduction	479
API Functions	479
Programming Example	482

26.1 Introduction

The CRC module provides functions to compute the CRC-8-CCITT and CRC-16 of a buffer of data. Support is provided for computing a running CRC, where a partial CRC is computed on one portion of the data, and then continued at a later time on another portion of the data. A running CRC is useful when computing the CRC on a stream of data that is coming in via a serial link for example).

A CRC is useful for detecting errors that occur during the transmission of data over a communications channel or during storage in a memory (such as flash). However, a CRC does not provide protection against an intentional modification or tampering of the data.

This module is contained in `driverlib/sw_crc.c`, with `driverlib/sw_crc.h` containing the API declarations for use by applications.

26.2 API Functions

Functions

- `uint16_t Crc16 (uint16_t ui16Crc, const uint8_t *pui8Data, uint32_t ui32Count)`
- `uint16_t Crc16Array (uint32_t ui32WordLen, const uint32_t *pui32Data)`
- `void Crc16Array3 (uint32_t ui32WordLen, const uint32_t *pui32Data, uint16_t *pui16Crc3)`
- `uint32_t Crc32 (uint32_t ui32Crc, const uint8_t *pui8Data, uint32_t ui32Count)`
- `uint8_t Crc8CCITT (uint8_t ui8Crc, const uint8_t *pui8Data, uint32_t ui32Count)`

26.2.1 Function Documentation

26.2.1.1 Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

```
uint16_t
Crc16(uint16_t ui16Crc,
      const uint8_t *pui8Data,
      uint32_t ui32Count)
```

Parameters:

ui16Crc is the starting CRC-16 value.

pui8Data is a pointer to the data buffer.

ui32Count is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-16 of the input buffer. The CRC-16 is computed in a running fashion, meaning that the entire data block that is to have its CRC-16 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then ***ui16Crc*** should be set to 0. If, however, the entire block of data is not available, then ***ui16Crc*** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as ***ui16Crc*** for the next portion of the data.

For example, to compute the CRC-16 of a block that has been split into three pieces, use the following:

```
ui16Crc = Crc16(0, pui8Data1, ui32Len1);
ui16Crc = Crc16(ui16Crc, pui8Data2, ui32Len2);
ui16Crc = Crc16(ui16Crc, pui8Data3, ui32Len3);
```

Computing a CRC-16 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The CRC-16 of the input data.

26.2.1.2 Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

```
uint16_t
Crc16Array(uint32_t ui32WordLen,
           const uint32_t *pui32Data)
```

Parameters:

ui32WordLen is the length of the array in words (the number of bytes divided by 4).
pui32Data is a pointer to the data buffer.

Description:

This function is a wrapper around the running CRC-16 function, providing the CRC-16 for a single block of data.

Returns:

The CRC-16 of the input data.

26.2.1.3 Crc16Array3

Calculates three CRC-16s of an array of words.

Prototype:

```
void
Crc16Array3(uint32_t ui32WordLen,
            const uint32_t *pui32Data,
            uint16_t *pui16Crc3)
```

Parameters:

ui32WordLen is the length of the array in words (the number of bytes divided by 4).

pui32Data is a pointer to the data buffer.

pui16Crc3 is a pointer to an array in which to place the three CRC-16 values.

Description:

This function is used to calculate three CRC-16s of the input buffer; the first uses every byte from the array, the second uses only the even-index bytes from the array (in other words, bytes 0, 2, 4, etc.), and the third uses only the odd-index bytes from the array (in other words, bytes 1, 3, 5, etc.).

Returns:

None

26.2.1.4 Crc32

Calculates the CRC-32 of an array of bytes.

Prototype:

```
uint32_t
Crc32(uint32_t ui32Crc,
      const uint8_t *pui8Data,
      uint32_t ui32Count)
```

Parameters:

ui32Crc is the starting CRC-32 value.

pui8Data is a pointer to the data buffer.

ui32Count is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-32 of the input buffer. The CRC-32 is computed in a running fashion, meaning that the entire data block that is to have its CRC-32 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ui32Crc** should be set to 0xFFFFFFFF. If, however, the entire block of data is not available, then **ui32Crc** should be set to 0xFFFFFFFF for the first portion of the data, and then the returned value should be passed back in as **ui32Crc** for the next portion of the data. Once all data has been passed to the function, the final CRC-32 can be obtained by inverting the last returned value.

For example, to compute the CRC-32 of a block that has been split into three pieces, use the following:

```
ui32Crc = Crc32(0xFFFFFFFF, pui8Data1, ui32Len1);
ui32Crc = Crc32(ui32Crc, pui8Data2, ui32Len2);
ui32Crc = Crc32(ui32Crc, pui8Data3, ui32Len3);
ui32Crc ^= 0xFFFFFFFF;
```

Computing a CRC-32 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The accumulated CRC-32 of the input data.

26.2.1.5 Crc8CCITT

Calculates the CRC-8-CCITT of an array of bytes.

Prototype:

```
uint8_t  
Crc8CCITT(uint8_t ui8Crc,  
           const uint8_t *pui8Data,  
           uint32_t ui32Count)
```

Parameters:

ui8Crc is the starting CRC-8-CCITT value.

pui8Data is a pointer to the data buffer.

ui32Count is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-8-CCITT of the input buffer. The CRC-8-CCITT is computed in a running fashion, meaning that the entire data block that is to have its CRC-8-CCITT computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ui8Crc** should be set to 0. If, however, the entire block of data is not available, then **ui8Crc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **ui8Crc** for the next portion of the data.

For example, to compute the CRC-8-CCITT of a block that has been split into three pieces, use the following:

```
ui8Crc = Crc8CCITT(0, pui8Data1, ui32Len1);  
ui8Crc = Crc8CCITT(ui8Crc, pui8Data2, ui32Len2);  
ui8Crc = Crc8CCITT(ui8Crc, pui8Data3, ui32Len3);
```

Computing a CRC-8-CCITT in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The CRC-8-CCITT of the input data.

26.3 Programming Example

The following example shows how to compute the CRC-16 of a buffer of data.

```
unsigned long ulIdx, ulValue;  
unsigned char pucData[256];  
  
//  
// Fill pucData with some data.  
//  
for(ulIdx = 0; ulIdx < 256; ulIdx++)  
{  
    pucData[ulIdx] = ulIdx;  
}  
  
//  
// Compute the CRC-16 of the data.  
//  
ulValue = Crc16(0, pucData, 256);
```

27 System Control

Introduction	483
API Functions	484
Programming Example	520

27.1 Introduction

System control determines the overall operation of the device. It controls the clocking of the device, the set of peripherals that are enabled, configuration of the device and its resets, and provides information about the device.

The members of the Tiva family have a varying peripheral set and memory sizes. The device has a set of read-only registers that indicate the size of the memories, the peripherals that are present, and the pins that are present for peripherals that have a varying number of pins. This information can be used to write adaptive software that can run on more than one member of the Tiva family.

The device can be clocked from several sources: an external oscillator, the main oscillator, the internal oscillator, the precision internal oscillator (PIOSC) or the PLL. The PLL can use any of the oscillators as its input. Because the internal oscillator has a very wide error range (+/- 50%), it cannot be used for applications that require specific timing; its real use is for detecting failures of the main oscillator and the PLL, and for applications that strictly respond to external events and do not use time-based peripherals (such as a UART). When using the PLL, the input clock frequency is constrained to specific frequencies that are specified in the device data sheet. When direct clocking with an external oscillator or the main oscillator, the frequency is constrained to between 0 Hz and 50 MHz (depending on the part). The frequency of the internal oscillator varies by device, with voltage, and with temperature. The internal oscillator provides no tuning or frequency measurement mechanism; its frequency is not adjustable.

Almost the entire device operates from a single clock. See the device data sheet for more information on how clocking for the various peripherals is configured.

Three modes of operation are supported by the Tiva family: run mode, sleep mode, and deep-sleep mode. In run mode, the processor is actively executing code. In sleep mode, the clocking of the device is unchanged but the processor no longer executes code (and is no longer clocked). In deep-sleep mode, the clocking of the device may change (depending upon the run mode clock configuration) and the processor no longer executes code (and is no longer clocked). An interrupt returns the device to run mode from one of the sleep modes; the sleep modes are entered upon request from the code.

The device has an internal LDO for generating the core power supply. On some devices, the output voltage of the LDO can be adjusted between 2.25 V and 2.75 V. Depending upon the application, lower voltage may be advantageous for its power savings, or higher voltage may be advantageous for its improved performance. The default setting of 2.5 V is a good compromise between the two, and should not be changed without careful consideration and evaluation.

There are several system events that, when detected, cause system control to reset the device. These events are a power-on, the input voltage dropping too low, an external reset, a software reset request, waking from hibernation, a watchdog timeout, a hardware system service request, and a main oscillator failure. The properties of some of these events can be configured, and the reason for a reset can be determined from system control. Not all of these reset causes are on all devices, see the device data sheet for more details.

Each peripheral in the device can be individually enabled, disabled, or reset. Additionally, the set of peripherals that remain enabled during sleep mode and deep-sleep mode can be configured, allowing custom sleep and deep-sleep modes to be defined. Care must be taken with deep-sleep mode, though, because in this mode, the PLL is no longer used and the system is clocked by the input crystal. Peripherals that depend on a particular input clock rate (such as a UART) require special consideration in deep-sleep mode due to a clock rate change; these peripherals must either be reconfigured upon entry to and exit from deep-sleep mode, or simply not enabled in deep-sleep mode. Some devices provide the option to clock some peripherals with the PIOSC, even while in deep-sleep mode so the peripheral clocking does not have to be reconfigured upon entry and exit.

There are various system events that, when detected, cause system control to generate a processor interrupt. These events are the PLL achieving lock, the internal LDO current limit being exceeded, the internal oscillator failing, the main oscillator failing, the input voltage dropping too low, the internal LDO voltage dropping too low, and the PLL failing. Not all of these interrupts are available on all Tiva devices, see the device data sheet for more details. Each of these interrupts can be individually enabled or disabled, and the sources must be cleared by the interrupt handler when they occur.

This driver is contained in `driverlib/sysctl.c`, with `driverlib/sysctl.h` containing the API declarations for use by applications.

27.2 API Functions

Functions

- void `SysCtlAltClkConfig` (uint32_t ui32Config)
- uint32_t `SysCtlClockFreqSet` (uint32_t ui32Config, uint32_t ui32SysClock)
- uint32_t `SysCtlClockGet` (void)
- void `SysCtlClockOutConfig` (uint32_t ui32Config, uint32_t ui32Div)
- void `SysCtlClockSet` (uint32_t ui32Config)
- void `SysCtlDeepSleep` (void)
- void `SysCtlDeepSleepClockConfigSet` (uint32_t ui32Div, uint32_t ui32Config)
- void `SysCtlDeepSleepClockSet` (uint32_t ui32Config)
- void `SysCtlDeepSleepPowerSet` (uint32_t ui32Config)
- void `SysCtlDelay` (uint32_t ui32Count)
- uint32_t `SysCtlFlashSectorSizeGet` (void)
- uint32_t `SysCtlFlashSizeGet` (void)
- void `SysCtlGPIOAHBDisable` (uint32_t ui32GPIOPeripheral)
- void `SysCtlGPIOAHBEnable` (uint32_t ui32GPIOPeripheral)
- void `SysCtlIntClear` (uint32_t ui32Ints)
- void `SysCtlIntDisable` (uint32_t ui32Ints)
- void `SysCtlIntEnable` (uint32_t ui32Ints)
- void `SysCtlIntRegister` (void (*pfnHandler)(void))
- uint32_t `SysCtlIntStatus` (bool bMasked)
- void `SysCtlIntUnregister` (void)
- uint32_t `SysCtlLDODeepSleepGet` (void)
- void `SysCtlLDODeepSleepSet` (uint32_t ui32Voltage)
- uint32_t `SysCtlLDOSleepGet` (void)

- void [SysCtlLDOSleepSet](#) (uint32_t ui32Voltage)
- void [SysCtlMOSCCConfigSet](#) (uint32_t ui32Config)
- void [SysCtlNMIClear](#) (uint32_t ui32Ints)
- uint32_t [SysCtlNMIStatus](#) (void)
- void [SysCtlPeripheralClockGating](#) (bool bEnable)
- void [SysCtlPeripheralDeepSleepDisable](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralDeepSleepEnable](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralDisable](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralEnable](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralPowerOff](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralPowerOn](#) (uint32_t ui32Peripheral)
- bool [SysCtlPeripheralPresent](#) (uint32_t ui32Peripheral)
- bool [SysCtlPeripheralReady](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralReset](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralSleepDisable](#) (uint32_t ui32Peripheral)
- void [SysCtlPeripheralSleepEnable](#) (uint32_t ui32Peripheral)
- uint32_t [SysCtlPIOSCCalibrate](#) (uint32_t ui32Type)
- uint32_t [SysCtlPWMClockGet](#) (void)
- void [SysCtlPWMClockSet](#) (uint32_t ui32Config)
- void [SysCtlReset](#) (void)
- uint32_t [SysCtlResetBehaviorGet](#) (void)
- void [SysCtlResetBehaviorSet](#) (uint32_t ui32Behavior)
- void [SysCtlResetCauseClear](#) (uint32_t ui32Causes)
- uint32_t [SysCtlResetCauseGet](#) (void)
- void [SysCtlSleep](#) (void)
- void [SysCtlSleepPowerSet](#) (uint32_t ui32Config)
- uint32_t [SysCtlSRAMSizeGet](#) (void)
- void [SysCtlUSBPLLDisable](#) (void)
- void [SysCtlUSBPLLEnable](#) (void)
- void [SysCtlVoltageEventClear](#) (uint32_t ui32Status)
- void [SysCtlVoltageEventConfig](#) (uint32_t ui32Config)
- uint32_t [SysCtlVoltageEventStatus](#) (void)

27.2.1 Detailed Description

The SysCtl API is broken up into eight groups of functions: those that provide device information, those that deal with device clocking, those that provide peripheral control, those that deal with the SysCtl interrupt, those that deal with the LDO, those that deal with sleep modes, those that deal with reset reasons, those that deal with the brown-out reset, and those that deal with clock verification timers.

Information about the device is provided by [SysCtlSRAMSizeGet\(\)](#), [SysCtlFlashSizeGet\(\)](#), and [SysCtlPeripheralPresent\(\)](#).

Clocking of the device is configured with [SysCtlClockSet\(\)](#) and [SysCtlPWMClockSet\(\)](#). Information about device clocking is provided by [SysCtlClockGet\(\)](#) and [SysCtlPWMClockGet\(\)](#).

Peripheral enabling and reset are controlled with [SysCtlPeripheralReset\(\)](#), [SysCtlPeripheralEnable\(\)](#), [SysCtlPeripheralDisable\(\)](#), [SysCtlPeripheralSleepEnable\(\)](#), [SysCtlPeripheralSleepDisable\(\)](#), [SysCtlPeripheralDeepSleepEnable\(\)](#), [SysCtlPeripheralDeepSleepDisable\(\)](#), and [SysCtlPeripheralClockGating\(\)](#).

The system control interrupt is managed with [SysCtlIntRegister\(\)](#), [SysCtlIntUnregister\(\)](#), [SysCtlIntEnable\(\)](#), [SysCtlIntDisable\(\)](#), [SysCtlIntClear\(\)](#), [SysCtlIntStatus\(\)](#).

The LDO is controlled with [SysCtlLDOSet\(\)](#) and [SysCtlLDOConfigSet\(\)](#). Its status is provided by [SysCtlLDOGet\(\)](#).

The device is put into sleep modes with [SysCtlSleep\(\)](#) and [SysCtlDeepSleep\(\)](#).

The reset reason is managed with [SysCtlResetCauseGet\(\)](#) and [SysCtlResetCauseClear\(\)](#). A software reset is performed with [SysCtlReset\(\)](#).

The brown-out reset is configured with [SysCtlBrownOutConfigSet\(\)](#).

The clock verification timers are managed with [SysCtlIOSCVerificationSet\(\)](#), [SysCtlMOSCVerificationSet\(\)](#), [SysCtlPLLVerificationSet\(\)](#), and [SysCtlClkVerificationClear\(\)](#).

27.2.2 Function Documentation

27.2.2.1 SysCtlAltClkConfig

Configures the alternate peripheral clock source.

Prototype:

```
void  
SysCtlAltClkConfig(uint32_t ui32Config)
```

Parameters:

ui32Config holds the configuration options for the alternate peripheral clock.

Description:

This function configures the alternate peripheral clock. The alternate peripheral clock is used to provide a known clock in all operating modes to peripherals that support using the alternate peripheral clock as an input clock. The *ui32Config* parameter value provides the clock input source using one of the following values:

- **SYSCTL_ALTCLK_PIOSC** - use the PIOSC as the alternate clock source (default).
- **SYSCTL_ALTCLK_RTCOSC** - use the Hibernate module RTC clock as the alternate clock source.
- **SYSCTL_ALTCLK_LFIOSC** - use the low-frequency internal oscillator as the alternate clock source.

Example: Select the Hibernate module RTC clock as the alternate clock source.

```
//  
// Select the Hibernate module RTC clock as the alternate clock source.  
//  
SysCtlAltClkConfig(SYSCTL_ALTCLK_RTCOSC);
```

Note:

The availability of the alternate peripheral clock varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.2 SysCtlClockFreqSet

Configures the system clock.

Prototype:

```
uint32_t
SysCtlClockFreqSet(uint32_t ui32Config,
                   uint32_t ui32SysClock)
```

Parameters:

ui32Config is the required configuration of the device clocking.

ui32SysClock is the requested processor frequency.

Description:

This function configures the main system clocking for the device. The input frequency, oscillator source, whether or not to enable the PLL, and the system clock divider are all configured with this function. This function configures the system frequency to the closest available divisor of one of the fixed PLL VCO settings provided in the *ui32Config* parameter. The caller sets the *ui32SysClock* parameter to request the system clock frequency, and this function then attempts to match this using the values provided in the *ui32Config* parameter. If this function cannot exactly match the requested frequency, it picks the closest frequency that is lower than the requested frequency. The *ui32Config* parameter provides the remaining configuration options using a set of defines that are a logical OR of several different values, many of which are grouped into sets where only one of the set can be chosen. This function returns the current system frequency which may not match the requested frequency.

The oscillator source is chosen with one of the following values:

- **SYSCTL_OSC_MAIN** to use an external crystal or oscillator.
- **SYSCTL_OSC_INT** to use the 16-MHz precision internal oscillator.
- **SYSCTL_OSC_INT30** to use the internal low frequency oscillator.
- **SYSCTL_OSC_EXT32** to use the hibernate modules 32.786-kHz oscillator. This option is only available on devices that include the hibernation module.

The system clock source is chosen with one of the following values:

- **SYSCTL_USE_PLL** is used to select the PLL output as the system clock.
- **SYSCTL_USE_OSC** is used to choose one of the oscillators as the system clock.

The PLL VCO frequency is chosen with one of the the following values:

- **SYSCTL_CFG_VCO_480** to set the PLL VCO output to 480-MHz
- **SYSCTL_CFG_VCO_320** to set the PLL VCO output to 320-MHz

Example: Configure the system clocking to be 40 MHz with a 320-MHz PLL setting using the 16-MHz internal oscillator.

```
SysCtlClockFreqSet(SYSCTL_OSC_INT | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_320,
40000000);
```

Note:

This function cannot be used with TM4C123 devices. For TM4C123 devices use the [SysCtlClockSet\(\)](#) function.

Returns:

The actual configured system clock frequency in Hz or zero if the value could not be changed due to a parameter error or PLL lock failure.

27.2.2.3 SysCtlClockGet

Gets the processor clock rate.

Prototype:

```
uint32_t  
SysCtlClockGet (void)
```

Description:

This function determines the clock rate of the processor clock, which is also the clock rate of the peripheral modules (with the exception of PWM, which has its own clock divider; other peripherals may have different clocking, see the device data sheet for details).

Note:

This cannot return accurate results if [SysCtlClockSet\(\)](#) has not been called to configure the clocking of the device, or if the device is directly clocked from a crystal (or a clock source) that is not one of the supported crystal frequencies. In the latter case, this function should be modified to directly return the correct system clock rate.

This function can only be called on TM4C123 devices. For TM4C129 devices, the return value from [SysCtlClockFreqSet\(\)](#) indicates the system clock frequency.

Returns:

The processor clock rate for TM4C123 devices only.

27.2.2.4 SysCtlClockOutConfig

Configures and enables or disables the clock output on the DIVSCLK pin.

Prototype:

```
void  
SysCtlClockOutConfig(uint32_t ui32Config,  
                      uint32_t ui32Div)
```

Parameters:

ui32Config holds the configuration options including enabling or disabling the clock output on the DIVSCLK pin.

ui32Div is the divisor for the clock selected in the *ui32Config* parameter.

Description:

This function selects the source for the DIVSCLK, enables or disables the clock output and provides an output divider value. The *ui32Div* parameter specifies the divider for the selected

clock source and has a valid range of 1-256. The *ui32Config* parameter configures the DIVSCLK output based on the following settings:

The first setting allows the output to be enabled or disabled.

- **SYSCTL_CLKOUT_EN** - enable the DIVSCLK output.
- **SYSCTL_CLKOUT_DIS** - disable the DIVSCLK output (default).

The next group of settings selects the source for the DIVSCLK.

- **SYSCTL_CLKOUT_SYSCLK** - use the current system clock as the source (default).
- **SYSCTL_CLKOUT_PIOSC** - use the PIOSC as the source.
- **SYSCTL_CLKOUT_MOSC** - use the MOSC as the source.

Example: Enable the PIOSC divided by 4 as the DIVSCLK output.

```
//  
// Enable the PIOSC divided by 4 as the DIVSCLK output.  
//  
SysCtlClockOutConfig(SYSCTL_DIVSCLK_EN | SYSCTL_DIVSCLK_SRC_PIOSC, 4);
```

Note:

The availability of the DIVSCLK output varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.5 SysCtlClockSet

Sets the clocking of the device.

Prototype:

```
void  
SysCtlClockSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required configuration of the device clocking.

Description:

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The *ui32Config* parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen.

The system clock divider is chosen with one of the following values: **SYSCTL_SYSDIV_1**, **SYSCTL_SYSDIV_2**, **SYSCTL_SYSDIV_3**, ... **SYSCTL_SYSDIV_64**.

The use of the PLL is chosen with either **SYSCTL_USE_PLL** or **SYSCTL_USE_OSC**.

The external crystal frequency is chosen with one of the following values: SYSCTL_XTAL_4MHZ ,	SYSCTL_XTAL_4_09MHZ ,	SYSCTL_XTAL_4_91MHZ ,
SYSCTL_XTAL_5MHZ ,	SYSCTL_XTAL_5_12MHZ ,	SYSCTL_XTAL_6MHZ ,
SYSCTL_XTAL_6_14MHZ ,	SYSCTL_XTAL_7_37MHZ ,	SYSCTL_XTAL_8MHZ ,
SYSCTL_XTAL_8_19MHZ ,	SYSCTL_XTAL_10MHZ ,	SYSCTL_XTAL_12MHZ ,

SYSCTL_XTAL_12_2MHZ, **SYSCTL_XTAL_13_5MHZ**, **SYSCTL_XTAL_14_3MHZ**,
SYSCTL_XTAL_16MHZ, **SYSCTL_XTAL_16_3MHZ**, **SYSCTL_XTAL_18MHZ**,
SYSCTL_XTAL_20MHZ, **SYSCTL_XTAL_24MHZ**, or **SYSCTL_XTAL_25MHZ**. Values
below **SYSCTL_XTAL_5MHZ** are not valid when the PLL is in operation.

The oscillator source is chosen with one of the following values: **SYSCTL_OSC_MAIN**, **SYSCTL_OSC_INT**, **SYSCTL_OSC_INT4**, **SYSCTL_OSC_INT30**, or **SYSCTL_OSC_EXT32**. **SYSCTL_OSC_EXT32** is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

The internal and main oscillators are disabled with the **SYSCTL_INT_OSC_DIS** and **SYSCTL_MAIN_OSC_DIS** flags, respectively. The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device is prevented by the hardware.

To clock the system from an external source (such as an external crystal oscillator), use **SYSCTL_USE_OSC | SYSCTL_OSC_MAIN**. To clock the system from the main oscillator, use **SYSCTL_USE_OSC | SYSCTL_OSC_MAIN**. To clock the system from the PLL, use **SYSCTL_USE_PLL | SYSCTL_OSC_MAIN**, and select the appropriate crystal with one of the **SYSCTL_XTAL_xxx** values.

Note:

This function should only be called on TM4C123 devices. For all other devices use the [SysCtlClockFreqSet\(\)](#) function.

If selecting the PLL as the system clock source (that is, via **SYSCTL_USE_PLL**), this function polls the PLL lock interrupt to determine when the PLL has locked. If an interrupt handler for the system control interrupt is in place, and it responds to and clears the PLL lock interrupt, this function delays until its timeout has occurred instead of completing as soon as PLL lock is achieved.

Returns:

None.

27.2.2.6 SysCtlDeepSleep

Puts the processor into deep-sleep mode.

Prototype:

```
void
SysCtlDeepSleep (void)
```

Description:

This function places the processor into deep-sleep mode; it does not return until the processor returns to run mode. The peripherals that are enabled via [SysCtlPeripheralDeepSleepEnable\(\)](#) continue to operate and can wake up the processor (if automatic clock gating is enabled with [SysCtlPeripheralClockGating\(\)](#), otherwise all peripherals continue to operate).

Returns:

None.

27.2.2.7 SysCtlDeepSleepClockConfigSet

Sets the clock configuration of the device while in deep-sleep mode.

Prototype:

```
void
SysCtlDeepSleepClockConfigSet(uint32_t ui32Div,
                               uint32_t ui32Config)
```

Parameters:

ui32Div is the clock divider when in deep-sleep mode.

ui32Config is the configuration of the device clocking while in deep-sleep mode.

Description:

This function configures the clocking of the device while in deep-sleep mode. The *ui32Config* parameter selects the oscillator and the *ui32Div* parameter sets the clock divider used in deep-sleep mode. The valid values for the *ui32Div* parameter range from 1 to 1024, however not all Tiva microcontrollers support this full range. This function replaces the [SysCtlDeepSleepClockSet\(\)](#) function and can be used on Tiva devices that support deep-sleep mode.

The oscillator source is chosen from one of the following values: **SYSCTL_DSPL_OSC_MAIN**, **SYSCTL_DSPL_OSC_INT**, **SYSCTL_DSPL_OSC_INT30**, or **SYSCTL_DSPL_OSC_EXT32**. The **SYSCTL_DSPL_OSC_EXT32** option is only available on devices with the hibernation module, and then only when the hibernation module is enabled.

The precision internal oscillator can be powered down in deep-sleep mode by specifying **SYSCTL_DSPL_PIOSC_PD**. The precision internal oscillator is not powered down if it is required for operation while in deep-sleep (based on other configuration settings).

The main oscillator can be powered down in deep-sleep mode by specifying **SYSCTL_DSPL_MOSC_PD**. The main oscillator is not powered down if it is required for operation while in deep-sleep (based on other configuration settings).

Note:

The availability of deep-sleep clocking configuration and the configuration values vary with the Tiva device in use. Please consult the data sheet for the device you are using to determine whether the desired configuration options are available and to determine the valid range for the clock divider.

Returns:

None.

27.2.2.8 SysCtlDeepSleepClockSet

Sets the clocking of the device while in deep-sleep mode.

Prototype:

```
void
SysCtlDeepSleepClockSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required configuration of the device clocking while in deep-sleep mode.

Description:

This function configures the clocking of the device while in deep-sleep mode. The oscillator to be used and the system clock divider are configured with this function.

The *ui32Config* parameter is the logical OR of the following values:

The system clock divider is chosen from one of the following values: **SYSCTL_DSPL_DIV_1**, **SYSCTL_DSPL_DIV_2**, **SYSCTL_DSPL_DIV_3**, ... **SYSCTL_DSPL_DIV_64**.

The oscillator source is chosen from one of the following values: **SYSCTL_DSPL_OSC_MAIN**, **SYSCTL_DSPL_OSC_INT**, **SYSCTL_DSPL_OSC_INT30**, or **SYSCTL_DSPL_OSC_EXT32**. **SYSCTL_OSC_EXT32** is only available on devices with the hibernation module, and then only when the hibernation module has been enabled.

The precision internal oscillator can be powered down in deep-sleep mode by specifying **SYSCTL_DSPL_PIOSC_PD**. The precision internal oscillator is not powered down if it is required for operation while in deep-sleep (based on other configuration settings.)

Note:

This function should only be called on TM4C123 devices. For other devices use the [SysCtlDeepSleepClockConfigSet\(\)](#) function.

The availability of deep-sleep clocking configuration varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

27.2.2.9 SysCtlDeepSleepPowerSet

Configures the power to the flash and SRAM while in deep-sleep mode.

Prototype:

```
void  
SysCtlDeepSleepPowerSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required flash and SRAM power configuration.

Description:

This function allows the power configuration of the flash and SRAM while in deep-sleep mode to be set. The *ui32Config* parameter is the logical OR of the flash power configuration and the SRAM power configuration.

The flash power configuration is specified as either:

- **SYSCTL_FLASH_NORMAL** - The flash is left in fully powered mode, providing fast wake-up time but higher power consumption.
- **SYSCTL_FLASH_LOW_POWER** - The flash is in low power mode, providing reduced power consumption but longer wake-up time.

The SRAM power configuration is specified as one of:

- **SYSCTL_LDO_SLEEP** - The LDO is in sleep mode.
- **SYSCTL_TEMP_LOW_POWER** - The temperature sensor in low power mode.

- **SYSCTL_SRAM_NORMAL** - The SRAM is left in fully powered mode, providing fast wake-up time but higher power consumption.
- **SYSCTL_SRAM_STANDBY** - The SRAM is placed into a lower power mode, providing reduced power consumption but longer wake-up time.
- **SYSCTL_SRAM_LOW_POWER** - The SRAM is placed into lowest power mode, providing further reduced power consumption but longer wake-up time.

Note:

The availability of this feature varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

27.2.2.10 SysCtlDelay



Provides a small delay.

Prototype:

```
void
SysCtlDelay(uint32_t ui32Count)
```

Parameters:

ui32Count is the number of delay loop iterations to perform.

Description:

This function provides a means of generating a delay by executing a simple 3 instruction cycle loop a given number of times. It is written in assembly to keep the loop instruction count consistent across tool chains.

It is important to note that this function does NOT provide an accurate timing mechanism. Although the delay loop is 3 instruction cycles long, the execution time of the loop will vary dramatically depending upon the application's interrupt environment (the loop will be interrupted unless run with interrupts disabled and this is generally an unwise thing to do) and also the current system clock rate and flash timings (wait states and the operation of the prefetch buffer affect the timing).

For better accuracy, the ROM version of this function may be used. This version will not suffer from flash- and prefetch buffer-related timing variability but will still be delayed by interrupt service routines.

For best accuracy, a system timer should be used with code either polling for a particular timer value being exceeded or processing the timer interrupt to determine when a particular time period has elapsed.

Returns:

None.

27.2.2.11 SysCtlFlashSectorSizeGet

Gets the size of a single eraseable sector of flash.

Prototype:

```
uint32_t  
SysCtlFlashSectorSizeGet (void)
```

Description:

This function determines the flash sector size on the Tiva device. This size determines the erase granularity of the device flash.

Returns:

The number of bytes in a single flash sector.

27.2.2.12 SysCtlFlashSizeGet

Gets the size of the flash.

Prototype:

```
uint32_t  
SysCtlFlashSizeGet (void)
```

Description:

This function determines the size of the flash on the Tiva device.

Returns:

The total number of bytes of flash.

27.2.2.13 SysCtlGPIOAHBDisable

Disables access to a GPIO peripheral via the AHB.

Prototype:

```
void  
SysCtlGPIOAHBDisable(uint32_t ui32GPIOPeripheral)
```

Parameters:

ui32GPIOPeripheral is the GPIO peripheral to disable.

Description:

This function disables the specified GPIO peripheral for access from the Advanced Host Bus (AHB). Once disabled, the GPIO peripheral is accessed from the legacy Advanced Peripheral Bus (APB).

The *ui32GPIOPeripheral* argument must be only one of the following values: **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, or **SYSCTL_PERIPH_GPIOJ**.

Note:

Some devices allow disabling AHB access to GPIO ports that are only present on the AHB. Disabling AHB access to these ports will disable access to these GPIO ports. On some devices, all GPIO ports are only available on AHB.

Returns:

None.

27.2.2.14 SysCtlGPIOAHBEnable

Enables access to a GPIO peripheral via the AHB.

Prototype:

```
void
SysCtlGPIOAHBEnable(uint32_t ui32GPIOPeripheral)
```

Parameters:

ui32GPIOPeripheral is the GPIO peripheral to enable.

Description:

This function is used to enable the specified GPIO peripheral to be accessed from the Advanced Host Bus (AHB) instead of the legacy Advanced Peripheral Bus (APB). When a GPIO peripheral is enabled for AHB access, the **_AHB_BASE** form of the base address should be used for GPIO functions. For example, instead of using **GPIO_PORTA_BASE** as the base address for GPIO functions, use **GPIO_PORTA_AHB_BASE** instead.

The *ui32GPIOPeripheral* argument must be only one of the following values:
SYSCTL_PERIPH_GPIOA, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**,
SYSCTL_PERIPH_GPIOD, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**,
SYSCTL_PERIPH_GPIOG, **SYSCTL_PERIPH_GPIOH**, or **SYSCTL_PERIPH_GPIOJ**.

Note:

On some devices, all GPIO ports are only available on AHB.

Returns:

None.

27.2.2.15 SysCtlIntClear

Clears system control interrupt sources.

Prototype:

```
void
SysCtlIntClear(uint32_t ui32Ints)
```

Parameters:

ui32Ints is a bit mask of the interrupt sources to be cleared. Must be a logical OR of **SYSCTL_INT_BOR0**, **SYSCTL_INT_VDDA_OK**, **SYSCTL_INT_MOSC_PUP**, **SYSCTL_INT_USBPLL_LOCK**, **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_BOR1**.

Description:

The specified system control interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep it from being called again immediately on exit.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to

do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

The interrupt sources vary based on the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.16 SysCtlIntDisable

Disables individual system control interrupt sources.

Prototype:

```
void  
SysCtlIntDisable(uint32_t ui32Ints)
```

Parameters:

ui32Ints is a bit mask of the interrupt sources to be disabled. Must be a logical OR of **SYSCTL_INT_BOR0**, **SYSCTL_INT_VDDA_OK**, **SYSCTL_INT_MOSC_PUP**, **SYSCTL_INT_USBPLL_LOCK**, **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_BOR1**.

Description:

This function disables the indicated system control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note:

The interrupt sources vary based on the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.17 SysCtlIntEnable

Enables individual system control interrupt sources.

Prototype:

```
void  
SysCtlIntEnable(uint32_t ui32Ints)
```

Parameters:

ui32Ints is a bit mask of the interrupt sources to be enabled. Must be a logical OR of **SYSCTL_INT_BOR0**, **SYSCTL_INT_VDDA_OK**, **SYSCTL_INT_MOSC_PUP**, **SYSCTL_INT_USBPLL_LOCK**, **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_BOR1**.

Description:

This function enables the indicated system control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note:

The interrupt sources vary based on the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.18 SysCtlIntRegister

Registers an interrupt handler for the system control interrupt.

Prototype:

```
void
SysCtlIntRegister(void (*pfnHandler) (void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the system control interrupt occurs.

Description:

This function registers the handler to be called when a system control interrupt occurs. This function enables the global interrupt in the interrupt controller; specific system control interrupts must be enabled via [SysCtlIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [SysCtlIntClear\(\)](#).

System control can generate interrupts when the PLL achieves lock, if the internal LDO current limit is exceeded, if the internal oscillator fails, if the main oscillator fails, if the internal LDO output voltage droops too much, if the external voltage droops too much, or if the PLL fails.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Note:

The events that cause system control interrupts vary based on the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.19 SysCtlIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t
SysCtlIntStatus(bool bMasked)
```

Parameters:

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the system controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Note:

The interrupt sources vary based on the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

The current interrupt status, enumerated as a bit field of `SYSCTL_INT_BOR0`, `SYSCTL_INT_VDDA_OK`, `SYSCTL_INT_MOSC_PUP`, `SYSCTL_INT_USBPLL_LOCK`, `SYSCTL_INT_PLL_LOCK`, `SYSCTL_INT_MOSC_FAIL`, `SYSCTL_INT_BOR`, and/or `SYSCTL_INT_BOR1`.

27.2.2.20 SysCtlIntUnregister

Unregisters the interrupt handler for the system control interrupt.

Prototype:

```
void  
SysCtlIntUnregister(void)
```

Description:

This function unregisters the handler to be called when a system control interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

27.2.2.21 SysCtlLDODeepSleepGet

Returns the output voltage of the LDO when the device enters deep-sleep mode.

Prototype:

```
uint32_t  
SysCtlLDODeepSleepGet(void)
```

Description:

This function returns the output voltage of the LDO when the device is in deep-sleep mode, as specified by the control register.

Note:

The availability of this feature, the default LDO voltage, and the adjustment range varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

Returns the deep-sleep-mode voltage of the LDO; is one of **SYSCTL_LDO_0_90V**, **SYSCTL_LDO_0_95V**, **SYSCTL_LDO_1_00V**, **SYSCTL_LDO_1_05V**, **SYSCTL_LDO_1_10V**, **SYSCTL_LDO_1_15V**, or **SYSCTL_LDO_1_20V**.

27.2.2.22 SysCtlLDODeepSleepSet

Sets the output voltage of the LDO when the device enters deep-sleep mode.

Prototype:

```
void
SysCtlLDODeepSleepSet (uint32_t ui32Voltage)
```

Parameters:

ui32Voltage is the required output voltage from the LDO while in deep-sleep mode.

Description:

This function sets the output voltage of the LDO while in deep-sleep mode. The *ui32Voltage* parameter specifies the output voltage of the LDO and must be one of the following values: **SYSCTL_LDO_0_90V**, **SYSCTL_LDO_0_95V**, **SYSCTL_LDO_1_00V**, **SYSCTL_LDO_1_05V**, **SYSCTL_LDO_1_10V**, **SYSCTL_LDO_1_15V**, or **SYSCTL_LDO_1_20V**.

Note:

The availability of this feature, the default LDO voltage, and the adjustment range varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

27.2.2.23 SysCtlLDOSleepGet

Returns the output voltage of the LDO when the device enters sleep mode.

Prototype:

```
uint32_t
SysCtlLDOSleepGet (void)
```

Description:

This function determines the output voltage of the LDO while in sleep mode, as specified by the control register.

Note:

The availability of this feature, the default LDO voltage, and the adjustment range varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

Returns the sleep-mode voltage of the LDO and is one of **SYSCTL_LDO_0_90V**, **SYSCTL_LDO_0_95V**, **SYSCTL_LDO_1_00V**, **SYSCTL_LDO_1_05V**, **SYSCTL_LDO_1_10V**, **SYSCTL_LDO_1_15V**, or **SYSCTL_LDO_1_20V**.

27.2.2.24 SysCtlLDOSleepSet

Sets the output voltage of the LDO when the device enters sleep mode.

Prototype:

```
void  
SysCtlLDOSleepSet(uint32_t ui32Voltage)
```

Parameters:

ui32Voltage is the required output voltage from the LDO while in sleep mode.

Description:

This function sets the output voltage of the LDO while in sleep mode. The *ui32Voltage* parameter must be one of the following values: **SYSCTL_LDO_0_90V**, **SYSCTL_LDO_0_95V**, **SYSCTL_LDO_1_00V**, **SYSCTL_LDO_1_05V**, **SYSCTL_LDO_1_10V**, **SYSCTL_LDO_1_15V**, or **SYSCTL_LDO_1_20V**.

Note:

The availability of this feature, the default LDO voltage, and the adjustment range varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

27.2.2.25 SysCtlMOSCConfigSet

Sets the configuration of the main oscillator (MOSC) control.

Prototype:

```
void  
SysCtlMOSCConfigSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required configuration of the MOSC control.

Description:

This function configures the control of the main oscillator. The *ui32Config* is specified as the logical OR of the following values:

- **SYSCTL_MOSC_VALIDATE** enables the MOSC verification circuit that detects a failure of the main oscillator (such as a loss of the clock).
- **SYSCTL_MOSC_INTERRUPT** indicates that a MOSC failure should generate an interrupt instead of resetting the processor.
- **SYSCTL_MOSC_NO_XTAL** indicates that there is no crystal or oscillator connected to the OSC0/OSC1 pins, allowing power consumption to be reduced.
- **SYSCTL_MOSC_PWR_DIS** disable power to the main oscillator. If this parameter is not specified, the MOSC input remains powered.
- **SYSCTL_MOSC_LOWFREQ** MOSC is less than 10 MHz.
- **SYSCTL_MOSC_HIGHFREQ** MOSC is greater than 10 MHz.
- **SYSCTL_MOSC_SESRC** specifies that the MOSC is a single-ended oscillator connected to OSC0. If this parameter is not specified, the input is assumed to be a crystal.

Note:

The availability of MOSC control varies based on the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available. In addition, the capability of MOSC control varies based on the Tiva part in use.

Returns:

None.

27.2.2.26 SysCtlNMIClear

Clears NMI sources.

Prototype:

```
void
SysCtlNMIClear(uint32_t ui32Ints)
```

Parameters:

ui32Ints is a bit mask of the non-maskable interrupt sources.

Description:

This function clears the current NMI status specified in the *ui32Ints* parameter. The valid values for the *ui32Ints* parameter are a logical OR of the following values:

- **SYSCTL_NMI_MOSCFAIL** the main oscillator is not present or did not start.
- **SYSCTL_NMI_TAMPER** a tamper event has been detected.
- **SYSCTL_NMI_WDT0** watchdog 0 generated a timeout.
- **SYSCTL_NMI_WDT1** watchdog 1 generated a timeout.
- **SYSCTL_NMI_POWER** a power event occurred.
- **SYSCTL_NMI_EXTERNAL** an external NMI pin asserted.

Example: Clear all current NMI status flags.

```
//
// Clear all the current NMI sources.
//
SysCtlNMIClear(SysCtlNMIStatus());
```

Note:

The availability of the NMI status varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.27 SysCtlNMIStatus

Returns the current NMI status.

Prototype:

```
uint32_t
SysCtlNMIStatus(void)
```

Description:

This function returns the NMI status for the system controller. The valid values for the *ui32Ints* parameter are a logical OR of the following values:

- **SYSCTL_NMI_MOSCFAIL** the main oscillator is not present or did not start.
- **SYSCTL_NMI_TAMPER** a tamper event has been detected.
- **SYSCTL_NMI_WDT0** watchdog 0 generated a timeout.
- **SYSCTL_NMI_WDT1** watchdog 1 generated a timeout.
- **SYSCTL_NMI_POWER** a power event occurred.
- **SYSCTL_NMI_EXTERNAL** an external NMI pin asserted.

Example: Clear all current NMI status flags.

```
//  
// Clear all the current NMI sources.  
//  
SysCtlNMIClear(SysCtlNMIStatus());
```

Note:

The availability of the NMI status varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

The current NMI status.

27.2.2.28 SysCtlPeripheralClockGating

Controls peripheral clock gating in sleep and deep-sleep mode.

Prototype:

```
void  
SysCtlPeripheralClockGating(bool bEnable)
```

Parameters:

bEnable is a boolean that is **true** if the sleep and deep-sleep peripheral configuration should be used and **false** if not.

Description:

This function controls how peripherals are clocked when the processor goes into sleep or deep-sleep mode. By default, the peripherals are clocked the same as in run mode; if peripheral clock gating is enabled, they are clocked according to the configuration set by [SysCtlPeripheralSleepEnable\(\)](#), [SysCtlPeripheralSleepDisable\(\)](#), [SysCtlPeripheralDeepSleepEnable\(\)](#), and [SysCtlPeripheralDeepSleepDisable\(\)](#).

Returns:

None.

27.2.2.29 SysCtlPeripheralDeepSleepDisable

Disables a peripheral in deep-sleep mode.

Prototype:

```
void
SysCtlPeripheralDeepSleepDisable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to disable in deep-sleep mode.

Description:

This function causes a peripheral to stop operating when the processor goes into deep-sleep mode. Disabling peripherals while in deep-sleep mode helps to lower the current draw of the device, and can keep peripherals that require a particular clock frequency from operating when the clock changes as a result of entering deep-sleep mode. If enabled (via [SysCtlPeripheralEnable\(\)](#)), the peripheral automatically resumes operation when the processor leaves deep-sleep mode, maintaining its entire state from before deep-sleep mode was entered.

Deep-sleep mode clocking of peripherals must be enabled via [SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_CAN1,
SYSCTL_PERIPH_COMP0,	SYSCTL_PERIPH_EEPROM0,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EPIO,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C1,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C4,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C7,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_LCD0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_PWM1,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI0,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI3,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER2,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER5,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART0,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART3,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART6,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_USB0,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER0,
SYSCTL_PERIPH_WTIMER2,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER4,	SYSCTL_PERIPH_WTIMER3,
	SYSCTL_PERIPH_WTIMER5

Returns:

None.

27.2.2.30 SysCtlPeripheralDeepSleepEnable

Enables a peripheral in deep-sleep mode.

Prototype:

```
void  
SysCtlPeripheralDeepSleepEnable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to enable in deep-sleep mode.

Description:

This function allows a peripheral to continue operating when the processor goes into deep-sleep mode. Because the clocking configuration of the device may change, not all peripherals can safely continue operating while the processor is in deep-sleep mode. Those that must run at a particular frequency (such as a UART) do not work as expected if the clock changes. It is the responsibility of the caller to make sensible choices.

Deep-sleep mode clocking of peripherals must be enabled via [SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_CAN1,
SYSCTL_PERIPH_COMPO,	SYSCTL_PERIPH_EEPROM0,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EPIO,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C1,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C4,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C7,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_LCD0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_PWM1,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI0,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI3,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER2,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER5,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART0,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART3,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART6,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_USB0,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER0,
SYSCTL_PERIPH_WTIMER2,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER4,	SYSCTL_PERIPH_WTIMER3,
	SYSCTL_PERIPH_WTIMER5

Returns:

None.

27.2.2.31 SysCtlPeripheralDisable

Disables a peripheral.

Prototype:

```
void
SysCtlPeripheralDisable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to disable.

Description:

This function disables a peripheral. Once disabled, they do not operate or respond to register reads/writes.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_CAN1,
SYSCTL_PERIPH_COMP0,	SYSCTL_PERIPH_EEPROM0,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EPIO,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C1,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C4,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_I2C7,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_LCD0,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_PWM1,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI0,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_SSI3,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER2,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_TIMER5,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART0,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART3,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_UART6,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_USB0,
SYSCTL_PERIPH_WTIMER2,	SYSCTL_PERIPH_WTIMER0,
SYSCTL_PERIPH_WTIMER4,	SYSCTL_PERIPH_WTIMER1,
	SYSCTL_PERIPH_WTIMER3,
	SYSCTL_PERIPH_WTIMER5

Returns:

None.

27.2.2.32 SysCtlPeripheralEnable

Enables a peripheral.

Prototype:

```
void
SysCtlPeripheralEnable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to enable.

Description:

This function enables a peripheral. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_CAN1,	SYSCTL_PERIPH_CCM0,
SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_EEPROM0,	SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_COMP0,	SYSCTL_PERIPH_EPIO,	SYSCTL_PERIPH_GPIOA,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_GPIOC,	SYSCTL_PERIPH_GPIOD,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOF,	SYSCTL_PERIPH_GPIOG,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOJ,	SYSCTL_PERIPH_GPIOK,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOM,	SYSCTL_PERIPH_GPION,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOQ,	SYSCTL_PERIPH_GPIOR,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOT,	SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_I2C1,	SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C4,	SYSCTL_PERIPH_I2C5,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C7,	SYSCTL_PERIPH_I2C8,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_LCD0,	SYSCTL_PERIPH_ONEWIRE0,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_PWM1,	SYSCTL_PERIPH_QEI0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_SSI0,	SYSCTL_PERIPH_SSI1,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI3,	SYSCTL_PERIPH_TIMER0,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_TIMER2,	SYSCTL_PERIPH_TIMER3,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER5,	SYSCTL_PERIPH_TIMER6,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_UART0,	SYSCTL_PERIPH_UART1,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART3,	SYSCTL_PERIPH_UART4,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART6,	SYSCTL_PERIPH_UART7,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_USB0,	SYSCTL_PERIPH_WDOG0,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_WTIMER0,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER2,	SYSCTL_PERIPH_WTIMER3,
SYSCTL_PERIPH_WTIMER4,	SYSCTL_PERIPH_WTIMER4,	SYSCTL_PERIPH_WTIMER5

Note:

It takes five clock cycles after the write to enable a peripheral before the the peripheral is actually enabled. During this time, attempts to access the peripheral result in a bus fault. Care should be taken to ensure that the peripheral is not accessed during this brief time period.

Returns:

None.

27.2.2.33 SysCtlPeripheralPowerOff

Powers off a peripheral.

Prototype:

```
void  
SysCtlPeripheralPowerOff(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to be powered off.

Description:

This function allows the power to a peripheral to be turned off. The peripheral continues to receive power when its clock is enabled, but the power is removed when its clock is disabled.

The *ui32Peripheral* parameter must be only one of the following values:
SYSCTL_PERIPH_CAN0, SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY, SYSCTL_PERIPH_LCD0, SYSCTL_PERIPH_USB0

Note:

The ability to power off a peripheral varies based on the Tiva part in use. Please consult the data sheet for the part you are using to determine if this feature is available.

Returns:

None.

27.2.2.34 SysCtlPeripheralPowerOn

Powers on a peripheral.

Prototype:

```
void  
SysCtlPeripheralPowerOn(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to be powered on.

Description:

This function turns on the power to a peripheral. The peripheral continues to receive power even when its clock is not enabled.

The *ui32Peripheral* parameter must be only one of the following values:
SYSCTL_PERIPH_CAN0, SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY, SYSCTL_PERIPH_LCD0, SYSCTL_PERIPH_USB0

Note:

The ability to power off a peripheral varies based on the Tiva part in use. Please consult the data sheet for the part you are using to determine if this feature is available.

Returns:

None.

27.2.2.35 SysCtlPeripheralPresent

Determines if a peripheral is present.

Prototype:

```
bool  
SysCtlPeripheralPresent(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral in question.

Description:

This function determines if a particular peripheral is present in the device. Each member of the Tiva family has a different peripheral set; this function determines which peripherals are present on this device.

The `ui32Peripheral` parameter must be one of the following values:

- `SYSCTL_PERIPH_ADC0`, `SYSCTL_PERIPH_ADC1`,
- `SYSCTL_PERIPH_CAN0`, `SYSCTL_PERIPH_CAN1`,
- `SYSCTL_PERIPH_COMP0`, `SYSCTL_PERIPH_EEPROM0`,
- `SYSCTL_PERIPH_EPHY`, `SYSCTL_PERIPH_EPIO0`,
- `SYSCTL_PERIPH_GPIOB`, `SYSCTL_PERIPH_GPIOC`,
- `SYSCTL_PERIPH_GPIOE`, `SYSCTL_PERIPH_GPIOF`,
- `SYSCTL_PERIPH_GPIOH`, `SYSCTL_PERIPH_GPIOJ`,
- `SYSCTL_PERIPH_GPIOL`, `SYSCTL_PERIPH_GPIOM`,
- `SYSCTL_PERIPH_GPIOP`, `SYSCTL_PERIPH_GPIOQ`,
- `SYSCTL_PERIPH_GPIOS`, `SYSCTL_PERIPH_GPIOT`, `SYSCTL_PERIPH_HIBERNATE`,
- `SYSCTL_PERIPH_I2C0`, `SYSCTL_PERIPH_I2C1`, `SYSCTL_PERIPH_I2C2`,
- `SYSCTL_PERIPH_I2C3`, `SYSCTL_PERIPH_I2C4`, `SYSCTL_PERIPH_I2C5`,
- `SYSCTL_PERIPH_I2C6`, `SYSCTL_PERIPH_I2C7`, `SYSCTL_PERIPH_I2C8`,
- `SYSCTL_PERIPH_I2C9`, `SYSCTL_PERIPH_LCD0`, `SYSCTL_PERIPH_ONEWIRE0`,
- `SYSCTL_PERIPH_PWM0`, `SYSCTL_PERIPH_PWM1`, `SYSCTL_PERIPH_QEI0`,
- `SYSCTL_PERIPH_QEI1`, `SYSCTL_PERIPH_SSI0`, `SYSCTL_PERIPH_SSI1`,
- `SYSCTL_PERIPH_SSI2`, `SYSCTL_PERIPH_SSI3`, `SYSCTL_PERIPH_TIMER0`,
- `SYSCTL_PERIPH_TIMER1`, `SYSCTL_PERIPH_TIMER2`, `SYSCTL_PERIPH_TIMER3`,
- `SYSCTL_PERIPH_TIMER4`, `SYSCTL_PERIPH_TIMER5`, `SYSCTL_PERIPH_TIMER6`,
- `SYSCTL_PERIPH_TIMER7`, `SYSCTL_PERIPH_UART0`, `SYSCTL_PERIPH_UART1`,
- `SYSCTL_PERIPH_UART2`, `SYSCTL_PERIPH_UART3`, `SYSCTL_PERIPH_UART4`,
- `SYSCTL_PERIPH_UART5`, `SYSCTL_PERIPH_UART6`, `SYSCTL_PERIPH_UART7`,
- `SYSCTL_PERIPH_UDMA`, `SYSCTL_PERIPH_USB0`, `SYSCTL_PERIPH_WDOG0`,
- `SYSCTL_PERIPH_WDOG1`, `SYSCTL_PERIPH_WTIMER0`, `SYSCTL_PERIPH_WTIMER1`,
- `SYSCTL_PERIPH_WTIMER2`, `SYSCTL_PERIPH_WTIMER3`,
- `SYSCTL_PERIPH_WTIMER4`, or `SYSCTL_PERIPH_WTIMER5`

Returns:

Returns **true** if the specified peripheral is present and **false** if it is not.

27.2.2.36 SysCtlPeripheralReady

Determines if a peripheral is ready.

Prototype:

```
bool SysCtlPeripheralReady(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral in question.

Description:

This function determines if a particular peripheral is ready to be accessed. The peripheral may be in a non-ready state if it is not enabled, is being held in reset, or is in the process of becoming ready after being enabled or taken out of reset.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_CAN1,	SYSCTL_PERIPH_EEPROM0,	SYSCTL_PERIPH_CCM0,
SYSCTL_PERIPH_EEPROM0,	SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_GPIOA,	SYSCTL_PERIPH_GPIOA,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,	SYSCTL_PERIPH_GPIOB,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,	SYSCTL_PERIPH_GPIOG,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,	SYSCTL_PERIPH_GPIOK,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,	SYSCTL_PERIPH_GPION,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,	SYSCTL_PERIPH_GPIOR,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,	SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C1,	SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C4,	SYSCTL_PERIPH_I2C5,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C7,	SYSCTL_PERIPH_I2C8,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_LCD0,	SYSCTL_PERIPH_ONEWIRE0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_PWM1,	SYSCTL_PERIPH_QEI0,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI0,	SYSCTL_PERIPH_SSI1,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI3,	SYSCTL_PERIPH_TIMERO,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER2,	SYSCTL_PERIPH_TIMER3,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER5,	SYSCTL_PERIPH_TIMER6,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART0,	SYSCTL_PERIPH_UART1,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART3,	SYSCTL_PERIPH_UART4,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART6,	SYSCTL_PERIPH_UART7,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_USB0,	SYSCTL_PERIPH_WDOG0,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER0,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER2,		SYSCTL_PERIPH_WTIMER3,
SYSCTL_PERIPH_WTIMER4,		SYSCTL_PERIPH_WTIMER5,

Note:

The ability to check for a peripheral being ready varies based on the Tiva part in use. Please consult the data sheet for the part you are using to determine if this feature is available.

Returns:

Returns **true** if the specified peripheral is ready and **false** if it is not.

27.2.2.37 SysCtlPeripheralReset

Performs a software reset of a peripheral.

Prototype:

```
void
SysCtlPeripheralReset(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to reset.

Description:

This function performs a software reset of the specified peripheral. An individual peripheral reset signal is asserted for a brief period and then de-asserted, returning the internal state of the peripheral to its reset condition.

The *ui32Peripheral* parameter must be only one of the following values:

SYSCTL_PERIPH_ADC0,	SYSCTL_PERIPH_ADC1,
----------------------------	----------------------------

SYSCTL_PERIPH_CAN0, SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_CCM0,
SYSCTL_PERIPH_COM0, SYSCTL_PERIPH_EEPROM0, SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY, SYSCTL_PERIPH_EPIO, SYSCTL_PERIPH_GPIOA,
SYSCTL_PERIPH_GPIOB, SYSCTL_PERIPH_GPIOC, SYSCTL_PERIPH_GPIOB,
SYSCTL_PERIPH_GPIOE, SYSCTL_PERIPH_GPIOF, SYSCTL_PERIPH_GPIOG,
SYSCTL_PERIPH_GPIOH, SYSCTL_PERIPH_GPIOJ, SYSCTL_PERIPH_GPIOK,
SYSCTL_PERIPH_GPIOL, SYSCTL_PERIPH_GPIOM, SYSCTL_PERIPH_GPIOI,
SYSCTL_PERIPH_GPIOP, SYSCTL_PERIPH_GPIOQ, SYSCTL_PERIPH_GPIOR,
SYSCTL_PERIPH_GPIOS, SYSCTL_PERIPH_GPIOT, SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_I2C0, SYSCTL_PERIPH_I2C1, SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_I2C3, SYSCTL_PERIPH_I2C4, SYSCTL_PERIPH_I2C5,
SYSCTL_PERIPH_I2C6, SYSCTL_PERIPH_I2C7, SYSCTL_PERIPH_I2C8,
SYSCTL_PERIPH_I2C9, SYSCTL_PERIPH_LCD0, SYSCTL_PERIPH_ONEWIRE0,
SYSCTL_PERIPH_PWM0, SYSCTL_PERIPH_PWM1, SYSCTL_PERIPH_QEI0,
SYSCTL_PERIPH_QEI1, SYSCTL_PERIPH_SSI0, SYSCTL_PERIPH_SSI1,
SYSCTL_PERIPH_SSI2, SYSCTL_PERIPH_SSI3, SYSCTL_PERIPH_TIMER0,
SYSCTL_PERIPH_TIMER1, SYSCTL_PERIPH_TIMER2, SYSCTL_PERIPH_TIMER3,
SYSCTL_PERIPH_TIMER4, SYSCTL_PERIPH_TIMER7, SYSCTL_PERIPH_TIMER6,
SYSCTL_PERIPH_TIMER5, SYSCTL_PERIPH_UART0, SYSCTL_PERIPH_UART1,
SYSCTL_PERIPH_UART2, SYSCTL_PERIPH_UART3, SYSCTL_PERIPH_UART4,
SYSCTL_PERIPH_UART5, SYSCTL_PERIPH_UART6, SYSCTL_PERIPH_UART7,
SYSCTL_PERIPH_UDMA, SYSCTL_PERIPH_USB0, SYSCTL_PERIPH_WDOG0,
SYSCTL_PERIPH_WDOG1, SYSCTL_PERIPH_WTIMER0, SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER2, SYSCTL_PERIPH_WTIMER3,
SYSCTL_PERIPH_WTIMER4, or SYSCTL_PERIPH_WTIMER5

Returns:

None.

27.2.2.38 SysCtlPeripheralSleepDisable

Disables a peripheral in sleep mode.

Prototype:

```
void  
SysCtlPeripheralSleepDisable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to disable in sleep mode.

Description:

This function causes a peripheral to stop operating when the processor goes into sleep mode. Disabling peripherals while in sleep mode helps to lower the current draw of the device. If enabled (via [SysCtlPeripheralEnable\(\)](#)), the peripheral automatically resumes operation when the processor leaves sleep mode, maintaining its entire state from before sleep mode was entered.

Sleep mode clocking of peripherals must be enabled via [SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The *ui32Peripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**,

SYSCTL_PERIPH_CAN0,	SYSCTL_PERIPH_CAN1,	SYSCTL_PERIPH_CCM0,
SYSCTL_PERIPH_COMPO,	SYSCTL_PERIPH_EEPROM0,	SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EPIO,	SYSCTL_PERIPH_GPIOA,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,	SYSCTL_PERIPH_GPIOD,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,	SYSCTL_PERIPH_GPIOG,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,	SYSCTL_PERIPH_GPIOK,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,	SYSCTL_PERIPH_GPION,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,	SYSCTL_PERIPH_GPIOR,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,	SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C1,	SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C4,	SYSCTL_PERIPH_I2C5,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C7,	SYSCTL_PERIPH_I2C8,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_LCD0,	SYSCTL_PERIPH_ONEWIRE0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_PWM1,	SYSCTL_PERIPH_QEI0,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI0,	SYSCTL_PERIPH_SSI1,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI3,	SYSCTL_PERIPH_TIMER0,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER2,	SYSCTL_PERIPH_TIMER3,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER5,	SYSCTL_PERIPH_TIMER6,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART0,	SYSCTL_PERIPH_UART1,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART3,	SYSCTL_PERIPH_UART4,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART6,	SYSCTL_PERIPH_UART7,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_USB0,	SYSCTL_PERIPH_WDOG0,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER0,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER2,		SYSCTL_PERIPH_WTIMER3,
SYSCTL_PERIPH_WTIMER4, or SYSCTL_PERIPH_WTIMER5		

Returns:

None.

27.2.2.39 SysCtlPeripheralSleepEnable

Enables a peripheral in sleep mode.

Prototype:

```
void
SysCtlPeripheralSleepEnable(uint32_t ui32Peripheral)
```

Parameters:*ui32Peripheral* is the peripheral to enable in sleep mode.**Description:**

This function allows a peripheral to continue operating when the processor goes into sleep mode. Because the clocking configuration of the device does not change, any peripheral can safely continue operating while the processor is in sleep mode and can therefore wake the processor from sleep mode.

Sleep mode clocking of peripherals must be enabled via [SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The *ui32Peripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CCM0**,

SYSCTL_PERIPH_COMPO,	SYSCTL_PERIPH_EEPROM0,	SYSCTL_PERIPH_EMAC,
SYSCTL_PERIPH_EPHY,	SYSCTL_PERIPH_EPIO,	SYSCTL_PERIPH_GPIOA,
SYSCTL_PERIPH_GPIOB,	SYSCTL_PERIPH_GPIOC,	SYSCTL_PERIPH_GPIOD,
SYSCTL_PERIPH_GPIOE,	SYSCTL_PERIPH_GPIOF,	SYSCTL_PERIPH_GPIOG,
SYSCTL_PERIPH_GPIOH,	SYSCTL_PERIPH_GPIOJ,	SYSCTL_PERIPH_GPIOK,
SYSCTL_PERIPH_GPIOL,	SYSCTL_PERIPH_GPIOM,	SYSCTL_PERIPH_GPION,
SYSCTL_PERIPH_GPIOP,	SYSCTL_PERIPH_GPIOQ,	SYSCTL_PERIPH_GPIOR,
SYSCTL_PERIPH_GPIOS,	SYSCTL_PERIPH_GPIOT,	SYSCTL_PERIPH_HIBERNATE,
SYSCTL_PERIPH_I2C0,	SYSCTL_PERIPH_I2C1,	SYSCTL_PERIPH_I2C2,
SYSCTL_PERIPH_I2C3,	SYSCTL_PERIPH_I2C4,	SYSCTL_PERIPH_I2C5,
SYSCTL_PERIPH_I2C6,	SYSCTL_PERIPH_I2C7,	SYSCTL_PERIPH_I2C8,
SYSCTL_PERIPH_I2C9,	SYSCTL_PERIPH_LCD0,	SYSCTL_PERIPH_ONEWIRE0,
SYSCTL_PERIPH_PWM0,	SYSCTL_PERIPH_PWM1,	SYSCTL_PERIPH_QEIO,
SYSCTL_PERIPH_QEI1,	SYSCTL_PERIPH_SSI0,	SYSCTL_PERIPH_SSI1,
SYSCTL_PERIPH_SSI2,	SYSCTL_PERIPH_SSI3,	SYSCTL_PERIPH_TIMER0,
SYSCTL_PERIPH_TIMER1,	SYSCTL_PERIPH_TIMER2,	SYSCTL_PERIPH_TIMER3,
SYSCTL_PERIPH_TIMER4,	SYSCTL_PERIPH_TIMER5,	SYSCTL_PERIPH_TIMER6,
SYSCTL_PERIPH_TIMER7,	SYSCTL_PERIPH_UART0,	SYSCTL_PERIPH_UART1,
SYSCTL_PERIPH_UART2,	SYSCTL_PERIPH_UART3,	SYSCTL_PERIPH_UART4,
SYSCTL_PERIPH_UART5,	SYSCTL_PERIPH_UART6,	SYSCTL_PERIPH_UART7,
SYSCTL_PERIPH_UDMA,	SYSCTL_PERIPH_USB0,	SYSCTL_PERIPH_WDOG0,
SYSCTL_PERIPH_WDOG1,	SYSCTL_PERIPH_WTIMER0,	SYSCTL_PERIPH_WTIMER1,
SYSCTL_PERIPH_WTIMER2,		SYSCTL_PERIPH_WTIMER3,
SYSCTL_PERIPH_WTIMER4,		SYSCTL_PERIPH_WTIMER5

Returns:

None.

27.2.2.40 SysCtlPIOSCCalibrate

Calibrates the precision internal oscillator.

Prototype:

```
uint32_t
SysCtlPIOSCCalibrate(uint32_t ui32Type)
```

Parameters:

ui32Type is the type of calibration to perform.

Description:

This function performs a calibration of the PIOSC. There are three types of calibration available; the desired calibration type as specified in **ui32Type** is one of:

- **SYSCTL_PIOSC_CAL_AUTO** to perform automatic calibration using the 32-kHz clock from the hibernate module as a reference. This type is only possible on parts that have a hibernate module, and then only if it is enabled, a 32.768-kHz clock source is attached to the XOSC0/1 pins and the hibernate module's RTC is also enabled.
- **SYSCTL_PIOSC_CAL_FACT** to reset the PIOSC calibration to the factory provided calibration.
- **SYSCTL_PIOSC_CAL_USER** to set the PIOSC calibration to a user-supplied value. The value to be used is ORed into the lower 7-bits of this value, with 0x40 being the “nominal”

value (in other words, if everything were perfect, 0x40 provides exactly 16 MHz). Values larger than 0x40 slow down PIOSC, and values smaller than 0x40 speed up PIOSC.

Returns:

Returns 1 if the calibration was successful and 0 if it failed.

27.2.2.41 SysCtlPWMClockGet

Gets the current PWM clock configuration.

Prototype:

```
uint32_t
SysCtlPWMClockGet (void)
```

Description:

This function returns the current PWM clock configuration.

Returns:

Returns the current PWM clock configuration; is one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

Note:

This function should only be used with TM4C123 devices. For other TM4C devices, the [PWM-ClockGet\(\)](#) function should be used.

27.2.2.42 SysCtlPWMClockSet

Sets the PWM clock configuration.

Prototype:

```
void
SysCtlPWMClockSet (uint32_t ui32Config)
```

Parameters:

ui32Config is the configuration for the PWM clock; it must be one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

Description:

This function configures the rate of the clock provided to the PWM module as a ratio of the processor clock. This clock is used by the PWM module to generate PWM signals; its rate forms the basis for all PWM signals.

Note:

This function should only be used with TM4C123 devices. For other TM4C devices, the [PWM-ClockSet\(\)](#) function should be used.

The clocking of the PWM is dependent on the system clock rate as configured by [SysCtlClockSet\(\)](#).

Returns:

None.

27.2.2.43 SysCtlReset

Resets the device.

Prototype:

```
void  
SysCtlReset (void)
```

Description:

This function performs a software reset of the entire device. The processor and all peripherals are reset and all device registers are returned to their default values (with the exception of the reset cause register, which maintains its current value but has the software reset bit set as well).

Returns:

This function does not return.

27.2.2.44 SysCtlResetBehaviorGet

Returns the current types of reset issued due to reset events.

Prototype:

```
uint32_t  
SysCtlResetBehaviorGet (void)
```

Description:

This function returns the types of resets issued when a configurable reset occurs. The value returned is a logical OR combination of the valid values that are described in the documentation for the *ui32Behavior* parameter of the [SysCtlResetBehaviorSet\(\)](#) function.

Note:

This function should only be used with Flurry-class devices.

Returns:

The reset behaviors for all configurable resets.

27.2.2.45 SysCtlResetBehaviorSet

Sets the type of reset issued due to certain reset events.

Prototype:

```
void  
SysCtlResetBehaviorSet (uint32_t ui32Behavior)
```

Parameters:

ui32Behavior specifies the types of resets for each of the configurable reset events.

Description:

This function sets the types of reset issued when a configurable reset event occurs. The reset events that are configurable are: Watchdog 0 or 1, a brown out and the external RSTn pin. The valid actions are either a system reset or a full POR sequence. See the data sheet for more

information on the differences between a full POR and a system reset. All reset behaviors can be configured with a single call using the logical OR of the values defined below. Any reset option that is not specifically set remains configured for its default behavior. Either POR or system reset can be selected for each reset cause.

Valid values are logical combinations of the following:

- **SYSCTL_ONRST_WDOG0_POR** configures a Watchdog 0 reset to perform a full POR.
- **SYSCTL_ONRST_WDOG0_SYS** configures a Watchdog 0 reset to perform a system reset.
- **SYSCTL_ONRST_WDOG1_POR** configures a Watchdog 1 reset to perform a full POR.
- **SYSCTL_ONRST_WDOG1_SYS** configures a Watchdog 1 reset to perform a system reset.
- **SYSCTL_ONRST_BOR_POR** configures a brown-out reset to perform a full POR.
- **SYSCTL_ONRST_BOR_SYS** configures a brown-out reset to perform a system reset.
- **SYSCTL_ONRST_EXT_POR** configures an external pin reset to perform a full POR.
- **SYSCTL_ONRST_EXT_SYS** configures an external pin reset to perform a system reset.

Example: Set Watchdog 0 reset to trigger a POR and a brown-out reset to trigger a system reset while leaving the remaining resets with their default behaviors.

```
SysCtlResetBehaviorSet(SYSCTL_ONRST_WDOG0_POR | SYSCTL_ONRST_BOR_SYS);
```

Note:

This function cannot be used with TM4C123 devices.

Returns:

None.

27.2.2.46 SysCtlResetCauseClear

Clears reset reasons.

Prototype:

```
void
SysCtlResetCauseClear(uint32_t ui32Causes)
```

Parameters:

ui32Causes are the reset causes to be cleared; must be a logical OR of **SYSCTL_CAUSE_HSRVREQ**, **SYSCTL_CAUSE_HIB**, **SYSCTL_CAUSE_WDOG1**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG0**, **SYSCTL_CAUSE_BOR**, **SYSCTL_CAUSE_POR**, and/or **SYSCTL_CAUSE_EXT**.

Description:

This function clears the specified sticky reset reasons. Once cleared, another reset for the same reason can be detected, and a reset for a different reason can be distinguished (instead of having two reset causes set). If the reset reason is used by an application, all reset causes should be cleared after they are retrieved with [SysCtlResetCauseGet\(\)](#).

Returns:

None.

27.2.2.47 SysCtlResetCauseGet

Gets the reason for a reset.

Prototype:

```
uint32_t  
SysCtlResetCauseGet (void)
```

Description:

This function returns the reason(s) for a reset. Because the reset reasons are sticky until either cleared by software or a power-on reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason is a logical OR of **SYSCTL_COUSE_HSRVREQ**, **SYSCTL_COUSE_HIB**, **SYSCTL_COUSE_WDOG1**, **SYSCTL_COUSE_SW**, **SYSCTL_COUSE_WDOG0**, **SYSCTL_COUSE_BOR**, **SYSCTL_COUSE_POR**, and/or **SYSCTL_COUSE_EXT**.

Returns:

Returns the reason(s) for a reset.

27.2.2.48 SysCtlSleep

Puts the processor into sleep mode.

Prototype:

```
void  
SysCtlSleep (void)
```

Description:

This function places the processor into sleep mode; it does not return until the processor returns to run mode. The peripherals that are enabled via [SysCtlPeripheralSleepEnable\(\)](#) continue to operate and can wake up the processor (if automatic clock gating is enabled with [SysCtlPeripheralClockGating\(\)](#), otherwise all peripherals continue to operate).

Returns:

None.

27.2.2.49 SysCtlSleepPowerSet

Configures the power to the flash and SRAM while in sleep mode.

Prototype:

```
void  
SysCtlSleepPowerSet (uint32_t ui32Config)
```

Parameters:

ui32Config is the required flash and SRAM power configuration.

Description:

This function allows the power configuration of the flash and SRAM while in sleep mode to be set. The **ui32Config** parameter is the logical OR of the flash power configuration and the SRAM power configuration.

The flash power configuration is specified as either:

- **SYSCTL_FLASH_NORMAL** - The flash is left in fully powered mode, providing fast wake-up time but higher power consumption.
- **SYSCTL_FLASH_LOW_POWER** - The flash is in low power mode, providing reduced power consumption but longer wake-up time.

The SRAM power configuration is specified as one of:

- **SYSCTL_SRAM_NORMAL** - The SRAM is left in fully powered mode, providing fast wake-up time but higher power consumption.
- **SYSCTL_SRAM_STANDBY** - The SRAM is placed into a lower power mode, providing reduced power consumption but longer wake-up time.
- **SYSCTL_SRAM_LOW_POWER** - The SRAM is placed into lowest power mode, providing further reduced power consumption but longer wake-up time.

Note:

The availability of this feature varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

27.2.2.50 SysCtlSRAMSizeGet

Gets the size of the SRAM.

Prototype:

```
uint32_t
SysCtlSRAMSizeGet (void)
```

Description:

This function determines the size of the SRAM on the Tiva device.

Returns:

The total number of bytes of SRAM.

27.2.2.51 SysCtlUSBPLLDisable

Powers down the USB PLL.

Prototype:

```
void
SysCtlUSBPLLDisable (void)
```

Description:

This function disables the USB controller's PLL, which is used by its physical layer. The USB registers are still accessible, but the physical layer no longer functions.

Note:

This function should only be called on TM4C123 devices.

Returns:

None.

27.2.2.52 SysCtlUSBPLLEnable

Powers up the USB PLL.

Prototype:

```
void  
SysCtlUSBPLLEnable(void)
```

Description:

This function enables the USB controller's PLL, which is used by its physical layer. This call is necessary before connecting to any external devices.

Note:

This function should only be called on TM4C123 devices.

Returns:

None.

27.2.2.53 SysCtlVoltageEventClear

Clears the voltage event status.

Prototype:

```
void  
SysCtlVoltageEventClear(uint32_t ui32Status)
```

Parameters:

ui32Status is a bit mask of the voltage events to clear.

Description:

This function clears the current voltage events status for the values specified in the **ui32Status** parameter. The **ui32Status** value must be a logical OR of the following values:

- **SYSCTL_VESTAT_VDDBOR** a brown-out event occurred on the VDD rail.
- **SYSCTL_VESTAT_VDDABOR** a brown-out event occurred on the VDDA rail.

Example: Clear the current voltage event status.

```
//  
// Clear all the current voltage events.  
//  
SysCtlVoltageEventClear(SysCtlVoltageEventStatus());
```

Note:

The availability of voltage event status varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns:

None.

27.2.2.54 SysCtlVoltageEventConfig

Configures the response to system voltage events.

Prototype:

```
void
SysCtlVoltageEventConfig(uint32_t ui32Config)
```

Parameters:

ui32Config holds the configuration options for the voltage events.

Description:

This function configures the response to voltage-related events. These events are triggered when the voltage rails drop below certain levels. The *ui32Config* parameter provides the configuration for the voltage events and is a combination of the **SYSCTL_VEVENT_*** values.

The response to a brown out on the VDDA rail is set by using one of the following values:

- **SYSCTL_VEVENT_VDDABO_NONE** - There is no action taken on a VDDA brown out.
- **SYSCTL_VEVENT_VDDABO_INT** - A system interrupt is generated when a VDDA brown out occurs.
- **SYSCTL_VEVENT_VDDABO_NMI** - An NMI is generated when a VDDA brown out occurs.
- **SYSCTL_VEVENT_VDDABO_RST** - A reset is generated when a VDDA brown out occurs. The type of reset that is generated is controlled by the **SYSCTL_ONRST_BOR_*** setting passed into the [SysCtlResetBehaviorSet\(\)](#) function.

The response to a brown out on the VDD rail is set by using one of the following values:

- **SYSCTL_VEVENT_VDDBO_NONE** - There is no action taken on a VDD brown out.
- **SYSCTL_VEVENT_VDDBO_INT** - A system interrupt is generated when a VDD brown out occurs.
- **SYSCTL_VEVENT_VDDBO_NMI** - An NMI is generated when a VDD brown out occurs.
- **SYSCTL_VEVENT_VDDBO_RST** - A reset is generated when a VDD brown out occurs. The type of reset that is generated is controlled by the **SYSCTL_ONRST_BOR_*** setting passed into the [SysCtlResetBehaviorSet\(\)](#) function.

Example: Configure the voltage events to trigger an interrupt on a VDDA brown out, an NMI on a VDDC brown out and a reset on a VDD brown out.

```
//
// Configure the BOR rest to trigger a full POR. This is needed because
// the SysCtlVoltageEventConfig() call is triggering a reset so the type
// of reset is specified by this call.
//
SysCtlResetBehaviorSet(SYSCTL_ONRST_BOR_POR);

//
// Trigger an interrupt on a VDDA brown out and a reset on a VDD brown out.
//
SysCtlVoltageEventConfig(SYSCTL_VEVENT_VDDABO_INT |
                         SYSCTL_VEVENT_VDDBO_RST);
```

Returns:

None.

27.2.2.55 SysCtlVoltageEventStatus

Returns the voltage event status.

Prototype:

```
uint32_t  
SysCtlVoltageEventStatus (void)
```

Description:

This function returns the voltage event status for the system controller. The value returned is a logical OR of the following values:

- **SYSCTL_VESTAT_VDDBOR** a brown-out event occurred on the VDD rail.
- **SYSCTL_VESTAT_VDDABOR** a brown-out event occurred on the VDDA rail.

The values returned from this function can be passed to the [SysCtlVoltageEventClear\(\)](#) to clear the current voltage event status. Because voltage events are not cleared due to a reset, the voltage event status must be cleared by calling [SysCtlVoltageEventClear\(\)](#).

Example: Clear the current voltage event status.

```
uint32_t ui32VoltageEvents;  
  
//  
// Read the current voltage event status.  
//  
ui32VoltageEvents = SysCtlVoltageEventStatus();  
  
//  
// Clear all the current voltage events.  
//  
SysCtlVoltageEventClear(ui32VoltageEvents);
```

Returns:

The current voltage event status.

Note:

The availability of voltage events varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

27.3 Programming Example

The following example shows how to use the SysCtl API to configure the device for normal operation.

```
//  
// Configure the device to run at 20 MHz from the PLL using a 4 MHz crystal  
// as the input.  
//  
SysCtlClockSet(SYSCLOCK_SYSDIV_10 | SYSCLOCK_USE_PLL | SYSCLOCK_XTAL_4MHZ |  
    SYSCLOCK_OSC_MAIN);  
  
//  
// Enable the GPIO blocks and the SSI.  
//  
SysCtlPeripheralEnable(SYSCLOCK_PERIPH_GPIOA);
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI);

// 
// Enable the GPIO blocks and the SSI in sleep mode.
//
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_SSI);

// 
// Enable peripheral clock gating.
//
SysCtlPeripheralClockGating(true);
```


28 System Exception Module

Introduction	523
API Functions	523
Programming Example	526

28.1 Introduction

The system exception module driver provides methods for manipulating the behavior of the system exception module that handles system-level Cortex-M4 FPU exceptions. The exceptions are underflow, overflow, divide by zero, invalid operation, input denormal, and inexact exception. The application can optionally choose to enable one or more of these interrupts and use the interrupt handler to decide upon a course of action to be taken in each case. All the interrupt events are ORed together before being sent to the interrupt controller, so the System Exception module can only generate a single interrupt request to the controller at any given time.

This driver is contained in `driverlib/sysexc.c`, with `driverlib/sysexc.h` containing the API declarations for use by applications.

28.2 API Functions

Functions

- void [SysExclIntClear](#) (uint32_t ui32IntFlags)
- void [SysExclIntDisable](#) (uint32_t ui32IntFlags)
- void [SysExclIntEnable](#) (uint32_t ui32IntFlags)
- void [SysExclIntRegister](#) (void (*pfnHandler)(void))
- uint32_t [SysExclIntStatus](#) (bool bMasked)
- void [SysExclIntUnregister](#) (void)

28.2.1 Detailed Description

The system exception module interrupts are managed with [SysExclIntRegister\(\)](#), [SysExclIntUnregister\(\)](#), [SysExclIntEnable\(\)](#), [SysExclIntDisable\(\)](#), [SysExclIntStatus\(\)](#), and [SysExclIntClear\(\)](#).

28.2.2 Function Documentation

28.2.2.1 SysExclIntClear

Clears system exception interrupt sources.

Prototype:

```
void
SysExclIntClear(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

This function clears the specified system exception interrupt sources, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being recognized again immediately upon exit.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **SYSEXC_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXC_INT_FP_OFC** - Floating-point overflow exception interrupt
- **SYSEXC_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXC_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXC_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXC_INT_FP_IDC** - Floating-point input denormal exception interrupt

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

28.2.2.2 SysExcIntDisable

Disables individual system exception interrupt sources.

Prototype:

```
void  
SysExcIntDisable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated system exception interrupt sources. Only sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **SYSEXC_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXC_INT_FP_OFC** - Floating-point overflow exception interrupt
- **SYSEXC_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXC_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXC_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXC_INT_FP_IDC** - Floating-point input denormal exception interrupt

Returns:

None.

28.2.2.3 SysExcIntEnable

Enables individual system exception interrupt sources.

Prototype:

```
void
SysExcIntEnable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated system exception interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The ***ui32IntFlags*** parameter is the logical OR of any of the following:

- **SYSEXC_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXC_INT_FP_OFC** - Floating-point overflow exception interrupt
- **SYSEXC_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXC_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXC_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXC_INT_FP_IDC** - Floating-point input denormal exception interrupt

Returns:

None.

28.2.2.4 SysExcIntRegister

Registers an interrupt handler for the system exception interrupt.

Prototype:

```
void
SysExcIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the system exception interrupt occurs.

Description:

This function places the address of the system exception interrupt handler into the interrupt vector table in SRAM. This function also enables the global interrupt in the interrupt controller; specific system exception interrupts must be enabled via [SysExcIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

28.2.2.5 SysExcIntStatus

Gets the current system exception interrupt status.

Prototype:

```
uint32_t  
SysExcIntStatus (bool bMasked)
```

Parameters:

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function returns the system exception interrupt status. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current system exception interrupt status, enumerated as the logical OR of **SYSEXC_INT_FP_IJC**, **SYSEXC_INT_FP_OFC**, **SYSEXC_INT_FP_UFC**, **SYSEXC_INT_FP_IOC**, **SYSEXC_INT_FP_DZC**, and **SYSEXC_INT_FP_IDC**.

28.2.2.6 SysExcIntUnregister

Unregisters the system exception interrupt handler.

Prototype:

```
void  
SysExcIntUnregister (void)
```

Description:

This function removes the system exception interrupt handler from the vector table in SRAM. This function also masks off the system exception interrupt in the interrupt controller so that the interrupt handler is no longer called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

28.3 Programming Example

The following example shows how to use the system exception module API to register an interrupt handler and enable an interrupt.

```
//  
// The interrupt handler function.  
//  
extern void SysExcIntHandler(void);  
  
//  
// Register the interrupt handler function for the system exception  
// interrupt.  
//  
SysExcIntRegister(SysExcIntHandler);  
  
//  
// Enable the Floating-point overflow exception.  
//  
SysExcIntEnable(SYSEXC_INT_FP_OFC);
```


29 System Tick (SysTick)

Introduction	529
API Functions	529
Programming Example	533

29.1 Introduction

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M microprocessor. Its intended purpose is to provide a periodic interrupt for an RTOS, but it can be used for other simple timing purposes.

The SysTick interrupt handler does not need to clear the SysTick interrupt source as it is cleared automatically by the NVIC when the SysTick interrupt handler is called.

This driver is contained in `driverlib/systick.c`, with `driverlib/systick.h` containing the API declarations for use by applications.

29.2 API Functions

Functions

- void `SysTickDisable` (void)
- void `SysTickEnable` (void)
- void `SysTickIntDisable` (void)
- void `SysTickIntEnable` (void)
- void `SysTickIntRegister` (void (*pfnHandler)(void))
- void `SysTickIntUnregister` (void)
- uint32_t `SysTickPeriodGet` (void)
- void `SysTickPeriodSet` (uint32_t ui32Period)
- uint32_t `SysTickValueGet` (void)

29.2.1 Detailed Description

The SysTick API is fairly simple, like SysTick itself. There are functions for configuring and enabling SysTick (`SysTickEnable()`, `SysTickDisable()`, `SysTickPeriodSet()`, `SysTickPeriodGet()`, and `SysTickValueGet()`) and functions for dealing with an interrupt handler for SysTick (`SysTickIntRegister()`, `SysTickIntUnregister()`, `SysTickIntEnable()`, and `SysTickIntDisable()`).

29.2.2 Function Documentation

29.2.2.1 `SysTickDisable`

Disables the SysTick counter.

Prototype:

```
void  
SysTickDisable(void)
```

Description:

This function stops the SysTick counter. If an interrupt handler has been registered, it is not called until SysTick is restarted.

Returns:

None.

29.2.2.2 SysTickEnable

Enables the SysTick counter.

Prototype:

```
void  
SysTickEnable(void)
```

Description:

This function starts the SysTick counter. If an interrupt handler has been registered, it is called when the SysTick counter rolls over.

Note:

Calling this function causes the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to [SysTickPeriodSet\(\)](#). If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written to force the reload. Any write to this register clears the SysTick counter to 0 and causes a reload with the supplied period on the next clock.

Returns:

None.

29.2.2.3 SysTickIntDisable

Disables the SysTick interrupt.

Prototype:

```
void  
SysTickIntDisable(void)
```

Description:

This function disables the SysTick interrupt, preventing it from being reflected to the processor.

Returns:

None.

29.2.2.4 SysTickIntEnable

Enables the SysTick interrupt.

Prototype:

```
void
SysTickIntEnable(void)
```

Description:

This function enables the SysTick interrupt, allowing it to be reflected to the processor.

Note:

The SysTick interrupt handler is not required to clear the SysTick interrupt source because it is cleared automatically by the NVIC when the interrupt handler is called.

Returns:

None.

29.2.2.5 SysTickIntRegister

Registers an interrupt handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the SysTick interrupt occurs.

Description:

This function registers the handler to be called when a SysTick interrupt occurs.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

29.2.2.6 SysTickIntUnregister

Unregisters the interrupt handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntUnregister(void)
```

Description:

This function unregisters the handler to be called when a SysTick interrupt occurs.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

29.2.2.7 SysTickPeriodGet

Gets the period of the SysTick counter.

Prototype:

```
uint32_t  
SysTickPeriodGet (void)
```

Description:

This function returns the rate at which the SysTick counter wraps, which equates to the number of processor clocks between interrupts.

Returns:

Returns the period of the SysTick counter.

29.2.2.8 SysTickPeriodSet

Sets the period of the SysTick counter.

Prototype:

```
void  
SysTickPeriodSet (uint32_t ui32Period)
```

Parameters:

ui32Period is the number of clock ticks in each period of the SysTick counter and must be between 1 and 16, 777, 216, inclusive.

Description:

This function sets the rate at which the SysTick counter wraps, which equates to the number of processor clocks between interrupts.

Note:

Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and causes a reload with the *ui32Period* supplied here on the next clock after SysTick is enabled.

Returns:

None.

29.2.2.9 SysTickCountGet

Gets the current value of the SysTick counter.

Prototype:

```
uint32_t  
SysTickCountGet (void)
```

Description:

This function returns the current value of the SysTick counter, which is a value between the period - 1 and zero, inclusive.

Returns:

Returns the current value of the SysTick counter.

29.3 Programming Example

The following example shows how to use the SysTick API to configure the SysTick counter and read its value.

```
uint32_t ui32Value;

//
// Configure and enable the SysTick counter.
//
SysTickPeriodSet(1000);
SysTickEnable();

//
// Delay for some time...
//

//
// Read the current SysTick value.
//
ui32Value = SysTickValueGet();
```


30 Timer

Introduction	535
API Functions	536
Programming Example	558

30.1 Introduction

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

The timer module provides two half-width timers/counters that can be configured to operate independently as timers or event counters or to operate as a combined full-width timer or Real Time Clock (RTC). Some timers provide 16-bit half-width timers and a 32-bit full-width timer, while others provide 32-bit half-width timers and a 64-bit full-width timer. For the purposes of this API, the two half-width timers provided by a timer module are referred to as TimerA and TimerB, and the full-width timer is referred to as TimerA.

When configured as either a full-width or half-width timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured in one-shot mode, the timer ceases counting when it reaches zero when counting down or the load value when counting up. If configured in continuous mode, the timer counts to zero (counting down) or the load value (counting up), then reloads and continues counting. When configured as a full-width timer, the timer can also be configured to operate as an RTC. In this mode, the timer expects to be driven by a 32.768-KHz external clock, which is divided down to produce 1 second clock ticks.

When in half-width mode, the timer can also be configured for event capture or as a Pulse Width Modulation (PWM) generator. When configured for event capture, the timer acts as a counter. It can be configured to either count the time between events or the events themselves. The type of event being counted can be configured as a positive edge, a negative edge, or both edges. When a timer is configured as a PWM generator, the input signal used to capture events becomes an output signal, and the timer drives an edge-aligned pulse onto that signal.

The timer module also provides the ability to control other functional parameters, such as output inversion, output triggers, and timer behavior during stalls.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured, or that a certain number of events have been captured. Interrupts can also be generated when the timer has counted down to zero or when the timer matches a certain value.

On some parts, the counters from multiple timer modules can be synchronized. Synchronized counters are useful in PWM and edge time capture modes. In PWM mode, the PWM outputs from multiple timers can be in lock-step by having the same load value and synchronizing the counters (meaning that the counters always have the same value). Similarly, by using the same load value and synchronized counters in edge time capture mode, the absolute time between two input edges can be easily measured.

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

30.2 API Functions

Functions

- `uint32_t TimerADCEventGet (uint32_t ui32Base)`
- `void TimerADCEventSet (uint32_t ui32Base, uint32_t ui32ADCEvent)`
- `uint32_t TimerClockSourceGet (uint32_t ui32Base)`
- `void TimerClockSourceSet (uint32_t ui32Base, uint32_t ui32Source)`
- `void TimerConfigure (uint32_t ui32Base, uint32_t ui32Config)`
- `void TimerControlEvent (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Event)`
- `void TimerControlLevel (uint32_t ui32Base, uint32_t ui32Timer, bool bInvert)`
- `void TimerControlStall (uint32_t ui32Base, uint32_t ui32Timer, bool bStall)`
- `void TimerControlTrigger (uint32_t ui32Base, uint32_t ui32Timer, bool bEnable)`
- `void TimerControlWaitOnTrigger (uint32_t ui32Base, uint32_t ui32Timer, bool bWait)`
- `void TimerDisable (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint32_t TimerDMAEventGet (uint32_t ui32Base)`
- `void TimerDMAEventSet (uint32_t ui32Base, uint32_t ui32DMAEvent)`
- `void TimerEnable (uint32_t ui32Base, uint32_t ui32Timer)`
- `void TimerIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void TimerIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void TimerIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void TimerIntRegister (uint32_t ui32Base, uint32_t ui32Timer, void (*pfnHandler)(void))`
- `uint32_t TimerIntStatus (uint32_t ui32Base, bool bMasked)`
- `void TimerIntUnregister (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint32_t TimerLoadGet (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint64_t TimerLoadGet64 (uint32_t ui32Base)`
- `void TimerLoadSet (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)`
- `void TimerLoadSet64 (uint32_t ui32Base, uint64_t ui64Value)`
- `uint32_t TimerMatchGet (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint64_t TimerMatchGet64 (uint32_t ui32Base)`
- `void TimerMatchSet (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)`
- `void TimerMatchSet64 (uint32_t ui32Base, uint64_t ui64Value)`
- `uint32_t TimerPrescaleGet (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint32_t TimerPrescaleMatchGet (uint32_t ui32Base, uint32_t ui32Timer)`
- `void TimerPrescaleMatchSet (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)`
- `void TimerPrescaleSet (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Value)`
- `void TimerRTCDisable (uint32_t ui32Base)`
- `void TimerRTCEnable (uint32_t ui32Base)`
- `void TimerSynchronize (uint32_t ui32Base, uint32_t ui32Timers)`
- `void TimerUpdateMode (uint32_t ui32Base, uint32_t ui32Timer, uint32_t ui32Config)`
- `uint32_t TimerValueGet (uint32_t ui32Base, uint32_t ui32Timer)`
- `uint64_t TimerValueGet64 (uint32_t ui32Base)`

30.2.1 Detailed Description

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 is handled by [TimerConfigure\(\)](#), which performs the high level setup of the timer module; that is, it is used to set up full- or half-width modes, and to select between PWM, capture, and timer operations. Timer control is performed by [TimerEnable\(\)](#), [TimerDisable\(\)](#), [TimerControlLevel\(\)](#), [TimerControlTrigger\(\)](#), [TimerControlEvent\(\)](#), [TimerControlStall\(\)](#), [TimerRTCEnable\(\)](#), and [TimerRTCDisable\(\)](#).

Timer content is managed with [TimerLoadSet\(\)](#), [TimerLoadGet\(\)](#), [TimerLoadSet64\(\)](#), [TimerLoadGet64\(\)](#), [TimerPrescaleSet\(\)](#), [TimerPrescaleGet\(\)](#), [TimerMatchSet\(\)](#), [TimerMatchGet\(\)](#), [TimerMatchSet64\(\)](#), [TimerMatchGet64\(\)](#), [TimerPrescaleMatchSet\(\)](#), [TimerPrescaleMatchGet\(\)](#), [TimerValueGet\(\)](#), [TimerValueGet64\(\)](#), and [TimerSynchronize\(\)](#).

The interrupt handler for the Timer interrupt is managed with [TimerIntRegister\(\)](#) and [TimerIntUnregister\(\)](#). The individual interrupt sources within the timer module are managed with [TimerIntEnable\(\)](#), [TimerIntDisable\(\)](#), [TimerIntStatus\(\)](#), and [TimerIntClear\(\)](#).

30.2.2 Function Documentation

30.2.2.1 TimerADCEventGet

Returns the events that can cause an ADC trigger event.

Prototype:

```
uint32_t
TimerADCEventGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function returns the timer events that can cause an ADC trigger event. The ADC trigger events are the logical OR of any of the following values:

- **TIMER_ADC_MODEMATCH_B** - The mode match ADC trigger for timer B is enabled.
- **TIMER_ADC_CAPEVENT_B** - The capture event ADC trigger for timer B is enabled.
- **TIMER_ADC_CAPMATCH_B** - The capture match ADC trigger for timer B is enabled.
- **TIMER_ADC_TIMEOUT_B** - The timeout ADC trigger for timer B is enabled.
- **TIMER_ADC_MODEMATCH_A** - The mode match ADC trigger for timer A is enabled.
- **TIMER_ADC_RTC_A** - The RTC ADC trigger for timer A is enabled.
- **TIMER_ADC_CAPEVENT_A** - The capture event ADC trigger for timer A is enabled.
- **TIMER_ADC_CAPMATCH_A** - The capture match ADC trigger for timer A is enabled.
- **TIMER_ADC_TIMEOUT_A** - The timeout ADC trigger for timer A is enabled.

Note:

The ability to specify ADC event triggers varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

The timer events that trigger the ADC.

30.2.2.2 TimerADCEventSet

Enables the events that can cause an ADC trigger event.

Prototype:

```
void  
TimerADCEventSet(uint32_t ui32Base,  
                  uint32_t ui32ADCEvent)
```

Parameters:

ui32Base is the base address of the timer module.

ui32ADCEvent is a bit mask of the events that can cause an ADC trigger event.

Description:

This function enables the timer events that can cause an ADC trigger event. The ADC trigger events are specified in the *ui32ADCEvent* parameter by passing in the logical OR of any of the following values:

- **TIMER_ADC_MODEMATCH_B** - Enables the mode match ADC trigger for timer B.
- **TIMER_ADC_CAPEVENT_B** - Enables the capture event ADC trigger for timer B.
- **TIMER_ADC_CAPMATCH_B** - Enables the capture match ADC trigger for timer B.
- **TIMER_ADC_TIMEOUT_B** - Enables the timeout ADC trigger for timer B.
- **TIMER_ADC_MODEMATCH_A** - Enables the mode match ADC trigger for timer A.
- **TIMER_ADC_RTC_A** - Enables the RTC ADC trigger for timer A.
- **TIMER_ADC_CAPEVENT_A** - Enables the capture event ADC trigger for timer A.
- **TIMER_ADC_CAPMATCH_A** - Enables the capture match ADC trigger for timer A.
- **TIMER_ADC_TIMEOUT_A** - Enables the timeout ADC trigger for timer A.

Note:

The ability to specify ADC event triggers varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

30.2.2.3 TimerClockSourceGet

Returns the clock source for the specified timer module.

Prototype:

```
uint32_t  
TimerClockSourceGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function returns the clock source for the specified timer module. The possible clock sources are the system clock (**TIMER_CLOCK_SYSTEM**) or the precision internal oscillator (**TIMER_CLOCK_PIOSC**).

Note:

The ability to specify the timer clock source varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

Returns either **TIMER_CLOCK_SYSTEM** or **TIMER_CLOCK_PIOSC**.

30.2.2.4 TimerClockSourceSet

Sets the clock source for the specified timer module.

Prototype:

```
void
TimerClockSourceSet(uint32_t ui32Base,
                    uint32_t ui32Source)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Source is the clock source for the timer module.

Description:

This function sets the clock source for both timer A and timer B for the given timer module. The possible clock sources are the system clock (**TIMER_CLOCK_SYSTEM**) or the precision internal oscillator (**TIMER_CLOCK_PIOSC**).

Note:

The ability to specify the timer clock source varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

30.2.2.5 TimerConfigure

Configures the timer(s).

Prototype:

```
void
TimerConfigure(uint32_t ui32Base,
               uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Config is the configuration for the timer.

Description:

This function configures the operating mode of the timer(s). The timer module is disabled before being configured and is left in the disabled state. The timer can be configured to be a single full-width timer by using the **TIMER_CFG_*** values or a pair of half-width timers using the **TIMER_CFG_A_*** and **TIMER_CFG_B_*** values passed in the *ui32Config* parameter.

The configuration is specified in *ui32Config* as one of the following values:

- **TIMER_CFG_ONE_SHOT** - Full-width one-shot timer
- **TIMER_CFG_ONE_SHOT_UP** - Full-width one-shot timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_PERIODIC** - Full-width periodic timer
- **TIMER_CFG_PERIODIC_UP** - Full-width periodic timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_RTC** - Full-width real time clock timer
- **TIMER_CFG_SPLIT_PAIR** - Two half-width timers

When configured for a pair of half-width timers, each timer is separately configured. The first timer is configured by setting *ui32Config* to the result of a logical OR operation between one of the following values and *ui32Config*:

- **TIMER_CFG_A_ONE_SHOT** - Half-width one-shot timer
- **TIMER_CFG_A_ONE_SHOT_UP** - Half-width one-shot timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_PERIODIC** - Half-width periodic timer
- **TIMER_CFG_A_PERIODIC_UP** - Half-width periodic timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_CAP_COUNT** - Half-width edge count capture
- **TIMER_CFG_A_CAP_COUNT_UP** - Half-width edge count capture that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_CAP_TIME** - Half-width edge time capture
- **TIMER_CFG_A_CAP_TIME_UP** - Half-width edge time capture that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_PWM** - Half-width PWM output

Some Tiva devices also allow configuring an action when the timers reach their timeout. Please consult the data sheet for the part you are using to determine whether configuring actions on timers is available.

One of the following can be combined with the **TIMER_CFG_*** values to enable an action on timer A:

- **TIMER_CFG_A_ACT_TOINTD** - masks the timeout interrupt of timer A.
- **TIMER_CFG_A_ACT_NONE** - no additional action on timeout of timer A.
- **TIMER_CFG_A_ACT_TOGGLE** - toggle CCP on timeout of timer A.
- **TIMER_CFG_A_ACT_SETTO** - set CCP on timeout of timer A.
- **TIMER_CFG_A_ACT_CLRTO** - clear CCP on timeout of timer A.
- **TIMER_CFG_A_ACT_SETTOGTO** - set CCP immediately and then toggle it on timeout of timer A.
- **TIMER_CFG_A_ACT_CLRTOGTO** - clear CCP immediately and then toggle it on timeout of timer A.
- **TIMER_CFG_A_ACT_SETCLRTO** - set CCP immediately and then clear it on timeout of timer A.
- **TIMER_CFG_A_ACT_CLRSETTO** - clear CCP immediately and then set it on timeout of timer A.

One of the following can be combined with the **TIMER_CFG_*** values to enable an action on timer B:

- **TIMER_CFG_B_ACT_TOINTD** - masks the timeout interrupt of timer B.

- **TIMER_CFG_B_ACT_NONE** - no additional action on timeout of timer B.
- **TIMER_CFG_B_ACT_TOGGLE** - toggle CCP on timeout of timer B.
- **TIMER_CFG_B_ACT_SETTO** - set CCP on timeout of timer B.
- **TIMER_CFG_B_ACT_CLRTO** - clear CCP on timeout of timer B.
- **TIMER_CFG_B_ACT_SETTOGTO** - set CCP immediately and then toggle it on timeout of timer B.
- **TIMER_CFG_B_ACT_CLRTOGTO** - clear CCP immediately and then toggle it on timeout of timer B.
- **TIMER_CFG_B_ACT_SETCLRTO** - set CCP immediately and then clear it on timeout of timer B.
- **TIMER_CFG_B_ACT_CLRSETTO** - clear CCP immediately and then set it on timeout of timer B.

Similarly, the second timer is configured by setting *ui32Config* to the result of a logical OR operation between one of the corresponding **TIMER_CFG_B_*** values and *ui32Config*.

Returns:

None.

30.2.2.6 TimerControlEvent

Controls the event type.

Prototype:

```
void
TimerControlEvent (uint32_t ui32Base,
                   uint32_t ui32Timer,
                   uint32_t ui32Event)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ui32Event specifies the type of event; must be one of **TIMER_EVENT_POS_EDGE**, **TIMER_EVENT_NEG_EDGE**, or **TIMER_EVENT_BOTH_EDGES**.

Description:

This function configures the signal edge(s) that triggers the timer when in capture mode.

Returns:

None.

30.2.2.7 TimerControlLevel

Controls the output level.

Prototype:

```
void
TimerControlLevel (uint32_t ui32Base,
                   uint32_t ui32Timer,
                   bool bInvert)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bInvert specifies the output level.

Description:

This function configures the PWM output level for the specified timer. If the **bInvert** parameter is **true**, then the timer's output is made active low; otherwise, it is made active high.

Returns:

None.

30.2.2.8 TimerControlStall

Controls the stall handling.

Prototype:

```
void  
TimerControlStall(uint32_t ui32Base,  
                  uint32_t ui32Timer,  
                  bool bStall)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bStall specifies the response to a stall signal.

Description:

This function controls the stall response for the specified timer. If the **bStall** parameter is **true**, then the timer stops counting if the processor enters debug mode; otherwise the timer keeps running while in debug mode.

Returns:

None.

30.2.2.9 TimerControlTrigger

Enables or disables the ADC trigger output.

Prototype:

```
void  
TimerControlTrigger(uint32_t ui32Base,  
                     uint32_t ui32Timer,  
                     bool bEnable)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bEnable specifies the desired ADC trigger state.

Description:

This function controls the ADC trigger output for the specified timer. If the ***bEnable*** parameter is **true**, then the timer's ADC output trigger is enabled; otherwise it is disabled.

Returns:

None.

30.2.2.10 TimerControlWaitOnTrigger

Controls the wait on trigger handling.

Prototype:

```
void
TimerControlWaitOnTrigger(uint32_t ui32Base,
                          uint32_t ui32Timer,
                          bool bWait)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bWait specifies if the timer should wait for a trigger input.

Description:

This function controls whether or not a timer waits for a trigger input to start counting. When enabled, the previous timer in the trigger chain must count to its timeout in order for this timer to start counting. Refer to the part's data sheet for a description of the trigger chain.

Note:

This functionality is not available on all parts. This function should not be used for Timer 0A or Wide Timer 0A.

Returns:

None.

30.2.2.11 TimerDisable

Disables the timer(s).

Prototype:

```
void
TimerDisable(uint32_t ui32Base,
             uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to disable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This function disables operation of the timer module.

Returns:

None.

30.2.2.12 TimerDMAEventGet

Returns the events that can trigger a uDMA request.

Prototype:

```
uint32_t  
TimerDMAEventGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function returns the timer events that can trigger the start of a uDMA sequence. The uDMA trigger events are the logical OR of the following values:

- **TIMER_DMA_MODEMATCH_B** - Enables the mode match uDMA trigger for timer B.
- **TIMER_DMA_CAPEVENT_B** - Enables the capture event uDMA trigger for timer B.
- **TIMER_DMA_CAPMATCH_B** - Enables the capture match uDMA trigger for timer B.
- **TIMER_DMA_TIMEOUT_B** - Enables the timeout uDMA trigger for timer B.
- **TIMER_DMA_MODEMATCH_A** - Enables the mode match uDMA trigger for timer A.
- **TIMER_DMA_RTC_A** - Enables the RTC uDMA trigger for timer A.
- **TIMER_DMA_CAPEVENT_A** - Enables the capture event uDMA trigger for timer A.
- **TIMER_DMA_CAPMATCH_A** - Enables the capture match uDMA trigger for timer A.
- **TIMER_DMA_TIMEOUT_A** - Enables the timeout uDMA trigger for timer A.

Note:

The ability to specify uDMA event triggers varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

The timer events that trigger the uDMA.

30.2.2.13 TimerDMAEventSet

Enables the events that can trigger a uDMA request.

Prototype:

```
void  
TimerDMAEventSet(uint32_t ui32Base,  
                  uint32_t ui32DMAEvent)
```

Parameters:

ui32Base is the base address of the timer module.

ui32DMAEvent is a bit mask of the events that can trigger uDMA.

Description:

This function enables the timer events that can trigger the start of a uDMA sequence. The uDMA trigger events are specified in the ***ui32DMAEvent*** parameter by passing in the logical OR of the following values:

- **TIMER_DMA_MODEMATCH_B** - The mode match uDMA trigger for timer B is enabled.
- **TIMER_DMA_CAPEVENT_B** - The capture event uDMA trigger for timer B is enabled.
- **TIMER_DMA_CAPMATCH_B** - The capture match uDMA trigger for timer B is enabled.
- **TIMER_DMA_TIMEOUT_B** - The timeout uDMA trigger for timer B is enabled.
- **TIMER_DMA_MODEMATCH_A** - The mode match uDMA trigger for timer A is enabled.
- **TIMER_DMA_RTC_A** - The RTC uDMA trigger for timer A is enabled.
- **TIMER_DMA_CAPEVENT_A** - The capture event uDMA trigger for timer A is enabled.
- **TIMER_DMA_CAPMATCH_A** - The capture match uDMA trigger for timer A is enabled.
- **TIMER_DMA_TIMEOUT_A** - The timeout uDMA trigger for timer A is enabled.

Note:

The ability to specify uDMA event triggers varies with the Tiva part in use. Please consult the data sheet for the part you are using to determine whether this support is available.

Returns:

None.

30.2.2.14 TimerEnable

Enables the timer(s).

Prototype:

```
void
TimerEnable(uint32_t ui32Base,
            uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to enable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This function enables operation of the timer module. The timer must be configured before it is enabled.

Returns:

None.

30.2.2.15 TimerIntClear

Clears timer interrupt sources.

Prototype:

```
void  
TimerIntClear(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified timer interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [TimerIntEnable\(\)](#).

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

30.2.2.16 TimerIntDisable

Disables individual timer interrupt sources.

Prototype:

```
void  
TimerIntDisable(uint32_t ui32Base,  
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [TimerIntEnable\(\)](#).

Returns:

None.

30.2.2.17 TimerIntEnable

Enables individual timer interrupt sources.

Prototype:

```
void
TimerIntEnable(uint32_t ui32Base,
               uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter must be the logical OR of any combination of the following:

- **TIMER_TIMB_DMA** - Timer B uDMA complete
- **TIMER_TIMA_DMA** - Timer A uDMA complete
- **TIMER_CAPB_EVENT** - Capture B event interrupt
- **TIMER_CAPB_MATCH** - Capture B match interrupt
- **TIMER_TIMB_TIMEOUT** - Timer B timeout interrupt
- **TIMER_RTC_MATCH** - RTC interrupt mask
- **TIMER_CAPA_EVENT** - Capture A event interrupt
- **TIMER_CAPA_MATCH** - Capture A match interrupt
- **TIMER_TIMA_TIMEOUT** - Timer A timeout interrupt

Returns:

None.

30.2.2.18 TimerIntRegister

Registers an interrupt handler for the timer interrupt.

Prototype:

```
void
TimerIntRegister(uint32_t ui32Base,
                  uint32_t ui32Timer,
                  void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

pfnHandler is a pointer to the function to be called when the timer interrupt occurs.

Description:

This function registers the handler to be called when a timer interrupt occurs. In addition, this function enables the global interrupt in the interrupt controller; specific timer interrupts must be enabled via [TimerIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [TimerIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

30.2.2.19 TimerIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t  
TimerIntStatus(uint32_t ui32Base,  
               bool bMasked)
```

Parameters:

ui32Base is the base address of the timer module.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This function returns the interrupt status for the timer module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of values described in [TimerIntEnable\(\)](#).

30.2.2.20 TimerIntUnregister

Unregisters an interrupt handler for the timer interrupt.

Prototype:

```
void  
TimerIntUnregister(uint32_t ui32Base,  
                    uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This function unregisters the handler to be called when a timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler is no longer called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

30.2.2.21 TimerLoadGet

Gets the timer load value.

Prototype:

```
uint32_t
TimerLoadGet(uint32_t ui32Base,
            uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for full-width operation.

Description:

This function gets the currently programmed interval load value for the specified timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers and for half-width modes of 32/64-bit timers. Use [TimerLoadGet64\(\)](#) for full-width modes of 32/64-bit timers.

Returns:

Returns the load value for the timer.

30.2.2.22 TimerLoadGet64

Gets the timer load value for a 64-bit timer.

Prototype:

```
uint64_t
TimerLoadGet64(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function gets the currently programmed interval load value for the specified 64-bit timer.

Returns:

Returns the load value for the timer.

30.2.2.23 TimerLoadSet

Sets the timer load value.

Prototype:

```
void
TimerLoadSet(uint32_t ui32Base,
            uint32_t ui32Timer,
            uint32_t ui32Value)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for full-width operation.

ui32Value is the load value.

Description:

This function configures the timer load value; if the timer is running then the value is immediately loaded into the timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers and for half-width modes of 32/64-bit timers. Use [TimerLoadSet64\(\)](#) for full-width modes of 32/64-bit timers.

Returns:

None.

30.2.2.24 TimerLoadSet64

Sets the timer load value for a 64-bit timer.

Prototype:

```
void  
TimerLoadSet64(uint32_t ui32Base,  
                uint64_t ui64Value)
```

Parameters:

ui32Base is the base address of the timer module.

ui64Value is the load value.

Description:

This function configures the timer load value for a 64-bit timer; if the timer is running, then the value is immediately loaded into the timer.

Returns:

None.

30.2.2.25 TimerMatchGet

Gets the timer match value.

Prototype:

```
uint32_t  
TimerMatchGet(uint32_t ui32Base,  
              uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for full-width operation.

Description:

This function gets the match value for the specified timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers and for half-width modes of 32/64-bit timers. Use [TimerMatchGet64\(\)](#) for full-width modes of 32/64-bit timers.

Returns:

Returns the match value for the timer.

30.2.2.26 TimerMatchGet64

Gets the timer match value for a 64-bit timer.

Prototype:

```
uint64_t
TimerMatchGet64(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function gets the match value for the specified timer.

Returns:

Returns the match value for the timer.

30.2.2.27 TimerMatchSet

Sets the timer match value.

Prototype:

```
void
TimerMatchSet(uint32_t ui32Base,
              uint32_t ui32Timer,
              uint32_t ui32Value)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for full-width operation.

ui32Value is the match value.

Description:

This function configures the match value for a timer. This value is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal. On some Tiva devices, match interrupts can also be generated in periodic and one-shot modes.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers and for half-width modes of 32/64-bit timers. Use [TimerMatchSet64\(\)](#) for full-width modes of 32/64-bit timers.

Returns:

None.

30.2.2.28 TimerMatchSet64

Sets the timer match value for a 64-bit timer.

Prototype:

```
void  
TimerMatchSet64(uint32_t ui32Base,  
                 uint64_t ui64Value)
```

Parameters:

ui32Base is the base address of the timer module.
ui64Value is the match value.

Description:

This function configures the match value for a timer. This value is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal.

Returns:

None.

30.2.2.29 TimerPrescaleGet

Gets the timer prescale value.

Prototype:

```
uint32_t  
TimerPrescaleGet(uint32_t ui32Base,  
                  uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.
ui32Timer specifies the timer; must be one of **TIMER_A** or **TIMER_B**.

Description:

This function gets the value of the input clock prescaler. The prescaler is only operational when in half-width mode and is used to extend the range of the half-width timer modes. The prescaler provides the least significant bits when counting down in periodic and one-shot modes; in all other modes, the prescaler provides the most significant bits.

Note:

The availability of the prescaler varies with the Tiva part and timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

The value of the timer prescaler.

30.2.2.30 TimerPrescaleMatchGet

Gets the timer prescale match value.

Prototype:

```
uint32_t
TimerPrescaleMatchGet (uint32_t ui32Base,
                      uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer; must be one of **TIMER_A** or **TIMER_B**.

Description:

This function gets the value of the input clock prescaler match value. When in a half-width mode that uses the counter match and prescaler, the prescale match effectively extends the range of the match. The prescaler provides the least significant bits when counting down in periodic and one-shot modes; in all other modes, the prescaler provides the most significant bits.

Note:

The availability of the prescaler match varies with the Tiva part and timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

The value of the timer prescale match.

30.2.2.31 TimerPrescaleMatchSet

Sets the timer prescale match value.

Prototype:

```
void
TimerPrescaleMatchSet (uint32_t ui32Base,
                      uint32_t ui32Timer,
                      uint32_t ui32Value)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ui32Value is the timer prescale match value which must be between 0 and 255 (inclusive) for 16/32-bit timers and between 0 and 65535 (inclusive) for 32/64-bit timers.

Description:

This function configures the value of the input clock prescaler match value. When in a half-width mode that uses the counter match and the prescaler, the prescale match effectively extends

the range of the match. The prescaler provides the least significant bits when counting down in periodic and one-shot modes; in all other modes, the prescaler provides the most significant bits.

Note:

The availability of the prescaler match varies with the Tiva part and timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

30.2.2.32 TimerPrescaleSet

Sets the timer prescale value.

Prototype:

```
void  
TimerPrescaleSet (uint32_t ui32Base,  
                  uint32_t ui32Timer,  
                  uint32_t ui32Value)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ui32Value is the timer prescale value which must be between 0 and 255 (inclusive) for 16/32-bit timers and between 0 and 65535 (inclusive) for 32/64-bit timers.

Description:

This function configures the value of the input clock prescaler. The prescaler is only operational when in half-width mode and is used to extend the range of the half-width timer modes. The prescaler provides the least significant bits when counting down in periodic and one-shot modes; in all other modes, the prescaler provides the most significant bits.

Note:

The availability of the prescaler varies with the Tiva part and timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

30.2.2.33 TimerRTCDisable

Disables RTC counting.

Prototype:

```
void  
TimerRTCDisable (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function causes the timer to stop counting when in RTC mode.

Returns:

None.

30.2.2.34 TimerRTCEnable

Enables RTC counting.

Prototype:

```
void
TimerRTCEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function causes the timer to start counting when in RTC mode. If not configured for RTC mode, this function does nothing.

Returns:

None.

30.2.2.35 TimerSynchronize

Synchronizes the counters in a set of timers.

Prototype:

```
void
TimerSynchronize(uint32_t ui32Base,
                  uint32_t ui32Timers)
```

Parameters:

ui32Base is the base address of the timer module. This parameter must be the base address of Timer0 (in other words, **TIMER0_BASE**).

ui32Timers is the set of timers to synchronize.

Description:

This function synchronizes the counters in a specified set of timers. When a timer is running in half-width mode, each half can be included or excluded in the synchronization event. When a timer is running in full-width mode, only the A timer can be synchronized (specifying the B timer has no effect).

The *ui32Timers* parameter is the logical OR of any of the following defines:

- **TIMER_0A_SYNC**
- **TIMER_0B_SYNC**
- **TIMER_1A_SYNC**
- **TIMER_1B_SYNC**
- **TIMER_2A_SYNC**

- **TIMER_2B_SYNC**
- **TIMER_3A_SYNC**
- **TIMER_3B_SYNC**
- **TIMER_4A_SYNC**
- **TIMER_4B_SYNC**
- **TIMER_5A_SYNC**
- **TIMER_5B_SYNC**
- **WTIMER_0A_SYNC**
- **WTIMER_0B_SYNC**
- **WTIMER_1A_SYNC**
- **WTIMER_1B_SYNC**
- **WTIMER_2A_SYNC**
- **WTIMER_2B_SYNC**
- **WTIMER_3A_SYNC**
- **WTIMER_3B_SYNC**
- **WTIMER_4A_SYNC**
- **WTIMER_4B_SYNC**
- **WTIMER_5A_SYNC**
- **WTIMER_5B_SYNC**

Note:

This functionality is not available on all parts.

Returns:

None.

30.2.2.36 TimerUpdateMode

This function configures the update of timer load and match settings.

Prototype:

```
void  
TimerUpdateMode(uint32_t ui32Base,  
                uint32_t ui32Timer,  
                uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ui32Config is a combination of the updates methods for the timers specified in the *ui32Timer* parameter.

Description:

This function configures how the timer updates the timer load and match values for the timers. The *ui32Timer* values can be **TIMER_A**, **TIMER_B**, or **TIMER_BOTH** to apply the settings in *ui32Config* to either timer or both timers. If the timer is not split then the **TIMER_A** should be used.

The *ui32Config* values affects when the [TimerLoadSet\(\)](#) and [TimerLoadSet64\(\)](#) values take effect.

- **TIMER_UP_LOAD_IMMEDIATE** is the default mode that causes the [TimerLoadSet\(\)](#) or [TimerLoadSet64\(\)](#) to update the timer counter immediately.
- **TIMER_UP_LOAD_TIMEOUT** causes the [TimerLoadSet\(\)](#) or [TimerLoadSet64\(\)](#) to update the timer when it counts down to zero.

Similarly the *ui32Config* value affects when the [TimerMatchSet\(\)](#) and [TimerMatchSet64\(\)](#) values take effect.

- **TIMER_UP_MATCH_IMMEDIATE** is the default mode that causes the [TimerMatchSet\(\)](#) or [TimerMatchSet64\(\)](#) to update the timer match value immediately.
- **TIMER_UP_MATCH_TIMEOUT** causes the [TimerMatchSet\(\)](#) or [TimerMatchSet64\(\)](#) to update the timer match value when it counts down to zero.

Note:

These settings have no effect if the timer is not in count down mode and are mostly useful when operating in PWM mode to allow for synchronous update of timer match and load values.

Returns:

None.

30.2.2.37 TimerValueGet

Gets the current timer value.

Prototype:

```
uint32_t
TimerValueGet(uint32_t ui32Base,
              uint32_t ui32Timer)
```

Parameters:

ui32Base is the base address of the timer module.

ui32Timer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for full-width operation.

Description:

This function reads the current value of the specified timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers and for half-width modes of 32/64-bit timers. Use [TimerValueGet64\(\)](#) for full-width modes of 32/64-bit timers.

Returns:

Returns the current value of the timer.

30.2.2.38 TimerValueGet64

Gets the current 64-bit timer value.

Prototype:

```
uint64_t
TimerValueGet64(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the timer module.

Description:

This function reads the current value of the specified timer.

Returns:

Returns the current value of the timer.

30.3 Programming Example

The following example shows how to use the timer API to configure the timer as a half-width one shot timer and a half-width edge capture counter.

```
//  
// Configure TimerA as a half-width one-shot timer, and TimerB as a  
// half-width edge capture counter.  
//  
TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_ONE_SHOT |  
    TIMER_CFG_B_CAP_COUNT));  
  
//  
// Set the count time for the the one-shot timer (TimerA).  
//  
TimerLoadSet(TIMER0_BASE, TIMER_A, 3000);  
  
//  
// Configure the counter (TimerB) to count both edges.  
//  
TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_BOTH_EDGES);  
  
//  
// Enable the timers.  
//  
TimerEnable(TIMER0_BASE, TIMER_BOTH);
```

31 UART

Introduction	559
API Functions	559
Programming Example	583

31.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Tiva UART modules. Functions are provided to configure and control the UART modules, to send and receive data, and to manage interrupts for the UART modules.

The Tiva UART performs the functions of parallel-to-serial and serial-to-parallel conversions. It is very similar in functionality to a 16C550 UART, but is not register-compatible.

Some of the features of the Tiva UART are:

- A 16x12 bit receive FIFO and a 16x8 bit transmit FIFO.
- Programmable baud rate generator.
- Automatic generation and stripping of start, stop, and parity bits.
- Line break generation and detection.
- Programmable serial interface
 - 5, 6, 7, or 8 data bits
 - even, odd, stick, or no parity bit generation and detection
 - 1 or 2 stop bit generation
 - baud rate generation, from DC to processor clock/16
- Modem control/flow control
- IrDA serial-IR (SIR) encoder/decoder.
- uDMA interface
- 9-bit operation

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

31.2 API Functions

Functions

- void `UART9BitAddrSend` (`uint32_t ui32Base, uint8_t ui8Addr`)
- void `UART9BitAddrSet` (`uint32_t ui32Base, uint8_t ui8Addr, uint8_t ui8Mask`)
- void `UART9BitDisable` (`uint32_t ui32Base`)
- void `UART9BitEnable` (`uint32_t ui32Base`)
- void `UARTBreakCtl` (`uint32_t ui32Base, bool bBreakState`)
- bool `UARTBusy` (`uint32_t ui32Base`)

- `int32_t UARTCharGet (uint32_t ui32Base)`
- `int32_t UARTCharGetNonBlocking (uint32_t ui32Base)`
- `void UARTCharPut (uint32_t ui32Base, unsigned char ucData)`
- `bool UARTCharPutNonBlocking (uint32_t ui32Base, unsigned char ucData)`
- `bool UARTCharsAvail (uint32_t ui32Base)`
- `uint32_t UARTClockSourceGet (uint32_t ui32Base)`
- `void UARTClockSourceSet (uint32_t ui32Base, uint32_t ui32Source)`
- `void UARTConfigGetExpClk (uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t *pui32Baud, uint32_t *pui32Config)`
- `void UARTConfigSetExpClk (uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t ui32Baud, uint32_t ui32Config)`
- `void UARTDisable (uint32_t ui32Base)`
- `void UARTDisableSIR (uint32_t ui32Base)`
- `void UARTDMADisable (uint32_t ui32Base, uint32_t ui32DMAFlags)`
- `void UARTDMAEnable (uint32_t ui32Base, uint32_t ui32DMAFlags)`
- `void UARTEnable (uint32_t ui32Base)`
- `void UARTEnableSIR (uint32_t ui32Base, bool bLowPower)`
- `void UARTFIFODisable (uint32_t ui32Base)`
- `void UARTFIFOEnable (uint32_t ui32Base)`
- `void UARTFIFOLevelGet (uint32_t ui32Base, uint32_t *pui32TxLevel, uint32_t *pui32RxLevel)`
- `void UARTFIFOLevelSet (uint32_t ui32Base, uint32_t ui32TxLevel, uint32_t ui32RxLevel)`
- `uint32_t UARTFlowControlGet (uint32_t ui32Base)`
- `void UARTFlowControlSet (uint32_t ui32Base, uint32_t ui32Mode)`
- `void UARTIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void UARTIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void UARTIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)`
- `void UARTIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))`
- `uint32_t UARTIntStatus (uint32_t ui32Base, bool bMasked)`
- `void UARTIntUnregister (uint32_t ui32Base)`
- `void UARTRModemControlClear (uint32_t ui32Base, uint32_t ui32Control)`
- `uint32_t UARTRModemControlGet (uint32_t ui32Base)`
- `void UARTRModemControlSet (uint32_t ui32Base, uint32_t ui32Control)`
- `uint32_t UARTRModemStatusGet (uint32_t ui32Base)`
- `uint32_t UARTRParityModeGet (uint32_t ui32Base)`
- `void UARTRParityModeSet (uint32_t ui32Base, uint32_t ui32Parity)`
- `void UARTRxErrorClear (uint32_t ui32Base)`
- `uint32_t UARTRxErrorGet (uint32_t ui32Base)`
- `void UARTRSmartCardDisable (uint32_t ui32Base)`
- `void UARTRSmartCardEnable (uint32_t ui32Base)`
- `bool UARTSpaceAvail (uint32_t ui32Base)`
- `uint32_t UARTRTxIntModeGet (uint32_t ui32Base)`
- `void UARTRTxIntModeSet (uint32_t ui32Base, uint32_t ui32Mode)`

31.2.1 Detailed Description

The UART API provides the set of functions required to implement an interrupt-driven UART driver. These functions may be used to control any of the available UART ports on a Tiva microcontroller and can be used with one port without causing conflicts with the other port.

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.

The clock source for the baud rate generator is handled by the [UARTClockSourceSet\(\)](#) and [UARTClockSourceGet\(\)](#) functions.

Configuration and control of the UART are handled by the [UARTConfigGetExpClk\(\)](#), [UARTConfigSetExpClk\(\)](#), [UARTDisable\(\)](#), [UARTEnable\(\)](#), [UARTParityModeGet\(\)](#), and [UARTParityModeSet\(\)](#) functions. The DMA interface can be enabled or disabled by the [UARTDMAEnable\(\)](#) and [UARTDMDisable\(\)](#) functions.

Sending and receiving data via the UART is handled by the [UARTCharGet\(\)](#), [UARTCharGetNonBlocking\(\)](#), [UARTCharPut\(\)](#), [UARTCharPutNonBlocking\(\)](#), [UARTBreakCtl\(\)](#), [UARTCharsAvail\(\)](#), and [UARTSpaceAvail\(\)](#) functions.

Managing the UART interrupts is handled by the [UARTIntClear\(\)](#), [UARTIntDisable\(\)](#), [UARTIntEnable\(\)](#), [UARTIntRegister\(\)](#), [UARTIntStatus\(\)](#), and [UARTIntUnregister\(\)](#) functions.

The 9-bit operation mode is handled by the [UART9BitEnable\(\)](#), [UART9BitDisable\(\)](#), [UART9BitAddrSet\(\)](#), and [UART9BitAddrSend\(\)](#) functions.

The [UARTConfigSet\(\)](#), [UARTConfigGet\(\)](#), [UARTCharNonBlockingGet\(\)](#), and [UARTCharNonBlockingPut\(\)](#) APIs from previous versions of the peripheral driver library have been replaced by the [UARTConfigSetExpClk\(\)](#), [UARTConfigGetExpClk\(\)](#), [UARTCharGetNonBlocking\(\)](#), and [UARTCharPutNonBlocking\(\)](#) APIs, respectively. Macros have been provided in `uart.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

31.2.2 Function Documentation

31.2.2.1 [UART9BitAddrSend](#)

Sends an address character from the specified port when operating in 9-bit mode.

Prototype:

```
void
UART9BitAddrSend(uint32_t ui32Base,
                  uint8_t ui8Addr)
```

Parameters:

ui32Base is the base address of the UART port.
ui8Addr is the address to be transmitted.

Description:

This function waits until all data has been sent from the specified port and then sends the given address as an address byte. It then waits until the address byte has been transmitted before returning.

The normal data functions ([UARTCharPut\(\)](#), [UARTCharPutNonBlocking\(\)](#), [UARTCharGet\(\)](#), and [UARTCharGetNonBlocking\(\)](#)) are used to send and receive data characters in 9-bit mode.

Note:

The availability of 9-bit mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.2 UART9BitAddrSet

Sets the device address(es) for 9-bit mode.

Prototype:

```
void  
UART9BitAddrSet(uint32_t ui32Base,  
                 uint8_t ui8Addr,  
                 uint8_t ui8Mask)
```

Parameters:

ui32Base is the base address of the UART port.

ui8Addr is the device address.

ui8Mask is the device address mask.

Description:

This function configures the device address or range of device addresses that respond to requests on the 9-bit UART port. The received address is masked with the mask and then compared against the given address, allowing either a single address (if **ui8Mask** is 0xff) or a set of addresses to be matched.

Note:

The availability of 9-bit mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.3 UART9BitDisable

Disables 9-bit mode on the specified UART.

Prototype:

```
void  
UART9BitDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function disables the 9-bit operational mode of the UART.

Note:

The availability of 9-bit mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.4 UART9BitEnable

Enables 9-bit mode on the specified UART.

Prototype:

```
void
UART9BitEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function enables the 9-bit operational mode of the UART.

Note:

The availability of 9-bit mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.5 UARTBreakCtl

Causes a BREAK to be sent.

Prototype:

```
void
UARTBreakCtl(uint32_t ui32Base,
              bool bBreakState)
```

Parameters:

ui32Base is the base address of the UART port.

bBreakState controls the output level.

Description:

Calling this function with **bBreakState** set to **true** asserts a break condition on the UART. Calling this function with **bBreakState** set to **false** removes the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

Returns:

None.

31.2.2.6 UARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:

```
bool  
UARTBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit FIFO is empty and all bits of the last transmitted character, including all stop bits, have left the hardware shift register.

Returns:

Returns **true** if the UART is transmitting or **false** if all transmissions are complete.

31.2.2.7 UARTCharGet

Waits for a character from the specified port.

Prototype:

```
int32_t  
UARTCharGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

Returns:

Returns the character read from the specified port, cast as a *int32_t*.

31.2.2.8 UARTCharGetNonBlocking

Receives a character from the specified port.

Prototype:

```
int32_t  
UARTCharGetNonBlocking(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function gets a character from the receive FIFO for the specified port.

Returns:

Returns the character read from the specified port, cast as a *int32_t*. A **-1** is returned if there are no characters present in the receive FIFO. The [UARTCharsAvail\(\)](#) function should be called before attempting to call this function.

31.2.2.9 UARTCharPut

Waits to send a character from the specified port.

Prototype:

```
void
UARTCharPut (uint32_t ui32Base,
             unsigned char ucData)
```

Parameters:

ui32Base is the base address of the UART port.

ucData is the character to be transmitted.

Description:

This function sends the character *ucData* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

Returns:

None.

31.2.2.10 UARTCharPutNonBlocking

Sends a character to the specified port.

Prototype:

```
bool
UARTCharPutNonBlocking (uint32_t ui32Base,
                        unsigned char ucData)
```

Parameters:

ui32Base is the base address of the UART port.

ucData is the character to be transmitted.

Description:

This function writes the character *ucData* to the transmit FIFO for the specified port. This function does not block, so if there is no space available, then a **false** is returned and the application must retry the function later.

Returns:

Returns **true** if the character was successfully placed in the transmit FIFO or **false** if there was no space available in the transmit FIFO.

31.2.2.11 UARTCharsAvail

Determines if there are any characters in the receive FIFO.

Prototype:

```
bool  
UARTCharsAvail(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns a flag indicating whether or not there is data available in the receive FIFO.

Returns:

Returns **true** if there is data in the receive FIFO or **false** if there is no data in the receive FIFO.

31.2.2.12 UARTClockSourceGet

Gets the baud clock source for the specified UART.

Prototype:

```
uint32_t  
UARTClockSourceGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the baud clock source for the specified UART. The possible baud clock source are the system clock (**UART_CLOCK_SYSTEM**) or the precision internal oscillator (**UART_CLOCK_PIOSC**).

Note:

The ability to specify the UART baud clock source varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.13 UARTClockSourceSet

Sets the baud clock source for the specified UART.

Prototype:

```
void  
UARTClockSourceSet(uint32_t ui32Base,  
                   uint32_t ui32Source)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Source is the baud clock source for the UART.

Description:

This function allows the baud clock source for the UART to be selected. The possible clock source are the system clock (**UART_CLOCK_SYSTEM**) or the precision internal oscillator (**UART_CLOCK_PIOSC**).

Changing the baud clock source changes the baud rate generated by the UART. Therefore, the baud rate should be reconfigured after any change to the baud clock source.

Note:

The ability to specify the UART baud clock source varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.14 **UARTConfigGetExpClk**

Gets the current configuration of a UART.

Prototype:

```
void
UARTConfigGetExpClk(uint32_t ui32Base,
                     uint32_t ui32UARTClk,
                     uint32_t *pui32Baud,
                     uint32_t *pui32Config)
```

Parameters:

ui32Base is the base address of the UART port.

ui32UARTClk is the rate of the clock supplied to the UART module.

pui32Baud is a pointer to storage for the baud rate.

pui32Config is a pointer to storage for the data format.

Description:

This function determines the baud rate and data format for the UART, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an “official” baud rate. The data format returned in *pui32Config* is enumerated the same as the *ui32Config* parameter of [UARTConfigSetExpClk\(\)](#).

The peripheral clock is the same as the processor clock. The frequency of the system clock is the value returned by [SysCtlClockGet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtlClockGet\(\)](#)).

For Tiva parts that have the ability to specify the UART baud clock source (via [UARTClockSourceSet\(\)](#)), the peripheral clock can be changed to PIOSC. In this case, the peripheral clock should be specified as 16, 000, 000 (the nominal rate of PIOSC).

Returns:

None.

31.2.2.15 UARTConfigSetExpClk

Sets the configuration of a UART.

Prototype:

```
void  
UARTConfigSetExpClk(uint32_t ui32Base,  
                     uint32_t ui32UARTClk,  
                     uint32_t ui32Baud,  
                     uint32_t ui32Config)
```

Parameters:

ui32Base is the base address of the UART port.

ui32UARTClk is the rate of the clock supplied to the UART module.

ui32Baud is the desired baud rate.

ui32Config is the data format for the port (number of data bits, number of stop bits, and parity).

Description:

This function configures the UART for operation in the specified data format. The baud rate is provided in the *ui32Baud* parameter and the data format in the *ui32Config* parameter.

The *ui32Config* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **UART_CONFIG_WLEN_8**, **UART_CONFIG_WLEN_7**, **UART_CONFIG_WLEN_6**, and **UART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **UART_CONFIG_STOP_ONE** and **UART_CONFIG_STOP_TWO** select one or two stop bits (respectively). **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, and **UART_CONFIG_PAR_ZERO** select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

The peripheral clock is the same as the processor clock. The frequency of the system clock is the value returned by [SysCtlClockGet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtlClockGet\(\)](#)).

For Tiva parts that have the ability to specify the UART baud clock source (via [UARTClockSourceSet\(\)](#)), the peripheral clock can be changed to PIOSC. In this case, the peripheral clock should be specified as 16, 000, 000 (the nominal rate of PIOSC).

Returns:

None.

31.2.2.16 UARTDisable

Disables transmitting and receiving.

Prototype:

```
void  
UARTDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function disables the UART, waits for the end of transmission of the current character, and flushes the transmit FIFO.

Returns:

None.

31.2.2.17 `UARTDisableSIR`

Disables SIR (IrDA) mode on the specified UART.

Prototype:

```
void
UARTDisableSIR(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function disables SIR(IrDA) mode on the UART. This function only has an effect if the UART has not been enabled by a call to [UARTEnable\(\)](#). The call [UARTEnableSIR\(\)](#) must be made before a call to [UARTConfigSetExpClk\(\)](#) because the [UARTConfigSetExpClk\(\)](#) function calls the [UARTEnable\(\)](#) function. Another option is to call [UARTDisable\(\)](#) followed by [UARTEnableSIR\(\)](#) and then enable the UART by calling [UARTEnable\(\)](#).

Note:

The availability of SIR (IrDA) operation varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.18 `UARTDMADisable`

Disable UART uDMA operation.

Prototype:

```
void
UARTDMADisable(uint32_t ui32Base,
                uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the UART port.

ui32DMAFlags is a bit mask of the uDMA features to disable.

Description:

This function is used to disable UART uDMA features that were enabled by [UARTDMAEnable\(\)](#). The specified UART uDMA features are disabled. The ***ui32DMAFlags*** parameter is the logical OR of any of the following values:

- **UART_DMA_RX** - disable uDMA for receive

- **UART_DMA_TX** - disable uDMA for transmit
- **UART_DMA_ERR_RXSTOP** - do not disable uDMA receive on UART error

Returns:

None.

31.2.2.19 **UARTDMAEnable**

Enable UART uDMA operation.

Prototype:

```
void  
UARTDMAEnable(uint32_t ui32Base,  
              uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the UART port.

ui32DMAFlags is a bit mask of the uDMA features to enable.

Description:

The specified UART uDMA features are enabled. The UART can be configured to use uDMA for transmit or receive and to disable receive if an error occurs. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- **UART_DMA_RX** - enable uDMA for receive
- **UART_DMA_TX** - enable uDMA for transmit
- **UART_DMA_ERR_RXSTOP** - disable uDMA receive on UART error

Note:

The uDMA controller must also be set up before DMA can be used with the UART.

Returns:

None.

31.2.2.20 **UARTEnable**

Enables transmitting and receiving.

Prototype:

```
void  
UARTEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function enables the UART and its transmit and receive FIFOs.

Returns:

None.

31.2.2.21 UARTEnableSIR

Enables SIR (IrDA) mode on the specified UART.

Prototype:

```
void
UARTEnableSIR(uint32_t ui32Base,
               bool bLowPower)
```

Parameters:

ui32Base is the base address of the UART port.

bLowPower indicates if SIR Low Power Mode is to be used.

Description:

This function enables SIR (IrDA) mode on the UART. If the *bLowPower* flag is set, then SIR low power mode will be selected as well. This function only has an effect if the UART has not been enabled by a call to [UARTEnable\(\)](#). The call [UARTEnableSIR\(\)](#) must be made before a call to [UARTConfigSetExpClk\(\)](#) because the [UARTConfigSetExpClk\(\)](#) function calls the [UARTEnable\(\)](#) function. Another option is to call [UARTDisable\(\)](#) followed by [UARTEnableSIR\(\)](#) and then enable the UART by calling [UARTEnable\(\)](#).

Note:

The availability of SIR (IrDA) operation varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.22 UARTFIFODisable

Disables the transmit and receive FIFOs.

Prototype:

```
void
UARTFIFODisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function disables the transmit and receive FIFOs in the UART.

Returns:

None.

31.2.2.23 UARTFIFOEnable

Enables the transmit and receive FIFOs.

Prototype:

```
void
UARTFIFOEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function enables the transmit and receive FIFOs in the UART.

Returns:

None.

31.2.2.24 UARTFIFOLevelGet

Gets the FIFO level at which interrupts are generated.

Prototype:

```
void  
UARTFIFOLevelGet (uint32_t ui32Base,  
                    uint32_t *pui32TxLevel,  
                    uint32_t *pui32RxLevel)
```

Parameters:

ui32Base is the base address of the UART port.

pui32TxLevel is a pointer to storage for the transmit FIFO level, returned as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

pui32RxLevel is a pointer to storage for the receive FIFO level, returned as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function gets the FIFO level at which transmit and receive interrupts are generated.

Returns:

None.

31.2.2.25 UARTFIFOLevelSet

Sets the FIFO level at which interrupts are generated.

Prototype:

```
void  
UARTFIFOLevelSet (uint32_t ui32Base,  
                   uint32_t ui32TxLevel,  
                   uint32_t ui32RxLevel)
```

Parameters:

ui32Base is the base address of the UART port.

ui32TxLevel is the transmit FIFO interrupt level, specified as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

ui32RxLevel is the receive FIFO interrupt level, specified as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function configures the FIFO level at which transmit and receive interrupts are generated.

Returns:

None.

31.2.2.26 UARTFlowControlGet

Returns the UART hardware flow control mode currently in use.

Prototype:

```
uint32_t
UARTFlowControlGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the current hardware flow control mode.

Note:

The availability of hardware flow control varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

Returns the current flow control mode in use. This value is a logical OR combination of values **UART_FLOWCONTROL_TX** if transmit (CTS) flow control is enabled and **UART_FLOWCONTROL_RX** if receive (RTS) flow control is in use. If hardware flow control is disabled, **UART_FLOWCONTROL_NONE** is returned.

31.2.2.27 UARTFlowControlSet

Sets the UART hardware flow control mode to be used.

Prototype:

```
void
UARTFlowControlSet(uint32_t ui32Base,
                   uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Mode indicates the flow control modes to be used. This parameter is a logical OR combination of values **UART_FLOWCONTROL_TX** and **UART_FLOWCONTROL_RX** to enable hardware transmit (CTS) and receive (RTS) flow control or **UART_FLOWCONTROL_NONE** to disable hardware flow control.

Description:

This function configures the required hardware flow control modes. If **ui32Mode** contains flag **UART_FLOWCONTROL_TX**, data is only transmitted if the incoming CTS signal is asserted. If **ui32Mode** contains flag **UART_FLOWCONTROL_RX**, the RTS output is controlled by the hardware and is asserted only when there is space available in the receive FIFO. If no hardware flow control is required, **UART_FLOWCONTROL_NONE** should be passed.

Note:

The availability of hardware flow control varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.28 UARTIntClear

Clears UART interrupt sources.

Prototype:

```
void  
UARTIntClear(uint32_t ui32Base,  
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the UART port.

ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified UART interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [UARTIntEnable\(\)](#).

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

31.2.2.29 UARTIntDisable

Disables individual UART interrupt sources.

Prototype:

```
void  
UARTIntDisable(uint32_t ui32Base,  
               uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the UART port.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [UARTIntEnable\(\)](#).

Returns:

None.

31.2.2.30 UARTIntEnable

Enables individual UART interrupt sources.

Prototype:

```
void
UARTIntEnable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the UART port.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **UART_INT_9BIT** - 9-bit Address Match interrupt
- **UART_INT_OE** - Overrun Error interrupt
- **UART_INT_BE** - Break Error interrupt
- **UART_INT_PE** - Parity Error interrupt
- **UART_INT_FE** - Framing Error interrupt
- **UART_INT_RT** - Receive Timeout interrupt
- **UART_INT_TX** - Transmit interrupt
- **UART_INT_RX** - Receive interrupt
- **UART_INT_DSR** - DSR interrupt
- **UART_INT_DCD** - DCD interrupt
- **UART_INT_CTS** - CTS interrupt
- **UART_INT_RI** - RI interrupt

Returns:

None.

31.2.2.31 UARTIntRegister

Registers an interrupt handler for a UART interrupt.

Prototype:

```
void  
UARTIntRegister(uint32_t ui32Base,  
                void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the UART port.

pfnHandler is a pointer to the function to be called when the UART interrupt occurs.

Description:

This function does the actual registering of the interrupt handler. This function enables the global interrupt in the interrupt controller; specific UART interrupts must be enabled via [UART-IntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

31.2.2.32 UARTIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t  
UARTIntStatus(uint32_t ui32Base,  
              bool bMasked)
```

Parameters:

ui32Base is the base address of the UART port.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function returns the interrupt status for the specified UART. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in [UARTIntEnable\(\)](#).

31.2.2.33 UARTIntUnregister

Unregisters an interrupt handler for a UART interrupt.

Prototype:

```
void  
UARTIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function does the actual unregistering of the interrupt handler. It clears the handler to be called when a UART interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

31.2.2.34 UARTModemControlClear

Clears the states of the DTR and/or RTS modem control signals.

Prototype:

```
void
UARTModemControlClear(uint32_t ui32Base,
                      uint32_t ui32Control)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Control is a bit-mapped flag indicating which modem control bits should be set.

Description:

This function clears the states of the DTR or RTS modem handshake outputs from the UART.

The *ui32Control* parameter is the logical OR of any of the following:

- **UART_OUTPUT_DTR** - The modem control DTR signal
- **UART_OUTPUT_RTS** - The modem control RTS signal

Note:

The availability of hardware modem handshake signals varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.35 UARTModemControlGet

Gets the states of the DTR and RTS modem control signals.

Prototype:

```
uint32_t
UARTModemControlGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the current states of each of the two UART modem control signals, DTR and RTS.

Note:

The availability of hardware modem handshake signals varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

Returns the states of the handshake output signals. This value is a logical OR combination of values **UART_OUTPUT_RTS** and **UART_OUTPUT_DTR** where the presence of each flag indicates that the associated signal is asserted.

31.2.2.36 UARTModemControlSet

Sets the states of the DTR and/or RTS modem control signals.

Prototype:

```
void  
UARTModemControlSet(uint32_t ui32Base,  
                     uint32_t ui32Control)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Control is a bit-mapped flag indicating which modem control bits should be set.

Description:

This function configures the states of the DTR or RTS modem handshake outputs from the UART.

The *ui32Control* parameter is the logical OR of any of the following:

- **UART_OUTPUT_DTR** - The modem control DTR signal
- **UART_OUTPUT_RTS** - The modem control RTS signal

Note:

The availability of hardware modem handshake signals varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.37 UARTModemStatusGet

Gets the states of the RI, DCD, DSR and CTS modem status signals.

Prototype:

```
uint32_t
UARTModemStatusGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the current states of each of the four UART modem status signals, RI, DCD, DSR and CTS.

Note:

The availability of hardware modem handshake signals varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

Returns the states of the handshake output signals. This value is a logical OR combination of values **UART_INPUT_RI**, **UART_INPUT_DCD**, **UART_INPUT_CTS** and **UART_INPUT_DSR** where the presence of each flag indicates that the associated signal is asserted.

31.2.2.38 UARTParityModeGet

Gets the type of parity currently being used.

Prototype:

```
uint32_t
UARTParityModeGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function gets the type of parity used for transmitting data and expected when receiving data.

Returns:

Returns the current parity settings, specified as one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**.

31.2.2.39 UARTParityModeSet

Sets the type of parity.

Prototype:

```
void
UARTParityModeSet (uint32_t ui32Base,
                   uint32_t ui32Parity)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Parity specifies the type of parity to use.

Description:

This function configures the type of parity to use for transmitting and expect when receiving. The *ui32Parity* parameter must be one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**. The last two parameters allow direct control of the parity bit; it is always either one or zero based on the mode.

Returns:

None.

31.2.2.40 UARTRxErrorClear

Clears all reported receiver errors.

Prototype:

```
void  
UARTRxErrorClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function is used to clear all receiver error conditions reported via [UARTRxErrorGet\(\)](#). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

Returns:

None.

31.2.2.41 UARTRxErrorGet

Gets current receiver errors.

Prototype:

```
uint32_t  
UARTRxErrorGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to [UARTCharGet\(\)](#) or [UARTCharGetNonBlocking\(\)](#) with the exception that the overrun error is set immediately when the overrun occurs rather than when a character is next read.

Returns:

Returns a logical OR combination of the receiver error flags, **UART_RXERROR_FRAMING**, **UART_RXERROR_PARITY**, **UART_RXERROR_BREAK** and **UART_RXERROR_OVERRUN**.

31.2.2.42 UARTSmartCardDisable

Disables ISO7816 smart card mode on the specified UART.

Prototype:

```
void
UARTSmartCardDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function clears the SMART (ISO7816 smart card) bit in the UART control register.

Note:

The availability of ISO7816 smart card mode varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.43 UARTSmartCardEnable

Enables ISO7816 smart card mode on the specified UART.

Prototype:

```
void
UARTSmartCardEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function enables the SMART control bit for the ISO7816 smart card mode on the UART. This call also sets 8-bit word length and even parity as required by ISO7816.

Note:

The availability of ISO7816 smart card mode varies with the Tiva part and UART in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.2.2.44 UARTSpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:

```
bool
UARTSpaceAvail(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns a flag indicating whether or not there is space available in the transmit FIFO.

Returns:

Returns **true** if there is space available in the transmit FIFO or **false** if there is no space available in the transmit FIFO.

31.2.2.45 UARTTxIntModeGet

Returns the current operating mode for the UART transmit interrupt.

Prototype:

```
uint32_t  
UARTTxIntModeGet (uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

This function returns the current operating mode for the UART transmit interrupt. The return value is **UART_TXINT_MODE_EOT** if the transmit interrupt is currently configured to be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter. The return value is **UART_TXINT_MODE_FIFO** if the interrupt is configured to be asserted based on the level of the transmit FIFO.

Note:

The availability of end-of-transmission mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

Returns **UART_TXINT_MODE_FIFO** or **UART_TXINT_MODE_EOT**.

31.2.2.46 UARTTxIntModeSet

Sets the operating mode for the UART transmit interrupt.

Prototype:

```
void  
UARTTxIntModeSet (uint32_t ui32Base,  
                   uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Mode is the operating mode for the transmit interrupt. It may be **UART_TXINT_MODE_EOT** to trigger interrupts when the transmitter is idle or **UART_TXINT_MODE_FIFO** to trigger based on the current transmit FIFO level.

Description:

This function allows the mode of the UART transmit interrupt to be set. By default, the transmit interrupt is asserted when the FIFO level falls past a threshold set via a call to [UARTFIFOLevelSet\(\)](#). Alternatively, if this function is called with *ui32Mode* set to **UART_TXINT_MODE_EOT**, the transmit interrupt is asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter.

Note:

The availability of end-of-transmission mode varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available.

Returns:

None.

31.3 Programming Example

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

```

// Initialize the UART. Set the baud rate, number of data bits, turn off
// parity, number of stop bits, and stick mode.
//
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                     (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                      UART_CONFIG_PAR_NONE));

//
// Enable the UART.
//
UARTEnable(UART0_BASE);

//
// Check for characters. Spin here until a character is placed
// into the receive FIFO.
//
while(!UARTCharsAvail(UART0_BASE))
{
}

//
// Get the character(s) in the receive FIFO.
//
while(UARTCharGetNonBlocking(UART0_BASE))
{
}

//
// Put a character in the output buffer.
//
UARTCharPut(UART0_BASE, 'c');

//
// Disable the UART.
//
UARTDisable(UART0_BASE);

```


32 uDMA Controller

Introduction	585
API Functions	586
Programming Example	606

32.1 Introduction

The Micro Direct Memory Access (uDMA) API provides functions to configure the Tiva uDMA controller. The uDMA controller is designed to work with the ARM Cortex-M processor and provides an efficient and low-overhead means of transferring blocks of data in the system.

The uDMA controller has the following features:

- dedicated channels for supported peripherals
- one channel each for receive and transmit for devices with receive and transmit paths
- dedicated channel for software initiated data transfers
- channels can be independently configured and operated
- an arbitration scheme that is configurable per channel
- two levels of priority
- subordinate to Cortex-M processor bus usage
- data sizes of 8, 16, or 32 bits
- address increment of byte, half-word, word, or none
- maskable device requests
- optional software initiated transfers on any channel
- interrupt on transfer completion

The uDMA controller supports several different transfer modes, allowing for complex transfer schemes. The following transfer modes are provided:

- **Basic** mode performs a simple transfer when a request is asserted by a device. This mode is appropriate to use with peripherals where the peripheral asserts the request signal whenever data should be transferred. The transfer pauses if the request is de-asserted, even if the transfer is not complete.
- **Auto-request** mode performs a simple transfer that is started by a request, but always completes the entire transfer, even if the request is de-asserted. This mode is appropriate to use with software-initiated transfers.
- **Ping-Pong** mode is used to transfer data to or from two buffers, switching from one buffer to the other as each buffer fills. This mode is appropriate to use with peripherals as a way to ensure a continuous flow of data to or from the peripheral. However, it is more complex to set up and requires code to manage the ping-pong buffers in the interrupt handler.
- **Memory scatter-gather** mode is a complex mode that provides a way to set up a list of transfer “tasks” for the uDMA controller. Blocks of data can be transferred to and from arbitrary locations in memory.

- **Peripheral scatter-gather** mode is similar to memory scatter-gather mode except that it is controlled by a peripheral request.

Detailed explanation of the various transfer modes is beyond the scope of this document. Please refer to the device data sheet for more information on the operation of the uDMA controller.

The naming convention for the microDMA controller is to use the Greek letter “mu” to represent “micro”. For the purposes of this document, and in the software library function names, a lower case “u” will be used in place of “mu” when the controller is referred to as “uDMA”.

This driver is contained in `driverlib/udma.c`, with `driverlib/udma.h` containing the API declarations for use by applications.

32.2 API Functions

Defines

- `uDMATaskStructEntry(ui32TransferCount, ui32ItemSize, ui32SrcIncrement, pvSrcAddr, ui32DstIncrement, pvDstAddr, ui32ArbSize, ui32Mode)`

Functions

- `void uDMAChannelAssign (uint32_t ui32Mapping)`
- `void uDMAChannelAttributeDisable (uint32_t ui32ChannelNum, uint32_t ui32Attr)`
- `void uDMAChannelAttributeEnable (uint32_t ui32ChannelNum, uint32_t ui32Attr)`
- `uint32_t uDMAChannelAttributeGet (uint32_t ui32ChannelNum)`
- `void uDMAChannelControlSet (uint32_t ui32ChannelStructIndex, uint32_t ui32Control)`
- `void uDMAChannelDisable (uint32_t ui32ChannelNum)`
- `void uDMAChannelEnable (uint32_t ui32ChannelNum)`
- `bool uDMAChannelIsEnabled (uint32_t ui32ChannelNum)`
- `uint32_t uDMAChannelModeGet (uint32_t ui32ChannelStructIndex)`
- `void uDMAChannelRequest (uint32_t ui32ChannelNum)`
- `void uDMAChannelScatterGatherSet (uint32_t ui32ChannelNum, uint32_t ui32TaskCount, void *pvTaskList, uint32_t ui32IsPeriphSG)`
- `void uDMAChannelSelectDefault (uint32_t ui32DefPeriphs)`
- `void uDMAChannelSelectSecondary (uint32_t ui32SecPeriphs)`
- `uint32_t uDMAChannelSizeGet (uint32_t ui32ChannelStructIndex)`
- `void uDMAChannelTransferSet (uint32_t ui32ChannelStructIndex, uint32_t ui32Mode, void *pvSrcAddr, void *pvDstAddr, uint32_t ui32TransferSize)`
- `void * uDMAControlAlternateBaseGet (void)`
- `void * uDMAControlBaseGet (void)`
- `void uDMAControlBaseSet (void *psControlTable)`
- `void uDMADisable (void)`
- `void uDMAEnable (void)`
- `void uDMAErrorStatusClear (void)`
- `uint32_t uDMAErrorStatusGet (void)`

- void [uDMAIntClear](#) (uint32_t ui32ChanMask)
- void [uDMAIntRegister](#) (uint32_t ui32IntChannel, void (*pfnHandler)(void))
- uint32_t [uDMAIntStatus](#) (void)
- void [uDMAIntUnregister](#) (uint32_t ui32IntChannel)

32.2.1 Detailed Description

The uDMA API functions provide a means to enable and configure the Tiva uDMA controller to perform DMA transfers.

The general order of function calls to set up and perform a uDMA transfer is the following:

- [uDMAEnable\(\)](#) is called once to enable the controller.
- [uDMAControlBaseSet\(\)](#) is called once to set the channel control table.
- [uDMAChannelAttributeEnable\(\)](#) is called once or infrequently to configure the behavior of the channel.
- [uDMAChannelControlSet\(\)](#) is used to set up characteristics of the data transfer. It is only called once if the nature of the data transfer does not change.
- [uDMAChannelTransferSet\(\)](#) is used to set the buffer pointers and size for a transfer. It is called before each new transfer.
- [uDMAChannelEnable\(\)](#) enables a channel to perform data transfers.
- [uDMAChannelRequest\(\)](#) is used to initiate a software based transfer. This is normally not used for peripheral based transfers.

In order to use the uDMA controller, you must first enable it by calling [uDMAEnable\(\)](#). You can later disable it, if no longer needed, by calling [uDMADisable\(\)](#).

Once the uDMA controller is enabled, you must tell it where to find the channel control structures in system memory by using the function [uDMAControlBaseSet\(\)](#) and passing a pointer to the base of the channel control structure. The control structure must be allocated by the application. One way to do allocate the control structure is to declare an array of data type `int8_t` or `uint8_t`. In order to support all channels and transfer modes, the control table array should be 1024 bytes, but it can be fewer depending on transfer modes used and number of channels actually used.

Note:

The control table must be aligned on a 1024-byte boundary.

The uDMA controller supports multiple channels. Each channel has a set of attribute flags to control certain uDMA features and channel behavior. The attribute flags are configured with the function [uDMAChannelAttributeEnable\(\)](#) and cleared with [uDMAChannelAttributeDisable\(\)](#). The setting of the channel attribute flags can be queried using the function [uDMAChannelAttributeGet\(\)](#).

Next, the control parameters of the DMA transfer must be configured. These parameters control the size and address increment of the data items to be transferred. The function [uDMAChannelControlSet\(\)](#) is used to set up these control parameters.

All of the functions mentioned so far are used only once or infrequently to set up the uDMA channel and transfer. In order to configure the transfer addresses, transfer size, and transfer mode, use the function [uDMAChannelTransferSet\(\)](#). This function must be called for each new transfer. Once everything is set up, the channel is enabled by calling [uDMAChannelEnable\(\)](#), which must be done

before each new transfer. The uDMA controller automatically disables the channel at the completion of a transfer. A channel can be manually disabled by using [uDMAChannelDisable\(\)](#).

There are additional functions that can be used to query the status of a channel, either from an interrupt handler or in polling fashion. The function [uDMAChannelSizeGet\(\)](#) is used to find the amount of data remaining to transfer on a channel. This value is zero when a transfer is complete. The function [uDMAChannelModeGet\(\)](#) can be used to find the transfer mode of a uDMA channel. This function is usually used to see if the mode indicates stopped, meaning that a transfer has completed on a channel that was previously running. The function [uDMAChannelsEnabled\(\)](#) can be used to determine if a particular channel is enabled.

If the application is using run-time interrupt registration (see [IntRegister\(\)](#)), then the function [uDMAIntRegister\(\)](#) can be used to install an interrupt handler for the uDMA controller. This function also enables the interrupt on the system interrupt controller. If compile-time interrupt registration is used, then call the function [IntEnable\(\)](#) to enable uDMA interrupts. When an interrupt handler has been installed with [uDMAIntRegister\(\)](#), it can be removed by calling [uDMAIntUnregister\(\)](#).

This interrupt handler is only for software-initiated transfers or errors. uDMA interrupts for a peripheral occur on the peripheral's dedicated interrupt channel and should be handled by the peripheral interrupt handler. It is not necessary to acknowledge or clear uDMA interrupt sources. They are cleared automatically when they are serviced.

The uDMA interrupt handler should use the function [uDMAErrorStatusGet\(\)](#) to test if a uDMA error occurred. If so, the interrupt must be cleared by calling [uDMAErrorStatusClear\(\)](#).

Note:

Many of the API functions take a channel parameter that includes the logical OR of one of the values **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose the primary or alternate control structure. For Basic and Auto transfer modes, only the primary control structure is needed. The alternate control structure is only needed for complex transfer modes of Ping-pong or Scatter-gather. Refer to the device data sheet for detailed information about transfer modes.

Special considerations for using scatter-gather operations

In order to use the scatter-gather modes of the uDMA controller, you must prepare a "task" list in memory that describes the scatter-gather operations. There is a helper macro, [uDMATaskStructEntry](#) provided to help create the initialization values for the task list structure. Please see the documentation for this macro which includes a code snippet showing how it is used.

Once the task list is prepared, the appropriate uDMA channel must be configured for a scatter-gather operation. The best way to do this is to use the function [uDMAChannelScatterGatherSet\(\)](#). Alternatively, the functions [uDMAChannelControlSet\(\)](#) followed by [uDMAChannelTransferSet\(\)](#) can also be used.

Note:

The scatter-gather task list must be resident in SRAM. The uDMA controller cannot read from flash memory.

About uDMA Channel Function Parameters

Many of the uDMA API functions require a channel number as a parameter. There are two different uses of the channel number. In some cases, it is the number of the uDMA channel and is used to read or write registers within the uDMA controller. In this case, it is simply the channel number with no additional qualifier.

However, in other cases the channel number that is supplied as a parameter is really an index into

the uDMA channel control structure. Because every uDMA channel has a primary and an alternate channel control structure, this index must also be specified as part of the channel number. The index is specified by passing a value for the channel parameter that is the logical OR of the actual channel number and one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**. The default is the same as **UDMA_PRI_SELECT** so if you do not specify, the primary channel control structure is used, which is the right thing in most cases.

Note:

When **UDMA_ALT_SELECT** is specified, what is really happening is that channel index 32-63 is being used, because the alternate channel control structures for channels 0-31 are located at index locations 32-63 in the channel control table.

Here is an example of the first case. In this example, a uDMA channel is enabled, and only the channel number is used because this is programming a register in the uDMA controller.

```
uDMAChannelEnable(UDMA_CHANNEL_UART0RX);
```

Here is an example of the second case. In this example, the channel control structure is to be modified to configure some transfer parameters. Therefore in addition to specifying the channel index, the primary or alternate control structure must also be selected.

```
uDMAChannelControlSet(UDMA_CHANNEL_UART0RX | UDMA_PRI_SELECT, ...);
```

In order to help make it clear when one or the other form is to be used, the parameters are named differently in the API description. For functions that require just the channel number, the name of the parameter is *ulChannelNum*. For functions that require the channel index of the channel control structure, the name of the parameter is *ulChannelStructIdx*.

Selecting uDMA Channels

The uDMA controller has 32 channels, and therefore most of the API functions take a channel number with a value from 0-31 or a channel index with a value from 0-63 (the 32-63 is specified with the logical OR of the channel number with **UDMA_ALT_SELECT**). In order to avoid the need for hardcoded channel numbers in code, macros are provided that map channel names to channel numbers.

To use the default channel mapping, you may use one of the following choices whenever a channel number or index is needed. This list is all the possible channels that are defined by the API. However not all channels are available on all parts, depending on which peripherals are available on the part and which of those support uDMA. Please consult the data sheet for your specific part to see which uDMA channels are supported.

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit
- **UDMA_CHANNEL_ETH0RX** for ethernet receive
- **UDMA_CHANNEL_ETH0TX** for ethernet transmit
- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel

- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_ADC0** for ADC0 sequencer 0
- **UDMA_CHANNEL_ADC1** for ADC0 sequencer 1
- **UDMA_CHANNEL_ADC2** for ADC0 sequencer 2
- **UDMA_CHANNEL_ADC3** for ADC0 sequencer 3
- **UDMA_CHANNEL_TMR0A** for Timer 0A
- **UDMA_CHANNEL_TMR0B** for Timer 0B
- **UDMA_CHANNEL_TMR1A** for Timer 1A
- **UDMA_CHANNEL_TMR1B** for Timer 1B
- **UDMA_CHANNEL_I2S0RX** for I2S receive
- **UDMA_CHANNEL_I2S0TX** for I2S transmit
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

Some Tiva parts also provide a secondary channel mapping. For those parts, each channel has a secondary peripheral mapping, allowing more choices in channel mapping and to allow some additional peripherals to use uDMA that are not available in the default mapping.

In order to select the default or secondary channel mapping, use the functions [uDMAChannelSelectDefault\(\)](#) or [uDMAChannelSelectSecondary\(\)](#). Each channel can be configured individually to use the default or secondary mapping.

For example, the default for channel 0 is USBEP1RX. However this channel also has a secondary mapping to UART2RX. If an application requires use of uDMA with UART2 and does not use USB, then this channel could be remapped to UART2RX with the following function call:

```
uDMAChannelSelectSecondary (UDMA_DEF_USBEP1RX_SEC_UART2RX) ;
```

For channels that have been configured to use the secondary mapping, there is a set of macros to use for specifying the channel. Here is the list of channels when secondary mapping is used. As before, this is the full list, the actual channels available depend on which specific Tiva part is used.

- **UDMA_SEC_CHANNEL_UART2RX_0** for UART2 receive using uDMA channel 0
- **UDMA_SEC_CHANNEL_UART2TX_1** for UART2 transmit using uDMA channel 1
- **UDMA_SEC_CHANNEL_TMR3A** for Timer 3A
- **UDMA_SEC_CHANNEL_TMR3B** for Timer 3B
- **UDMA_SEC_CHANNEL_TMR2A_4** for Timer 2A using uDMA channel 4
- **UDMA_SEC_CHANNEL_TMR2B_5** for Timer 2B using uDMA channel 5
- **UDMA_SEC_CHANNEL_TMR2A_6** for Timer 2A using uDMA channel 6
- **UDMA_SEC_CHANNEL_TMR2B_7** for Timer 2B using uDMA channel 7

- **UDMA_SEC_CHANNEL_UART1RX** for UART1 receive
- **UDMA_SEC_CHANNEL_UART1TX** for UART1 transmit
- **UDMA_SEC_CHANNEL_SSI1RX** for SSI1 receive
- **UDMA_SEC_CHANNEL_SSI1TX** for SSI1 transmit
- **UDMA_SEC_CHANNEL_UART2RX_12** for UART2 receive using uDMA channel 12
- **UDMA_SEC_CHANNEL_UART2TX_13** for UART2 transmit using uDMA channel 13
- **UDMA_SEC_CHANNEL_TMR2A_14** for Timer 2A using uDMA channel 14
- **UDMA_SEC_CHANNEL_TMR2B_15** for Timer 2B using uDMA channel 15
- **UDMA_SEC_CHANNEL_TMR1A** for Timer 1A
- **UDMA_SEC_CHANNEL_TMR1B** for Timer 1B
- **UDMA_SEC_CHANNEL_EPI0RX** for EPI read
- **UDMA_SEC_CHANNEL_EPI0TX** for EPI write
- **UDMA_SEC_CHANNEL_ADC10** for ADC1 sequencer 0
- **UDMA_SEC_CHANNEL_ADC11** for ADC1 sequencer 1
- **UDMA_SEC_CHANNEL_ADC12** for ADC1 sequencer 2
- **UDMA_SEC_CHANNEL_ADC13** for ADC1 sequencer 3
- **UDMA_SEC_CHANNEL_SW** for the software dedicated uDMA channel

Further, some Tiva parts provide up to five possible channel assignments. For those parts, us the [uDMAChannelAssign\(\)](#) function to configure the channel assignments.

32.2.2 Define Documentation

32.2.2.1 uDMATaskStructEntry

A helper macro for building scatter-gather task table entries.

Definition:

```
#define uDMATaskStructEntry(ui32TransferCount,
                           ui32ItemSize,
                           ui32SrcIncrement,
                           pvSrcAddr,
                           ui32DstIncrement,
                           pvDstAddr,
                           ui32ArbSize,
                           ui32Mode)
```

Parameters:

ui32TransferCount is the count of items to transfer for this task.
ui32ItemSize is the bit size of the items to transfer for this task.
ui32SrcIncrement is the bit size increment for source data.
pvSrcAddr is the starting address of the data to transfer.
ui32DstIncrement is the bit size increment for destination data.
pvDstAddr is the starting address of the destination data.
ui32ArbSize is the arbitration size to use for the transfer task.

ui32Mode is the transfer mode for this task.

Description:

This macro is intended to be used to help populate a table of uDMA tasks for a scatter-gather transfer. This macro will calculate the values for the fields of a task structure entry based on the input parameters.

There are specific requirements for the values of each parameter. No checking is done so it is up to the caller to ensure that correct values are used for the parameters.

The *ui32TransferCount* parameter is the number of items that will be transferred by this task. It must be in the range 1-1024.

The *ui32ItemSize* parameter is the bit size of the transfer data. It must be one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32**.

The *ui32SrcIncrement* parameter is the increment size for the source data. It must be one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE**.

The *pvSrcAddr* parameter is a void pointer to the beginning of the source data.

The *ui32DstIncrement* parameter is the increment size for the destination data. It must be one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_DST_INC_NONE**.

The *pvDstAddr* parameter is a void pointer to the beginning of the location where the data will be transferred.

The *ui32ArbSize* parameter is the arbitration size for the transfer, and must be one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, and so on up to **UDMA_ARB_1024**. This is used to select the arbitration size in powers of 2, from 1 to 1024.

The *ui32Mode* parameter is the mode to use for this transfer task. It must be one of **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**. Note that normally all tasks will be one of the scatter-gather modes while the last task is a task list will be AUTO or BASIC.

This macro is intended to be used to initialize individual entries of a structure of tDMAControlTable type, like this:

```
tDMAControlTable MyTaskList[] =
{
    uDMATaskStructEntry(Task1Count,    UDMA_SIZE_8,
                        UDMA_SRC_INC_8,   MySourceBuf,
                        UDMA_DST_INC_8,   MyDestBuf,
                        UDMA_ARB_8,      UDMA_MODE_MEM_SCATTER_GATHER),
    uDMATaskStructEntry(Task2Count,    ...),
}
```

Returns:

Nothing; this is not a function.

32.2.3 Function Documentation

32.2.3.1 uDMAChannelAssign

Assigns a peripheral mapping for a uDMA channel.

Prototype:

```
void
uDMAChannelAssign(uint32_t ui32Mapping)
```

Parameters:

ui32Mapping is a macro specifying the peripheral assignment for a channel.

Description:

This function assigns a peripheral mapping to a uDMA channel. It is used to select which peripheral is used for a uDMA channel. The parameter *ui32Mapping* should be one of the macros named **UDMA_CHn_ttt** from the header file *udma.h*. For example, to assign uDMA channel 0 to the UART2 RX channel, the parameter should be the macro **UDMA_CH0_UART2RX**.

Please consult the Tiva data sheet for a table showing all the possible peripheral assignments for the uDMA channels for a particular device.

Note:

This function is only available on devices that have the DMA Channel Map Select registers (DMACHMAP0-3). Please consult the data sheet for your part.

Returns:

None.

32.2.3.2 uDMAChannelAttributeDisable

Disables attributes of a uDMA channel.

Prototype:

```
void
uDMAChannelAttributeDisable(uint32_t ui32ChannelNum,
                            uint32_t ui32Attr)
```

Parameters:

ui32ChannelNum is the channel to configure.

ui32Attr is a combination of attributes for the channel.

Description:

This function is used to disable attributes of a uDMA channel.

The *ui32Attr* parameter is the logical OR of any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:

None.

32.2.3.3 uDMAChannelAttributeEnable

Enables attributes of a uDMA channel.

Prototype:

```
void  
uDMAChannelAttributeEnable(uint32_t ui32ChannelNum,  
                           uint32_t ui32Attr)
```

Parameters:

ui32ChannelNum is the channel to configure.

ui32Attr is a combination of attributes for the channel.

Description:

This function is used to enable attributes of a uDMA channel.

The *ui32Attr* parameter is the logical OR of any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel (it is very unlikely that this flag should be used).
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:

None.

32.2.3.4 uDMAChannelAttributeGet

Gets the enabled attributes of a uDMA channel.

Prototype:

```
uint32_t  
uDMAChannelAttributeGet(uint32_t ui32ChannelNum)
```

Parameters:

ui32ChannelNum is the channel to configure.

Description:

This function returns a combination of flags representing the attributes of the uDMA channel.

Returns:

Returns the logical OR of the attributes of the uDMA channel, which can be any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

32.2.3.5 uDMAChannelControlSet

Sets the control parameters for a uDMA channel control structure.

Prototype:

```
void
uDMAChannelControlSet(uint32_t ui32ChannelStructIndex,
                      uint32_t ui32Control)
```

Parameters:

ui32ChannelStructIndex is the logical OR of the uDMA channel number with **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ui32Control is logical OR of several control values to set the control parameters for the channel.

Description:

This function is used to set control parameters for a uDMA transfer. These parameters are typically not changed often.

The ***ui32ChannelStructIndex*** parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The ***ui32Control*** parameter is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. The choices available for each of these values is described below.

Choose the data size from one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32** to select a data size of 8, 16, or 32 bits.

Choose the source address increment from one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE** to select an address increment of 8-bit bytes, 16-bit half-words, 32-bit words, or to select non-incrementing.

Choose the destination address increment from one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_DST_INC_NONE** to select an address increment of 8-bit bytes, 16-bit half-words, 32-bit words, or to select non-incrementing.

The arbitration size determines how many items are transferred before the uDMA controller re-arbitrates for the bus. Choose the arbitration size from one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, **UDMA_ARB_8**, through **UDMA_ARB_1024** to select the arbitration size from 1 to 1024 items, in powers of 2.

The value **UDMA_NEXT_USEBURST** is used to force the channel to only respond to burst requests at the tail end of a scatter-gather transfer.

Note:

The address increment cannot be smaller than the data size.

Returns:

None.

32.2.3.6 uDMAChannelDisable

Disables a uDMA channel for operation.

Prototype:

```
void  
uDMAChannelDisable(uint32_t ui32ChannelNum)
```

Parameters:

ui32ChannelNum is the channel number to disable.

Description:

This function disables a specific uDMA channel. Once disabled, a channel cannot respond to uDMA transfer requests until re-enabled via [uDMAChannelEnable\(\)](#).

Returns:

None.

32.2.3.7 uDMAChannelEnable

Enables a uDMA channel for operation.

Prototype:

```
void  
uDMAChannelEnable(uint32_t ui32ChannelNum)
```

Parameters:

ui32ChannelNum is the channel number to enable.

Description:

This function enables a specific uDMA channel for use. This function must be used to enable a channel before it can be used to perform a uDMA transfer.

When a uDMA transfer is completed, the channel is automatically disabled by the uDMA controller. Therefore, this function should be called prior to starting up any new transfer.

Returns:

None.

32.2.3.8 uDMAChannelsEnabled

Checks if a uDMA channel is enabled for operation.

Prototype:

```
bool  
uDMAChannelIsEnabled(uint32_t ui32ChannelNum)
```

Parameters:

ui32ChannelNum is the channel number to check.

Description:

This function checks to see if a specific uDMA channel is enabled. This function can be used to check the status of a transfer, as the channel is automatically disabled at the end of a transfer.

Returns:

Returns **true** if the channel is enabled, **false** if disabled.

32.2.3.9 uDMAChannelModeGet

Gets the transfer mode for a uDMA channel control structure.

Prototype:

```
uint32_t
uDMAChannelModeGet (uint32_t ui32ChannelStructIndex)
```

Parameters:

ui32ChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the transfer mode for the uDMA channel and to query the status of a transfer on a channel. When the transfer is complete the mode is **UDMA_MODE_STOP**.

Returns:

Returns the transfer mode of the specified channel and control structure, which is one of the following values: **UDMA_MODE_STOP**, **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_PINGPONG**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**.

32.2.3.10 uDMAChannelRequest

Requests a uDMA channel to start a transfer.

Prototype:

```
void
uDMAChannelRequest (uint32_t ui32ChannelNum)
```

Parameters:

ui32ChannelNum is the channel number on which to request a uDMA transfer.

Description:

This function allows software to request a uDMA channel to begin a transfer. This function could be used for performing a memory-to-memory transfer, or if for some reason a transfer needs to be initiated by software instead of the peripheral associated with that channel.

Note:

If the channel is **UDMA_CHANNEL_SW** and interrupts are used, then the completion is signaled on the uDMA dedicated interrupt. If a peripheral channel is used, then the completion is signaled on the peripheral's interrupt.

Returns:

None.

32.2.3.11 uDMAChannelScatterGatherSet

Configures a uDMA channel for scatter-gather mode.

Prototype:

```
void  
uDMAChannelScatterGatherSet(uint32_t ui32ChannelNum,  
                           uint32_t ui32TaskCount,  
                           void *pvTaskList,  
                           uint32_t ui32IsPeriphSG)
```

Parameters:

ui32ChannelNum is the uDMA channel number.

ui32TaskCount is the number of scatter-gather tasks to execute.

pvTaskList is a pointer to the beginning of the scatter-gather task list.

ui32IsPeriphSG is a flag to indicate it is a peripheral scatter-gather transfer (else it is memory scatter-gather transfer)

Description:

This function is used to configure a channel for scatter-gather mode. The caller must have already set up a task list and must pass a pointer to the start of the task list as the *pvTaskList* parameter. The *ui32TaskCount* parameter is the count of tasks in the task list, not the size of the task list. The flag *bIsPeriphSG* should be used to indicate if scatter-gather should be configured for peripheral or memory operation.

See also:

[uDMA_TaskStructEntry](#)

Returns:

None.

32.2.3.12 uDMAChannelSelectDefault

Selects the default peripheral for a set of uDMA channels.

Prototype:

```
void  
uDMAChannelSelectDefault(uint32_t ui32DefPeriphs)
```

Parameters:

ui32DefPeriphs is the logical OR of the uDMA channels for which to use the default peripheral, instead of the secondary peripheral.

Description:

This function is used to select the default peripheral assignment for a set of uDMA channels.

The parameter *ui32DefPeriphs* can be the logical OR of any of the following macros. If one of the macros below is in the list passed to this function, then the default peripheral (marked as *_DEF_*) is selected.

- **UDMA_DEF_USBEP1RX_SEC_UART2RX**
- **UDMA_DEF_USBEP1TX_SEC_UART2TX**
- **UDMA_DEF_USBEP2RX_SEC_TMR3A**
- **UDMA_DEF_USBEP2TX_SEC_TMR3B**
- **UDMA_DEF_USBEP3RX_SEC_TMR2A**
- **UDMA_DEF_USBEP3TX_SEC_TMR2B**

- UDMA_DEF_ETH0RX_SEC_TMR2A
- UDMA_DEF_ETH0TX_SEC_TMR2B
- UDMA_DEF_UART0RX_SEC_UART1RX
- UDMA_DEF_UART0TX_SEC_UART1TX
- UDMA_DEF_SSI0RX_SEC_SSI1RX
- UDMA_DEF_SSI0TX_SEC_SSI1TX
- UDMA_DEF_RESERVED_SEC_UART2RX
- UDMA_DEF_RESERVED_SEC_UART2TX
- UDMA_DEF_ADC00_SEC_TMR2A
- UDMA_DEF_ADC01_SEC_TMR2B
- UDMA_DEF_ADC02_SEC_RESERVED
- UDMA_DEF_ADC03_SEC_RESERVED
- UDMA_DEF_TMR0A_SEC_TMR1A
- UDMA_DEF_TMR0B_SEC_TMR1B
- UDMA_DEF_TMR1A_SEC_EPIORX
- UDMA_DEF_TMR1B_SEC_EPIOTX
- UDMA_DEF_UART1RX_SEC_RESERVED
- UDMA_DEF_UART1TX_SEC_RESERVED
- UDMA_DEF_SSI1RX_SEC_ADC10
- UDMA_DEF_SSI1TX_SEC_ADC11
- UDMA_DEF_RESERVED_SEC_ADC12
- UDMA_DEF_RESERVED_SEC_ADC13
- UDMA_DEF_I2S0RX_SEC_RESERVED
- UDMA_DEF_I2S0TX_SEC_RESERVED

Returns:

None.

32.2.3.13 uDMAChannelSelectSecondary

Selects the secondary peripheral for a set of uDMA channels.

Prototype:

```
void
uDMAChannelSelectSecondary(uint32_t ui32SecPeriphs)
```

Parameters:

ui32SecPeriphs is the logical OR of the uDMA channels for which to use the secondary peripheral, instead of the default peripheral.

Description:

This function is used to select the secondary peripheral assignment for a set of uDMA channels. By selecting the secondary peripheral assignment for a channel, the default peripheral assignment is no longer available for that channel.

The parameter *ui32SecPeriphs* can be the logical OR of any of the following macros. If one of the macros below is in the list passed to this function, then the secondary peripheral (marked as **_SEC_**) is selected.

- UDMA_DEF_USBEP1RX_SEC_UART2RX
- UDMA_DEF_USBEP1TX_SEC_UART2TX
- UDMA_DEF_USBEP2RX_SEC_TMR3A
- UDMA_DEF_USBEP2TX_SEC_TMR3B
- UDMA_DEF_USBEP3RX_SEC_TMR2A
- UDMA_DEF_USBEP3TX_SEC_TMR2B
- UDMA_DEF_ETH0RX_SEC_TMR2A
- UDMA_DEF_ETH0TX_SEC_TMR2B
- UDMA_DEF_UART0RX_SEC_UART1RX
- UDMA_DEF_UART0TX_SEC_UART1TX
- UDMA_DEF_SSI0RX_SEC_SSI1RX
- UDMA_DEF_SSI0TX_SEC_SSI1TX
- UDMA_DEF_RESERVED_SEC_UART2RX
- UDMA_DEF_RESERVED_SEC_UART2TX
- UDMA_DEF_ADC00_SEC_TMR2A
- UDMA_DEF_ADC01_SEC_TMR2B
- UDMA_DEF_ADC02_SEC_RESERVED
- UDMA_DEF_ADC03_SEC_RESERVED
- UDMA_DEF_TMR0A_SEC_TMR1A
- UDMA_DEF_TMR0B_SEC_TMR1B
- UDMA_DEF_TMR1A_SEC_EPIO_RX
- UDMA_DEF_TMR1B_SEC_EPIO_TX
- UDMA_DEF_UART1RX_SEC_RESERVED
- UDMA_DEF_UART1TX_SEC_RESERVED
- UDMA_DEF_SSI1RX_SEC_ADC10
- UDMA_DEF_SSI1TX_SEC_ADC11
- UDMA_DEF_RESERVED_SEC_ADC12
- UDMA_DEF_RESERVED_SEC_ADC13
- UDMA_DEF_I2S0RX_SEC_RESERVED
- UDMA_DEF_I2S0TX_SEC_RESERVED

Returns:

None.

32.2.3.14 uDMAChannelSizeGet

Gets the current transfer size for a uDMA channel control structure.

Prototype:

```
uint32_t  
uDMAChannelSizeGet(uint32_t ui32ChannelStructIndex)
```

Parameters:

ui32ChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the uDMA transfer size for a channel. The transfer size is the number of items to transfer, where the size of an item might be 8, 16, or 32 bits. If a partial transfer has already occurred, then the number of remaining items is returned. If the transfer is complete, then 0 is returned.

Returns:

Returns the number of items remaining to transfer.

32.2.3.15 uDMAChannelTransferSet

Sets the transfer parameters for a uDMA channel control structure.

Prototype:

```
void
uDMAChannelTransferSet (uint32_t ui32ChannelStructIndex,
                        uint32_t ui32Mode,
                        void *pvSrcAddr,
                        void *pvDstAddr,
                        uint32_t ui32TransferSize)
```

Parameters:

ui32ChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ui32Mode is the type of uDMA transfer.

pvSrcAddr is the source address for the transfer.

pvDstAddr is the destination address for the transfer.

ui32TransferSize is the number of data items to transfer.

Description:

This function is used to configure the parameters for a uDMA transfer. These parameters are not typically changed often. The function [uDMAChannelControlSet\(\)](#) MUST be called at least once for this channel prior to calling this function.

The **ui32ChannelStructIndex** parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The **ui32Mode** parameter should be one of the following values:

- **UDMA_MODE_STOP** stops the uDMA transfer. The controller sets the mode to this value at the end of a transfer.
- **UDMA_MODE_BASIC** to perform a basic transfer based on request.
- **UDMA_MODE_AUTO** to perform a transfer that always completes once started even if the request is removed.
- **UDMA_MODE_PINGPONG** to set up a transfer that switches between the primary and alternate control structures for the channel. This mode allows use of ping-pong buffering for uDMA transfers.
- **UDMA_MODE_MEM_SCATTER_GATHER** to set up a memory scatter-gather transfer.
- **UDMA_MODE_PER_SCATTER_GATHER** to set up a peripheral scatter-gather transfer.

The *pvSrcAddr* and *pvDstAddr* parameters are pointers to the first location of the data to be transferred. These addresses should be aligned according to the item size. The compiler takes care of this alignment if the pointers are pointing to storage of the appropriate data type.

The *ui32TransferSize* parameter is the number of data items, not the number of bytes.

The two scatter-gather modes, memory and peripheral, are actually different depending on whether the primary or alternate control structure is selected. This function looks for the **UDMA_PRI_SELECT** and **UDMA_ALT_SELECT** flag along with the channel number and sets the scatter-gather mode as appropriate for the primary or alternate control structure.

The channel must also be enabled using [uDMAChannelEnable\(\)](#) after calling this function. The transfer does not begin until the channel has been configured and enabled. Note that the channel is automatically disabled after the transfer is completed, meaning that [uDMAChannelEnable\(\)](#) must be called again after setting up the next transfer.

Note:

Great care must be taken to not modify a channel control structure that is in use or else the results are unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the [uDMAChannelModeGet\(\)](#) returns **UDMA_MODE_STOP**. For PINGPONG or one of the SCATTER_GATHER modes, it is safe to modify the primary or alternate control structure only when the other is being used. The [uDMAChannelModeGet\(\)](#) function returns **UDMA_MODE_STOP** when a channel control structure is inactive and safe to modify.

Returns:

None.

32.2.3.16 uDMAControlAlternateBaseGet

Gets the base address for the channel control table alternate structures.

Prototype:

```
void *  
uDMAControlAlternateBaseGet (void)
```

Description:

This function gets the base address of the second half of the channel control table that holds the alternate control structures for each channel.

Returns:

Returns a pointer to the base address of the second half of the channel control table.

32.2.3.17 uDMAControlBaseGet

Gets the base address for the channel control table.

Prototype:

```
void *  
uDMAControlBaseGet (void)
```

Description:

This function gets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel.

Returns:

Returns a pointer to the base address of the channel control table.

32.2.3.18 uDMAControlBaseSet

Sets the base address for the channel control table.

Prototype:

```
void  
uDMAControlBaseSet (void *psControlTable)
```

Parameters:

psControlTable is a pointer to the 1024-byte-aligned base address of the uDMA channel control table.

Description:

This function configures the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel. The table must be aligned on a 1024-byte boundary. The base address must be configured before any of the channel functions can be used.

The size of the channel control table depends on the number of uDMA channels and the transfer modes that are used. Refer to the introductory text and the microcontroller datasheet for more information about the channel control table.

Returns:

None.

32.2.3.19 uDMADisable

Disables the uDMA controller for use.

Prototype:

```
void  
uDMADisable (void)
```

Description:

This function disables the uDMA controller. Once disabled, the uDMA controller cannot operate until re-enabled with [uDMAEnable\(\)](#).

Returns:

None.

32.2.3.20 uDMAEnable

Enables the uDMA controller for use.

Prototype:

```
void  
uDMAEnable (void)
```

Description:

This function enables the uDMA controller. The uDMA controller must be enabled before it can be configured and used.

Returns:

None.

32.2.3.21 uDMAErrorStatusClear

Clears the uDMA error interrupt.

Prototype:

```
void  
uDMAErrorStatusClear (void)
```

Description:

This function clears a pending uDMA error interrupt. This function should be called from within the uDMA error interrupt handler to clear the interrupt.

Returns:

None.

32.2.3.22 uDMAErrorStatusGet

Gets the uDMA error status.

Prototype:

```
uint32_t  
uDMAErrorStatusGet (void)
```

Description:

This function returns the uDMA error status. It should be called from within the uDMA error interrupt handler to determine if a uDMA error occurred.

Returns:

Returns non-zero if a uDMA error is pending.

32.2.3.23 uDMAIntClear

Clears uDMA interrupt status.

Prototype:

```
void  
uDMAIntClear(uint32_t ui32ChanMask)
```

Parameters:

ui32ChanMask is a 32-bit mask with one bit for each uDMA channel.

Description:

This function clears bits in the uDMA interrupt status register according to which bits are set in ***ui32ChanMask***. There is one bit for each channel. If a bit is set in ***ui32ChanMask***, then that corresponding channel's interrupt status is cleared (if it was set).

Note:

This function is only available on devices that have the DMA Channel Interrupt Status Register (DMACHIS). Please consult the data sheet for your part. Devices without the DMACHIS register have uDMA done status in the interrupt registers in the peripheral memory maps.

Returns:

None.

32.2.3.24 uDMAIntRegister

Registers an interrupt handler for the uDMA controller.

Prototype:

```
void
uDMAIntRegister(uint32_t ui32IntChannel,
                 void (*pfnHandler)(void))
```

Parameters:

ui32IntChannel identifies which uDMA interrupt is to be registered.

pfnHandler is a pointer to the function to be called when the interrupt is activated.

Description:

This function registers and enables the handler to be called when the uDMA controller generates an interrupt. The ***ui32IntChannel*** parameter should be one of the following:

- **UDMA_INT_SW** to register an interrupt handler to process interrupts from the uDMA software channel (UDMA_CHANNEL_SW)
- **UDMA_INT_ERR** to register an interrupt handler to process uDMA error interrupts

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Note:

The interrupt handler for the uDMA is for transfer completion when the channel UDMA_CHANNEL_SW is used and for error interrupts. The interrupts for each peripheral channel are handled through the individual peripheral interrupt handlers.

Returns:

None.

32.2.3.25 uDMAIntStatus

Gets the uDMA controller channel interrupt status.

Prototype:

```
uint32_t  
uDMAIntStatus(void)
```

Description:

This function is used to get the interrupt status of the uDMA controller. The returned value is a 32-bit bit mask that indicates which channels are requesting an interrupt. This function can be used from within an interrupt handler to determine or confirm which uDMA channel has requested an interrupt.

Note:

This function is only available on devices that have the DMA Channel Interrupt Status Register (DMACHIS). Please consult the data sheet for your part.

Returns:

Returns a 32-bit mask which indicates requesting uDMA channels. There is a bit for each channel and a 1 indicates that the channel is requesting an interrupt. Multiple bits can be set.

32.2.3.26 uDMAIntUnregister

Unregisters an interrupt handler for the uDMA controller.

Prototype:

```
void  
uDMAIntUnregister(uint32_t ui32IntChannel)
```

Parameters:

ui32IntChannel identifies which uDMA interrupt to unregister.

Description:

This function disables and unregisters the handler to be called for the specified uDMA interrupt. The *ui32IntChannel* parameter should be one of **UDMA_INT_SW** or **UDMA_INT_ERR** as documented for the function [uDMAIntRegister\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

32.3 Programming Example

The following example sets up the uDMA controller to perform a software initiated memory-to-memory transfer:

```
//  
// The application must allocate the channel control table. This one is a  
// full table for all modes and channels.  
// NOTE: This table must be 1024-byte aligned.  
//  
uint8_t pui8DMAControlTable[1024];
```

```
//  
// Source and destination buffers used for the DMA transfer.  
//  
uint8_t pui8SourceBuffer[256];  
uint8_t pui8DestBuffer[256];  
  
//  
// Enable the uDMA controller.  
//  
uDMAEnable();  
  
//  
// Set the base for the channel control table.  
//  
uDMAControlBaseSet(&pui8DMAControlTable[0]);  
  
//  
// No attributes must be set for a software-based transfer. The attributes  
// are cleared by default, but are explicitly cleared here, in case they  
// were set elsewhere.  
//  
uDMAChannelAttributeDisable(UDMA_CHANNEL_SW, UDMA_CONFIG_ALL);  
  
//  
// Now set up the characteristics of the transfer for 8-bit data size, with  
// source and destination increments in bytes, and a byte-wise buffer copy.  
// A bus arbitration size of 8 is used.  
//  
uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,  
                      UDMA_SIZE_8 | UDMA_SRC_INC_8 |  
                      UDMA_DST_INC_8 | UDMA_ARB_8);  
  
//  
// The transfer buffers and transfer size are now configured. The transfer  
// uses AUTO mode, which means that the transfer automatically runs to  
// completion after the first request.  
//  
uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,  
                      UDMA_MODE_AUTO, pui8SourceBuffer, pui8DestBuffer,  
                      sizeof(pui8DestBuffer));  
  
//  
// Finally, the channel must be enabled. Because this is a software-  
// initiated transfer, a request must also be made. The request starts the  
// transfer.  
//  
uDMAChannelEnable(UDMA_CHANNEL_SW);  
uDMAChannelRequest(UDMA_CHANNEL_SW);
```


33 USB Controller

Introduction	609
API Functions	609
Using uDMA with USB	652
Using integrated USB DMA	656
USB LPM Features	671
USB ULPI Features	684
Programming Example	688

33.1 Introduction

The USB APIs provide a set of functions that are used to access the Tiva USB device, host and/or device, or OTG controllers. The APIs are split into groups according to the functionality provided by the USB controller present in the microcontroller. The groups are the following: USBDev, USBHost, USBOTG, USBDMA, USBEndpoint, and USBFIFO. The APIs in the USBDev group are used when the USB controller is operating as a Device. The APIs in the USBHost group are used when the USB controller is operating as a Host. The USBOTG APIs are used when configuring a USB controller that supports OTG mode. With USB OTG controllers, once the mode of the USB controller is configured, the device or host APIs are used. The remainder of the APIs are used for both USB host and USB device controllers. The USBEndpoint APIs are used to configure and access the endpoints, the USBDMA APIs are used to configure and operate the DMA controller within the USB module, and the USBFIFO APIs are used to configure the size and location of the FIFOs.

33.2 General USB API Functions

Functions

- void [USBClockDisable](#) (uint32_t ui32Base)
- void [USBClockEnable](#) (uint32_t ui32Base, uint32_t ui32Div, uint32_t ui32Flags)
- uint32_t [USBControllerVersion](#) (uint32_t ui32Base)
- uint32_t [USBDevAddrGet](#) (uint32_t ui32Base)
- void [USBDevAddrSet](#) (uint32_t ui32Base, uint32_t ui32Address)
- void [USBDevConnect](#) (uint32_t ui32Base)
- void [USBDevDisconnect](#) (uint32_t ui32Base)
- void [USBDevEndpointConfigGet](#) (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t *pu32MaxPacketSize, uint32_t *pu32Flags)
- void [USBDevEndpointConfigSet](#) (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPacketSize, uint32_t ui32Flags)
- void [USBDevEndpointDataAck](#) (uint32_t ui32Base, uint32_t ui32Endpoint, bool bIsLastPacket)
- void [USBDevEndpointStall](#) (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void [USBDevEndpointStallClear](#) (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)

- void **USBDevEndpointStatusClear** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBDevMode** (uint32_t ui32Base)
- uint32_t **USBDevSpeedGet** (uint32_t ui32Base)
- uint32_t **USBEndpointDataAvail** (uint32_t ui32Base, uint32_t ui32Endpoint)
- int32_t **USBEndpointDataGet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t *pui8Data, uint32_t *pu32Size)
- int32_t **USBEndpointDataPut** (uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t *pui8Data, uint32_t ui32Size)
- int32_t **USBEndpointDataSend** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32TransType)
- void **USBEndpointDataToggleClear** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBEndpointDMAChannel** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Channel)
- void **USBEndpointDMAConfigSet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Config)
- void **USBEndpointDMADisable** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBEndpointDMAEnable** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBEndpointPacketCountSet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Count)
- uint32_t **USBEndpointStatus** (uint32_t ui32Base, uint32_t ui32Endpoint)
- uint32_t **USBFIFOAddrGet** (uint32_t ui32Base, uint32_t ui32Endpoint)
- void **USBFIFOConfigGet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t *pui32FIFOAddress, uint32_t *pui32FIFOSize, uint32_t ui32Flags)
- void **USBFIFOConfigSet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32FIFOAddress, uint32_t ui32FIFOSize, uint32_t ui32Flags)
- void **USBFIFOFlush** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- uint32_t **USBFrameNumberGet** (uint32_t ui32Base)
- void **USBHighSpeed** (uint32_t ui32Base, bool bEnable)
- uint32_t **USBHostAddrGet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBHostAddrSet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags)
- void **USBHostEndpointConfig** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPayload, uint32_t ui32NAKPollInterval, uint32_t ui32TargetEndpoint, uint32_t ui32Flags)
- void **USBHostEndpointDataAck** (uint32_t ui32Base, uint32_t ui32Endpoint)
- void **USBHostEndpointDataToggle** (uint32_t ui32Base, uint32_t ui32Endpoint, bool bDataToggle, uint32_t ui32Flags)
- void **USBHostEndpointPing** (uint32_t ui32Base, uint32_t ui32Endpoint, bool bEnable)
- void **USBHostEndpointSpeed** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBHostEndpointStatusClear** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- uint32_t **USBHostHubAddrGet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void **USBHostHubAddrSet** (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags)
- void **USBHostMode** (uint32_t ui32Base)
- void **USBHostPwrConfig** (uint32_t ui32Base, uint32_t ui32Flags)

- void **USBHostPwrDisable** (uint32_t ui32Base)
- void **USBHostPwrEnable** (uint32_t ui32Base)
- void **USBHostPwrFaultDisable** (uint32_t ui32Base)
- void **USBHostPwrFaultEnable** (uint32_t ui32Base)
- void **USBHostRequestIN** (uint32_t ui32Base, uint32_t ui32Endpoint)
- void **USBHostRequestINClear** (uint32_t ui32Base, uint32_t ui32Endpoint)
- void **USBHostRequestStatus** (uint32_t ui32Base)
- void **USBHostReset** (uint32_t ui32Base, bool bStart)
- void **USBHostResume** (uint32_t ui32Base, bool bStart)
- uint32_t **USBHostSpeedGet** (uint32_t ui32Base)
- void **USBHostSuspend** (uint32_t ui32Base)
- void **USBIntDisableControl** (uint32_t ui32Base, uint32_t ui32Flags)
- void **USBIntDisableEndpoint** (uint32_t ui32Base, uint32_t ui32Flags)
- void **USBIntEnableControl** (uint32_t ui32Base, uint32_t ui32Flags)
- void **USBIntEnableEndpoint** (uint32_t ui32Base, uint32_t ui32Flags)
- void **USBIntRegister** (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t **USBIntStatusControl** (uint32_t ui32Base)
- uint32_t **USBIntStatusEndpoint** (uint32_t ui32Base)
- void **USBIntUnregister** (uint32_t ui32Base)
- void **USBModeConfig** (uint32_t ui32Base, uint32_t ui32Mode)
- uint32_t **USBModeGet** (uint32_t ui32Base)
- uint32_t **USBNumEndpointsGet** (uint32_t ui32Base)
- void **USBOTGMode** (uint32_t ui32Base)
- void **USBOTGSessionRequest** (uint32_t ui32Base, bool bStart)
- void **USBPHYPowerOff** (uint32_t ui32Base)
- void **USBPHYPowerOn** (uint32_t ui32Base)

33.2.1 Detailed Description

The USB APIs provide all of the functions needed by an application to implement a USB device or USB host stack. The APIs abstract the IN/OUT nature of endpoints based on the type of USB controller that is in use. Any API that uses the IN/OUT terminology complies with the standard USB interpretation of these terms. For example, an OUT endpoint on a microcontroller that has only a device interface actually receives data on this endpoint, while a microcontroller that has a host interface actually transmits data on an OUT endpoint.

Another important fact to understand is that all endpoints in the USB controller, whether host or device, have two "sides" to them, allowing each endpoint to both transmit and receive data. An application can use a single endpoint for both IN and OUT transactions. For example: In device mode, endpoint 1 can be configured to have BULK IN and BULK OUT handled by endpoint 1. It is important to note that the endpoint number used is the endpoint number reported to the host. For microcontrollers with host controllers, the application can use an endpoint to communicate with both IN and OUT endpoints of different types as well. For example: Endpoint 2 can be used to communicate with one device's interrupt IN endpoint and another device's bulk OUT endpoint at the same time. This configuration effectively gives the application one dedicated control endpoint for IN or OUT control transactions on endpoint 0 and seven IN endpoints and seven OUT endpoints.

The USB controller has a global FIFO memory space that can be allocated to endpoints. The overall size of the FIFO RAM is 2048 or 4096 bytes, depending on the Tiva device used. It is important

to note that the first 64 bytes of this memory are dedicated to endpoint 0 for control transactions. The remaining 1984 or 4032 bytes are configurable however the application requires. The FIFO configuration is usually set up at the beginning of the application and not modified once the USB controller is in use. The FIFO configuration uses the USBFIFOConfig() API to configure the starting address and the size of the FIFOs that are dedicated to each endpoint.

Example: FIFO Configuration

0-64 - endpoint 0 IN/OUT (64 bytes).
64-576 - endpoint 1 IN (512 bytes).
576-1088 - endpoint 1 OUT (512 bytes).
1088-1600 - endpoint 2 IN (512 bytes).

```
//  
// FIFO for endpoint 1 IN starts at address 64 and is 512 bytes in size.  
//  
USBFIFOConfig(USB0_BASE,  USB_EP_1,  64,  USB_FIFO_SZ_512,  USB_EP_DEV_IN);  
  
//  
// FIFO for endpoint 1 OUT starts at address 576 and is 512 bytes in size.  
//  
USBFIFOConfig(USB0_BASE,  USB_EP_1,  576,  USB_FIFO_SZ_512,  USB_EP_DEV_OUT);  
  
//  
// FIFO for endpoint 2 IN starts at address 1088 and is 512 bytes in size.  
//  
USBFIFOConfig(USB0_BASE,  USB_EP_2,  1088,  USB_FIFO_SZ_512,  USB_EP_DEV_IN);
```

33.2.2 Function Documentation

33.2.2.1 USBClockDisable

Disables the clocking of the USB controller's PHY.

Prototype:

```
void  
USBClockDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables the USB PHY clock. This function should not be called in applications where the USB controller is used.

Example: Disable the USB PHY clock input.

```
//  
// Disable clocking of the USB controller's PHY.  
//  
USBClockDisable(USB0_BASE);
```

Note:

The ability to configure the USB PHY clock is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.2.2.2 USBClockEnable

Configures and enables the clocking to the USB controller's PHY.

Prototype:

```
void
USBClockEnable(uint32_t ui32Base,
               uint32_t ui32Div,
               uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Div specifies the divider for the internal USB PHY clock.

ui32Flags configures the internal USB PHY clock and specifies the clock source for a ULPI-connected PHY.

Description:

This function configures and enables the USB PHY clock. In addition, for systems that use a ULPI-connected external PHY, this function configures the source for the PHY clock. The *ui32Flags* parameter specifies the clock source with the following values:

- **USB_CLOCK_INTERNAL** uses the internal PLL combined with the *ui32Div* value to generate the USB clock that is used by the internal USB PHY. In addition, when using an external ULPI-connected USB PHY, the specified clock is output on the USB0CLK pin.
- **USB_CLOCK_EXTERNAL** specifies that USB0CLK is an input from the ULPI-connected external PHY.

The *ui32Div* parameter is used to specify a divider for the internal clock if the **USB_CLOCK_INTERNAL** is specified and is ignored if **USB_CLOCK_EXTERNAL** is specified. When the **USB_CLOCK_INTERNAL** is specified, the *ui32Div* value must be set so that the PLL_VCO/*ui32Div* results in a 60-MHz clock.

Example: Enable the USB clock with a 480-MHz PLL setting.

```
//
// Enable the USB clock using a 480-MHz PLL.
// (480-MHz/8 = 60-MHz)
//
USBClockEnable(USB0_BASE, 8, USB_CLOCK_INTERNAL);
```

Note:

The ability to configure the USB PHY clock is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.2.2.3 USBControllerVersion

Returns the version of the USB controller.

Prototype:

```
uint32_t  
USBControllerVersion(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the version number of the USB controller, which can be used to adjust for slight differences between the USB controllers in the Tiva family. The values that are returned are **USB_CONTROLLER_VER_0** and **USB_CONTROLLER_VER_1**.

Note:

The most significant difference between **USB_CONTROLLER_VER_0** and **USB_CONTROLLER_VER_1** is that **USB_CONTROLLER_VER_1** supports the USB controller's own bus master DMA controller, while the **USB_CONTROLLER_VER_0** only supports using the uDMA controller with the USB module.

Example: Get the version of the Tiva USB controller.

```
uint32_t ui32Version;  
  
//  
// Retrieve the version of the Tiva USB controller.  
//  
ui32Version = USBControllerVersion(USBO_BASE);
```

Returns:

This function returns one of the **USB_CONTROLLER_VER_** values.

33.2.2.4 USBDevAddrGet

Returns the current device address in device mode.

Prototype:

```
uint32_t  
USBDevAddrGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current device address. This address was set by a call to [USBDevAddrSet\(\)](#).

Note:

This function must only be called in device mode.

Returns:

The current device address.

33.2.2.5 USBDevAddrSet

Sets the address in device mode.

Prototype:

```
void  
USBDevAddrSet (uint32_t ui32Base,  
                uint32_t ui32Address)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Address is the address to use for a device.

Description:

This function configures the device address on the USB bus. This address was likely received via a SET ADDRESS command from the host controller.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.6 USBDevConnect

Connects the USB controller to the bus in device mode.

Prototype:

```
void  
USBDevConnect (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function causes the soft connect feature of the USB controller to be enabled. Call [USB-DevDisconnect\(\)](#) to remove the USB device from the bus.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.7 USBDevDisconnect

Removes the USB controller from the bus in device mode.

Prototype:

```
void  
USBDevDisconnect (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function causes the soft connect feature of the USB controller to remove the device from the USB bus. A call to [USBDevConnect\(\)](#) is needed to reconnect to the bus.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.8 USBDevEndpointConfigGet

Gets the current configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfigGet(uint32_t ui32Base,
                        uint32_t ui32Endpoint,
                        uint32_t *pui32MaxPacketSize,
                        uint32_t *pui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

pui32MaxPacketSize is a pointer which is written with the maximum packet size for this endpoint.

pui32Flags is a pointer which is written with the current endpoint settings. On entry to the function, this pointer must contain either **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** to indicate whether the IN or OUT endpoint is to be queried.

Description:

This function returns the basic configuration for an endpoint in device mode. The values returned in ***pui32MaxPacketSize** and ***pui32Flags** are equivalent to the **ui32MaxPacketSize** and **ui32Flags** previously passed to [USBDevEndpointConfigSet\(\)](#) for this endpoint.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.9 USBDevEndpointConfigSet

Sets the configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfigSet(uint32_t ui32Base,
```

```
    uint32_t ui32Endpoint,
    uint32_t ui32MaxPacketSize,
    uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32MaxPacketSize is the maximum packet size for this endpoint.

ui32Flags are used to configure other endpoint settings.

Description:

This function sets the basic configuration for an endpoint in device mode. Endpoint zero does not have a dynamic configuration, so this function must not be called for endpoint zero. The *ui32Flags* parameter determines some of the configuration while the other parameters provide the rest.

The **USB_EP_MODE**_flags define what the type is for the specified endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The **USB_EP_DMA_MODE**_flags determine the type of DMA access to the endpoint data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" or the "Using the integrated USB DMA Controller" section for more information on DMA configuration depending on the type of DMA that is supported by the USB controller.

When configuring an IN endpoint, the **USB_EP_AUTO_SET** bit can be specified to cause the automatic transmission of data on the USB bus as soon as *ui32MaxPacketSize* bytes of data are written into the FIFO for this endpoint. This option is commonly used with DMA (both on devices with integrated USB DMA as well as those that use uDMA) as no interaction is required to start the transmission of data.

When configuring an OUT endpoint, the **USB_EP_AUTO_REQUEST** bit is specified to trigger the request for more data once the FIFO has been drained enough to receive *ui32MaxPacketSize* more bytes of data. Also for OUT endpoints, the **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this option is not used, this flag must be manually cleared via a call to [USBDevEndpointStatusClear\(\)](#). Both of these settings can be used to remove the need for extra calls when using the controller with DMA.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.10 USBDevEndpointDataAck

Acknowledge that data was read from the specified endpoint's FIFO in device mode.

Prototype:

```
void
USBDevEndpointDataAck(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      bool bIsLastPacket)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

bIsLastPacket indicates if this packet is the last one.

Description:

This function acknowledges that the data was read from the endpoint's FIFO. The *bIsLastPacket* parameter is set to a **true** value if this is the last in a series of data packets on endpoint zero. The *bIsLastPacket* parameter is not used for endpoints other than endpoint zero. This call can be used if processing is required between reading the data and acknowledging that the data has been read.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.11 USBDevEndpointStall

Stalls the specified endpoint in device mode.

Prototype:

```
void
USBDevEndpointStall(uint32_t ui32Base,
                     uint32_t ui32Endpoint,
                     uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies the endpoint to stall.

ui32Flags specifies whether to stall the IN or OUT endpoint.

Description:

This function causes the endpoint number passed in to go into a stall condition. If the *ui32Flags* parameter is **USB_EP_DEV_IN**, then the stall is issued on the IN portion of this endpoint. If the *ui32Flags* parameter is **USB_EP_DEV_OUT**, then the stall is issued on the OUT portion of this endpoint.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.12 USBDevEndpointStallClear

Clears the stall condition on the specified endpoint in device mode.

Prototype:

```
void
USBDevEndpointStallClear(uint32_t ui32Base,
                         uint32_t ui32Endpoint,
                         uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies which endpoint to remove the stall condition.

ui32Flags specifies whether to remove the stall condition from the IN or the OUT portion of this endpoint.

Description:

This function causes the endpoint number passed in to exit the stall condition. If the *ui32Flags* parameter is **USB_EP_DEV_IN**, then the stall is cleared on the IN portion of this endpoint. If the *ui32Flags* parameter is **USB_EP_DEV_OUT**, then the stall is cleared on the OUT portion of this endpoint.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.13 USBDevEndpointStatusClear

Clears the status bits in this endpoint in device mode.

Prototype:

```
void
USBDevEndpointStatusClear(uint32_t ui32Base,
                          uint32_t ui32Endpoint,
                          uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags are the status bits that are cleared.

Description:

This function clears the status of any bits that are passed in the *ui32Flags* parameter. The *ui32Flags* parameter can take the value returned from the [USBEndpointStatus\(\)](#) call.

Note:

This function must only be called in device mode.

Returns:

None.

33.2.2.14 USBDevMode

Change the mode of the USB controller to device.

Prototype:

```
void  
USBDevMode (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function changes the mode of the USB controller to device mode.

Note:

This function must only be called on microcontrollers that support OTG operation.

Returns:

None.

33.2.2.15 USBDevSpeedGet

Returns the current speed of the USB controller in device mode.

Prototype:

```
uint32_t  
USBDevSpeedGet (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the operating speed of the connection to the USB host controller. This function returns either **USB_HIGH_SPEED** or **USB_FULL_SPEED** to indicate the connection speed in device mode.

Example: Get the USB connection speed.

```
//  
// Get the connection speed of the USB controller.  
//  
USBDevSpeedGet (USB0_BASE);
```

Note:

This function must only be called in device mode.

Returns:

Returns either **USB_HIGH_SPEED** or **USB_FULL_SPEED**.

33.2.2.16 USBEndpointDataAvail

Determines the number of bytes of data available in a specified endpoint's FIFO.

Prototype:

```
uint32_t
USBEndpointDataAvail(uint32_t ui32Base,
                      uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:

This function returns the number of bytes of data currently available in the FIFO for the specified receive (OUT) endpoint. It may be used prior to calling [USBEndpointDataGet\(\)](#) to determine the size of buffer required to hold the newly-received packet.

Returns:

This call returns the number of bytes available in a specified endpoint FIFO.

33.2.2.17 USBEndpointDataGet

Retrieves data from the specified endpoint's FIFO.

Prototype:

```
int32_t
USBEndpointDataGet(uint32_t ui32Base,
                   uint32_t ui32Endpoint,
                   uint8_t *pui8Data,
                   uint32_t *pui32Size)
```

Parameters:

ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
pui8Data is a pointer to the data area used to return the data from the FIFO.
pui32Size is initially the size of the buffer passed into this call via the *pui8Data* parameter. It is set to the amount of data returned in the buffer.

Description:

This function returns the data from the FIFO for the specified endpoint. The *pui32Size* parameter indicates the size of the buffer passed in the *pui32Data* parameter. The data in the *pui32Size* parameter is changed to match the amount of data returned in the *pui8Data* parameter. If a zero-byte packet is received, this call does not return an error but instead just returns a zero in the *pui32Size* parameter. The only error case occurs when there is no data packet available.

Returns:

This call returns 0, or -1 if no packet was received.

33.2.2.18 USBEndpointDataPut

Puts data into the specified endpoint's FIFO.

Prototype:

```
int32_t  
USBEndpointDataPut(uint32_t ui32Base,  
                    uint32_t ui32Endpoint,  
                    uint8_t *pui8Data,  
                    uint32_t ui32Size)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

pui8Data is a pointer to the data area used as the source for the data to put into the FIFO.

ui32Size is the amount of data to put into the FIFO.

Description:

This function puts the data from the *pui8Data* parameter into the FIFO for this endpoint. If a packet is already pending for transmission, then this call does not put any of the data into the FIFO and returns -1. Care must be taken to not write more data than can fit into the FIFO allocated by the call to [USBFIFOConfigSet\(\)](#).

Returns:

This call returns 0 on success, or -1 to indicate that the FIFO is in use and cannot be written.

33.2.2.19 USBEndpointDataSend

Starts the transfer of data from an endpoint's FIFO.

Prototype:

```
int32_t  
USBEndpointDataSend(uint32_t ui32Base,  
                     uint32_t ui32Endpoint,  
                     uint32_t ui32TransType)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32TransType is set to indicate what type of data is being sent.

Description:

This function starts the transfer of data from the FIFO for a specified endpoint. This function is called if the **USB_EP_AUTO_SET** bit was not enabled for the endpoint. Setting the *ui32TransType* parameter allows the appropriate signaling on the USB bus for the type of transaction being requested. The *ui32TransType* parameter must be one of the following:

- **USB_TRANS_OUT** for OUT transaction on any endpoint in host mode.
- **USB_TRANS_IN** for IN transaction on any endpoint in device mode.
- **USB_TRANS_IN_LAST** for the last IN transaction on endpoint zero in a sequence of IN transactions.
- **USB_TRANS_SETUP** for setup transactions on endpoint zero.
- **USB_TRANS_STATUS** for status results on endpoint zero.

Returns:

This call returns 0 on success, or -1 if a transmission is already in progress.

33.2.2.20 USBEndpointDataToggleClear

Sets the data toggle on an endpoint to zero.

Prototype:

```
void
USBEndpointDataToggleClear(uint32_t ui32Base,
                           uint32_t ui32Endpoint,
                           uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.
ui32Endpoint specifies the endpoint to reset the data toggle.
ui32Flags specifies whether to access the IN or OUT endpoint.

Description:

This function causes the USB controller to clear the data toggle for an endpoint. This call is not valid for endpoint zero and can be made with host or device controllers.

The *ui32Flags* parameter must be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

33.2.2.21 USBEndpointDMAChannel

Sets the DMA channel to use for a specified endpoint.

Prototype:

```
void
USBEndpointDMAChannel(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.
ui32Endpoint specifies which endpoint's FIFO address to return.
ui32Channel specifies which DMA channel to use for which endpoint.

Description:

This function is used to configure which DMA channel to use with a specified endpoint. Receive DMA channels can only be used with receive endpoints and transmit DMA channels can only be used with transmit endpoints. As a result, the 3 receive and 3 transmit DMA channels can be mapped to any endpoint other than 0. The values that are passed into the *ui32Channel* value are the UDMA_CHANNEL_USBEP* values defined in udma.h.

Note:

This function only has an effect on microcontrollers that have the ability to change the DMA channel for an endpoint. Calling this function on other devices has no effect.

Returns:

None.

33.2.2.22 USBEndpointDMAConfigSet

Configure the DMA settings for an endpoint.

Prototype:

```
void
USBEndpointDMAConfigSet(uint32_t ui32Base,
                        uint32_t ui32Endpoint,
                        uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Config specifies the configuration options for an endpoint.

Description:

This function configures the DMA settings for a specified endpoint without changing other options that may already be configured. In order for the DMA transfer to be enabled, the [USBEndpointDMAEnable\(\)](#) function must be called before starting the DMA transfer. The configuration options are passed in the *ui32Config* parameter and can have the values described below.

One of the following values to specify direction:

- **USB_EP_HOST_OUT** or **USB_EP_DEV_IN** - This setting is used with DMA transfers from memory to the USB controller.
- **USB_EP_HOST_IN** or **USB_EP_DEV_OUT** - This setting is used with DMA transfers from the USB controller to memory.

One of the following values:

- **USB_EP_DMA_MODE_0(default)** - This setting is typically used for transfers that do not span multiple packets or when interrupts are required for each packet.
- **USB_EP_DMA_MODE_1** - This setting is typically used for transfers that span multiple packets and do not require interrupts between packets.

Values only used with **USB_EP_HOST_OUT** or **USB_EP_DEV_IN**:

- **USB_EP_AUTO_SET** - This setting is used to allow transmit DMA transfers to automatically be sent when a full packet is loaded into a FIFO. This is needed with **USB_EP_DMA_MODE_1** to ensure that packets go out when the FIFO becomes full and the DMA has more data to send.

Values only used with **USB_EP_HOST_IN** or **USB_EP_DEV_OUT**:

- **USB_EP_AUTO_CLEAR** - This setting is used to allow receive DMA transfers to automatically be acknowledged as they are received. This is needed with **USB_EP_DMA_MODE_1** to ensure that packets continue to be received and acknowledged when the FIFO is emptied by the DMA transfer.

Values only used with **USB_EP_HOST_IN**:

- **USB_EP_AUTO_REQUEST** - This setting is used to allow receive DMA transfers to automatically request a new IN transaction when the previous transfer has emptied the FIFO. This is typically used in conjunction with **USB_EP_AUTO_CLEAR** so that receive DMA transfers can continue without interrupting the main processor.

Example: Set endpoint 1 receive endpoint to automatically acknowledge request and automatically generate a new IN request in host mode.

```
//  
// Configure endpoint 1 for receiving multiple packets using DMA.  
//  
USBEndpointDMAConfigSet(USB0_BASE, USB_EP_1, USB_EP_HOST_IN |  
                         USB_EP_DMA_MODE_1 |  
                         USB_EP_AUTO_CLEAR |  
                         USB_EP_AUTO_REQUEST);
```

Example: Set endpoint 2 transmit endpoint to automatically send each packet in host mode when spanning multiple packets.

```
//  
// Configure endpoint 1 for transmitting multiple packets using DMA.  
//  
USBEndpointDMAConfigSet(USB0_BASE, USB_EP_2, USB_EP_HOST_OUT |  
                         USB_EP_DMA_MODE_1 |  
                         USB_EP_AUTO_SET);
```

Returns:

None.

33.2.2.23 USBEndpointDMADisable

Disable DMA on a specified endpoint.

Prototype:

```
void  
USBEndpointDMADisable(uint32_t ui32Base,  
                      uint32_t ui32Endpoint,  
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags specifies which direction to disable.

Description:

This function disables DMA on a specified endpoint to allow non-DMA USB transactions to generate interrupts normally. The **ui32Flags** parameter must be **USB_EP_DEV_IN** or **USB_EP_DEV_OUT**; all other bits are ignored.

Returns:

None.

33.2.2.24 USBEndpointDMAEnable

Enable DMA on a specified endpoint.

Prototype:

```
void
USBEndpointDMAEnable(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags specifies which direction and what mode to use when enabling DMA.

Description:

This function enables DMA on a specified endpoint and configures the mode according to the values in the *ui32Flags* parameter. The *ui32Flags* parameter must have **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** set. Once this function is called the only DMA or error interrupts are generated by the USB controller.

Note:

If this function is called when an endpoint is configured in DMA mode 0 the USB controller does not generate an interrupt.

Returns:

None.

33.2.2.25 USBEndpointPacketCountSet

Sets the number of packets to request when transferring multiple bulk packets.

Prototype:

```
void
USBEndpointPacketCountSet(uint32_t ui32Base,
                           uint32_t ui32Endpoint,
                           uint32_t ui32Count)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint index to target for this write.

ui32Count is the number of packets to request.

Description:

This function sets the number of consecutive bulk packets to request when transferring multiple bulk packets with DMA.

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.2.2.26 USBEndpointStatus

Returns the current status of an endpoint.

Prototype:

```
uint32_t
USBEndpointStatus(uint32_t ui32Base,
                  uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

Description:

This function returns the status of a specified endpoint. If any of these status bits must be cleared, then the [USBDevEndpointStatusClear\(\)](#) or the [USBHostEndpointStatusClear\(\)](#) functions must be called.

The following are the status flags for host mode:

- **USB_HOST_IN_PID_ERROR** - PID error on the specified endpoint.
- **USB_HOST_IN_NOT_COMP** - The device failed to respond to an IN request.
- **USB_HOST_IN_STALL** - A stall was received on an IN endpoint.
- **USB_HOST_IN_DATA_ERROR** - There was a CRC or bit-stuff error on an IN endpoint in Isochronous mode.
- **USB_HOST_IN_NAK_TO** - NAKs received on this IN endpoint for more than the specified timeout period.
- **USB_HOST_IN_ERROR** - Failed to communicate with a device using this IN endpoint.
- **USB_HOST_IN_FIFO_FULL** - This IN endpoint's FIFO is full.
- **USB_HOST_IN_PKTRDY** - Data packet ready on this IN endpoint.
- **USB_HOST_OUT_NAK_TO** - NAKs received on this OUT endpoint for more than the specified timeout period.
- **USB_HOST_OUT_NOT_COMP** - The device failed to respond to an OUT request.
- **USB_HOST_OUT_STALL** - A stall was received on this OUT endpoint.
- **USB_HOST_OUT_ERROR** - Failed to communicate with a device using this OUT endpoint.
- **USB_HOST_OUT_FIFO_NE** - This endpoint's OUT FIFO is not empty.
- **USB_HOST_OUT_PKTPEND** - The data transfer on this OUT endpoint has not completed.
- **USB_HOST_EP0_NAK_TO** - NAKs received on endpoint zero for more than the specified timeout period.
- **USB_HOST_EP0_ERROR** - The device failed to respond to a request on endpoint zero.
- **USB_HOST_EP0_IN_STALL** - A stall was received on endpoint zero for an IN transaction.
- **USB_HOST_EP0_IN_PKTRDY** - Data packet ready on endpoint zero for an IN transaction.

The following are the status flags for device mode:

- **USB_DEV_OUT_SENT_STALL** - A stall was sent on this OUT endpoint.
- **USB_DEV_OUT_DATA_ERROR** - There was a CRC or bit-stuff error on an OUT endpoint.
- **USB_DEV_OUT_OVERRUN** - An OUT packet was not loaded due to a full FIFO.
- **USB_DEV_OUT_FIFO_FULL** - The OUT endpoint's FIFO is full.

- **USB_DEV_OUT_PKTRDY** - There is a data packet ready in the OUT endpoint's FIFO.
- **USB_DEV_IN_NOT_COMP** - A larger packet was split up, more data to come.
- **USB_DEV_IN_SENT_STALL** - A stall was sent on this IN endpoint.
- **USB_DEV_IN_UNDERRUN** - Data was requested on the IN endpoint and no data was ready.
- **USB_DEV_IN_FIFO_NE** - The IN endpoint's FIFO is not empty.
- **USB_DEV_IN_PKTPEND** - The data transfer on this IN endpoint has not completed.
- **USB_DEV_EP0_SETUP_END** - A control transaction ended before Data End condition was sent.
- **USB_DEV_EP0_SENT_STALL** - A stall was sent on endpoint zero.
- **USB_DEV_EP0_IN_PKTPEND** - The data transfer on endpoint zero has not completed.
- **USB_DEV_EP0_OUT_PKTRDY** - There is a data packet ready in endpoint zero's OUT FIFO.

Returns:

The current status flags for the endpoint depending on mode.

33.2.2.27 USBFIFOAddrGet

Returns the absolute FIFO address for a specified endpoint.

Prototype:

```
uint32_t  
USBFIFOAddrGet(uint32_t ui32Base,  
                uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies which endpoint's FIFO address to return.

Description:

This function returns the actual physical address of the FIFO. This address is needed when the USB is going to be used with the uDMA controller and the source or destination address must be set to the physical FIFO address for a specified endpoint. This function can also be used to provide the physical address to manually read data from an endpoints FIFO.

Returns:

None.

33.2.2.28 USBFIFOConfigGet

Returns the FIFO configuration for an endpoint.

Prototype:

```
void  
USBFIFOConfigGet(uint32_t ui32Base,  
                  uint32_t ui32Endpoint,  
                  uint32_t *pui32FIFOAddress,  
                  uint32_t *pui32FIFOSize,  
                  uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

pui32FIFOAddress is the starting address for the FIFO.

pui32FIFOSize is the size of the FIFO as specified by one of the **USB_FIFO_SZ_** values.

ui32Flags specifies what information to retrieve from the FIFO configuration.

Description:

This function returns the starting address and size of the FIFO for a specified endpoint. Endpoint zero does not have a dynamically configurable FIFO, so this function must not be called for endpoint zero. The *ui32Flags* parameter specifies whether the endpoint's OUT or IN FIFO must be read. If in host mode, the *ui32Flags* parameter must be **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode, the *ui32Flags* parameter must be either **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

33.2.2.29 USBFIFOConfigSet

Sets the FIFO configuration for an endpoint.

Prototype:

```
void
USBFIFOConfigSet(uint32_t ui32Base,
                  uint32_t ui32Endpoint,
                  uint32_t ui32FIFOAddress,
                  uint32_t ui32FIFOSize,
                  uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32FIFOAddress is the starting address for the FIFO.

ui32FIFOSize is the size of the FIFO specified by one of the **USB_FIFO_SZ_** values.

ui32Flags specifies what information to set in the FIFO configuration.

Description:

This function configures the starting FIFO RAM address and size of the FIFO for a specified endpoint. Endpoint zero does not have a dynamically configurable FIFO, so this function must not be called for endpoint zero. The *ui32FIFOSize* parameter must be one of the values in the **USB_FIFO_SZ_** values.

The *ui32FIFOAddress* value must be a multiple of 8 bytes and directly indicates the starting address in the USB controller's FIFO RAM. For example, a value of 64 indicates that the FIFO starts 64 bytes into the USB controller's FIFO memory. The *ui32Flags* value specifies whether the endpoint's OUT or IN FIFO must be configured. If in host mode, use **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode, use **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

33.2.2.30 USBFIFOFlush

Forces a flush of an endpoint's FIFO.

Prototype:

```
void  
USBFIFOFlush(uint32_t ui32Base,  
              uint32_t ui32Endpoint,  
              uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags specifies if the IN or OUT endpoint is accessed.

Description:

This function forces the USB controller to flush out the data in the FIFO. The function can be called with either host or device controllers and requires the **ui32Flags** parameter be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

33.2.2.31 USBFrameNumberGet

Gets the current frame number.

Prototype:

```
uint32_t  
USBFrameNumberGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the last frame number received.

Returns:

The last frame number received.

33.2.2.32 USBHighSpeed

Enables or disables USB high-speed negotiation.

Prototype:

```
void  
USBHighSpeed(uint32_t ui32Base,  
             bool bEnable)
```

Parameters:

ui32Base specifies the USB module base address.

bEnable specifies whether to enable or disable high-speed negotiation.

Description:

High-speed negotiations for both host and device mode are enabled when this function is called with the **bEnable** parameter set to **true**. In device mode this causes the device to negotiate for high speed when the USB controller receives a reset from the host. In host mode, the USB host enables high-speed negotiations when resetting the connected device. If **bEnable** is set to **false** the controller only operates only in full-speed or low-speed.

Example: Enable USB high-speed mode.

```
//  
// Enable USB high-speed mode.  
//  
USBHighSpeed(USBO_BASE, true);
```

Note:

This feature is not available on all Tiva devices and should only be called when the USB is connected to an external ULPI PHY. Please check the data sheet to determine if the USB controller can interface with a ULPI PHY.

Returns:

None.

33.2.2.33 USBHostAddrGet

Gets the current functional device address for an endpoint.

Prototype:

```
uint32_t  
USBHostAddrGet(uint32_t ui32Base,  
               uint32_t ui32Endpoint,  
               uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags determines if this is an IN or an OUT endpoint.

Description:

This function returns the current functional address that an endpoint is using to communicate with a device. The **ui32Flags** parameter determines if the IN or OUT endpoint's device address is returned.

Note:

This function must only be called in host mode.

Returns:

Returns the current function address being used by an endpoint.

33.2.2.34 USBHostAddrSet

Sets the functional address for the device that is connected to an endpoint in host mode.

Prototype:

```
void
USBHostAddrSet(uint32_t ui32Base,
                uint32_t ui32Endpoint,
                uint32_t ui32Addr,
                uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Addr is the functional address for the controller to use for this endpoint.

ui32Flags determines if this is an IN or an OUT endpoint.

Description:

This function configures the functional address for a device that is using this endpoint for communication. This **ui32Addr** parameter is the address of the target device that this endpoint is communicating with. The **ui32Flags** parameter indicates if the IN or OUT endpoint is set.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.35 USBHostEndpointConfig

Sets the base configuration for a host endpoint.

Prototype:

```
void
USBHostEndpointConfig(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32MaxPayload,
                      uint32_t ui32NAKPollInterval,
                      uint32_t ui32TargetEndpoint,
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32MaxPayload is the maximum payload for this endpoint.

ui32NAKPollInterval is either the NAK timeout limit or the polling interval, depending on the type of endpoint.

ui32TargetEndpoint is the endpoint that the host endpoint is targeting.

ui32Flags are used to configure other endpoint settings.

Description:

This function sets the basic configuration for the transmit or receive portion of an endpoint in host mode. The *ui32Flags* parameter determines some of the configuration while the other parameters provide the rest. The *ui32Flags* parameter determines whether this is an IN endpoint (**USB_EP_HOST_IN** or **USB_EP_DEV_IN**) or an OUT endpoint (**USB_EP_HOST_OUT** or **USB_EP_DEV_OUT**), whether this is a Full speed endpoint (**USB_EP_SPEED_FULL**) or a Low speed endpoint (**USB_EP_SPEED_LOW**).

The **USB_EP_MODE**_flags control the type of the endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The *ui32NAKPollInterval* parameter has different meanings based on the **USB_EP_MODE** value and whether or not this call is being made for endpoint zero or another endpoint. For endpoint zero or any Bulk endpoints, this value always indicates the number of frames to allow a device to NAK before considering it a timeout. If this endpoint is an isochronous or interrupt endpoint, this value is the polling interval for this endpoint.

For interrupt endpoints, the polling interval is the number of frames between interrupt IN requests to an endpoint and has a range of 1 to 255. For isochronous endpoints this value represents a polling interval of $2^{(ui32NAKPollInterval - 1)}$ frames. When used as a NAK timeout, the *ui32NAKPollInterval* value specifies $2^{(ui32NAKPollInterval - 1)}$ frames before issuing a time out.

There are two special time out values that can be specified when setting the *ui32NAKPollInterval* value. The first is **MAX_NAK_LIMIT**, which is the maximum value that can be passed in this variable. The other is **DISABLE_NAK_LIMIT**, which indicates that there is no limit on the number of NAKs.

The **USB_EP_DMA_MODE**_flags determine the type of DMA access to the endpoint data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" or the "Using the integrated USB DMA Controller" section for more information on DMA configuration depending on the type of DMA that is supported by the USB controller.

When configuring the OUT portion of an endpoint, the **USB_EP_AUTO_SET** bit is specified to cause the transmission of data on the USB bus to start as soon as the number of bytes specified by *ui32MaxPayload* has been written into the OUT FIFO for this endpoint.

When configuring the IN portion of an endpoint, the **USB_EP_AUTO_REQUEST** bit can be specified to trigger the request for more data once the FIFO has been drained enough to fit *ui32MaxPayload* bytes. The **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this option is not used, this flag must be manually cleared via a call to [USBDevEndpointStatusClear\(\)](#) or [USBHostEndpointStatusClear\(\)](#).

For interrupt endpoints in low or full speed mode, the polling interval (*ui32NAKPollInterval*) is the number of frames between interrupt IN requests to an endpoint and has a range of 1 to 255. For interrupt endpoints in high speed mode the polling interval is $2^{(ui32NAKPollInterval - 1)}$ microframes between interrupt IN requests to an endpoint and has a range of 1 to 16.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.36 USBHostEndpointDataAck

Acknowledge that data was read from the specified endpoint's FIFO in host mode.

Prototype:

```
void  
USBHostEndpointDataAck(uint32_t ui32Base,  
                      uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

Description:

This function acknowledges that the data was read from the endpoint's FIFO. This call is used if processing is required between reading the data and acknowledging that the data has been read.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.37 USBHostEndpointDataToggle

Sets the value data toggle on an endpoint in host mode.

Prototype:

```
void  
USBHostEndpointDataToggle(uint32_t ui32Base,  
                         uint32_t ui32Endpoint,  
                         bool bDataToggle,  
                         uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies the endpoint to reset the data toggle.

bDataToggle specifies whether to set the state to DATA0 or DATA1.

ui32Flags specifies whether to set the IN or OUT endpoint.

Description:

This function is used to force the state of the data toggle in host mode. If the value passed in the **bDataToggle** parameter is **false**, then the data toggle is set to the DATA0 state, and if it is **true** it is set to the DATA1 state. The **ui32Flags** parameter can be **USB_EP_HOST_IN** or **USB_EP_HOST_OUT** to access the desired portion of this endpoint. The **ui32Flags** parameter is ignored for endpoint zero.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.38 USBHostEndpointPing

Enables or disables ping tokens for an endpoint using high-speed control transfers in host mode.

Prototype:

```
void
USBHostEndpointPing(uint32_t ui32Base,
                    uint32_t ui32Endpoint,
                    bool bEnable)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies the endpoint to enable/disable ping tokens.

bEnable specifies whether enable or disable ping tokens.

Description:

This function configures the USB controller to either send or not send ping tokens during the data and status phase of high speed control transfers. The only supported value for **ui32Endpoint** is **USB_EP_0** because all control transfers are handled using this endpoint. If the **bEnable** is **true** then ping tokens are enabled, if **false** then ping tokens are disabled. This must be used if the controller must support communications with devices that do not support ping tokens in high speed mode.

Example: Disable ping transactions in host mode on endpoint 0.

```
//
// Disable ping transaction on endpoint 0.
//
USBHostEndpointPing(USB0_BASE,  USB_EP_0,  false);
```

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.39 USBHostEndpointSpeed

Changes the speed of the connection for a host endpoint.

Prototype:

```
void
USBHostEndpointSpeed(uint32_t ui32Base,
                     uint32_t ui32Endpoint,
                     uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags are used to configure other endpoint settings.

Description:

This function sets the USB speed for an IN or OUT endpoint in host mode. The *ui32Flags* parameter specifies the speed using one of the following values: **USB_EP_SPEED_LOW**, **USB_EP_SPEED_FULL**, or **USB_EP_SPEED_HIGH**. The *ui32Flags* parameter also specifies which direction is set by adding the logical OR in either **USB_EP_HOST_IN** or **USB_EP_HOST_OUT**. All other flags are ignored. This function is typically only used for endpoint 0, but could be used with other endpoints as well.

Example: Set host transactions on endpoint 0 to full speed..

```
//  
// Set host endpoint 0 transactions to full speed.  
//  
USBHostEndpointSpeed(USB0_BASE, USB_EP_0, USB_EP_SPEED_FULL);
```

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.40 USBHostEndpointStatusClear

Clears the status bits in this endpoint in host mode.

Prototype:

```
void  
USBHostEndpointStatusClear(uint32_t ui32Base,  
                           uint32_t ui32Endpoint,  
                           uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags are the status bits that are cleared.

Description:

This function clears the status of any bits that are passed in the *ui32Flags* parameter. The *ui32Flags* parameter can take the value returned from the [USBEndpointStatus\(\)](#) call.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.41 USBHostHubAddrGet

Gets the current device hub address for this endpoint.

Prototype:

```
uint32_t
USBHostHubAddrGet (uint32_t ui32Base,
                    uint32_t ui32Endpoint,
                    uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags determines if this is an IN or an OUT endpoint.

Description:

This function returns the current hub address that an endpoint is using to communicate with a device. The *ui32Flags* parameter determines if the device address for the IN or OUT endpoint is returned.

Note:

This function must only be called in host mode.

Returns:

This function returns the current hub address being used by an endpoint.

33.2.2.42 USBHostHubAddrSet

Sets the hub address for the device that is connected to an endpoint.

Prototype:

```
void
USBHostHubAddrSet (uint32_t ui32Base,
                    uint32_t ui32Endpoint,
                    uint32_t ui32Addr,
                    uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Addr is the hub address and port for the device using this endpoint. The hub address must be defined in bits 0 through 6 with the port number in bits 8 through 14.

ui32Flags determines if this is an IN or an OUT endpoint.

Description:

This function configures the hub address for a device that is using this endpoint for communication. The *ui32Flags* parameter determines if the device address for the IN or the OUT endpoint is configured by this call and sets the speed of the downstream device. Valid values are one of **USB_EP_HOST_OUT** or **USB_EP_HOST_IN** optionally ORed with **USB_EP_SPEED_LOW**.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.43 USBHostMode

Change the mode of the USB controller to host.

Prototype:

```
void  
USBHostMode (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function changes the mode of the USB controller to host mode.

Note:

This function must only be called on microcontrollers that support OTG operation.

Returns:

None.

33.2.2.44 USBHostPwrConfig

Sets the configuration for USB power fault.

Prototype:

```
void  
USBHostPwrConfig (uint32_t ui32Base,  
                  uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies the configuration of the power fault.

Description:

This function controls how the USB controller uses its external power control pins (USBnPFLT and USBnEPEN). The flags specify the power fault level sensitivity, the power fault action, and the power enable level and source.

One of the following can be selected as the power fault level sensitivity:

- **USB_HOST_PWRFLT_LOW** - An external power fault is indicated by the pin being driven low.
- **USB_HOST_PWRFLT_HIGH** - An external power fault is indicated by the pin being driven high.

One of the following can be selected as the power fault action:

- **USB_HOST_PWRFLT_EP_NONE** - No automatic action when power fault detected.

- **USB_HOST_PWRFLT_EP_TRI** - Automatically tri-state the USBnEPEN pin on a power fault.
- **USB_HOST_PWRFLT_EP_LOW** - Automatically drive USBnEPEN pin low on a power fault.
- **USB_HOST_PWRFLT_EP_HIGH** - Automatically drive USBnEPEN pin high on a power fault.

One of the following can be selected as the power enable level and source:

- **USB_HOST_PWREN_MAN_LOW** - USBnEPEN is driven low by the USB controller when [USBHostPwrEnable\(\)](#) is called.
- **USB_HOST_PWREN_MAN_HIGH** - USBnEPEN is driven high by the USB controller when [USBHostPwrEnable\(\)](#) is called.
- **USB_HOST_PWREN_AUTOLOW** - USBnEPEN is driven low by the USB controller automatically if [USBOTGSessionRequest\(\)](#) has enabled a session.
- **USB_HOST_PWREN_AUTOHIGH** - USBnEPEN is driven high by the USB controller automatically if [USBOTGSessionRequest\(\)](#) has enabled a session.

When using the VBUS glitch filter, the **USB_HOST_PWREN_FILTER** can be added to ignore small, short drops in VBUS level caused by high power consumption. This feature is mainly used to avoid causing VBUS errors caused by devices with high in-rush current.

Note:

This function must only be called on microcontrollers that support host mode or OTG operation. The **USB_HOST_PWREN_AUTOLOW** and **USB_HOST_PWREN_AUTOHIGH** parameters can only be specified on devices that support OTG operation.

Returns:

None.

33.2.2.45 USBHostPwrDisable

Disables the external power pin.

Prototype:

```
void
USBHostPwrDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables the USBnEPEN signal, which disables an external power supply in host mode operation.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.46 USBHostPwrEnable

Enables the external power pin.

Prototype:

```
void  
USBHostPwrEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function enables the USBnEPEN signal, which enables an external power supply in host mode operation.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.47 USBHostPwrFaultDisable

Disables power fault detection.

Prototype:

```
void  
USBHostPwrFaultDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables power fault detection in the USB controller.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.48 USBHostPwrFaultEnable

Enables power fault detection.

Prototype:

```
void  
USBHostPwrFaultEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function enables power fault detection in the USB controller. If the USBnPFLT pin is not in use, this function must not be used.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.49 USBHostRequestIN

Schedules a request for an IN transaction on an endpoint in host mode.

Prototype:

```
void
USBHostRequestIN(uint32_t ui32Base,
                  uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

Description:

This function schedules a request for an IN transaction. When the USB device being communicated with responds with the data, the data can be retrieved by calling [USBEndpointDataGet\(\)](#) or via a DMA transfer.

Note:

This function must only be called in host mode and only for IN endpoints.

Returns:

None.

33.2.2.50 USBHostRequestINClear

Clears a scheduled IN transaction for an endpoint in host mode.

Prototype:

```
void
USBHostRequestINClear(uint32_t ui32Base,
                      uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

Description:

This function clears a previously scheduled IN transaction if it is still pending. This function is used to safely disable any scheduled IN transactions if the endpoint specified by **ui32Endpoint** is reconfigured for communications with other devices.

Note:

This function must only be called in host mode and only for IN endpoints.

Returns:

None.

33.2.2.51 USBHostRequestStatus

Issues a request for a status IN transaction on endpoint zero.

Prototype:

```
void  
USBHostRequestStatus(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function is used to cause a request for a status IN transaction from a device on endpoint zero. This function can only be used with endpoint zero as that is the only control endpoint that supports this ability. This function is used to complete the last phase of a control transaction to a device and an interrupt is signaled when the status packet has been received.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.52 USBHostReset

Handles the USB bus reset condition.

Prototype:

```
void  
USBHostReset(uint32_t ui32Base,  
             bool bStart)
```

Parameters:

ui32Base specifies the USB module base address.

bStart specifies whether to start or stop signaling reset on the USB bus.

Description:

When this function is called with the *bStart* parameter set to **true**, this function causes the start of a reset condition on the USB bus. The caller must then delay at least 20ms before calling this function again with the *bStart* parameter set to **false**.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.53 USBHostResume

Handles the USB bus resume condition.

Prototype:

```
void
USBHostResume(uint32_t ui32Base,
              bool bStart)
```

Parameters:

ui32Base specifies the USB module base address.

bStart specifies if the USB controller is entering or leaving the resume signaling state.

Description:

When in device mode, this function brings the USB controller out of the suspend state. This call must first be made with the **bStart** parameter set to **true** to start resume signaling. The device application must then delay at least 10ms but not more than 15ms before calling this function with the **bStart** parameter set to **false**.

When in host mode, this function signals devices to leave the suspend state. This call must first be made with the **bStart** parameter set to **true** to start resume signaling. The host application must then delay at least 20ms before calling this function with the **bStart** parameter set to **false**. This action causes the controller to complete the resume signaling on the USB bus.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.54 USBHostSpeedGet

Returns the current speed of the USB device connected.

Prototype:

```
uint32_t
USBHostSpeedGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current speed of the USB bus in host mode.

Example: Get the USB connection speed.

```
//
// Get the connection speed of the device connected to the USB controller.
//
USBHostSpeedGet(USB0_BASE);
```

Note:

This function must only be called in host mode.

Returns:

Returns one of the following: **USB_LOW_SPEED**, **USB_FULL_SPEED**, **USB_HIGH_SPEED**, or **USB_UNDEF_SPEED**.

33.2.2.55 USBHostSuspend

Puts the USB bus in a suspended state.

Prototype:

```
void  
USBHostSuspend(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

When used in host mode, this function puts the USB bus in the suspended state.

Note:

This function must only be called in host mode.

Returns:

None.

33.2.2.56 USBIntDisableControl

Disables control interrupts on a specified USB controller.

Prototype:

```
void  
USBIntDisableControl(uint32_t ui32Base,  
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which control interrupts to disable.

Description:

This function disables the control interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which control interrupts to disable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTCTRL_*** and not any other **USB_INT** flags.

Returns:

None.

33.2.2.57 USBIntDisableEndpoint

Disables endpoint interrupts on a specified USB controller.

Prototype:

```
void
USBIntDisableEndpoint (uint32_t ui32Base,
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which endpoint interrupts to disable.

Description:

This function disables endpoint interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which endpoint interrupts to disable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTEP_*** and not any other **USB_INT** flags.

Returns:

None.

33.2.2.58 USBIntEnableControl

Enables control interrupts on a specified USB controller.

Prototype:

```
void
USBIntEnableControl (uint32_t ui32Base,
                     uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which control interrupts to enable.

Description:

This function enables the control interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which control interrupts to enable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTCTRL_*** and not any other **USB_INT** flags.

Returns:

None.

33.2.2.59 USBIntEnableEndpoint

Enables endpoint interrupts on a specified USB controller.

Prototype:

```
void
USBIntEnableEndpoint (uint32_t ui32Base,
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which endpoint interrupts to enable.

Description:

This function enables endpoint interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which endpoint interrupts to enable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTEP_*** and not any other **USB_INT** flags.

Returns:

None.

33.2.2.60 USBIntRegister

Registers an interrupt handler for the USB controller.

Prototype:

```
void  
USBIntRegister(uint32_t ui32Base,  
               void (*pfnHandler) (void))
```

Parameters:

ui32Base specifies the USB module base address.

pfnHandler is a pointer to the function to be called when a USB interrupt occurs.

Description:

This function registers the handler to be called when a USB interrupt occurs and enables the global USB interrupt in the interrupt controller. The specific desired USB interrupts must be enabled via a separate call to **USBIntEnable()**. It is the interrupt handler's responsibility to clear the interrupt sources via calls to [USBIntStatusControl\(\)](#) and [USBIntStatusEndpoint\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Returns:

None.

33.2.2.61 USBIntStatusControl

Returns the control interrupt status on a specified USB controller.

Prototype:

```
uint32_t  
USBIntStatusControl(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function reads control interrupt status for a USB controller. This call returns the current status for control interrupts only, the endpoint interrupt status is retrieved by calling [USBIntStatusEndpoint\(\)](#). The bit values returned are compared against the **USB_INTCTRL_*** values.

The following are the meanings of all **USB_INCTRL_** flags and the modes for which they are valid. These values apply to any calls to [USBIntStatusControl\(\)](#), [USBIntEnableControl\(\)](#), and [USBIntDisableControl\(\)](#). Some of these flags are only valid in the following modes as indicated in the parentheses: Host, Device, and OTG.

- **USB_INTCTRL_ALL** - A full mask of all control interrupt sources.
- **USB_INTCTRL_VBUS_ERR** - A VBUS error has occurred (Host Only).
- **USB_INTCTRL_SESSION** - Session Start Detected on A-side of cable (OTG Only).
- **USB_INTCTRL_SESSION_END** - Session End Detected (Device Only)
- **USB_INTCTRL_DISCONNECT** - Device Disconnect Detected (Host Only)
- **USB_INTCTRL_CONNECT** - Device Connect Detected (Host Only)
- **USB_INTCTRL_SOF** - Start of Frame Detected.
- **USB_INTCTRL_BABBLE** - USB controller detected a device signaling past the end of a frame (Host Only)
- **USB_INTCTRL_RESET** - Reset signaling detected by device (Device Only)
- **USB_INTCTRL_RESUME** - Resume signaling detected.
- **USB_INTCTRL_SUSPEND** - Suspend signaling detected by device (Device Only)
- **USB_INTCTRL_MODE_DETECT** - OTG cable mode detection has completed (OTG Only)
- **USB_INTCTRL_POWER_FAULT** - Power Fault detected (Host Only)

Note:

This call clears the source of all of the control status interrupts.

Returns:

Returns the status of the control interrupts for a USB controller.

33.2.2.62 USBIntStatusEndpoint

Returns the endpoint interrupt status on a specified USB controller.

Prototype:

```
uint32_t
USBIntStatusEndpoint(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function reads endpoint interrupt status for a USB controller. This call returns the current status for endpoint interrupts only, the control interrupt status is retrieved by calling [USBIntStatusControl\(\)](#). The bit values returned are compared against the **USB_INTEP_*** values. These values are grouped into classes for **USB_INTEP_HOST_*** and **USB_INTEP_DEV_*** values to handle both host and device modes with all endpoints.

Note:

This call clears the source of all of the endpoint interrupts.

Returns:

Returns the status of the endpoint interrupts for a USB controller.

33.2.2.63 USBIntUnregister

Unregisters an interrupt handler for the USB controller.

Prototype:

```
void  
USBIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function unregisters the interrupt handler. This function also disables the USB interrupt in the interrupt controller.

See also:

[IntRegister\(\)](#) for important information about registering or unregistering interrupt handlers.

Returns:

None.

33.2.2.64 USBModeConfig

Change the operating mode of the USB controller.

Prototype:

```
void  
USBModeConfig(uint32_t ui32Base,  
              uint32_t ui32Mode)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Mode specifies the operating mode of the USB OTG pins.

Description:

This function changes the operating modes of the USB controller. When operating in full OTG mode, the USB controller uses the VBUS and ID pins to detect mode and voltage changes. While these pins are primarily used in OTG mode, they can also affect the operation of host and device modes. In device mode, the USB controller can be configured to monitor or ignore VBUS. Monitoring VBUS allows the controller to determine if it has been disconnected from the host. In host mode, the USB controller uses the VBUS pin to detect loss of VBUS caused by excessive power draw due to a drop in the VBUS voltage. This call takes the place of [USBHostMode\(\)](#), [USBDevMode\(\)](#), and [USBOTGMode\(\)](#). The *ui32Mode* value should be one of the following values:

- **USB_MODE_OTG** enables operating in full OTG mode, VBUS and ID are used by the controller.

- **USB_MODE_HOST** enables operating only as a host with no monitoring of VBUS or ID pins.
- **USB_MODE_HOST_VBUS** enables operating only as a host with monitoring of VBUS pin. This configuration enables detection of VBUS droop while still forcing host mode.
- **USB_MODE_DEVICE** enables operating only as a device with no monitoring of VBUS or ID pins.
- **USB_MODE_DEVICE_VBUS** enables operating only as a device with monitoring of VBUS pin. This configuration enables disconnect detection while still forcing device mode.

Note:

Some of the options above are not available on some Tiva devices. Please check the data sheet to determine if the USB controller supports a particular mode.

Example: Force device mode but allow monitoring of the USB VBUS pin.

```
//  
// Force device mode but allow monitoring of VBUS for disconnect.  
//  
USBModeConfig(USB_MODE_DEVICE_VBUS);
```

Returns:

None.

33.2.2.65 USBModeGet

Returns the current operating mode of the controller.

Prototype:

```
uint32_t  
USBModeGet (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current operating mode on USB controllers with OTG or Dual mode functionality.

For OTG controllers:

The function returns one of the following values on OTG controllers:

USB_OTG_MODE_ASIDE_HOST indicates that the controller is in host mode on the A-side of the cable.

USB_OTG_MODE_ASIDE_DEV indicates that the controller is in device mode on the A-side of the cable.

USB_OTG_MODE_BSIDE_HOST indicates that the controller is in host mode on the B-side of the cable.

USB_OTG_MODE_BSIDE_DEV indicates that the controller is in device mode on the B-side of the cable. If an OTG session request is started with no cable in place, this mode is the default.

USB_OTG_MODE_NONE indicates that the controller is not attempting to determine its role in the system.

For Dual Mode controllers:

The function returns one of the following values:

USB_DUAL_MODE_HOST indicates that the controller is acting as a host.

USB_DUAL_MODE_DEVICE indicates that the controller acting as a device.

USB_DUAL_MODE_NONE indicates that the controller is not active as either a host or device.

Returns:

Returns **USB_OTG_MODE_ASIDE_HOST**, **USB_OTG_MODE_ASIDE_DEV**,
USB_OTG_MODE_BSIDE_HOST, **USB_OTG_MODE_BSIDE_DEV**,
USB_OTG_MODE_NONE, **USB_DUAL_MODE_HOST**, **USB_DUAL_MODE_DEVICE**,
or **USB_DUAL_MODE_NONE**.

33.2.2.66 USBNumEndpointsGet

Returns the number of USB endpoint pairs on the device.

Prototype:

```
uint32_t  
USBNumEndpointsGet (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the number of endpoint pairs supported by the USB controller corresponding to the passed base address. The value returned is the number of IN or OUT endpoints available and does not include endpoint 0 (the control endpoint). For example, if 15 is returned, there are 15 IN and 15 OUT endpoints available in addition to endpoint 0.

Returns:

Returns the number of IN or OUT endpoints available.

33.2.2.67 USBOTGMode

Changes the mode of the USB controller to OTG.

Prototype:

```
void  
USBOTGMode (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function changes the mode of the USB controller to OTG mode. This function is only valid on microcontrollers that have the OTG capabilities.

Returns:

None.

33.2.2.68 USBOTGSessionRequest

Starts or ends a session.

Prototype:

```
void  
USBOTGSessionRequest(uint32_t ui32Base,  
                      bool bStart)
```

Parameters:

ui32Base specifies the USB module base address.

bStart specifies if this call starts or ends a session.

Description:

This function is used in OTG mode to start a session request or end a session. If the **bStart** parameter is set to **true**, then this function starts a session and if it is **false** it ends a session.

Returns:

None.

33.2.2.69 USBPHYPowerOff

Powers off the internal USB PHY.

Prototype:

```
void  
USBPHYPowerOff(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function powers off the internal USB PHY, reducing the current consumption of the device. While in the powered-off state, the USB controller is unable to operate.

Returns:

None.

33.2.2.70 USBPHYPowerOn

Powers on the internal USB PHY.

Prototype:

```
void  
USBPHYPowerOn(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function powers on the internal USB PHY, enabling it return to normal operation. By default, the PHY is powered on, so this function must only be called if [USBPHYPowerOff\(\)](#) has previously been called.

Returns:

None.

33.3 Using USB with the uDMA Controller

On devices that do not have an integrated DMA in the USB controller, the uDMA can be used with the USB controller for either sending or receiving data with both host and device controllers. The uDMA controller cannot be used to access endpoint 0, however all other endpoints are capable of using the uDMA controller. The uDMA channel numbers for USB are defined by the following values:

- DMA_CHANNEL_USBEP1RX
- DMA_CHANNEL_USBEP1TX
- DMA_CHANNEL_USBEP2RX
- DMA_CHANNEL_USBEP2TX
- DMA_CHANNEL_USBEP3RX
- DMA_CHANNEL_USBEP3TX

For devices with more than 8 endpoints, the required endpoints must be assigned to one of the 3 DMA receive channels and 3 DMA transmit channels using the [USBEndpointDMAChannel\(\)](#) function.

Because the uDMA controller views transfers as either transmit or receive and the USB controller operates on IN/OUT transactions, some care must be taken to use the correct uDMA channel with the correct endpoint. USB host IN and USB device OUT endpoints both use receive uDMA channels while USB host OUT and USB device IN endpoints use transmit uDMA channels.

When configuring the endpoint, there are additional DMA settings required. When calling [USBDev-vEndpointConfigSet\(\)](#) for an endpoint that uses uDMA, extra flags must be added to the *ulFlags* parameter. These flags are one of **USB_EP_DMA_MODE_0** or **USB_EP_DMA_MODE_1** to control the mode of the DMA transaction, and likely **USB_EP_AUTO_SET** to allow the data to be transmitted automatically once a packet is ready. When using **USB_EP_DMA_MODE_0**, the USB controller only generates an interrupt when the full transfer is complete. As a result, the application must know the full transfer size before configuring the DMA transfer. In **USB_EP_DMA_MODE_1**, the USB controller generates DMA requests only when a full packet is transferred and interrupts the processor on any short packet. The short packet data remains in the USB FIFO, and the application must trigger the last transfer of data from the FIFO. The **USB_EP_AUTO_SET** is specified when using uDMA to prevent the need for application code to start the actual transfer of data on every full packet of data.

Example: Endpoint configuration for a device IN endpoint:

```

//
// Endpoint 1 is a device mode BULK IN endpoint using DMA.
//
USBDevEndpointConfigSet(USB0_BASE,    USB_EP_1,    64,
                        (USB_EP_MODE_BULK | USB_EP_DEV_IN |
                         USB_EP_DMA_MODE_0 | USB_EP_AUTO_SET));

```

Next, the application must configure the uDMA controller for the desired DMA transfer to the FIFO. To clear out any previous settings, the application calls DMAChannelAttributeClear(). Then the application calls DMAChannelAttributeSet() for the uDMA channel that corresponds to the endpoint and specify the **DMA_CONFIG_USEBURST** flag.

Note:

All uDMA transfers used by the USB controller must enable burst mode.

The application also provides the size of each DMA transaction, combined with the source and destination increments and the arbitration level for the uDMA controller.

Example: Configure endpoint 1 transmit channel.

```

//
// Set up the DMA for USB transmit.
//
DMAChannelAttributeClear(DMA_CHANNEL_USBEP1TX, DMA_CONFIG_ALL);

//
// Enable uDMA burst mode.
//
DMAChannelAttributeSet(DMA_CHANNEL_USBEP1TX, DMA_CONFIG_USEBURST);

//
// Data size is 8 bits and the source has a one byte increment.
// Destination has no increment as it is a FIFO.
//
DMAChannelControlSet(DMA_CHANNEL_USBEP1TX, DMA_DATA_SIZE_8, DMA_ADDR_INC_8,
                      DMA_ADDR_INC_NONE, DMA_ARB_64, 0);

```

The next step is to actually start the uDMA transfer once the data is ready to be sent. There are only two calls that the application must make to start a new transfer. For most cases, the previous uDMA configuration remains the same. The call to DMAChannelTransferSet() resets the source and destination addresses for the DMA transfer and specifies how much data to send. The call to DMAChannelEnable() actually allows the DMA controller to begin requesting data to fill the FIFO.

Example: Start the transfer of data on endpoint 1.

```

//
// Configure the address and size of the data to transfer.
//
DMAChannelTransferSet(DMA_CHANNEL_USBEP1TX, DMA_MODE_BASIC, pData,
                      USBFIFOAddr(USB0_BASE, USB_EP_1), 64);
//
// Start the transfer.
//
DMAChannelEnable(DMA_CHANNEL_USBEP1TX);

```

Because the uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, the application must perform an extra check to determine the actual source of the interrupt. It is important to note that the DMA interrupt does not mean that the USB transfer is complete, but only that the data has been transferred to the USB controller's FIFO. There is also an interrupt indicating that

the USB transfer is complete. However, both events must be handled in the same interrupt routine because if other code in the system holds off the USB interrupt routine, both the uDMA complete and transfer complete can occur before the USB interrupt handler is called. The USB has no status bit indicating that the interrupt was due to a DMA complete, which means that the application must remember if a DMA transaction was in progress. The example below shows the g_ulFlags global variable being used to remember that a DMA transfer was pending.

Example: Interrupt handling with uDMA.

```
if((g_ulFlags & EP1_DMA_IN_PEND) &&
   (DMAChannelModeGet(DMA_CHANNEL_USBEP1TX) == DMA_MODE_STOP))
{
    //
    // Handle the DMA complete case.
    //
    ...
}

//
// Get the interrupt status.
//
ulStatus = USBIntStatusEndpoint(USB0_BASE);

if(ulStatus & USB_INTEP_DEV_IN_1)
{
    //
    // Handler the transfer complete case.
    //
    ...
}
```

To use the USB device controller with an OUT endpoint, the application must use a receive uDMA channel. When calling [USBDevEndpointConfigSet\(\)](#) for an endpoint that uses uDMA, the application must set extra flags in the *ulFlags* parameter. The **USB_EP_DMA_MODE_0** and **USB_EP_DMA_MODE_1** parameters control the mode of the transaction, **USB_EP_AUTO_CLEAR** allows the data to be received automatically without manually acknowledging that the data has been read. If the transfer size is not known, **USB_EP_DMA_MODE_1** is used as it does not generate an interrupt when each packet is sent over USB and interrupts if a short packet is received. In **USB_EP_DMA_MODE_1**, the last short packet remains in the FIFO and must be read by software when the interrupt is received. If the full transfer size is known, **USB_EP_DMA_MODE_0** can be used because it does not interrupt the processor after each packet and completes even if the last packet is a short packet. The **USB_EP_AUTO_CLEAR** flag is normally specified when using uDMA to allow the USB controller to transfer multiple packets without interruption of the microcontroller. The example below configures endpoint 1 as a Device mode Bulk OUT endpoint using DMA mode 1 with a max packet size of 64 bytes.

Example: Configure endpoint 1 receive channel:

```
//
// Endpoint 1 is a device mode BULK OUT endpoint using DMA.
//
USBDevEndpointConfigSet(USB0_BASE, USB_EP_1, 64,
                        (USB_EP_DEV_OUT | USB_EP_MODE_BULK |
                         USB_EP_DMA_MODE_1 | USB_EP_AUTO_CLEAR));
```

Next the application is required to configure the uDMA controller to match the desired transfer. Like the transmit case, the first call to **DMAChannelAttributeClear()** is made to clear any

previous settings. This function is followed by a call to DMAChannelAttributeSet() with the **DMA_CONFIG_USEBURST** value.

Note:

All uDMA transfers used by the USB controller must use burst mode.

The final call configures the read access size to 8 bits wide, the source address increment to 0, the destination address increment to 8 bits and the uDMA arbitration size to 64 bytes.

Example: Configure endpoint 1 transmit channel.

```
//  
// Clear out any uDMA settings.  
//  
DMAChannelAttributeClear(DMA_CHANNEL_USBEP1RX, DMA_CONFIG_ALL);  
  
DMAChannelAttributeSet(DMA_CHANNEL_USBEP1RX, DMA_CONFIG_USEBURST);  
  
DMAChannelControlSet(DMA_CHANNEL_USBEP1RX, DMA_DATA_SIZE_8,  
DMA_ADDR_INC_NONE, DMA_ADDR_INC_8, DMA_ARB_64, 0);
```

The next step is to actually start the uDMA transfer. Unlike the transfer side, if the application is ready, the receive side can be set up right away to wait for incoming data. Like the transmit case, these calls are the only ones required to start a new transfer, because normally, the previous uDMA configuration can remain the same.

Example: Start requesting data on endpoint 1.

```
//  
// Configure the address and size of the data to transfer. The transfer  
// is from the USB FIFO for endpoint 0 to g_DataBufferIn.  
//  
DMAChannelTransferSet(DMA_CHANNEL_USBEP1RX, DMA_MODE_BASIC,  
USBFIFOAddr(USB0_BASE, USB_EP_1), g_DataBufferIn,  
64);  
  
//  
// Enable the uDMA channel and wait for data.  
//  
DMAChannelEnable(DMA_CHANNEL_USBEP1RX);
```

The uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, which means that the application must determine what the actual source of the interrupt was. It is possible that the USB interrupt does not indicate that the USB transfer was complete. The interrupt can also be generated by a short packet, error, or even a transmit complete. As a result, the application must check both receive cases to determine if the interrupt is related to receiving data on the endpoint. Because the USB has no status bit indicating that the interrupt was due to a DMA complete, the application must remember if a DMA transaction was in progress.

Example: Interrupt handling with uDMA.

```
//  
// Get the current interrupt status.  
//  
ulStatus = USBIntStatusEndpoint(USB0_BASE);  
  
if(ulStatus & USB_INTEP_DEV_OUT_1)  
{  
//
```

```
    // Handle a short packet.  
    //  
    ...  
}  
else if((g_ulFlags & EP1_DMA_OUT_PEND) &&  
       (DMAChannelModeGet(DMA_CHANNEL_USBEP1RX) == DMA_MODE_STOP)  
{  
    //  
    // Handle the DMA complete case.  
    //  
    ...  
  
    //  
    // Restart receive DMA if desired.  
    //  
    ...  
}
```

33.4 Using the integrated USB DMA Controller

Functions

- void * [USBDMAChannelAddressGet](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelAddressSet](#) (uint32_t ui32Base, uint32_t ui32Channel, void *pvAddress)
- void [USBDMAChannelConfigSet](#) (uint32_t ui32Base, uint32_t ui32Channel, uint32_t ui32Endpoint, uint32_t ui32Config)
- uint32_t [USBDMAChannelCountGet](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelCountSet](#) (uint32_t ui32Base, uint32_t ui32Channel, uint32_t ui32Count)
- void [USBDMAChannelDisable](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelEnable](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelIntDisable](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelIntEnable](#) (uint32_t ui32Base, uint32_t ui32Channel)
- uint32_t [USBDMAChannelIntStatus](#) (uint32_t ui32Base)
- uint32_t [USBDMAChannelStatus](#) (uint32_t ui32Base, uint32_t ui32Channel)
- void [USBDMAChannelStatusClear](#) (uint32_t ui32Base, uint32_t ui32Channel, uint32_t ui32Status)
- uint32_t [USBDMANumChannels](#) (uint32_t ui32Base)

33.4.1 Detailed Description

Some USB controllers include an integrated USB DMA controller for DMA access to the USB FIFOs that is used instead of the uDMA controller. The programming method for using the integrated USB DMA controller differs from that used with the uDMA controller to move USB FIFO data to and from system memory.

The integrated USB DMA controller has a basic set of functions that allows an application to determine whether this feature is available and if so, provide additional details about the USB DMA controller. An application can use the [USBDMANumChannels\(\)](#) function to determine if the USB

controller supports the integrated USB DMA controller by checking that the returned value is non-zero. Like most DMA controllers, the USB DMA controller is used to transfer data between the USB controller and system memory using a simple address and count value. While the transfer size can be specified as any number of bytes, the USB DMA controller can only perform accesses on 32-bit-aligned boundaries, so care must be taken to only specify transfer addresses in modulo-4 increments. The USB DMA controller also has a fixed number of DMA channels that can be dynamically assigned to any endpoint and can be used to either transmit or receive data. The total number of DMA channels available is determined using the [USBDMANumChannels\(\)](#) function. The channel numbers themselves are numbered from 0 to the maximum number of channels minus one. The **UDMA_CHANNEL_USBEPnRX** and **UDMA_CHANNEL_USBEPnTX** defines for the uDMA must not be used as the channel numbers when using the integrated USB DMA controller.

Each DMA channel can be configured with various settings that assign a channel to an endpoint and configure how it is used during a DMA transfer. The DMA channel is configured using the *ulConfig* parameter of the [USBDMACChannelConfigSet\(\)](#) function. To set the proper direction for the transfer, the **USB_DMA_CFG_DIR_TX** or **USB_DMA_CFG_DIR_RX** option must be added to the *ulConfig* parameter. The DMA burst lengths are specified in 32-bit word increments using one of the following values added to the *ulConfig* parameter: **USB_DMA_CFG_BURST_NONE** (default), **USB_DMA_CFG_BURST_4**, **USB_DMA_CFG_BURST_8**, or **USB_DMA_CFG_BURST_16**. The DMA mode is also used to control how the DMA controller handles interrupting the processor. **USB_DMA_CFG_MODE_0** is typically used when only a single packet is being sent or received using DMA as it triggers an interrupt per packet transferred. **USB_DMA_CFG_MODE_1** is used when multiple packets are being sent using DMA and triggers one completion interrupt per transfer when spanning multiple packets rather than triggering an interrupt per packet. In addition, when calling [USBDevEndpointConfigSet\(\)](#) for an endpoint that uses DMA, extra flags must be added to the *ulFlags* parameter. These flags are one of **USB_EP_DMA_MODE_0** or **USB_EP_DMA_MODE_1** to control the mode of the DMA transaction, and likely **USB_EP_AUTO_SET** to allow the data to be transmitted automatically once a packet is ready. When using **USB_EP_DMA_MODE_0**, the USB controller only generates an interrupt when the full transfer is complete. As a result, the application must know the full transfer size before configuring the DMA transfer. In **USB_EP_DMA_MODE_1**, the USB controller generates DMA requests only when a full packet is transferred and interrupts the processor on any short packet. The short packet data remains in the USB FIFO, and the application must trigger the last transfer of data from the FIFO. The **USB_EP_AUTO_SET** is specified when using uDMA to prevent the need for application code to start the actual transfer of data on every full packet of data.

The method for configuring the USB DMA controller to perform a DMA transfer varies based on the USB mode (Host or Device), the size of the transfer, and the direction of the transfer. It is important to understand the interrupt mechanisms when dealing with the USB DMA controller. There are two types of interrupts that occur. The first is the normal endpoint interrupt that indicates a USB packet transfer has completed, and the other is a USB DMA interrupt that indicates a USB DMA transfer has completed. A USB DMA transfer complete does not indicate that the packet transmission has completed, but rather that the transfer to or from the FIFO has completed. Depending on the situation, the application may have to wait for more than one interrupt to signal the completion of the USB transaction. The next sections describe how to use the integrated USB DMA controller to complete USB transactions in the most common scenarios.

33.4.2 Sending a Single Packet

The first type of transfer is a single packet transfer that is less than or equal to the maximum packet size for the endpoint. In this mode, the DMA controller copies data from memory into the USB FIFO for the endpoint. In most cases, it is the responsibility of the application to then manually trigger a

USB transfer when the DMA is complete using the [USBEndpointDataSend\(\)](#) function. However, if the **USB_EP_AUTO_SET** value is included as an option to the [USBDvEndpointConfigSet\(\)](#) or the [USBHostEndpointConfig\(\)](#) function and the transfer is of exactly the maximum packet size, then it is not necessary to manually trigger a USB transfer and doing so may cause an extra null packet to be sent.

If endpoints are not dynamically allocated by the application, some of the DMA and endpoint configuration can be treated as static and executed only once during initial configuration.

Example: Endpoint configuration for USB DMA transmit

```
//  
// Endpoint 1 uses Mode 0 and transmit.  
//  
USBEndpointDMAConfigSet(USB0_BASE,  USB_EP_1,  USB_EP_HOST_OUT |  
                         USB_EP_DMA_MODE_0);  
  
//  
// Assign endpoint 1 to DMA channel 0 using Mode 0, no bursting, for  
// transmit, and enable DMA interrupts.  
//  
USBDMAChannelConfigSet(USB0_BASE,  0,  USB_EP_1,  USB_DMA_CFG_MODE_0 |  
                         USB_DMA_CFG_BURST_NONE |  
                         USB_DMA_CFG_DIR_TX |  
                         USB_DMA_CFG_INT_EN);
```

The following actions must be performed every time a DMA transfer is ready to be scheduled.

Example: Sending a packet using DMA on channel 0.

```
//  
// Set the source address for the transfer to pvBuffer.  
//  
USBDMAChannelAddressSet(USB0_BASE,  0,  pvBuffer);  
  
//  
// Set the transfer size to 44 bytes and the packet count to 0.  
//  
USBDMAChannelCountSet(USB0_BASE,  0,  44);  
USBEndpointPacketCountSet(USB0_BASE,  0,  0)  
  
//  
// Enable the DMA transfer.  
//  
USBDMAChannelEnable();
```

Once the DMA transfer is started, the application must wait for a DMA completion interrupt. DMA completion triggers a normal USB controller interrupt, and the actual status for the DMA interrupt is returned by calling the [USBDMAChannelIntStatus\(\)](#) function. The interrupt handler must handle all pending DMA channels because the call to [USBDMAChannelIntStatus\(\)](#) automatically clears all pending DMA interrupts. Once the DMA status indicates complete, the last step is to schedule the USB transfer by calling the [USBEndpointDataSend\(\)](#) function as shown below and then wait for a second endpoint interrupt that signals the completion of the USB transfer. The second interrupt must check if the transfer has completed by calling the [USBIntStatusEndpoint\(\)](#) function to see if the interrupt was for the given endpoint.

Example: Starting a transfer on channel 0 after DMA completes.

```
//  
// Start the USB transfer.  
//  
USBEndpointDataSend(USB0_BASE,  USB_EP_1,  USB_TRANS_OUT);
```

33.4.3 Sending Multiple Packets

The next type of transfer is a multiple packet transfer when the transfer size is greater than the maximum packet size for the endpoint. In this mode, the DMA controller copies data from memory into the USB FIFO for the endpoint in blocks that are the size of the maximum packet. If the last packet is a short packet, it is the responsibility of the application to manually start the USB transfer when the DMA transfer is complete. Because the **USB_EP_AUTO_SET** is used when multiple packets are being sent, it is not necessary to manually trigger the final transfer if the transfer is of exactly the maximum packet size.

If endpoints are not dynamically allocated by the application, some of the DMA and endpoint configuration can be treated as static and executed only once during initial configuration.

Example: Static endpoint configuration for DMA multiple packet transmit.

```

//
// Endpoint 1 uses Mode 1 and transmit and enables automatic sending
// when a full packet is sent to the FIFO.
//
USBEndpointDMAConfigSet(USB0_BASE,  USB_EP_1,  USB_EP_HOST_OUT |
                        USB_EP_DMA_MODE_1 |
                        USB_EP_AUTO_SET);

//
// Assign endpoint 1 to DMA channel 1 using Mode 1, no bursting, transmit,
// and enable DMA interrupts.
//
USBDMAChannelConfigSet(USB0_BASE,  1,  USB_EP_1,  USB_DMA_CFG_MODE_1 |
                        USB_DMA_CFG_BURST_NONE |
                        USB_DMA_CFG_DIR_TX |
                        USB_DMA_CFG_INT_EN)

```

The following actions must be performed every time a DMA transfer is scheduled. If the transfer size is not an even multiple of the maximum packet size, then the **USBEndpointPacketCountSet()** is passed in $(\text{transfer_size}/\text{max_packet_size}) + 1$ otherwise the number of packets is $(\text{transfer_size}/\text{max_packet_size})$.

Example: Starting a DMA send multi-packet transfer on channel 1.

```

//
// Set the source address for the transfer to pvBuffer.
//
USBDMAChannelAddressSet(USB0_BASE,  1,  pvBuffer);

//
// Set the transfer size to 1024 bytes and the packet count to 16.
// The packet count does not require a + 1 because 1024/64 leaves no
// remaining bytes to send in a final packet.
//
USBDMAChannelCountSet(USB0_BASE,  0,  1024);
USBEndpointPacketCountSet(USB0_BASE,  0,  1024/64);

//
// Enable the DMA transfer.
//
USBDMAChannelEnable();

```

Once the DMA transfer is started, the application must wait for a DMA completion interrupt. The DMA completion triggers a normal USB controller interrupt, and the status for the DMA interrupt is returned by calling the **USBDMAChannelIntStatus()**. The interrupt handler must handle all pending

DMA channels because the call to [USBDMAChannelIntStatus\(\)](#) automatically clears all pending DMA interrupts. At this point, if the transfer size is a multiple of the maximum packet size, the transfer is complete and no other action is required. If there is a short packet at the end of the transfer, then the last step is to schedule the final USB transfer by calling the [USBEndpointDataSend\(\)](#) and then wait for a final endpoint interrupt to signal completion of the USB transfer. The transfer is complete when the application receives a final endpoint interrupt, which is returned by the [USBIntStatusEndpoint\(\)](#) function.

33.4.4 Receiving a Single Packet

Receiving packets is a little more complicated than sending because in some cases you do not know when the receive request occurs or how much data is being sent when you configure the endpoint. This section handles receiving a single packet transfer that is less than or equal to the maximum packet size for an endpoint. In this mode, the DMA controller copies data from the USB endpoint FIFO to system memory. The DMA transfer of a single packet must be started when the packet is received. If the **USB_EP_AUTO_CLEAR** option is included in the call to [USBEndpointDMAConfigSet\(\)](#), then the receive is acknowledged when the DMA transfer pulls the last data from the FIFO. The DMA transfer is started when the endpoint receive interrupt indicates that data is available in the FIFO. The examples below demonstrate the steps necessary to complete a DMA transfer of a single packet.

If endpoints are not dynamically allocated by the application, some of the DMA and endpoint configuration can be treated as static and executed only once during initial configuration.

Example: Static endpoint configuration for single packet DMA receive

```
//  
// Endpoint 2 uses Mode 0, receive, and enables automatic acknowledge  
// when a full packet is read from the FIFO.  
//  
USBEndpointDMAConfigSet(USB0_BASE, USB_EP_2, USB_EP_HOST_IN |  
                         USB_EP_DMA_MODE_0 |  
                         USB_EP_AUTO_CLEAR);  
  
//  
// Assign endpoint 2 to DMA channel 3 using Mode 0, no bursting, receive,  
// and enable DMA interrupts.  
//  
USBDMAChannelConfigSet(USB0_BASE, 3, USB_EP_2, USB_DMA_CFG_MODE_0 |  
                         USB_DMA_CFG_BURST_NONE |  
                         USB_DMA_CFG_DIR_RX |  
                         USB_DMA_CFG_INT_EN);  
  
//  
// Make sure that DMA is not enabled on the endpoint. If DMA is left  
// enabled, the endpoint interrupt does not occur.  
//  
USBEndpointDMADisable(USB0_BASE, 3, USB_EP_HOST_IN);
```

Once the endpoint interrupt indicates that a packet is ready for transfer, then the application takes the following steps to start the DMA transfer to receive the data from the FIFO.

Example: Starting a DMA receive transfer on channel 3.

```
//  
// Set the destination address for the transfer to pvBuffer.  
//  
USBDMAChannelAddressSet(USB0_BASE, 3, pvBuffer);
```

```

//
// Set the transfer size to 48 bytes and the packet count to 0.
//
USBDMACChannelCountSet(USB0_BASE, 3, 48);
USBEndpointPacketCountSet(USB0_BASE, 3, 0)

//
// Enable the DMA transfer.
//
USBDMAChannelEnable();

```

Now the application must wait for a final interrupt to indicate that the DMA has completed by checking the return value from the [USBDMACHannelIntStatus\(\)](#) function. If the endpoint was not configured with the **USB_EP_AUTO_CLEAR** function, then a final call to [USBDevEndpointDataAck\(\)](#) or [USBHostEndpointDataAck\(\)](#) is required to acknowledge the packet. The **USB_EP_AUTO_CLEAR** feature must not be used if the packet may need to be stalled.

33.4.5 Receiving a Multiple Packets

This section handles receiving multiple packets with a single DMA transfer. This method is used when the total size of the transfer is greater than the maximum packet size for the endpoint. If there is a trailing null packet or short packet, the final transfer must be manually acknowledged with a call to [USBDevEndpointDataAck\(\)](#) or [USBHostEndpointDataAck\(\)](#). Unlike the single packet, this transfer can be configured before the packet is received if the size of the transfer is known beforehand. Because this type of USB DMA uses mode 1, the transfer only starts if a full packet has been received. If a short packet is unexpectedly received, the transfer does not start, but the USB endpoint interrupt is still signaled so that the transfer can be started manually if needed. This mechanism is also needed anytime the final packet is a short packet. The examples below demonstrate the steps necessary to complete a DMA transfer of multiple packets.

If endpoints are not dynamically allocated by the application, some of the DMA and endpoint configuration can be treated as static and executed only once during initial configuration.

Example: Static endpoint configuration for DMA multiple packet receive

```

//
// If in USB device mode.
//
if(bUSBDeviceMode)
{
    USBEndpointDMAConfigSet(USB0_BASE, USB_EP_2, USB_EP_HOST_IN |
                           USB_EP_DMA_MODE_1 |
                           USB_EP_AUTO_CLEAR);
}
else
{
    //
    // In host mode, USB_EP_AUTO_REQUEST is needed to trigger new requests.
    //
    USBEndpointDMAConfigSet(USB0_BASE, USB_EP_2, USB_EP_HOST_IN |
                           USB_EP_AUTO_REQUEST |
                           USB_EP_DMA_MODE_1 |
                           USB_EP_AUTO_CLEAR);
}

//
// Assign endpoint 2 to DMA channel 3 using Mode 1, no bursting, receive,

```

```
// and enable DMA interrupts.  
//  
USBDMAChannelConfigSet(USB0_BASE, 3, USB_EP_2, USB_DMA_CFG_MODE_1 |  
                        USB_DMA_CFG_BURST_NONE |  
                        USB_DMA_CFG_DIR_RX |  
                        USB_DMA_CFG_INT_EN)  
  
//  
// Make sure that DMA is not enabled on the endpoint. If DMA is left  
// enabled, the endpoint interrupt does not occur.  
//  
USBEndpointDMADisable(USB0_BASE, 3, USB_EP_HOST_IN);
```

When the application is ready to start the multiple packet transfer, the application starts the transfer by taking the following steps.

Example: Starting a multiple packet DMA receive transfer on channel 3.

```
//  
// Set the destination address for the transfer to pvBuffer.  
//  
USBDMAChannelAddressSet(USB0_BASE, 3, pvBuffer);  
  
//  
// Set the transfer size to 512 bytes and the packet count to 8.  
//  
USBDMAChannelCountSet(USB0_BASE, 3, 512);  
USBEndpointPacketCountSet(USB0_BASE, 3, 512/64);  
  
//  
// Enable the DMA transfer.  
//  
USBEndpointDMAEnable(USB0_BASE, USB_EP_2, USB_EP_HOST_IN);  
USBDMAChannelEnable(USB0_BASE, 3);
```

Now the application waits for the interrupt to indicate that either the DMA has completed or that a short packet has been received. If [USBDMAChannelIntStatus\(\)](#) indicates that the transfer is complete, then there are no more steps to take. If there is no pending DMA interrupt, then a short packet was received, and the application can either manually read the data from the FIFO or switch to the steps above to trigger a single packet transfer to complete the USB transfer.

The remainder of the USB functions are not directly needed for USB DMA transfers but may be needed for other application-specific reasons. The first two of these remaining functions are [USBDMAChannelIntEnable\(\)](#) and [USBDMAChannelIntDisable\(\)](#), which are a pair of functions to enable and disable a specific DMA channel from generating interrupts. The [USBDMAChannelEnable\(\)](#) function is explained previously. The [USBDMAChannelDisable\(\)](#) function disables a specific DMA channel if necessary. In some cases, there is global DMA status that is reported to the application via the [USBDMAChannelStatus\(\)](#) function, which can be handled and cleared by calling the [USBDMAChannelStatusClear\(\)](#) function. See the documentation for the [USBDMAChannelStatusClear\(\)](#) function for more details on the possible status values.

33.4.6 Function Documentation

33.4.6.1 USBDMAChannelAddressGet

Returns the source or destination address for the specified integrated USB DMA channel.

Prototype:

```
void *
USBDMAChannelAddressGet(uint32_t ui32Base,
                        uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies the USB DMA channel.

Description:

This function returns the DMA address for the channel number specified in the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the DMA channel to query. This function must not be used on devices that return **USB_CONTROLLER_VER_0** from the **USBControllerVersion()** function.

Example: Get the transfer address for USB DMA channel 1.

```
void *pvBuffer;

//
// Retrieve the current DMA address for channel 1.
//
pvBuffer = USBDMAChannelAddressGet(USBO_BASE, 1);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

The current DMA address for a USB DMA channel.

33.4.6.2 USBDMAChannelAddressSet

Sets the source or destination address for an integrated USB DMA transfer on a specified channel.

Prototype:

```
void
USBDMAChannelAddressSet(uint32_t ui32Base,
                        uint32_t ui32Channel,
                        void *pvAddress)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to configure.

pvAddress specifies the source or destination address for the USB DMA transfer.

Description:

This function sets the source or destination address for the USB DMA channel number specified in the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the USB DMA channel. The *pvAddress* parameter is a source address if the transfer type for the DMA channel is transmit and a destination address if the transfer type is receive.

Example: Set the transfer address for USB DMA channel 1.

```
void *pvBuffer;  
  
//  
// Set the address for USB DMA channel 1.  
//  
USBDMAChannelAddressSet(USB0_BASE, 1, pvBuffer);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.3 USBDMAChannelConfigSet

Assigns and configures an endpoint to a specified integrated USB DMA channel.

Prototype:

```
void  
USBDMAChannelConfigSet(uint32_t ui32Base,  
                        uint32_t ui32Channel,  
                        uint32_t ui32Endpoint,  
                        uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to access.

ui32Endpoint is the endpoint to assign to the USB DMA channel.

ui32Config is used to specify the configuration of the USB DMA channel.

Description:

This function assigns an endpoint and configures the settings for a USB DMA channel. The *ui32Endpoint* parameter is one of the **USB_EP_*** values and the *ui32Channel* value is a zero-based index of the DMA channel to configure. The *ui32Config* parameter is a combination of the **USB_DMA_CFG_*** values using the following guidelines.

Use one of the following to set the DMA burst mode:

- **USB_DMA_CFG_BURST_NONE** disables bursting.
- **USB_DMA_CFG_BURST_4** sets the DMA burst size to 4 words.
- **USB_DMA_CFG_BURST_8** sets the DMA burst size to 8 words.
- **USB_DMA_CFG_BURST_16** sets the DMA burst size to 16 words.

Use one of the following to set the DMA mode:

- **USB_DMA_CFG_MODE_0** is typically used when only a single packet is being sent via DMA and triggers one completion interrupt per packet.
- **USB_DMA_CFG_MODE_1** is typically used when multiple packets are being sent via DMA and triggers one completion interrupt per transfer.

Use one of the following to set the direction of the transfer:

- **USB_DMA_CFG_DIR_RX** selects a DMA transfer from the endpoint to a memory location.

- **USB_DMA_CFG_DIR_TX** selects a DMA transfer to the endpoint from a memory location.

The following two optional settings allow an application to immediately enable the DMA transfer and/or DMA interrupts when configuring the DMA channel:

- **USB_DMA_CFG_INT_EN** enables interrupts for this channel immediately so that an added call to [USBDMAChannelIntEnable\(\)](#) is not necessary.
- **USB_DMA_CFG_EN** enables the DMA channel immediately so that an added call to [USBDMAChannelEnable\(\)](#) is not necessary.

Example: Assign channel 0 to endpoint 1 in DMA mode 0, 4 word burst, enable interrupts and immediately enable the transfer.

```

//
// Assign channel 0 to endpoint 1 in DMA mode 0, 4 word bursts,
// enable interrupts and immediately enable the transfer.
//
USBDMAChannelConfigSet(USB0_BASE, 0, USB_EP_1,
                        (USB_DMA_CFG_BURST_4 | USB_DMA_CFG_MODE0 |
                         USB_DMA_CFG_DIR_RX | USB_DMA_CFG_INT_EN |
                         USB_DMA_CFG_EN));

```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.4 USBDMAChannelCountGet

Returns the transfer count for an integrated USB DMA channel.

Prototype:

```

uint32_t
USBDMAChannelCountGet(uint32_t ui32Base,
                      uint32_t ui32Channel)

```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to access.

Description:

This function returns the USB DMA transfer count in bytes for the channel number specified in the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the DMA channel to query.

Example: Get the transfer count for USB DMA channel 1.

```

uint32_t ui32Count;

//
// Get the transfer count for USB DMA channel 1.
//
ui32Count = USBDMAChannelCountGet(USB0_BASE, 1);

```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

The current count for a USB DMA channel.

33.4.6.5 USBDMAChannelCountSet

Sets the transfer count for an integrated USB DMA channel.

Prototype:

```
void
USBDMAChannelCountSet(uint32_t ui32Base,
                      uint32_t ui32Channel,
                      uint32_t ui32Count)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to access.

ui32Count specifies the number of bytes to transfer.

Description:

This function sets the USB DMA transfer count in bytes for the channel number specified in the ***ui32Channel*** parameter. The ***ui32Channel*** value is a zero-based index of the DMA channel.

Example: Set the transfer count to 512 bytes for USB DMA channel 1.

```
//
// Set the transfer count to 512 bytes for USB DMA channel 1.
//
USBDMAChannelCountSet(USBO_BASE, 1, 512);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.6 USBDMAChannelDisable

Disables integrated USB DMA for a specified channel.

Prototype:

```
void
USBDMAChannelDisable(uint32_t ui32Base,
                      uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies the USB DMA channel to disable.

Description:

This function disables the USB DMA channel passed in the *ui32Channel* parameter. The *ui32Channel* parameter is a zero-based index of the DMA channel.

Example: Disable USB DMA channel 2.

```
//  
// Disable USB DMA channel 2.  
//  
USBDMAChannelDisable(2);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.7 USBDMAChannelEnable

Enables integrated USB DMA for a specified channel.

Prototype:

```
void  
USBDMAChannelEnable(uint32_t ui32Base,  
                     uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies the USB DMA channel to enable.

Description:

This function enables the USB DMA channel passed in the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the USB DMA channel.

Example: Enable USB DMA channel 2.

```
//  
// Enable USB DMA channel 2.  
//  
USBDMAChannelEnable(2);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.8 USBDMAChannellntDisable

Disable interrupts for a specified integrated USB DMA channel.

Prototype:

```
void
USBDMAChannelIntDisable(uint32_t ui32Base,
                         uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which USB DMA channel interrupt to disable.

Description:

This function disables the USB DMA channel interrupt based on the *ui32Channel* parameter.

The *ui32Channel* value is a zero-based index of the USB DMA channel.

Example: Disable the USB DMA channel 3 interrupt.

```
//
// Disable the USB DMA channel 3 interrupt
//
USBDMAChannelIntDisable(USB0_BASE, 3);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.9 USBDMAChannelIntEnable

Enable interrupts for a specified integrated USB DMA channel.

Prototype:

```
void
USBDMAChannelIntEnable(uint32_t ui32Base,
                        uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel interrupt to enable.

Description:

This function enables the USB DMA channel interrupt based on the *ui32Channel* parameter.

The *ui32Channel* value is a zero-based index of the USB DMA channel. Once enabled, the [USBDMAChannelIntStatus\(\)](#) function returns if a DMA channel has generated an interrupt.

Example: Enable the USB DMA channel 3 interrupt.

```
//
// Enable the USB DMA channel 3 interrupt
//
USBDMAChannelIntEnable(USB0_BASE, 3);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.10 USBDMAChannelIntStatus

Return the current status of the integrated USB DMA interrupts.

Prototype:

```
uint32_t
USBDMAChannelIntStatus(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current bit-mapped interrupt status for all USB DMA channel interrupt sources. Calling this function automatically clears all currently pending USB DMA interrupts.

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Example: Get the pending USB DMA interrupts.

```
uint32_t ui32Ints;
// Get the pending USB DMA interrupts.
// ui32Ints = USBDMAChannelIntStatus(USB0_BASE);
```

Returns:

The bit-mapped interrupts for the DMA channels.

33.4.6.11 USBDMAChannelStatus

Returns the current status for an integrated USB DMA channel.

Prototype:

```
uint32_t
USBDMAChannelStatus(uint32_t ui32Base,
                     uint32_t ui32Channel)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to query.

Description:

This function returns the current status for the USB DMA channel specified by the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the USB DMA channel to query.

Example: Get the current USB DMA status for channel 2.

```
uint32_t ui32Status;  
  
//  
// Get the current USB DMA status for channel 2.  
//  
ui32Status = USBDMAChannelStatus(USB0_BASE, 2);
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

Returns zero or **USB_DMACTL0_ERR** if there is a pending error condition on a DMA channel.

33.4.6.12 USBDMAChannelStatusClear

Clears the integrated USB DMA status for a specified channel.

Prototype:

```
void  
USBDMAChannelStatusClear(uint32_t ui32Base,  
                         uint32_t ui32Channel,  
                         uint32_t ui32Status)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Channel specifies which DMA channel to clear.

ui32Status holds the status bits to clear.

Description:

This function clears the USB DMA channel status for the channel specified by the *ui32Channel* parameter. The *ui32Channel* value is a zero-based index of the USB DMA channel to query. The *ui32Status* parameter specifies the status bits to clear and must be the valid values that are returned from a call to the **USBDMAChannelStatus()** function.

Example: Clear the current USB DMA status for channel 2.

```
//  
// Clear the any pending USB DMA status for channel 2.  
//  
USBDMAChannelStatusClear(USB0_BASE, 2, USBDMAChannelStatus(USB0_BASE, 2));
```

Note:

This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers.

Returns:

None.

33.4.6.13 USBDMANumChannels

Returns the available number of integrated USB DMA channels.

Prototype:

```
uint32_t
USBDMANumChannels(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the total number of DMA channels available when using the integrated USB DMA controller. This function returns 0 if the integrated controller is not present.

Example: Get the number of integrated DMA channels.

```
uint32_t ui32Count;

//
// Get the number of integrated DMA channels.
//
ui32Count = USBDMANumChannels(USB0_BASE);
```

Returns:

The number of integrated USB DMA channels or zero if the integrated USB DMA controller is not present.

33.5 USB Link Power Management Functions

Functions

- void [USBDevLPMConfig](#) (uint32_t ui32Base, uint32_t ui32Config)
- void [USBDevLPMDisable](#) (uint32_t ui32Base)
- void [USBDevLPMEnable](#) (uint32_t ui32Base)
- void [USBDevLPMRemoteWake](#) (uint32_t ui32Base)
- void [USBHostLPMConfig](#) (uint32_t ui32Base, uint32_t ui32ResumeTime, uint32_t ui32Config)
- void [USBHostLPMResume](#) (uint32_t ui32Base)
- void [USBHostLPMSend](#) (uint32_t ui32Base, uint32_t ui32Address, uint32_t ui32Endpoint)
- uint32_t [USBLPMEndpointGet](#) (uint32_t ui32Base)
- void [USBLPMIntDisable](#) (uint32_t ui32Base, uint32_t ui32Ints)
- void [USBLPMIntEnable](#) (uint32_t ui32Base, uint32_t ui32Ints)
- uint32_t [USBLPMIntStatus](#) (uint32_t ui32Base)
- uint32_t [USBLPMLinkStateGet](#) (uint32_t ui32Base)
- bool [USBLPMRemoteWakeEnabled](#) (uint32_t ui32Base)

33.5.1 Detailed Description

Some Tiva microcontrollers controllers include support for the USB Link Power Management (LPM) feature. This feature allows a USB host to put a device into sleep mode much faster than normally possible with the default USB suspend, which takes about 3ms. The Tiva USB controller only supports entering the L1 state, which is a sleep state and not the full suspend state (L2). The L1 state has no specific power requirements and, with LPM, can be entered and exited much faster

than the suspend state. Even with the L1 state implemented, the suspend state is still supported when the USB bus is idle for 3ms. The L1 sleep state is supported in both host and device modes including support for LPM remote wake. This section covers the support available in DriverLib for both USB host and device modes and all of the configuration options.

33.5.2 Host Mode LPM Support

The USB host mode support for LPM includes the ability to send a request for a device to enter LPM mode and to resume from the LPM-initiated sleep mode. The application can make a request at any time by calling the [USBHostLPMSend\(\)](#) function, which requests the device to enter LPM mode as soon as the command is sent on the USB bus. When the host wants to wake the device, it can call the [USBHostLPMResume\(\)](#) function, or if the device supports remote wake from LPM, the device can also initiate resume signaling.

The USB controller must be configured before use by calling the [USBHostLPMConfig\(\)](#) function. Two configuration options are available: the resume time and if the device is allowed to use the remote wake feature. The resume time is specified in microseconds ranging from 50us to 1200us and is passed in the *ulResumeTime* parameter of the [USBHostLPMConfig\(\)](#) function. The remaining option determines if the device is allowed to issue remote wake signaling by using the **USB_HOST_LPM_RMTWAKE** option. The **USB_HOST_LPM_L1** option is at present the only mode that is supported when using LPM. The following example demonstrates how to configure and enable USB LPM support.

Example: Configure LPM in host mode.

```
//  
// Enable LPM with 500us resume signaling and enable remote wake.  
//  
USBHostLPMConfig(USB0_BASE, 500, USB_HOST_LPM_RMTWAKE | USB_HOST_LPM_L1)  
  
//  
// Enable all LPM related interrupts for host mode.  
//  
USBLPMIntEnable(USB0_BASE, USB_INTLPM_RESUME | USB_INTLPM_INCOMPLETE |  
    USB_INTLPM_ACK | USB_INTLPM_NYET |  
    USB_INTLPM_STALL);
```

After an LPM request to enter the L1 sleep state, the device can respond in one of four ways. The possible responses all generate a USB interrupt if the interrupts are enabled with a call to [USBLPMIntEnable\(\)](#), or the application can poll for a response using the [USBLPMIntStatus\(\)](#) function. It is important to remember that any call to [USBLPMIntStatus\(\)](#) clears all currently pending interrupts, so any pending interrupts must be handled after this call. The following are the valid interrupt responses returned from a call to [USBLPMIntStatus\(\)](#):

- **USB_INTLPM_INCOMPLETE** - The device failed to respond to the LPM request.
- **USB_INTLPM_ACK** - The device received the response and accepted the command.
- **USB_INTLPM_NYET** - The device responded with NYET indicating that it is not prepared to handle the request at this time.
- **USB_INTLPM_STALL** - The device received the command but does not support this LPM request.
- **USB_INTLPM_RESUME** - The device has resumed from the L1 state due to a host request or a remote resume request.

33.5.3 Device Mode LPM Support

The USB device mode support for LPM includes the ability to receive requests to enter and exit the L1 sleep state. If the host enables remote wake from the L1 state, then the device can issue remote wake requests by calling the [USBDevLPMRemoteWake\(\)](#) function after a device has been placed into the L1 state by an LPM request from the host.

The USB device can be configured to handle the incoming LPM request in multiple ways depending on the application requirements. Before the USB controller can properly respond to LPM commands, it must first be configured by calling the [USBDevLPMConfig\(\)](#) function with the required configuration options. The three valid configurations that control how the USB controller responds to LPM requests from the host are:

- **USB_DEV_LPM_NONE** - Do not respond to any LPM requests, causing a timeout on the host side.
- **USB_DEV_LPM_EN** - Enable full LPM responses from the device.
- **USB_DEV_LPM_EXTONLY** - Receive extended packets, but do not handle LPM requests.

Example: Configure LPM in device mode.

```
//  
// Enable full LPM support in device mode.  
//  
USBDevLPMConfig(USBO_BASE, USB_DEV_LPM_EN)  
  
//  
// Enable all LPM related interrupts for host mode.  
//  
USBLPMIntEnable(USBO_BASE, USB_INTLPM_ERROR | USB_INTLPM_RESUME |  
                 USB_INTLPM_INCOMPLETE | USB_INTLPM_ACK |  
                 USB_INTLPM_NYET | USB_INTLPM_STALL);
```

When a USB device receives an LPM command from the host, the results are stored and an interrupt is triggered. The results of the last successful LPM request are returned by calling the [USBLPMLinkStateGet\(\)](#) function, and the targeted endpoint is returned by calling [USBLPMEndpointGet\(\)](#). These functions should be called to properly handle the incoming LPM request. The only valid link state change that is supported is a change to the remote wake feature, which is enabled if the [USBLPMLinkStateGet\(\)](#) return value has the **USB_DEV_LPM_LS_RMTWAKE** bit set.

Example: Handling an incoming LPM request.

```
unsigned long ulStatus;  
unsigned long ulEndpoint;  
  
//  
// Get the current link state and the targeted endpoint.  
//  
ulStatus = USBLPMLinkStateGet(USBO_BASE);  
ulEndpoint = USBLPMEndpointGet(USBO_BASE);  
  
//  
// Check if remote wake is enabled.  
//  
if(ulStatus & USB_DEV_LPM_LS_RMTWAKE)  
{  
    // Handle enable of remote wake.  
}
```

```
    else
    {
        // Handle disable of remote wake.
    }
```

If the USB host controller enables remote wake on the device by sending an LPM request with the remote wake feature enabled, then the device is allowed to send remote wake requests to the host. A remote wake is sent by the device calling the [USBDevLPMRemoteWake\(\)](#) function.

Example: Sending a remote wake from a USB device.

```
//
// Send a remote wake signal.
//
USBDevLPMRemoteWake(USB0_BASE);
```

33.5.4 Function Documentation

33.5.4.1 USBDevLPMConfig

Configures the USB device mode response to LPM requests.

Prototype:

```
void
USBDevLPMConfig(uint32_t ui32Base,
                 uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Config is the combination of configuration options for LPM transactions in device mode.

Description:

This function sets the global configuration options for LPM transactions in device mode and must be called before ever calling [USBDevLPMEnable\(\)](#) to set the configuration for LPM transactions. The configuration options in device mode are specified in the *ui32Config* parameter and include one of the following:

- **USB_DEV_LPM_NONE** disables the USB controller from responding to LPM transactions.
- **USB_DEV_LPM_EN** enables the USB controller to respond to LPM and extended transactions.
- **USB_DEV_LPM_EXONLY** enables the USB controller to respond to extended transactions, but not LPM transactions.

The *ui32Config* option can also optionally include the **USB_DEV_LPM_NAK** value to cause the USB controller to NAK all transactions other than an LPM transaction once the USB controller is in LPM suspend mode. If this value is not included in the *ui32Config* parameter, the USB controller does not respond in suspend mode.

The USB controller does not enter LPM suspend mode until the application calls the [USBDevLPMEnable\(\)](#) function.

Example: Enable LPM transactions and NAK while in LPM suspend mode.

```

//  

// Enable LPM transactions and NAK while in LPM suspend mode.  

//  

USBDevLPMConfig(USB0_BASE,  USB_DEV_LPM_NAK | USB_DEV_LPM_EN);

```

Note:

This function must only be called in device mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.2 USBDevLPMDisable

Disables the USB controller from responding to LPM suspend requests.

Prototype:

```

void  

USBDevLPMDisable(uint32_t ui32Base)

```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables the USB controller from responding to LPM transactions. When the device enters LPM L1 mode, the USB controller automatically disables responding to further LPM transactions.

Note:

If LPM transactions were enabled before calling this function, then an LPM request can still occur before this function returns. As a result, the application must continue to handle LPM requests until this function returns.

Example: Disable LPM suspend mode.

```

//  

// Disable LPM suspend mode.  

//  

USBDevLPMDisable(USB0_BASE);

```

Note:

This function must only be called in device mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.3 USBDevLPMEnable

Enables the USB controller to respond to LPM suspend requests.

Prototype:

```
void  
USBDevLPMEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function is used to automatically respond to an LPM sleep request from the USB host controller. If there is no data pending in any transmit FIFOs, then the USB controller acknowledges the packet and enters the LPM L1 state and generates the **USB_INTLPM_ACK** interrupt. If the USB controller has pending transmit data in at least one FIFO, then the USB controller responds with NYET and signals the **USB_INTLPM_INCOMPLETE** or **USB_INTLPM_NYET** depending on if data is pending in receive or transmit FIFOs. A call to [USBDevLPMEnable\(\)](#) is required after every LPM resume event to re-enable LPM mode.

Example: Enable LPM suspend mode.

```
//  
// Enable LPM suspend mode.  
//  
USBDevLPMEnable(USBO_BASE);
```

Note:

This function must only be called in device mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.4 USBDevLPMRemoteWake

Initiates remote wake signaling to request the device to leave LPM suspend mode.

Prototype:

```
void  
USBDevLPMRemoteWake(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function initiates remote wake signaling to request that the host wake a device that has entered an LPM-triggered low power mode.

Example: Initiate remote wake signaling.

```
//  
// Initiate remote wake signaling.  
//  
USBDevLPMRemoteWake(USBO_BASE);
```

Note:

This function must only be called in device mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.5 USBHostLPMConfig

Sets the global configuration for all LPM requests.

Prototype:

```
void
USBHostLPMConfig(uint32_t ui32Base,
                  uint32_t ui32ResumeTime,
                  uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32ResumeTime specifies the resume signaling duration in 75us increments.

ui32Config specifies the combination of configuration options for LPM transactions.

Description:

This function sets the global configuration options for LPM transactions and must be called at least once before ever calling [USBHostLPMSend\(\)](#). The *ui32ResumeTime* specifies the length of time that the host drives resume signaling on the bus in microseconds. The valid values for *ui32ResumeTime* are from 50us to 1175us in 75us increments. The remaining configuration is specified by the *ui32Config* parameter and includes the following options:

- **USB_HOST_LPM_RMTWAKE** allows the device to signal a remote wake from the LPM state.
- **USB_HOST_LPM_L1** is the LPM mode to enter and must always be included in the configuration.

Example: Set the LPM configuration to allow remote wake with a resume duration of 500us.

```
//
// Set the LPM configuration to allow remote wake with a resume
// duration of 500us.
//
USBHostLPMConfig(USB0_BASE, 500, USB_HOST_LPM_RMTWAKE | USB_HOST_LPM_L1);
```

Note:

This function must only be called in host mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.6 USBHostLPMResume

Initiates resume signaling to wake a device from LPM suspend mode.

Prototype:

```
void
USBHostLPMResume(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

In host mode, this function initiates resume signaling to wake a device that has entered an LPM-triggered low power mode. This LPM-triggered low power mode is entered when the [USBHostLPMSend\(\)](#) is called to put a specific device into a low power state.

Example: Initiate resume signaling.

```
//  
// Initiate resume signaling.  
//  
USBHostLPMSend(USB0_BASE);
```

Note:

This function must only be called in host mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.7 USBHostLPMSend

Sends an LPM request to a device at a specified address and endpoint number.

Prototype:

```
void  
USBHostLPMSend(uint32_t ui32Base,  
                uint32_t ui32Address,  
                uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Address is the target device address for the LPM request.

ui32Endpoint is the target endpoint for the LPM request.

Description:

This function sends an LPM request to a connected device in host mode. The *ui32Address* parameter specifies the device address and has a range of values from 1 to 127. The *ui32Endpoint* parameter specifies the endpoint on the device to which to send the LPM request and must be one of the **USB_EP_*** values. The function returns before the LPM request is sent, requiring the caller to poll the [USBLPMIntStatus\(\)](#) function or wait for an interrupt to signal completion of the LPM transaction. This function must only be called after the [USB-HostLPMConfig\(\)](#) has configured the LPM transaction settings.

Example: Send an LPM request to the device at address 1 on endpoint 0.

```
//  
// Send an LPM request to the device at address 1 on endpoint 0.  
//  
USBHostLPMSend(USB0_BASE, 1, USB_EP_0);
```

Note:

This function must only be called in host mode. The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.8 USBLPMEndpointGet

Returns the current LPM endpoint value.

Prototype:

```
uint32_t
USBLPMEndpointGet (uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current LPM endpoint value. The meaning of the value depends on the mode of operation of the USB controller. When in device mode, the value returned is the endpoint that received the last LPM transaction. When in host mode this is the endpoint that was last sent an LPM transaction, or the endpoint that is configured to be sent when the LPM transaction is triggered. The value returned is in the **USB_EP_[0-7]** value and a direct endpoint index.

Example: Get the endpoint for the last LPM transaction.

```
uint32_t ui32Endpoint;
//
// Get the endpoint number that received the LPM request.
//
ui32LinkState = USBLPMEndpointGet (USB0_BASE);
```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

The last endpoint to receive an LPM request in device mode or the endpoint that the host sends an LPM request as one of the **USB_EP_[0-7]** values.

33.5.4.9 USBLPMIntDisable

Disables LPM interrupts.

Prototype:

```
void
USBLPMIntDisable (uint32_t ui32Base,
                   uint32_t ui32Ints)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Ints specifies which LPM interrupts to disable.

Description:

This function disables the LPM interrupts specified in the ***ui32Ints*** parameter, preventing them from triggering a USB interrupt.

The valid interrupt status bits when the USB controller is acting as a host are the following:

- **USB_INTLPM_ERROR** a bus error occurred in the transmission of an LPM transaction.
- **USB_INTLPM_RESUME** the USB controller has resumed from LPM low power state.
- **USB_INTLPM_INCOMPLETE** the LPM transaction failed because a timeout occurred or there were bit errors in the response for three attempts.
- **USB_INTLPM_ACK** the device has acknowledged an LPM transaction.
- **USB_INTLPM_NYET** the device has responded with a NYET to an LPM transaction.
- **USB_INTLPM_STALL** the device has stalled an LPM transaction.

The valid interrupt status bits when the USB controller is acting as a device are the following:

- **USB_INTLPM_ERROR** an LPM transaction was received that has an unsupported link state field. The transaction was stalled, but the requested link state can still be read using the [USBLPMLinkStateGet\(\)](#) function.
- **USB_INTLPM_RESUME** the USB controller has resumed from the LPM low power state.
- **USB_INTLPM_INCOMPLETE** the USB controller responded to an LPM transaction with a NYET because data was still in the transmit FIFOs.
- **USB_INTLPM_ACK** the USB controller acknowledged an LPM transaction and is now in the LPM suspend mode.
- **USB_INTLPM_NYET** the USB controller responded to an LPM transaction with a NYET because LPM transactions are not yet enabled by a call to [USBDevLPMEnable\(\)](#).
- **USB_INTLPM_STALL** the USB controller has stalled an incoming LPM transaction.

Example: Disable all LPM interrupt sources.

```
//  
// Disable all LPM interrupt sources.  
//  
USBLPMIntDisable(USB0_BASE,  USB_INTLPM_ERROR | USB_INTLPM_RESUME |  
                    USB_INTLPM_INCOMPLETE | USB_INTLPM_ACK |  
                    USB_INTLPM_NYET | USB_INTLPM_STALL);
```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.10 USBLPMIntEnable

Enables LPM interrupts.

Prototype:

```
void
USBLPMIntEnable(uint32_t ui32Base,
                 uint32_t ui32Ints)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Ints specifies which LPM interrupts to enable.

Description:

This function enables a set of LPM interrupts so that they can trigger a USB interrupt. The *ui32Ints* parameter specifies which of the **USB_INTLPM_*** to enable.

The valid interrupt status bits when the USB controller is acting as a host are the following:

- **USB_INTLPM_ERROR** a bus error occurred in the transmission of an LPM transaction.
- **USB_INTLPM_RESUME** the USB controller has resumed from LPM low power state.
- **USB_INTLPM_INCOMPLETE** the LPM transaction failed because a timeout occurred or there were bit errors in the response for three attempts.
- **USB_INTLPM_ACK** the device has acknowledged an LPM transaction.
- **USB_INTLPM_NYET** the device has responded with a NYET to an LPM transaction.
- **USB_INTLPM_STALL** the device has stalled an LPM transaction.

The valid interrupt status bits when the USB controller is acting as a device are the following:

- **USB_INTLPM_ERROR** an LPM transaction was received that has an unsupported link state field. The transaction was stalled, but the requested link state can still be read using the [USBLPMLinkStateGet\(\)](#) function.
- **USB_INTLPM_RESUME** the USB controller has resumed from the LPM low power state.
- **USB_INTLPM_INCOMPLETE** the USB controller responded to an LPM transaction with a NYET because data was still in the transmit FIFOs.
- **USB_INTLPM_ACK** the USB controller acknowledged an LPM transaction and is now in the LPM suspend mode.
- **USB_INTLPM_NYET** the USB controller responded to an LPM transaction with a NYET because LPM transactions are not yet enabled by a call to [USBDevLPMEnable\(\)](#).
- **USB_INTLPM_STALL** the USB controller has stalled an incoming LPM transaction.

Example: Enable all LPM interrupt sources.

```
//
// Enable all LPM interrupt sources.
//
USBLPMIntEnable(USBO_BASE,  USB_INTLPM_ERROR | USB_INTLPM_RESUME |
                  USB_INTLPM_INCOMPLETE | USB_INTLPM_ACK |
                  USB_INTLPM_NYET | USB_INTLPM_STALL);
```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.5.4.11 USBLPMIntStatus

Returns the current LPM interrupt status.

Prototype:

```
uint32_t  
USBLPMIntStatus(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current LPM interrupt status for the USB controller.

The valid interrupt status bits when the USB controller is acting as a host are the following:

- **USB_INTLPM_ERROR** a bus error occurred in the transmission of an LPM transaction.
- **USB_INTLPM_RESUME** the USB controller has resumed from the LPM low power state.
- **USB_INTLPM_INCOMPLETE** the LPM transaction failed because a timeout occurred or there were bit errors in the response for three attempts.
- **USB_INTLPM_ACK** the device has acknowledged an LPM transaction.
- **USB_INTLPM_NYET** the device has responded with a NYET to an LPM transaction.
- **USB_INTLPM_STALL** the device has stalled an LPM transaction.

The valid interrupt status bits when the USB controller is acting as a device are the following:

- **USB_INTLPM_ERROR** an LPM transaction was received that has an unsupported link state field. The transaction was stalled, but the requested link state can still be read using the [USBLPMLinkStateGet\(\)](#) function.
- **USB_INTLPM_RESUME** the USB controller has resumed from the LPM low power state.
- **USB_INTLPM_INCOMPLETE** the USB controller responded to an LPM transaction with a NYET because data was still in the transmit FIFOs.
- **USB_INTLPM_ACK** the USB controller acknowledged an LPM transaction and is now in the LPM suspend mode.
- **USB_INTLPM_NYET** the USB controller responded to an LPM transaction with a NYET because LPM transactions are not yet enabled by a call to [USBDevLPMEnable\(\)](#).
- **USB_INTLPM_STALL** the USB controller has stalled an incoming LPM transaction.

Note:

This call clears the source of all LPM status interrupts, so the caller must take care to save the value returned because a subsequent call to [USBLPMIntStatus\(\)](#) does not return the previous value.

Example: Get the current LPM interrupt status.

```
uint32_t ui32LPMIntStatus;  
  
//  
// Get the current LPM interrupt status.  
//  
ui32LPMIntStatus = USBLPMIntStatus(USB0_BASE);  
  
//  
// Check if an LPM transaction was acknowledged.  
//
```

```

if(ui32LPMIntStatus & USB_INTLPM_ACK)
{
    //
    // Handle entering LPM suspend mode.
    //
    ...
}

```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

The current LPM interrupt status.

33.5.4.12 USBLPMLinkStateGet

Returns the current link state setting.

Prototype:

```

uint32_t
USBLPMLinkStateGet(uint32_t ui32Base)

```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current link state setting for the USB controller. When the controller is operating as a host, this link state is sent with an LPM request. When the controller is acting as a device, this link state was received by the last LPM transaction whether it was acknowledged or stalled because the requested LPM mode is not supported.

Example: Get the link state for the last LPM transaction.

```

uint32_t ui32LinkState;

//
// Get the endpoint number that received the LPM request.
//
ui32LinkState = USBLPMLinkStateGet(USB0_BASE);

//
// Check if this was a supported link state.
//
if(ui32LinkState == USB_HOST_LPM_L1)
{
    //
    // Handle the supported L1 link state.
    //
}
else
{
    //
    // Handle the unsupported link state.
    //
}

```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

The current LPM link state.

33.5.4.13 USBLPMRemoteWakeEnabled

Returns if remote wake is currently enabled.

Prototype:

```
bool  
USBLPMRemoteWakeEnabled(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current state of the remote wake setting for host or device mode operation. If the controller is acting as a host this returns the current setting that is sent to devices when LPM requests are sent to a device. If the controller is in device mode, this function returns the state of the last LPM request sent from the host and indicates if the host enabled remote wakeup.

Example: Issue remote wake if remote wake is enabled.

```
if (USBLPMRemoteWakeEnabled(USB0_BASE))  
{  
    USBDevLPMRemoteWake(USB0_BASE);  
}
```

Note:

The USB LPM feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

The **true** if remote wake is enabled or **false** if it is not.

33.6 USB UTMI Low Pin Interface (ULPI)

Functions

- void [USBULPIConfig](#) (uint32_t ui32Base, uint32_t ui32Config)
- void [USBULPIDisable](#) (uint32_t ui32Base)
- void [USBULPIEnable](#) (uint32_t ui32Base)
- uint8_t [USBULPIRegRead](#) (uint32_t ui32Base, uint8_t ui8Reg)
- void [USBULPIRegWrite](#) (uint32_t ui32Base, uint8_t ui8Reg, uint8_t ui8Data)

33.6.1 Detailed Description

Some Tiva USB controllers include support for connecting an external USB physical (PHY) interface that adds support for USB high speed operation when the internal PHY's full speed operation

does not provide the bandwidth needed by an application. The USB controller supports the USB 2.0 Transceiver Macrocell Interface (UTMI) Low Pin Interface (ULPI) contained in the USB 2.0 specification. The configuration options for the ULPI interface are set by calling the [USBULPIConfig\(\)](#) function. The options for configuring the ULPI interface include using the external PHY for VBUS detection by specifying **USB_ULPI_EXTVBUS** and enabling external VBUS over-current detection by specifying **USB_ULPI_EXTVBUS_IND**. The ULPI interface is not enabled by default so the application must call the [USBULPIEnable\(\)](#) function and can call the [USBULPIDisable\(\)](#) function if it must disable the ULPI interface. Normal operation does not require direct access to the external PHY, but if necessary the [USBULPIRegRead\(\)](#) and [USBULPIRegWrite\(\)](#) functions provide direct access to ULPI PHY registers.

Example: Configuring and Enabling a ULPI connected USB PHY.

```
//  
// Enable external VBUS and over-current detection and enable the  
// ULPI interface.  
//  
USBULPIConfig(USB0_BASE,  USB_ULPI_EXTVBUS | USB_ULPI_EXTVBUS_IND);  
USBULPIEnable(USB0_BASE);
```

33.6.2 Function Documentation

33.6.2.1 USBULPIConfig

Configures the USB controller's ULPI function.

Prototype:

```
void  
USBULPIConfig(uint32_t ui32Base,  
              uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Config contains the configuration options.

Description:

This function is used to configure the USB controller's ULPI function. The configuration options are set in the *ui32Config* parameter and are a logical OR of the following values:

- **USB_ULPI_EXTVBUS** enables the external ULPI PHY as the source for VBUS signaling.
- **USB_ULPI_EXTVBUS_IND** enables the external ULPI PHY to detect external VBUS over-current condition.

Example: Enable ULPI PHY with full VBUS control.

```
//  
// Enable ULPI PHY with full VBUS control.  
//  
USBULPIConfig(USB0_BASE,  USB_ULPI_EXTVBUS | USB_ULPI_EXTVBUS_IND);
```

Note:

The USB ULPI feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.6.2.2 USBULPIDisable

Disables the USB controller's ULPI function.

Prototype:

```
void  
USBULPIDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables the USB controller's ULPI function. Accesses to the external ULPI-connected PHY cannot succeed after this function has been called.

Example: Disable ULPI function.

```
//  
// Disable ULPI function.  
//  
USBULPIDisable(USB0_BASE);
```

Note:

The USB ULPI feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.6.2.3 USBULPIEnable

Enables the USB controller's ULPI function.

Prototype:

```
void  
USBULPIEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function enables the USB controller's ULPI function and must be called before attempting to access an external ULPI-connected USB PHY.

Example: Enable ULPI function.

```
//  
// Enable ULPI function.  
//  
USBULPIEnable(USB0_BASE);
```

Note:

The USB ULPI feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.6.2.4 USBULPIRegRead

Reads a register from an external ULPI-connected USB PHY.

Prototype:

```
uint8_t
USBULPIRegRead(uint32_t ui32Base,
                uint8_t ui8Reg)
```

Parameters:

ui32Base specifies the USB module base address.
ui8Reg specifies the register address to read.

Description:

This function reads the register address specified in the **ui8Reg** parameter using the ULPI function. This function is blocking and only returns when the read access completes. The function does not return if there is not a ULPI-connected USB PHY present.

Example: Read a register from the ULPI PHY.

```
uint8_t ui8Value;

//
// Read a register from the ULPI PHY register at 0x10.
//
ui8Value = USBULPIRegRead(USBO_BASE, 0x10);
```

Note:

The USB ULPI feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

The value of the requested ULPI register.

33.6.2.5 USBULPIRegWrite

Writes a value to a register on an external ULPI-connected USB PHY.

Prototype:

```
void
USBULPIRegWrite(uint32_t ui32Base,
                 uint8_t ui8Reg,
                 uint8_t ui8Data)
```

Parameters:

ui32Base specifies the USB module base address.

ui8Reg specifies the register address to write.
ui8Data specifies the data to write.

Description:

This function writes the register address specified in the *ui8Reg* parameter with the value specified in the *ui8Data* parameter using the ULPI function. This function is blocking and only returns when the write access completes. The function does not return if there is not a ULPI-connected USB PHY present.

Example: Write a register from the external ULPI PHY.

```
//  
// Write the ULPI PHY register at 0x10 with 0x20.  
//  
USBULPIRegWrite(USB0_BASE, 0x10, 0x20);
```

Note:

The USB ULPI feature is not available on all Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine if this feature is available.

Returns:

None.

33.7 Programming Example

This example code makes the calls necessary to configure endpoint 1, in device mode, as a bulk IN endpoint. The first call configures endpoint 1 to have a maximum packet size of 64 bytes and makes it a bulk IN endpoint. The call to **USBIFConfig()** configures the starting address to 64 bytes in and 64 bytes long. It also specifies **USB_EP_DEV_IN** to indicate a device mode IN endpoint. The next two calls demonstrate how to fill the data FIFO for this endpoint and then have it scheduled for transmission on the USB bus. The **USBEndpointDataPut()** call puts data into the FIFO but does not actually start the data transmission. The **USBEndpointDataSend()** call schedules the transmission to go out the next time the host controller requests data on this endpoint.

```
//  
// Configure Endpoint 1.  
//  
USBDevEndpointConfigSet(USB0_BASE, USB_EP_1, 64, DISABLE_NAK_LIMIT,  
                      USB_EP_MODE_BULK | USB_EP_DEV_IN);  
  
//  
// Configure FIFO as a device IN endpoint FIFO starting at address 64  
// and is 64 bytes in size.  
//  
USBIFConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_64, USB_EP_DEV_IN);  
  
...  
  
//  
// Put the data in the FIFO.  
//  
USBEndpointDataPut(USB0_BASE, USB_EP_1, pucData, 64);  
  
//  
// Start the transmission of data.  
//  
USBEndpointDataSend(USB0_BASE, USB_EP_1, USB_TRANS_IN);
```

34 Watchdog Timer

Introduction	689
API Functions	689
Programming Example	698

34.1 Introduction

The Watchdog Timer API provides a set of functions for using the Tiva watchdog timer modules. Functions are provided to deal with the watchdog timer interrupts, and to handle status and configuration of the watchdog timer.

A watchdog timer module's function is to prevent system hangs. The watchdog timer module consists of a 32-bit down counter, a programmable load register, interrupt generation logic, and a locking register. Once the watchdog timer has been configured, the lock register can be written to prevent the timer configuration from being inadvertently altered.

The watchdog timer can be configured to generate an interrupt to the processor after its first timeout, and to generate a reset signal after its second timeout. The watchdog timer module generates the first timeout signal when the 32-bit counter reaches the zero state after being enabled; enabling the counter also enables the watchdog timer interrupt. After the first timeout event, the 32-bit counter is reloaded with the value of the watchdog timer load register, and the timer resumes counting down from that value. If the timer counts down to its zero state again before the first timeout interrupt is cleared, and the reset signal has been enabled, the watchdog timer asserts its reset signal to the system. If the interrupt is cleared before the 32-bit counter reaches its second timeout, the 32-bit counter is loaded with the value in the load register, and counting resumes from that value. If the load register is written with a new value while the watchdog timer counter is counting, then the counter is loaded with the new value and continues counting.

On some parts, there are two watchdog timers: one that is clocked by the system clock and a second that is clocked by PIOSC.

On some parts, the watchdog timer can be configured to generate an NMI instead of a standard interrupt. If the watchdog timer has been configured to generate an NMI, the interrupt is still treated the same as if it were a standard interrupt; it must be enabled in order to be triggered, and it must be cleared inside the NMI handler.

This driver is contained in `driverlib/watchdog.c`, with `driverlib/watchdog.h` containing the API declarations for use by applications.

34.2 API Functions

Functions

- void [WatchdogEnable](#) (uint32_t ui32Base)
- void [WatchdogIntClear](#) (uint32_t ui32Base)
- void [WatchdogIntEnable](#) (uint32_t ui32Base)
- void [WatchdogIntRegister](#) (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t [WatchdogIntStatus](#) (uint32_t ui32Base, bool bMasked)

- void [WatchdogIntTypeSet](#) (uint32_t ui32Base, uint32_t ui32Type)
- void [WatchdogIntUnregister](#) (uint32_t ui32Base)
- void [WatchdogLock](#) (uint32_t ui32Base)
- bool [WatchdogLockState](#) (uint32_t ui32Base)
- uint32_t [WatchdogReloadGet](#) (uint32_t ui32Base)
- void [WatchdogReloadSet](#) (uint32_t ui32Base, uint32_t ui32LoadVal)
- void [WatchdogResetDisable](#) (uint32_t ui32Base)
- void [WatchdogResetEnable](#) (uint32_t ui32Base)
- bool [WatchdogRunning](#) (uint32_t ui32Base)
- void [WatchdogStallDisable](#) (uint32_t ui32Base)
- void [WatchdogStallEnable](#) (uint32_t ui32Base)
- void [WatchdogUnlock](#) (uint32_t ui32Base)
- uint32_t [WatchdogValueGet](#) (uint32_t ui32Base)

34.2.1 Detailed Description

The Watchdog Timer API is broken into two groups of functions: those that deal with interrupts, and those that handle status and configuration.

The Watchdog Timer interrupts are handled by the [WatchdogIntRegister\(\)](#), [WatchdogIntUnregister\(\)](#), [WatchdogIntEnable\(\)](#), [WatchdogIntClear\(\)](#), and [WatchdogIntStatus\(\)](#) functions.

Status and configuration functions for the Watchdog Timer module are [WatchdogEnable\(\)](#), [WatchdogRunning\(\)](#), [WatchdogLock\(\)](#), [WatchdogUnlock\(\)](#), [WatchdogLockState\(\)](#), [WatchdogReloadSet\(\)](#), [WatchdogReloadGet\(\)](#), [WatchdogValueGet\(\)](#), [WatchdogResetEnable\(\)](#), [WatchdogResetDisable\(\)](#), [WatchdogStallEnable\(\)](#), and [WatchdogStallDisable\(\)](#).

34.2.2 Function Documentation

34.2.2.1 WatchdogEnable

Enables the watchdog timer.

Prototype:

```
void  
WatchdogEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function enables the watchdog timer counter and interrupt.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.2 WatchdogIntClear

Clears the watchdog timer interrupt.

Prototype:

```
void  
WatchdogIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

The watchdog timer interrupt source is cleared, so that it no longer asserts.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted). This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.3 WatchdogIntEnable

Enables the watchdog timer interrupt.

Prototype:

```
void  
WatchdogIntEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function enables the watchdog timer interrupt.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.4 WatchdogIntRegister

Registers an interrupt handler for the watchdog timer interrupt.

Prototype:

```
void  
WatchdogIntRegister(uint32_t ui32Base,  
                     void (*pfnHandler) (void))
```

Parameters:

ui32Base is the base address of the watchdog timer module.

pfnHandler is a pointer to the function to be called when the watchdog timer interrupt occurs.

Description:

This function does the actual registering of the interrupt handler. This function also enables the global interrupt in the interrupt controller; the watchdog timer interrupt must be enabled via [WatchdogEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [WatchdogIntClear\(\)](#).

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Note:

For parts with a watchdog timer module that has the ability to generate an NMI instead of a standard interrupt, this function registers the standard watchdog interrupt handler. To register the NMI watchdog handler, use [IntRegister\(\)](#) to register the handler for the **FAULT_NMI** interrupt.

Returns:

None.

34.2.2.5 WatchdogIntStatus

Gets the current watchdog timer interrupt status.

Prototype:

```
uint32_t  
WatchdogIntStatus(uint32_t ui32Base,  
                  bool bMasked)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This function returns the interrupt status for the watchdog timer module. Either the raw interrupt status or the status of interrupt that is allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, where a 1 indicates that the watchdog interrupt is active, and a 0 indicates that it is not active.

34.2.2.6 WatchdogIntTypeSet

Sets the type of interrupt generated by the watchdog.

Prototype:

```
void  
WatchdogIntTypeSet(uint32_t ui32Base,  
                    uint32_t ui32Type)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

ui32Type is the type of interrupt to generate.

Description:

This function sets the type of interrupt that is generated if the watchdog timer expires. *ui32Type* can be either **WATCHDOG_INT_TYPE_INT** to generate a standard interrupt (the default) or **WATCHDOG_INT_TYPE_NMI** to generate a non-maskable interrupt (NMI).

When configured to generate an NMI, the watchdog interrupt must still be enabled with [WatchdogIntEnable\(\)](#), and it must still be cleared inside the NMI handler with [WatchdogIntClear\(\)](#).

Note:

The ability to select an NMI interrupt varies with the Tiva part in use. Please consult the datasheet for the part you are using to determine whether this support is available. This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.7 WatchdogIntUnregister

Unregisters an interrupt handler for the watchdog timer interrupt.

Prototype:

```
void  
WatchdogIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function does the actual unregistering of the interrupt handler. This function clears the handler to be called when a watchdog timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

[IntRegister\(\)](#) for important information about registering interrupt handlers.

Note:

For parts with a watchdog timer module that has the ability to generate an NMI instead of a standard interrupt, this function unregisters the standard watchdog interrupt handler. To unregister the NMI watchdog handler, use [IntUnregister\(\)](#) to unregister the handler for the **FAULT_NMI** interrupt.

Returns:

None.

34.2.2.8 WatchdogLock

Enables the watchdog timer lock mechanism.

Prototype:

```
void  
WatchdogLock(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function locks out write access to the watchdog timer registers.

Returns:

None.

34.2.2.9 WatchdogLockState

Gets the state of the watchdog timer lock mechanism.

Prototype:

```
bool  
WatchdogLockState(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function returns the lock state of the watchdog timer registers.

Returns:

Returns **true** if the watchdog timer registers are locked, and **false** if they are not locked.

34.2.2.10 WatchdogReloadGet

Gets the watchdog timer reload value.

Prototype:

```
uint32_t  
WatchdogReloadGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function gets the value that is loaded into the watchdog timer when the count reaches zero for the first time.

Returns:

None.

34.2.2.11 WatchdogReloadSet

Sets the watchdog timer reload value.

Prototype:

```
void  
WatchdogReloadSet(uint32_t ui32Base,  
                  uint32_t ui32LoadVal)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

ui32LoadVal is the load value for the watchdog timer.

Description:

This function configures the value to load into the watchdog timer when the count reaches zero for the first time; if the watchdog timer is running when this function is called, then the value is immediately loaded into the watchdog timer counter. If the *ui32LoadVal* parameter is 0, then an interrupt is immediately generated.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.12 WatchdogResetDisable

Disables the watchdog timer reset.

Prototype:

```
void  
WatchdogResetDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function disables the capability of the watchdog timer to issue a reset to the processor after a second timeout condition.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.13 WatchdogResetEnable

Enables the watchdog timer reset.

Prototype:

```
void  
WatchdogResetEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function enables the capability of the watchdog timer to issue a reset to the processor after a second timeout condition.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.14 WatchdogRunning

Determines if the watchdog timer is enabled.

Prototype:

```
bool  
WatchdogRunning(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function checks to see if the watchdog timer is enabled.

Returns:

Returns **true** if the watchdog timer is enabled and **false** if it is not.

34.2.2.15 WatchdogStallDisable

Disables stalling of the watchdog timer during debug events.

Prototype:

```
void  
WatchdogStallDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function disables the debug mode stall of the watchdog timer. By doing so, the watchdog timer continues to count regardless of the processor debug state.

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.16 WatchdogStallEnable

Enables stalling of the watchdog timer during debug events.

Prototype:

```
void  
WatchdogStallEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function allows the watchdog timer to stop counting when the processor is stopped by the debugger. By doing so, the watchdog is prevented from expiring (typically almost immediately from a human time perspective) and resetting the system (if reset is enabled). The watchdog instead expires after the appropriate number of processor cycles have been executed while debugging (or at the appropriate time after the processor has been restarted).

Note:

This function has no effect if the watchdog timer has been locked.

Returns:

None.

34.2.2.17 WatchdogUnlock

Disables the watchdog timer lock mechanism.

Prototype:

```
void  
WatchdogUnlock(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function enables write access to the watchdog timer registers.

Returns:

None.

34.2.2.18 WatchdogValueGet

Gets the current watchdog timer value.

Prototype:

```
uint32_t  
WatchdogValueGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the watchdog timer module.

Description:

This function reads the current value of the watchdog timer.

Returns:

Returns the current value of the watchdog timer.

34.3 Programming Example

The following example shows how to set up the watchdog timer API to reset the processor after two timeouts.

```
//  
// Check to see if the registers are locked, and if so, unlock them.  
//  
if(WatchdogLockState(WATCHDOG0_BASE) == true)  
{  
    WatchdogUnlock(WATCHDOG0_BASE);  
}  
  
//  
// Initialize the watchdog timer.  
//  
WatchdogReloadSet(WATCHDOG0_BASE, 0xFFFFE);  
  
//  
// Enable the reset.  
//  
WatchdogResetEnable(WATCHDOG0_BASE);  
  
//  
// Enable the watchdog timer.  
//  
WatchdogEnable(WATCHDOG0_BASE);  
  
//  
// Wait for the reset to occur.  
//  
while(1)  
{  
}
```

35 Using the ROM

Introduction	699
Direct ROM Calls	699
Mapped ROM Calls	700
Firmware Update	701

35.1 Introduction

Many Tiva devices have portions of the peripheral driver library stored in an on-chip ROM. By using the code in the on-chip ROM, more flash is available for use by the application. The boot loader is also contained within the ROM, which can be called by an application in order to start a firmware update.

35.2 Direct ROM Calls

In order to call the ROM, the following steps must be performed:

- The device on which the application is run must be defined using a preprocessor symbol, which can be done either within the source code or in the project that builds the application. The latter is more flexible if code is shared between projects.
- `driverlib/rom.h` is included by the source code desiring to call the ROM.
- The ROM version of a peripheral driver library function is called. For example, if `GPIODirModeSet()` is to be called in the ROM, `ROM_GPIODirModeSet()` is used instead.

A define is used to select the device being used because the set of functions available in the ROM must be a compile-time decision; checking at run-time does not provide any flash savings because both the ROM call and the flash version of the API would be in the application flash image.

The following defines are recognized by `driverlib/rom.h`:

`TARGET_IS_TM4C123_RA1` The application is being built to run on a TM4C123 devices, silicon revision A1.

`TARGET_IS_TM4C123_RA3` The application is being built to run on a TM4C123 devices, silicon revision A3.

`TARGET_IS_TM4C123_RB1` The application is being built to run on a TM4C123 devices, silicon revision B1.

`TARGET_IS_TM4C129_RA0` The application is being built to run on a TM4C129 devices, silicon revision A0.

`TARGET_IS_TM4C129_RA1` The application is being built to run on a TM4C129 devices, silicon revision A1.

Note:

The **TARGET_IS_*** macros can also control the mapping of interrupt names to interrupt numbers. See the [Interrupt Mapping](#) section of this document for more details on how these defines are used to determine interrupt mapping.

By using ROM_Function(), the ROM is explicitly called. If the function in question is not available in the ROM, a compiler error is produced.

See the *TivaWare ROM User's Guide* for the specific device for details of the APIs available in the ROM.

The following is an example of calling a function in the ROM, defining the device in question using a #define in the source instead of in the project file:

```
#define TARGET_IS_TM4C123_RA1
#include "driverlib/rom.h"
#include "driverlib/systick.h"

int
main(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();

    // ...
}
```

35.3 Mapped ROM Calls

When code is intended to be shared between projects, and some of the projects run on devices with a ROM and some run on devices without a ROM, it is convenient to have the code automatically call the ROM or the flash version of the API without having `#ifdef`s in the code. `rom_map.h` provides an automatic mapping feature for accessing the ROM. Similar to the ROM_Function() APIs provided by `rom.h`, a set of MAP_Function() APIs are provided. If the function is available in ROM, MAP_Function() simply calls ROM_Function(); otherwise it calls Function().

In order to use the mapped ROM calls, the following steps must be performed:

- Follow the above steps for including and using `driverlib/rom.h`.
- Include `driverlib/rom_map.h`.
- Continuing the above example, call `MAP_GPIODirModeSet()` in the source code.

As in the direct ROM call method, the choice of calling ROM versus the flash version is made at compile-time. The only APIs that are provided via the ROM mapping feature are ones that are available in the ROM, which is not every API available in the peripheral driver library.

The following is an example of calling a function in shared code, where the device in question is defined in the project file:

```
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/systick.h"

void
```

```
SetupSysTick (void)
{
    MAP_SysTickPeriodSet (0x1000);
    Map_SysTickEnable();
}
```

When built for a device that does not have a ROM, this example is equivalent to:

```
#include "driverlib/systick.h"

void
SetupSysTick (void)
{
    SysTickPeriodSet (0x1000);
    SysTickEnable();
}
```

When built for a device that has a ROM, however, this example is equivalent to:

```
#include "driverlib/rom.h"
#include "driverlib/systick.h"

void
SetupSysTick (void)
{
    ROM_SysTickPeriodSet (0x1000);
    ROM_SysTickEnable();
}
```

35.4 Firmware Update

Functions

- void [ROM_UpdateI2C](#) (void)
- void [ROM_UpdateSSI](#) (void)
- void [ROM_UpdateUART](#) (void)
- void [ROM_UpdateUSB](#) (uint8_t *pui8USBBootROMInfo)

35.4.1 Detailed Description

There are a set of APIs in the ROM for restarting the boot loader in order to commence a firmware update. Multiple calls are provided because each selects a particular interface to be used for the update process, bypassing the interface selection step of the normal boot loader (including the auto-bauding in the UART interface).

See the *TivaWare ROM User's Guide* for the specific device for details of the firmware update APIs in the ROM.

35.4.2 Function Documentation

35.4.2.1 ROM_UpdateI2C

Starts an update over the I2C0 interface.

Prototype:

```
void  
ROM_UpdateI2C(void)
```

Description:

Calling this function commences an update of the firmware via the I2C0 interface. This function assumes that the I2C0 interface has already been configured and is currently operational. The I2C0 slave is used for data transfer, and the I2C0 master is used to monitor bus busy conditions (therefore, both must be enabled).

Returns:

Never returns.

35.4.2.2 ROM_UpdateSSI

Starts an update over the SSI0 interface.

Prototype:

```
void  
ROM_UpdateSSI(void)
```

Description:

Calling this function commences an update of the firmware via the SSI0 interface. This function assumes that the SSI0 interface has already been configured and is currently operational.

Returns:

Never returns.

35.4.2.3 ROM_UpdateUART

Starts an update over the UART0 interface.

Prototype:

```
void  
ROM_UpdateUART(void)
```

Description:

Calling this function commences an update of the firmware via the UART0 interface. This function assumes that the UART0 interface has already been configured and is currently operational.

Returns:

Never returns.

35.4.2.4 ROM_UpdateUSB

Starts an update over the USB interface.

Prototype:

```
void
ROM_UpdateUSB(uint8_t *pui8USBBBootROMInfo)
```

Parameters:

pui8USBBBootROMInfo is the optional override for some of the USB device and configuration descriptor values.

Description:

Calling this function commences an update of the firmware via the USB interface. This function assumes that the USB interface has already been configured and the device is being clocked by the PLL. By using specifying a non-zero *pui8USBBBootROMInfo* value, the vendor ID, product ID, bus-versus self-powered, maximum power, device version, and USB strings can be customized.

pui8USBBBootROMInfo[0] is the LSB of the Vendor ID.

pui8USBBBootROMInfo[1] is the MSB of the Vendor ID.

pui8USBBBootROMInfo[2] is the LSB of the Product ID.

pui8USBBBootROMInfo[3] is the MSB of the Product ID.

pui8USBBBootROMInfo[4] is the LSB of the Device release number(BCD).

pui8USBBBootROMInfo[5] is the MSB of the Device release number(BCD).

pui8USBBBootROMInfo[6] is the USB power flags for the device, like `USB_CONF_ATTR_SELF_PWR`.

pui8USBBBootROMInfo[7] is the maximum power for the device in 2mA increments.

pui8USBBBootROMInfo[8] is the start of the custom string descriptors.

Example: Custom Descriptor

```
//
// Default string descriptors.
//
const uint8_t g_pcCustomDescriptor[] =
{
    //
    // VID (0xbbaa)
    //
    0xaa, 0xbb,

    //
    // PID (0xddcc)
    //
    0xcc, 0xdd,

    //
    // Device Release(BCD 0x0120)
    //
    0x20, 0x01,

    //
    // Power configuration(Bus Powered).
    //
}
```

```
USB_CONF_ATTR_BUS_PWR,  
  
//  
// Power in 2mA increments(250mA).  
//  
250/2,  
  
//  
// Start of string descriptor.  
//  
2 + (1 * 2),  
USB_DTYPE_STRING,  
USBShort(USB_LANG_EN_US),  
  
//  
// Texas Instruments Incorporated  
//  
2 + (30 * 2),  
USB_DTYPE_STRING,  
'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,  
't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0, ' ', 0,  
'I', 0, 'n', 0, 'c', 0, 'o', 0, 'r', 0, 'p', 0, 'o', 0, 'r', 0, 'a', 0,  
't', 0, 'e', 0, 'd', 0,  
  
//  
// Stellaris Device Firmware Update Serial  
//  
2 + (27 * 2),  
USB_DTYPE_STRING,  
'T', 0, 'i', 0, 'v', 0, 'a', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0,  
'c', 0, 'e', 0, ' ', 0, 'F', 0, 'i', 0, 'r', 0, 'm', 0, 'w', 0, 'a', 0,  
'r', 0, 'e', 0, ' ', 0, 'U', 0, 'p', 0, 'd', 0, 'a', 0, 't', 0, 'e', 0,  
  
//  
// 00000000  
//  
2 + (8 * 2) ,  
USB_DTYPE_STRING,  
'0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '0', 0  
};
```

Returns:

Never returns.

36 Error Handling

Invalid arguments and error conditions are handled in a non-traditional manner in the peripheral driver library. Typically, a function would check its arguments to make sure that they are valid (if required; some may be unconditionally valid such as a 32-bit value used as the load value for a 32-bit timer). If an invalid argument is provided, an error code would be returned. The caller then has to check the return code from each invocation of the function to make sure that it succeeded.

This method results in a sizable amount of argument-checking code in each function and return-code-checking code at each call site. For a self-contained application, this extra code becomes an unneeded burden once the application is debugged. Having a means of removing it allows the final code to be smaller and therefore run faster.

In the peripheral driver library, most functions do not return errors ([FlashProgram\(\)](#), [FlashErase\(\)](#), [FlashProtectSet\(\)](#), and [FlashProtectSave\(\)](#) are the notable exceptions). Argument checking is done via a call to the `ASSERT` macro (provided in `driverlib/debug.h`). This macro has the usual definition of an assert macro; it takes an expression that “must” be true. By making this macro be empty, the argument checking is removed from the code.

There are two definitions of the `ASSERT` macro provided in `driverlib/debug.h`; one that is empty (used for normal situations) and one that evaluates the expression (used when the library is built with debugging). The debug version calls the `_error_` function whenever the expression is not true, passing the file name and line number of the `ASSERT` macro invocation. The `_error_` function is prototyped in `driverlib/debug.h` and must be provided by the application because it is the application’s responsibility to deal with error conditions.

By setting a breakpoint on the `_error_` function, the debugger immediately stops whenever an error occurs anywhere in the application (something that would be very difficult to do with other error checking methods). When the debugger stops, the arguments to the `_error_` function and the backtrace of the stack pinpoint the function that found an error, what it found to be a problem, and where it was called from. As an example:

```
void
UARTParityModeSet(uint32_t ui32Base, uint32_t ui32Parity)
{
    //
    // Check the arguments.
    //
    ASSERT((ui32Base == UART0_BASE) || (ui32Base == UART1_BASE) ||
           (ui32Base == UART2_BASE));
    ASSERT((ui32Parity == UART_CONFIG_PAR_NONE) ||
           (ui32Parity == UART_CONFIG_PAR_EVEN) ||
           (ui32Parity == UART_CONFIG_PAR_ODD) ||
           (ui32Parity == UART_CONFIG_PAR_ONE) ||
           (ui32Parity == UART_CONFIG_PAR_ZERO));
```

Each argument is individually checked, so the line number of the failing `ASSERT` indicates the argument that is invalid. The debugger is able to display the values of the arguments (from the stack backtrace) as well as the caller of the function that had the argument error. This method allows the problem to be quickly identified at the cost of a small amount of code.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
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
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com