TivaWare™ Peripheral Driver Library

USER'S GUIDE

SW-DRL-UG-1.0

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 1.0 of this document, last updated on April 11, 2013.

Table of Contents

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™

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

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_CR0_R` is used to access the `CR0` 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_CR0_SCR_M` and `SSI_CR0_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_CR0_FRF_M` macro defines the bit field, and the `SSI_CR0_FRF_NMW`, `SSI_CR0_FRF_TI`, and `SSI_CR0_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

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 definitions 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 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 fucntion 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)

4.1 Introduction

The analog to digital converter (ADC) API provides a set of functions for dealing with 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 definitions for use by applications.

4.2 API Functions

Functions

- 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 bInterrupt)
- void ADCHardwareOversampleConfigure (uint32_t ui32Base, uint32_t ui32Factor)
- void ADCIntClear (uint32_t ui32Base, uint32_t ui32SequenceNum)
- void ADCIntDisable (uint32_t ui32Base, uint32_t ui32SequenceNum)
- void ADCIntEnable (uint32_t ui32Base, uint32_t ui32SequenceNum)
- void ADCIntRegister (uint32_t ui32Base, uint32_t ui32SequenceNum, void (*pfnHandler)(void))
- uint32_t ADCIntStatus (uint32_t ui32Base, uint32_t ui32SequenceNum, 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 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 *pui32Buffer, 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 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.

4.2.2.2 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.

4.2.2.3 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.

4.2.2.4 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.

4.2.2.5 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.

4.2.2.6 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.

4.2.2.7 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.

4.2.2.8 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.

4.2.2.9 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.

4.2.2.10 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.

4.2.2.11 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.

4.2.2.12 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.

4.2.2.13 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.

4.2.2.14 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.

4.2.2.15 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**.

4.2.2.16 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.

4.2.2.17 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.

4.2.2.18 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.

4.2.2.19 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.

4.2.2.20 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).

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.

4.2.2.21 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.

4.2.2.22 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 should be disabled before it is configured.

Returns:
None.

4.2.2.23 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.

4.2.2.24 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.

4.2.2.25 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.

4.2.2.26 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.

4.2.2.27 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.

4.2.2.28 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.

4.2.2.29 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.

4.2.2.30 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.

4.2.2.31 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.
//
ADCProcessorTrigger(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 Controller Area Network (CAN)

5.1 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 definitions for use by applications.

5.2 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)
- void 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)

5.2.1 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 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 the 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 contains 32 message objects that 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 preloaded 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

5.2.2 Data Structure Documentation

5.2.2.1 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.

5.2.2.2 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.

5.2.3 Define Documentation

5.2.3.1 CAN_INT_ERROR

Definition:

```
#define CAN_INT_ERROR
```

Description:

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

5.2.3.2 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 will be generated by the CAN controller.

5.2.3.3 CAN_INT_STATUS

Definition:

```
#define CAN_INT_STATUS
```

Description:

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

5.2.3.4 CAN_STATUS_BUS_OFF

Definition:

```
#define CAN_STATUS_BUS_OFF
```

Description:

CAN controller has entered a Bus Off state.

5.2.3.5 CAN_STATUS_EPASS

Definition:

```
#define CAN_STATUS_EPASS
```

Description:

CAN controller error level has reached error passive level.

5.2.3.6 CAN_STATUS_EWARN

Definition:

```
#define CAN_STATUS_EWARN
```

Description:

CAN controller error level has reached warning level.

5.2.3.7 CAN_STATUS_LEC_ACK

Definition:

```
#define CAN_STATUS_LEC_ACK
```

Description:

An acknowledge error has occurred.

5.2.3.8 CAN_STATUS_LEC_BIT0

Definition:

```
#define CAN_STATUS_LEC_BIT0
```

Description:

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

5.2.3.9 CAN_STATUS_LEC_BIT1

Definition:

```
#define CAN_STATUS_LEC_BIT1
```

Description:

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

5.2.3.10 CAN_STATUS_LEC_CRC

Definition:

```
#define CAN_STATUS_LEC_CRC
```

Description:

A CRC error has occurred.

5.2.3.11 CAN_STATUS_LEC_FORM

Definition:

```
#define CAN_STATUS_LEC_FORM
```

Description:

A formatting error has occurred.

5.2.3.12 CAN_STATUS_LEC_MASK

Definition:

```
#define CAN_STATUS_LEC_MASK
```

Description:

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

5.2.3.13 CAN_STATUS_LEC_MSK

Definition:

```
#define CAN_STATUS_LEC_MSK
```

Description:

This is the mask for the last error code field.

5.2.3.14 CAN_STATUS_LEC_NONE

Definition:

```
#define CAN_STATUS_LEC_NONE
```

Description:

There was no error.

5.2.3.15 CAN_STATUS_LEC_STUFF

Definition:

```
#define CAN_STATUS_LEC_STUFF
```

Description:

A bit stuffing error has occurred.

5.2.3.16 CAN_STATUS_RXOK

Definition:

```
#define CAN_STATUS_RXOK
```

Description:

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

5.2.3.17 CAN_STATUS_TXOK

Definition:

```
#define CAN_STATUS_TXOK
```

Description:

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

5.2.3.18 MSG_OBJ_DATA_LOST

Definition:

```
#define MSG_OBJ_DATA_LOST
```

Description:

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

5.2.3.19 MSG_OBJ_EXTENDED_ID

Definition:

```
#define MSG_OBJ_EXTENDED_ID
```

Description:

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

5.2.3.20 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.

5.2.3.21 MSG_OBJ_NEW_DATA

Definition:

```
#define MSG_OBJ_NEW_DATA
```

Description:

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

5.2.3.22 MSG_OBJ_NO_FLAGS

Definition:

```
#define MSG_OBJ_NO_FLAGS
```

Description:

This indicates that a message object has no flags set.

5.2.3.23 MSG_OBJ_REMOTE_FRAME

Definition:

```
#define MSG_OBJ_REMOTE_FRAME
```

Description:

This indicates that a message object is a remote frame.

5.2.3.24 MSG_OBJ_RX_INT_ENABLE

Definition:

```
#define MSG_OBJ_RX_INT_ENABLE
```

Description:

This indicates that receive interrupts should be enabled, or are enabled.

5.2.3.25 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.

5.2.3.26 MSG_OBJ_TX_INT_ENABLE

Definition:

```
#define MSG_OBJ_TX_INT_ENABLE
```

Description:

This indicates that transmit interrupts should be enabled, or are enabled.

5.2.3.27 MSG_OBJ_USE_DIR_FILTER

Definition:

```
#define MSG_OBJ_USE_DIR_FILTER
```

Description:

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

5.2.3.28 MSG_OBJ_USE_EXT_FILTER

Definition:

```
#define MSG_OBJ_USE_EXT_FILTER
```

Description:

This indicates that a message object will use 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.

5.2.3.29 MSG_OBJ_USE_ID_FILTER

Definition:

```
#define MSG_OBJ_USE_ID_FILTER
```

Description:

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

5.2.4 Enumeration Documentation

5.2.4.1 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.

5.2.4.2 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.

5.2.4.3 tMsgObjType

Description:

This definition is used to determine the type of message object that will be 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.

5.2.5 Function Documentation

5.2.5.1 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 should 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 int16_t. 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* should 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.

5.2.5.2 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.

5.2.5.3 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:

bit_time_q = ui32SyncPropPhase1Seg + 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.

5.2.5.4 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.

5.2.5.5 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() should have been called to initialize the controller and the CAN bus clock should be configured by calling CANBitTimingSet().

Returns:
None.

5.2.5.6 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.

5.2.5.7 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.

5.2.5.8 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 should 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 should only be 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.

5.2.5.9 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.

5.2.5.10 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** will generate 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.

5.2.5.11 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() should be used to enable CAN interrupts on the interrupt controller.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

5.2.5.12 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 should be 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.

5.2.5.13 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.

5.2.5.14 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.

5.2.5.15 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.

5.2.5.16 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 should be non-zero even for a remote frame; it should 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 should be 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.

5.2.5.17 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.

5.2.5.18 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.

5.2.5.19 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 should be 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.

5.3 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_RXTX_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.

5.4 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);

...
```

6 EEPROM

6.1 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 definitions for use by applications.

6.1.1 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.

6.1.1.1 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.

6.1.1.2 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.

6.1.1.3 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.

6.2 API Functions

Defines

- EEPROM_INIT_ERROR
- EEPROM_INIT_OK
- EEPROM_INIT_RETRY
- EEPROM_INT_PROGRAM
- EEPROM_PROT_NA_LNA_URW
- EEPROM_PROT_RO_LNA_URO
- EEPROM_PROT_RW_LRO_URW
- EEPROM_PROT_SUPERVISOR_ONLY
- EEPROM_RC_INVPL

- 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)

6.2.1 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.

6.2.2 Define Documentation

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 EEPROM_INIT_RETRY

Definition:

```
#define EEPROM_INIT_RETRY
```

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. The EEPROM peripheral has recovered its state but the last write operation may have been lost. The application must check the validity of data it has written and retry any writes as required.

6.2.2.4 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.

6.2.2.5 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.

6.2.2.6 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.

6.2.2.7 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.

6.2.2.8 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.

6.2.2.9 EEPROM_RC_INVPL

Definition:

```
#define EEPROM_RC_INVPL
```

Description:

This return code bit indicates that the EEPROM programming state machine failed to write a value due to the voltage level dropping below that required for EEPROM programming. The operation may be retried once the voltage stabilizes.

6.2.2.10 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.

6.2.2.11 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.

6.2.2.12 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.

6.2.2.13 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.

6.2.2.14 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.

6.2.2.15 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.

6.2.2.16 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.

6.2.3 Function Documentation

6.2.3.1 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.

6.2.3.2 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 using of block hiding need not contain any embedded passwords which could be found through disassembly.

Returns:

None.

6.2.3.3 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.

6.2.3.4 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_ts 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING** to indicate status and error conditions.

6.2.3.5 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**.

6.2.3.6 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING** to indicate status and error conditions.

6.2.3.7 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.

6.2.3.8 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 to check for errors resulting from power failure during a previous write operation. The function detects these errors and performs as much recovery as possible before returning information to the caller on whether or not a previous data write was lost and must be retried.

In cases where **EEPROM_INIT_RETRY** is returned, the application is responsible for determining which data write may have been lost and rewriting this data. If **EEPROM_INIT_ERROR** is returned, the EEPROM was unable to recover its state. This condition may or may not be resolved on future resets depending upon the cause of the fault. For example, if the supply voltage is unstable, retrying the operation once the voltage is stabilized may clear the error.

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

Returns:

Returns **EEPROM_INIT_OK** if no errors were detected, **EEPROM_INIT_RETRY** if a previous write operation may have been interrupted by a power or reset event or **EEPROM_INIT_ERROR** if the EEPROM peripheral cannot currently recover from an interrupted write or erase operation.

6.2.3.9 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.

6.2.3.10 EEPROMIntDisable

Disables the EEPROM interrupt.

Prototype:

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

Parameters:

ui32IntFlags indicates which EEPROM interrupt source to disable. This must be **EEPROM_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.

6.2.3.11 EEPROMIntEnable

Enables the EEPROM interrupt.

Prototype:
```
void
EEPROMIntEnable(uint32_t ui32IntFlags)
```

Parameters:
ui32IntFlags indicates which EEPROM interrupt source to enable. This must be **EEPROM_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.

6.2.3.12 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.

6.2.3.13 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

6.2.3.14 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

6.2.3.15 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE** and **EEPROM_RC_WORKING**. Flags **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING** are expected in normal operation and do not indicate an error.

6.2.3.16 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.

6.2.3.17 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.

6.2.3.18 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_INVPL**, **EEPROM_RC_WRBUSY**, **EEPROM_RC_NOPERM**, **EEPROM_RC_WKCOPY**, **EEPROM_RC_WKERASE**, and **EEPROM_RC_WORKING**.

6.3 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));
```

7 External Peripheral Interface (EPI)

7.1 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 definitions for use by applications.

7.2 API Functions

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().

7.3 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(EPI0_BASE, 0);

//
// Select SDRAM mode.
//
EPIModeSet(EPI0_BASE, EPI_MODE_SDRAM);

//
// Configure SDRAM mode.
//
EPIConfigSDRAMSet(EPI0_BASE, (EPI_SDRAM_CORE_FREQ_50_100 |
                  EPI_SDRAM_FULL_POWER | EPI_SDRAM_SIZE_64MBIT), 1024);

//
// Set the address map.
//
EPIAddressMapSet(EPI0_BASE, EPI_ADDR_RAM_SIZE_256MB | EPI_ADDR_RAM_BASE_6);

//
// Wait for the EPI initialization to complete.
//
while(HWREG(EPI0_BASE + EPI_O_STAT) &  EPI_STAT_INITSEQ)
{
    //
```

```
    // Wait for SDRAM initialization to complete.
    //
}

//
// At this point, the SDRAM is accessible and available for use.
//
```

8 Fan Controller

8.1 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 definitions for use by applications.

8.2 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.

8.3 Programming Example

```
//
// Enable the Fan peripheral
//
SysCtlPeriphalEnable(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);
```

9 Flash

9.1 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 1 kB blocks that can be individually erased. Erasing a block causes the entire contents of the block to be reset to all ones. These blocks are paired into a set of 2 kB blocks that can be individually protected. 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.

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. In order to do this on some older devices, the flash controller must know the clock rate of the system in order to be able to time the number of micro-seconds certain signals are asserted. On these devices, the number of clock cycles per micro-second must be provided to the flash controller for it to accomplish this 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; the interrupt ensures that invalid accesses are not silently ignored, hiding potential bugs. The flash protection can be applied without being permanently enabled; this, along with the interrupt, 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, or 512 KB.

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

9.2 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)

9.2.1 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().

9.2.2 Function Documentation

9.2.2.1 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 1-kB block of the on-chip flash. After erasing, the block is filled with 0xFF bytes. Read-only and execute-only blocks cannot be erased.

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.

9.2.2.2 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. Can be any of the **FLASH_INT_PROGRAM** or **FLASH_INT_AMISC** values.

Description:

The specified flash controller interrupt sources are cleared, so that they no int32_ter 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.

9.2.2.3 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. Can be any of the **FLASH_INT_PROGRAM** or **FLASH_INT_ACCESS** values.

Description:

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.

9.2.2.4 FlashIntEnable

Enables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntEnable(uint32_t ui32IntFlags)
```

Parameters:

ui32IntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **FLASH_INT_PROGRAM** or **FLASH_INT_ACCESS** values.

Description:

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.

9.2.2.5 FlashIntRegister

Registers an interrupt handler for the flash interrupt.

Prototype:

```
void
FlashIntRegister(void (*pfnHandler)(void))
```

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.

9.2.2.6 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_PROGRAM** and **FLASH_INT_ACCESS**.

9.2.2.7 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 int32_ter called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

9.2.2.8 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. Each word in a page of flash can only be programmed one time between an erase of that page; programming a word multiple times results in an unpredictable value in that word of 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.

9.2.2.9 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 2-kB block of flash. Each 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.

9.2.2.10 FlashProtectSave

Saves the flash protection settings.

Prototype:
```
int32_t
FlashProtectSave(void)
```

Description:
This function makes the currently programmed flash protection settings permanent. On some devices, 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.

9.2.2.11 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 2-kB block of flash. 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.

9.2.2.12 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.

9.2.2.13 FlashUserSave

Saves the user registers.

Prototype:

```
int32_t
FlashUserSave(void)
```

Description:

This function makes the currently programmed user register settings permanent. On some devices, this operation is non-reversible; a chip reset or power cycle does not change this setting.

This function does not return until the protection has been saved.

Returns:
Returns 0 on success, or -1 if a hardware error is encountered.

9.2.2.14 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.

9.3 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));
```

10 Floating-Point Unit (FPU)

10.1 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 definitions for use by applications.

10.2 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)

10.2.1 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()).

10.2.2 Function Documentation

10.2.2.1 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.

10.2.2.2 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.

10.2.2.3 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.

10.2.2.4 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.

10.2.2.5 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.

10.2.2.6 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.

10.2.2.7 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.

10.2.2.8 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.

10.2.2.9 FPUStackingEnable

Enables the stacking of floating-point registers.

Prototype:

```
void
FPUStackingEnable(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.

10.3 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();
```

11 GPIO

11.1 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, they default to being an input.
- 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, they default to 2 mA drive strength.
- Optional weak pull-up or pull-down resistors. On reset, they default to a weak pull-up on Sandstorm-class devices, and default to disabled on all other devices.
- Optional open-drain operation. On reset, they default to standard push/pull operation.
- Can be configured to be a GPIO or a peripheral pin. On reset, they default to being GPIOs. Note that not all pins on all parts have peripheral functions, in which case the pin is only useful as a GPIO (that is, when configured for peripheral function the pin will not do anything useful).

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 upon (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).

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

11.2 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 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 GPIOPinTypeFan (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 GPIOPinTypeI2S (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 GPIOPinWrite (uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val)

11.2.1 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(). There are also convenience functions for configuring the pin in the required or recommended configuration for a particular peripheral; these are GPIOPinTypeCAN(), GPIOPinTypeComparator(), GPIOPinTypeGPIOInput(), GPIOPinTypeGPIOOutput(), GPIOPinTypeGPIOOutputOD(), GPIOPinTypeI2C(), GPIOPinTypePWM(), GPIOPinTypeQEI(), GPIOPinTypeSSI(), GPIOPinTypeTimer(), and GPIOPinTypeUART().

The GPIO interrupts are handled with GPIOIntTypeSet(), GPIOIntTypeGet(), GPIOIntEnable(), GPIOIntDisable(), GPIOIntStatus(), GPIOIntClear(), GPIOIntRegister(), and GPIOIntUnregister().

The GPIO pin state is accessed with GPIOPinRead() and GPIOPinWrite().

11.2.2 Function Documentation

11.2.2.1 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().

Note:

This function is not available on all devices, consult the data sheet to ensure that the device you are using supports GPIO ADC Control.

Returns:

None.

11.2.2.2 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.

Note:

This function is not available on all devices, consult the data sheet to ensure that the device you are using supports GPIO ADC Control.

Returns:

None.

11.2.2.3 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().

11.2.2.4 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.

Returns:

None.

11.2.2.5 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().

Note:

This function is not available on all devices, consult the data sheet to ensure that the device you are using supports GPIO DMA Control.

Returns:

None.

11.2.2.6 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.

Note:

This function is not available on all devices, consult the data sheet to ensure that the device you are using supports GPIO DMA Control.

Returns:

None.

11.2.2.7 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.

11.2.2.8 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.

Returns:

None.

11.2.2.9 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.

Returns:
None.

11.2.2.10 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.

11.2.2.11 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.

11.2.2.12 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().

11.2.2.13 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.

11.2.2.14 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.

11.2.2.15 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

11.2.2.16 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**

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.

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**

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 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.

11.2.2.17 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_IS_**<**partno**> define enables the appropriate set of defines for the device that is being used.

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.

11.2.2.18 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.

11.2.2.19 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.

Returns:

None.

11.2.2.20 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.21 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.22 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.23 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 two 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.24 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.25 GPIOPinTypeFan

Configures pin(s) for use by the fan module.

Prototype:
```
void
GPIOPinTypeFan(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 fan pins must be properly configured for the fan 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 a fan pin; it only configures a fan pin for proper operation. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.26 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.

Returns:

None.

11.2.2.27 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.

Returns:
None.

11.2.2.28 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.

Returns:
None.

11.2.2.29 GPIOPinTypeI2C

Configures pin(s) for use 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(s).

Description:
The I2C pins must be properly configured for the I2C 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 I2C pin; it only configures an I2C pin for proper operation. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.30 GPIOPinTypeI2CSCL

Configures pin(s) 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(s).

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(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 should only be used for Blizzard-class devices. It cannot be used to turn any pin into an I2C SCL pin; it only configures an I2C SCL pin for proper operation. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.31 GPIOPinTypeI2S

Configures pin(s) for use by the I2S peripheral.

Prototype:
```
void
GPIOPinTypeI2S(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:

Some I2S pins must be properly configured for the I2S peripheral to function correctly. This function provides a typical configuration for the digital I2S 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 I2S pin; it only configures a I2S pin for proper operation. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.32 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.33 GPIOPinTypePECIRx

Configures a pin for receive use by the PECI 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 PECI receive pin must be properly configured for the PECI 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 receive pin; it only configures a PECI receive pin for proper operation. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.34 GPIOPinTypePECITx

Configures a pin for transmit use by the PECI 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 PECI transmit pin must be properly configured for the PECI 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.35 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. Devices wtih flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.36 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.37 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.38 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.39 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.40 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:

Some USB analog pins must be properly configured for the USB peripheral to function correctly. This function provides the proper configuration for any USB pin(s). This can also be used to configure the EPEN and PFAULT pins so that they are no longer used by the USB controller.

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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:

None.

11.2.2.41 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:

Some 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. Devices with flexible pin muxing also require a GPIOPinConfigure() function call.

Returns:
None.

11.2.2.42 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.

11.3 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.

```
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 will be a 0, and pin 3 will be a 1.
//
GPIOPinWrite(GPIO_PORTA_BASE,
             (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
              GPIO_PIN_4 | GPIO_PIN_5),
             0xF8);

//
// Enable the pin interrupts.
//
GPIOIntEnable(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);
```

12 Hibernation Module

12.1 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
- Trim register for fine tuning the RTC rate
- One or two RTC match registers for generating RTC events
- External **WAKE** pin to initiate a wake-up
- Low-battery detection
- 16 or 64 32-bit words of battery-backed memory
- Programmable interrupts for hibernation events

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

12.2 API Functions

Functions

- uint32_t HibernateBatCheckDone (void)
- void HibernateBatCheckStart (void)
- void HibernateClockConfig (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)
- uint32_t HibernateWakeGet (void)
- void HibernateWakeSet (uint32_t ui32WakeFlags)

12.2.1 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 two match registers 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. On devices that include the subseconds counter, 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.

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 (see processor datasheet for a list of GPIOs that support this function) or the failure of the external oscillator if used as a clock source.

The tamper GPIOs are configured to use with HibernateTamperIOEnable() and HibernateTamperIODisable(). The external oscillator state can be retrieved with HibernateTamperExtOscValid(). If an external oscillator failure was detected, a recovery attempt can be triggered with HibernateTamperExtOscRecover().

The module always reponds 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. 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.

12.2.2 Function Documentation

12.2.2.1 HibernateBatCheckDone

Returns if a forced battery check has completed.

Prototype:

```
uint32_t
```

```
HibernateBatCheckDone(void)
```

Description:

This function returns if 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.

12.2.2.2 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.

12.2.2.3 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 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 24pF filter capacitor is used with a crystal.
- **HIBERNATE_OSC_LOWDRIVE** specifies a lower drive strength when a 12pF filter capacitor is used with a crystal.

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_LOWDRIVE** 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.

12.2.2.4 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.

Note:
The amount of memory available in the Hibernation module varies across Tiva devices. Please consult the data sheet for the Tiva device that you are using to determine the amount of memory available in the Hibernation module.

Returns:
None.

12.2.2.5 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 which is available 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.

12.2.2.6 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.

12.2.2.7 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.

12.2.2.8 HibernateGPIORetentionDisable

Disables GPIO retention after wake from hibernate.

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.

12.2.2.9 HibernateGPIORetentionEnable

Enables GPIO retention after wake from hibernate.

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.

12.2.2.10 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.

12.2.2.11 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.

12.2.2.12 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.

12.2.2.13 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

Returns:

None.

12.2.2.14 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.

12.2.2.15 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.

Returns:
Returns the interrupt status as a bit field with the values as described in the HibernateIntEnable() function.

12.2.2.16 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.

12.2.2.17 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.

12.2.2.18 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.

12.2.2.19 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 also trigger an interrupt. Optionally, hibernation can be aborted if a low-battery 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.

12.2.2.20 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 HibernateLowBatSet() 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.

12.2.2.21 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.

12.2.2.22 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.

12.2.2.23 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.

12.2.2.24 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.

Returns:

Returns the value of the requested match register.

12.2.2.25 HibernateRTCMatchSet

Sets the value of the RTC match 0 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.

12.2.2.26 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 count seconds if the hardware is configured correctly. The RTC must be enabled by calling HibernateRTCEnable() before calling this function.

Returns:
None.

12.2.2.27 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.

Returns:
The current RTC sub second count in 1/32768 seconds.

12.2.2.28 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.

Returns:
Returns the value of the requested sub section match register.

12.2.2.29 HibernateRTCSSMatchSet

Sets the value of the RTC sub second match 0 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.

Returns:

None.

12.2.2.30 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.

12.2.2.31 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.

12.2.2.32 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 one of the RTC matches occurs
- **HIBERNATE_WAKE_LOW_BAT** - wake from hibernation due to a low-battery level being detected

Note:

The **HIBERNATE_WAKE_LOW_BAT** parameter is only available on some Tiva devices.

Returns:

Returns flags indicating the configured wake conditions.

12.2.2.33 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.

Returns:

None.

12.3 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)
    {
        //
        // Wakeup was due to WAKE pin assertion.
        //
    }
    if(ui32Status & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // Wakeup was due to RTC match register.
        //
    }

    //
    // Restore program state information that was saved prior to
    // hibernation.
    //
    HibernateDataGet(pui32NVData, 64);

    //
    // Now that wakeup 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 a hibernation with wake up at a future time:

```
uint32_t ui32Status;
uint32_t pui32NVData[64];

//
```

```
// Need to 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_LOWDRIVE);
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 64 words
// of data, only as much as is actually needed by the program.
//
HibernateDataSet(pui32NVData, 64);

//
// Configure to wake on RTC match.
//
HibernateWakeSet(HIBERNATE_WAKE_RTC);

//
// Request hibernation.  The following call may return since 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_LOWDRIVE);
    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.
//

// ...
```

13 Inter-Integrated Circuit (I2C)

13.1 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 two speeds: Standard (100 kbps) and Fast (400 kbps).

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.

13.1.1 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.

13.1.2 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 definitions for use by applications.

13.2 API Functions

Functions

- void I2CIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- void I2CIntUnregister (uint32_t ui32Base)
- 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 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 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 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)

13.2.1 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.

13.2.2 Function Documentation

13.2.2.1 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 Master 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.

13.2.2.2 I2CIntUnregister

Unregisters an interrupt handler for the I2C module.

Prototype:

```
void
I2CIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Master 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 r controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

13.2.2.3 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 Master 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**.

13.2.2.4 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 Master 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**.

13.2.2.5 I2CMasterControl

Controls the state of the I2C Master module.

Prototype:

```
void
I2CMasterControl(uint32_t ui32Base,
                 uint32_t ui32Cmd)
```

Parameters:

ui32Base is the base address of the I2C Master module.

ui32Cmd command to be issued to the I2C Master module.

Description:

This function is used to control the state of the Master module 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**

Returns:
None.

13.2.2.6 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 Master 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.

13.2.2.7 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 Master module.
ui8Data data to be transmitted from the I2C Master.

Description:
This function places the supplied data into I2C Master Data Register.

Returns:
None.

13.2.2.8 I2CMasterDisable

Disables the I2C master block.

Prototype:

```
void
I2CMasterDisable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Master module.

Description:

This function disables operation of the I2C master block.

Returns:

None.

13.2.2.9 I2CMasterEnable

Enables the I2C Master block.

Prototype:

```
void
I2CMasterEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Master module.

Description:

This function enables operation of the I2C Master block.

Returns:

None.

13.2.2.10 I2CMasterErr

Gets the error status of the I2C Master module.

Prototype:

```
uint32_t
I2CMasterErr(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Master module.

Description:

This function is used to obtain the error status of the Master module 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**.

13.2.2.11 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 Master 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.

13.2.2.12 I2CMasterIntClear

Clears I2C Master interrupt sources.

Prototype:

```
void
I2CMasterIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Master 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.

13.2.2.13 I2CMasterIntClearEx

Clears I2C Master interrupt sources.

Prototype:
```
void
I2CMasterIntClearEx(uint32_t ui32Base,
                    uint32_t ui32IntFlags)
```

Parameters:
ui32Base is the base address of the I2C Master 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.

13.2.2.14 I2CMasterIntDisable

Disables the I2C Master interrupt.

Prototype:
```
void
I2CMasterIntDisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the I2C Master module.

Description:
This function disables the I2C Master interrupt source.

Returns:
None.

13.2.2.15 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 Master 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.

13.2.2.16 I2CMasterIntEnable

Enables the I2C Master interrupt.

Prototype:
```
void
I2CMasterIntEnable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the I2C Master module.

Description:
This function enables the I2C Master interrupt source.

Returns:
None.

13.2.2.17 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 Master 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_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.

13.2.2.18 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 Master 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 Master 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.

13.2.2.19 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 Master 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 Master 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().

13.2.2.20 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 Master 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.

13.2.2.21 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 Master 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.

13.2.2.22 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 Master 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 100kHz SCL frequency, *ui32Value* would be 0x7d.

Note:

Not all Tiva devices support this function. Please consult the device data sheet to determine if this feature is supported.

Returns:

None.

13.2.2.23 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 Slave 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.

13.2.2.24 I2CSlaveACKValueSet

Writes the ACK value.

Prototype:
```
void
I2CSlaveACKValueSet(uint32_t ui32Base,
                    bool bACK)
```

Parameters:
ui32Base is the base address of the I2C Slave 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.

13.2.2.25 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 Slave 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.

13.2.2.26 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 Slave 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.

13.2.2.27 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 Slave 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.

13.2.2.28 I2CSlaveDisable

Disables the I2C slave block.

Prototype:
```
void
I2CSlaveDisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the I2C Slave module.

Description:
This function disables operation of the I2C slave block.

Returns:
None.

13.2.2.29 I2CSlaveEnable

Enables the I2C Slave block.

Prototype:
```
void
I2CSlaveEnable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the I2C Slave module.

Description:
This fucntion enables operation of the I2C Slave block.

Returns:
None.

13.2.2.30 I2CSlaveInit

Initializes the I2C Slave block.

Prototype:
```
void
I2CSlaveInit(uint32_t ui32Base,
             uint8_t ui8SlaveAddr)
```

Parameters:
ui32Base is the base address of the I2C Slave 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.

13.2.2.31 I2CSlaveIntClear

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClear(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Slave 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.

13.2.2.32 I2CSlaveIntClearEx

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClearEx(uint32_t ui32Base,
                   uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the I2C Slave 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.

13.2.2.33 I2CSlaveIntDisable

Disables the I2C Slave interrupt.

Prototype:
```
void
I2CSlaveIntDisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the I2C Slave module.

Description:
This function disables the I2C Slave interrupt source.

Returns:
None.

13.2.2.34 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 Slave 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.

13.2.2.35 I2CSlaveIntEnable

Enables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Slave module.

Description:

This function enables the I2C Slave interrupt source.

Returns:

None.

13.2.2.36 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 Slave 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_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.

13.2.2.37 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 Slave 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 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.

13.2.2.38 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 Slave 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 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 I2CSlaveIntEnableEx().

13.2.2.39 I2CSlaveStatus

Gets the I2C Slave module status

Prototype:

```
uint32_t
I2CSlaveStatus(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the I2C Slave 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 module, **I2C_SLAVE_ACT_RREQ** to indicate that an I2C master has sent data to the I2C Slave module, **I2C_SLAVE_ACT_TREQ** to indicate that an I2C master has requested that the I2C Slave module 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.

13.3 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))
{
}
```

14 Interrupt Controller (NVIC)

14.1 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 should be placed appropriately with a linker script.

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

14.2 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)

14.2.1 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 should not be enabled before a handler has been registered, and interrupt sources should 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.

14.2.2 Function Documentation

14.2.2.1 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. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Returns:

None.

14.2.2.2 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. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Returns:

None.

14.2.2.3 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.

Returns:

A non-zero value if the interrupt is enabled.

14.2.2.4 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`.

Returns:

Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

14.2.2.5 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.

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`.

Returns:
Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

14.2.2.6 IntPendClear

Un-pends an interrupt.

Prototype:
```
void
IntPendClear(uint32_t ui32Interrupt)
```

Parameters:
ui32Interrupt specifies the interrupt to be un-pended.

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.

Returns:
None.

14.2.2.7 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. 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.

Returns:
None.

14.2.2.8 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. See IntPrioritySet() for a definition of the priority value.

Returns:

Returns the interrupt priority, or -1 if an invalid interrupt was specified.

14.2.2.9 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.

Returns:

The number of bits of preemptable priority.

14.2.2.10 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.

Returns:

None.

14.2.2.11 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 N bits of the priority level (where N is 3 for the Tiva C and E Series family), so any prioritization must be performed in those bits.

Returns:

Returns the value of the interrupt priority level mask.

14.2.2.12 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.

The hardware priority mechanism only looks at the upper N bits of the priority level (where N is 3 for the Tiva C and E Series family), so any prioritization must be performed in those bits.

Returns:

None.

14.2.2.13 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. 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.

The hardware priority mechanism only looks at the upper N bits of the priority level (where N is 3 for the Tiva C and E Series family), 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.

Returns:

None.

14.2.2.14 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. 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.

Returns:
None.

14.2.2.15 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 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.

14.2.2.16 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 interrupt source is automatically disabled (via IntDisable()) if necessary.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

14.3 Programming Example

The following example shows how to use the Interrupt Controller API to register an interrupt handler and enable the interrupt.

```
//
// The interrupt handler function.
//
extern void IntHandler(void);

//
// Register the interrupt handler function for interrupt 5.
//
IntRegister(5, IntHandler);

//
// Enable interrupt 5.
//
IntEnable(5);

//
// Enable interrupt 5.
//
IntMasterEnable();
```

15 Low Pin Count Interface (LPC)

15.1 Introduction

The LPC API provides functions to use the LPC module available in the Tiva family of microcontrollers. The LPC module provides for up to 8 channels on the LPC bus, including one channel which can be configured as a COMx port.

Some features of the LPC module are:

- configurable channel type (endpoint, mailbox)
- configurable channel pool size
- interrupt and uDMA support

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

15.2 API Functions

16 Memory Protection Unit (MPU)

16.1 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 definitions for use by applications.

16.2 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)

16.2.1 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 be 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_MPU** 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().

16.2.2 Function Documentation

16.2.2.1 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.

16.2.2.2 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.

16.2.2.3 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.

16.2.2.4 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.

16.2.2.5 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().

16.2.2.6 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.

16.2.2.7 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.

16.2.2.8 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 MPURegionSet() function.

This function can be used to save the configuration of a region for later use with the MPURegionSet() function. The region's enable state is preserved in the attributes that are saved.

Returns:

None.

16.2.2.9 MPURegionSet

Sets up the access rules for a specific region.

Prototype:

```
void
MPURegionSet(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 subregions, 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.

16.3 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]);
}
```

17 Platform Environment Control Interface (PECI)

17.1 Introduction

The PECI API provides functions to use the PECI module available in the Tiva microcontroller family. The PECI module provides for 2 microprocessors with 2 domains each for a total of up to 4 domains.

Some features of the PECI module are:

- configurable PECI baud and polling rates
- configurable interrupts and thresholds

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

17.2 API Functions

18 Pulse Width Modulator (PWM)

18.1 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 definitions for use by applications.

18.2 API Functions

Functions

- 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)
- 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)

18.2.1 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.

18.2.2 Function Documentation

18.2.2.1 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.

18.2.2.2 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.

18.2.2.3 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.

18.2.2.4 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**.

When running on a device supporting extended PWM fault handling, 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. On a device without extended PWM fault handling, the interrupt is directly related to the state of the single FAULT pin.

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.

18.2.2.5 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.

pfnIntHandler 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 *pfnIntHandler* 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 PWMIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

18.2.2.6 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.

18.2.2.7 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. PWMGenPeriodSet() and PWMPulseWidthSet() should be called after any changes to the counter mode of a generator.

Returns:
None.

18.2.2.8 PWMGenDisable

Disables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGenDisable(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.

18.2.2.9 PWMGenEnable

Enables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGenEnable(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.

18.2.2.10 PWMGenFaultClear

Clears one or more latched fault triggers for a given PWM generator.

Prototype:

```
void
PWMGenFaultClear(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_MODE_FAULT_LATCHED** in parameter *ui32Config*.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

18.2.2.11 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_MODE_FAULT_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.

18.2.2.12 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_MODE_FAULT_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**.

18.2.2.13 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 **PWM_FAULT_GROUP_0** or **PWM_FAULT_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 **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**.

18.2.2.14 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 **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

ui32FaultTriggers defines the set of inputs that are to contribute towards generation of the fault signal to the given PWM generator. For **PWM_FAULT_GROUP_0**, this is the logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**. For **PWM_FAULT_GROUP_1**, this 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**.

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.

18.2.2.15 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.

18.2.2.16 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 PWMGenIntTrigEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

18.2.2.17 PWMGenIntStatus

Gets interrupt status for the specified PWM generator block.

Prototype:

```
uint32_t
PWMGenIntStatus(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.

18.2.2.18 PWMGenIntTrigDisable

Disables interrupts for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigDisable(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.

18.2.2.19 PWMGenIntTrigEnable

Enables interrupts and triggers for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigEnable(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.

18.2.2.20 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.

18.2.2.21 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.

18.2.2.22 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.

18.2.2.23 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.

18.2.2.24 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.

18.2.2.25 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**.

18.2.2.26 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 int8_tacteristics of the selected PWM outputs. The outputs are selected using the parameter *ui32PWMOutBits*. The parameter *bFaultSuppress* determines the fault handling chacteristics 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 PWMOutputFaultLevel(). 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.

18.2.2.27 PWMOutputFaultLevel

Specifies the level of PWM outputs suppressed in response to a fault condition.

Prototype:

```
void
PWMOutputFaultLevel(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 PWMOutputFault() are unaffected by this function.

Note:
This function is available only on devices which support extended PWM fault handling.

Returns:
None.

18.2.2.28 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.

18.2.2.29 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.

18.2.2.30 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.

18.2.2.31 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.

18.2.2.32 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.

18.2.2.33 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.

18.3 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.
//
PWMGenPeriodSet(PWM_BASE, PWM_GEN_0, 400);

//
// 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.
//
PWMGenEnable(PWM_BASE, PWM_GEN_0);

//
// Enable the outputs.
//
PWMOutputState(PWM_BASE, (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
```

19 Quadrature Encoder (QEI)

19.1 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 definitions for use by applications.

19.2 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 QEIIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void QEIIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t QEIIntStatus (uint32_t ui32Base, bool bMasked)
- void QEIIntUnregister (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)

19.2.1 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 QEIIntRegister() and QEIIntUnregister(). The individual interrupt sources within the QEI module are managed with QEIIntEnable(), QEIIntDisable(), QEIIntStatus(), and QEIIntClear().

19.2.2 Function Documentation

19.2.2.1 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.

19.2.2.2 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.

19.2.2.3 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.

19.2.2.4 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.

19.2.2.5 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.

19.2.2.6 QEIIntClear

Clears quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntClear(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_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:

The specified quadrature encoder interrupt sources are cleared, so that they no int32_ter 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.

19.2.2.7 QEIIntDisable

Disables individual quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntDisable(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_INTERROR**, **QEI_INTDIR**, **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.

19.2.2.8 QEIIntEnable

Enables individual quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntEnable(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_INTERROR**, **QEI_INTDIR**, **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.

19.2.2.9 QEIIntRegister

Registers an interrupt handler for the quadrature encoder interrupt.

Prototype:

```
void
QEIIntRegister(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 QEIIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source via QEIIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

19.2.2.10 QEIIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t
QEIIntStatus(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_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, and **QEI_INTINDEX**.

19.2.2.11 QEIIntUnregister

Unregisters an interrupt handler for the quadrature encoder interrupt.

Prototype:

```
void
QEIIntUnregister(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.

19.2.2.12 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.

19.2.2.13 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.

19.2.2.14 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.

19.2.2.15 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.

19.2.2.16 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.

19.2.2.17 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.

19.3 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);
```

20 Synchronous Serial Interface (SSI)

20.1 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™, National Semiconductor® Microwire, or the Texas Instruments® synchronous serial interface frame formats. 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 definitions for use by applications.

20.2 API Functions

Functions

- 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 SSIIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)
- void SSIIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void SSIIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void SSIIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t SSIIntStatus (uint32_t ui32Base, bool bMasked)
- void SSIIntUnregister (uint32_t ui32Base)

20.2.1 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 SSIIntClear(), SSIIntDisable(), SSIIntEnable(), SSIIntRegister(), SSIIntStatus(), and SSIIntUnregister() functions.

The SSIConfig(), SSIDataNonBlockingGet(), and SSIDataNonBlockingPut() APIs from previous versions of the peripheral driver library have been replaced by the SSIConfigSetExpClk(), SSIDataGetNonBlocking(), 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.

20.2.2 Function Documentation

20.2.2.1 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 port.

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.

20.2.2.2 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 port.

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 will be either **SSI_CLOCK_SYSTEM** or **SSI_CLOCK_PIOSC**.

20.2.2.3 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 port.

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.

20.2.2.4 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**. 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 $>=$ 2 $*$ bit rate (master mode); this speed cannot exceed 25 MHz.
- FSSI $>=$ 12 $*$ bit rate or 6 $*$ bit rate (slave modes), depending on the capability of the specific microcontroller

where FSSI is the frequency of the clock supplied to the SSI module.

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.

20.2.2.5 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 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:
None.

20.2.2.6 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 *ui32Data* 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.

20.2.2.7 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.

20.2.2.8 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 SSIConfigSetExpClk(). 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.

20.2.2.9 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.

20.2.2.10 SSIDMADisable

Disables SSI DMA operation.

Prototype:
```
void
SSIDMADisable(uint32_t ui32Base,
              uint32_t ui32DMAFlags)
```

Parameters:
ui32Base is the base address of the SSI port.
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.

20.2.2.11 SSIDMAEnable

Enables SSI DMA operation.

Prototype:

```
void
SSIDMAEnable(uint32_t ui32Base,
             uint32_t ui32DMAFlags)
```

Parameters:

ui32Base is the base address of the SSI port.

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.

20.2.2.12 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.

20.2.2.13 SSIIntClear

Clears SSI interrupt sources.

Prototype:

```
void
SSIIntClear(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.

20.2.2.14 SSIIntDisable

Disables individual SSI interrupt sources.

Prototype:

```
void
SSIIntDisable(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.

20.2.2.15 SSIIntEnable

Enables individual SSI interrupt sources.

Prototype:

```
void
SSIIntEnable(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.

20.2.2.16 SSIIntRegister

Registers an interrupt handler for the synchronous serial interface.

Prototype:

```
void
SSIIntRegister(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 SSIIntEnable(). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via SSIIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

20.2.2.17 SSIIntStatus

Gets the current interrupt status.

Prototype:

```
uint32_t
SSIIntStatus(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**.

20.2.2.18 SSIIntUnregister

Unregisters an interrupt handler for the synchronous serial interface.

Prototype:

```
void
SSIIntUnregister(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.

20.3 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.
//
SSIConfigSetExpClk(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++;
}
```

21 System Control

21.1 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 PIOSC is not available on all Tiva devices. 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 periphersals 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 the input voltage dropping too low, the LDO voltage dropping too low, an external reset, a software reset request, a watchdog timeout, 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 upon a particular input clock rate (such as a UART) cannot operate as expected in deep-sleep mode due to the 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 definitions for use by applications.

21.2 API Functions

Functions

- uint32_t SysCtlADCSpeedGet (void)
- void SysCtlADCSpeedSet (uint32_t ui32Speed)
- uint32_t SysCtlClockGet (void)
- void SysCtlClockSet (uint32_t ui32Config)
- void SysCtlDeepSleep (void)
- void SysCtlDeepSleepClockSet (uint32_t ui32Config)
- void SysCtlDelay (uint32_t ui32Count)
- 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)
- void SysCtlMOSCConfigSet (uint32_t ui32Config)
- 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)
- void SysCtlResetCauseClear (uint32_t ui32Causes)
- uint32_t SysCtlResetCauseGet (void)
- void SysCtlSleep (void)
- uint32_t SysCtlSRAMSizeGet (void)
- void SysCtlUSBPLLDisable (void)
- void SysCtlUSBPLLEnable (void)

21.2.1 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().

21.2.2 Function Documentation

21.2.2.1 SysCtlADCSpeedGet

Gets the sample rate of the ADC.

Prototype:

```
uint32_t
SysCtlADCSpeedGet(void)
```

Description:

This function gets the current sample rate of the ADC.

Returns:

Returns the current ADC sample rate; is one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

21.2.2.2 SysCtlADCSpeedSet

Sets the sample rate of the ADC.

Prototype:

```
void
SysCtlADCSpeedSet(uint32_t ui32Speed)
```

Parameters:

ui32Speed is the desired sample rate of the ADC; must be one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

Description:

This function configures the rate at which the ADC samples are captured by the ADC block. The sampling speed may be limited by the hardware, so the sample rate may end up being slower than requested. SysCtlADCSpeedGet() returns the actual speed in use.

Returns:

None.

21.2.2.3 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.

Returns:
The processor clock rate.

21.2.2.4 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:
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.

21.2.2.5 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.

21.2.2.6 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_DSLP_DIV_1**, **SYSCTL_DSLP_DIV_2**, **SYSCTL_DSLP_DIV_3**, ... **SYSCTL_DSLP_DIV_64**.

The oscillator source is chosen from one of the following values: **SYSCTL_DSLP_OSC_MAIN**, **SYSCTL_DSLP_OSC_INT**, **SYSCTL_DSLP_OSC_INT30**, or **SYSCTL_DSLP_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_DSLP_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:
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.

21.2.2.7 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 constant length delay. It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

Returns:
None.

21.2.2.8 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.

21.2.2.9 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.

Returns:

None.

21.2.2.10 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**.

Returns:

None.

21.2.2.11 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_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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 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).

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.

21.2.2.12 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_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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.

21.2.2.13 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_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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.

21.2.2.14 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.

21.2.2.15 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_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and **SYSCTL_INT_PLL_FAIL**.

21.2.2.16 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.

21.2.2.17 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 connected to the OSC0/OSC1 pins, allowing power consumption to be reduced.

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.

21.2.2.18 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.

21.2.2.19 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.20 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.21 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.22 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_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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**.

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.

21.2.2.23 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_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**,

SYSCTL_PERIPH_EEPROM0, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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**.

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.

21.2.2.24 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_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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**.

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.

21.2.2.25 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 only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.26 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_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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**.

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.

21.2.2.27 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.28 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.29 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_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **SYSCTL_PERIPH_FAN0**, **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_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_LPC0**, **SYSCTL_PERIPH_PECI0**, **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_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.

21.2.2.30 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 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.

21.2.2.31 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**.

21.2.2.32 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:

The clocking of the PWM is dependent upon the system clock rate as configured by SysCtlClockSet().

Returns:

None.

21.2.2.33 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.

21.2.2.34 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_LDO**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG**, **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.

21.2.2.35 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_CAUSE_LDO**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG**, **SYSCTL_CAUSE_BOR**, **SYSCTL_CAUSE_POR**, and/or **SYSCTL_CAUSE_EXT**.

Returns:
Returns the reason(s) for a reset.

21.2.2.36 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.

21.2.2.37 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.

21.2.2.38 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.

Returns:

None.

21.2.2.39 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.

Returns:

None.

21.3 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(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_XTAL_4MHZ |
               SYSCTL_OSC_MAIN);

//
// Enable the GPIO blocks and the SSI.
//
SysCtlPeripheralEnable(SYSCTL_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);
```

22 System Exception Module

22.1 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 definitions for use by applications.

22.2 API Functions

Functions

- void SysExcIntClear (uint32_t ui32IntFlags)
- void SysExcIntDisable (uint32_t ui32IntFlags)
- void SysExcIntEnable (uint32_t ui32IntFlags)
- void SysExcIntRegister (void (*pfnHandler)(void))
- uint32_t SysExcIntStatus (bool bMasked)
- void SysExcIntUnregister (void)

22.2.1 Detailed Description

The system exception module interrupts are managed with SysExcIntRegister(), SysExcIntUnregister(), SysExcIntEnable(), SysExcIntDisable(), SysExcIntStatus(), and SysExcIntClear().

22.2.2 Function Documentation

22.2.2.1 SysExcIntClear

Clears system exception interrupt sources.

Prototype:

```
void
SysExcIntClear(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.

22.2.2.2 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.

22.2.2.3 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.

22.2.2.4 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.

22.2.2.5 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_IXC**, **SYSEXC_INT_FP_OFC**, **SYSEXC_INT_FP_UFC**, **SYSEXC_INT_FP_IOC**, **SYSEXC_INT_FP_DZC**, and **SYSEXC_INT_FP_IDC**.

22.2.2.6 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.

22.3 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);
```

23 System Tick (SysTick)

23.1 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 definitions for use by applications.

23.2 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)

23.2.1 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()).

23.2.2 Function Documentation

23.2.2.1 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.

23.2.2.2 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.

23.2.2.3 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.

23.2.2.4 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.

23.2.2.5 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.

23.2.2.6 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.

23.2.2.7 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.

23.2.2.8 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.

23.2.2.9 SysTickValueGet

Gets the current value of the SysTick counter.

Prototype:
```
uint32_t
SysTickValueGet(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.

23.3 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();
```

24 Timer

24.1 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 definitions for use by applications.

24.2 API Functions

Functions

- 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)
- 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)
- uint32_t TimerValueGet (uint32_t ui32Base, uint32_t ui32Timer)
- uint64_t TimerValueGet64 (uint32_t ui32Base)

24.2.1 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().

24.2.2 Function Documentation

24.2.2.1 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

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.

24.2.2.2 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.

24.2.2.3 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.

24.2.2.4 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.

24.2.2.5 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.

24.2.2.6 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.

24.2.2.7 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.

24.2.2.8 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.

24.2.2.9 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.

24.2.2.10 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.

24.2.2.11 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_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.

24.2.2.12 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.

24.2.2.13 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().

24.2.2.14 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 int32_ter called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

24.2.2.15 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.

24.2.2.16 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.

24.2.2.17 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.

24.2.2.18 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.

24.2.2.19 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.

24.2.2.20 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.

24.2.2.21 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.

24.2.2.22 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.

24.2.2.23 TimerPrescaleGet

Get 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.

24.2.2.24 TimerPrescaleMatchGet

Get 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.

24.2.2.25 TimerPrescaleMatchSet

Set 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.

24.2.2.26 TimerPrescaleSet

Set 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.

24.2.2.27 TimerRTCDisable

Disable 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.

24.2.2.28 TimerRTCEnable

Enable 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.

24.2.2.29 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.

24.2.2.30 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.

24.2.2.31 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.

24.3 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);
```

25 UART

25.1 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.
- DMA interface
- 9-bit operation

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

25.2 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 UARTModemControlClear (uint32_t ui32Base, uint32_t ui32Control)
- uint32_t UARTModemControlGet (uint32_t ui32Base)
- void UARTModemControlSet (uint32_t ui32Base, uint32_t ui32Control)
- uint32_t UARTModemStatusGet (uint32_t ui32Base)
- uint32_t UARTParityModeGet (uint32_t ui32Base)
- void UARTParityModeSet (uint32_t ui32Base, uint32_t ui32Parity)
- void UARTRxErrorClear (uint32_t ui32Base)
- uint32_t UARTRxErrorGet (uint32_t ui32Base)
- void UARTSmartCardDisable (uint32_t ui32Base)
- void UARTSmartCardEnable (uint32_t ui32Base)
- bool UARTSpaceAvail (uint32_t ui32Base)
- uint32_t UARTTxIntModeGet (uint32_t ui32Base)
- void UARTTxIntModeSet (uint32_t ui32Base, uint32_t ui32Mode)

25.2.1 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 UARTDMADisable() 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.

25.2.2 Function Documentation

25.2.2.1 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.

25.2.2.2 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.

25.2.2.3 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.

25.2.2.4 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.

25.2.2.5 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.

25.2.2.6 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.

25.2.2.7 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*.

25.2.2.8 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.

25.2.2.9 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.

25.2.2.10 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.

25.2.2.11 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.

25.2.2.12 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.

25.2.2.13 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.

25.2.2.14 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.

25.2.2.15 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.

25.2.2.16 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.

25.2.2.17 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.

25.2.2.18 UARTDMADisable

Disable UART DMA 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 DMA features to disable.

Description:
This function is used to disable UART DMA features that were enabled by UARTDMAEnable(). The specified UART DMA features are disabled. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- UART_DMA_RX - disable DMA for receive

- UART_DMA_TX - disable DMA for transmit
- UART_DMA_ERR_RXSTOP - do not disable DMA receive on UART error

Returns:
None.

25.2.2.19 UARTDMAEnable

Enable UART DMA 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 DMA features to enable.

Description:
The specified UART DMA features are enabled. The UART can be configured to use DMA 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 DMA for receive
- UART_DMA_TX - enable DMA for transmit
- UART_DMA_ERR_RXSTOP - disable DMA receive on UART error

Note:
The uDMA controller must also be set up before DMA can be used with the UART.

Returns:
None.

25.2.2.20 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.

25.2.2.21 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.

25.2.2.22 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.

25.2.2.23 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 functions enables the transmit and receive FIFOs in the UART.

Returns:
None.

25.2.2.24 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.

25.2.2.25 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.

25.2.2.26 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.

25.2.2.27 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.

25.2.2.28 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.

25.2.2.29 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.

25.2.2.30 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.

25.2.2.31 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 UARTIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

25.2.2.32 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().

25.2.2.33 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.

25.2.2.34 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.

25.2.2.35 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.

25.2.2.36 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.

25.2.2.37 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.

25.2.2.38 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**.

25.2.2.39 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.

25.2.2.40 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.

25.2.2.41 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**.

25.2.2.42 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.

25.2.2.43 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.

25.2.2.44 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.

25.2.2.45 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**.

25.2.2.46 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.

25.3 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);
```

26 uDMA Controller

26.1 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 definitions for use by applications.

26.2 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)

26.2.1 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 uDMAChannel-ControlSet() 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 uDMAChannelIsEnabled() 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.

26.2.2 Define Documentation

26.2.2.1 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.

26.2.3 Function Documentation

26.2.3.1 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_tttt** 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.

26.2.3.2 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.

26.2.3.3 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.

26.2.3.4 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.

26.2.3.5 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.

26.2.3.6 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.

26.2.3.7 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.

26.2.3.8 uDMAChannelIsEnabled

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.

26.2.3.9 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**.

26.2.3.10 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.

26.2.3.11 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:

uDMATaskStructEntry

Returns:

None.

26.2.3.12 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_EPI0RX**
- **UDMA_DEF_TMR1B_SEC_EPI0TX**
- **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.

26.2.3.13 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_EPI0RX**
- **UDMA_DEF_TMR1B_SEC_EPI0TX**
- **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.

26.2.3.14 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.

26.2.3.15 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 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.

26.2.3.16 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.

26.2.3.17 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.

26.2.3.18 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.

26.2.3.19 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.

26.2.3.20 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.

26.2.3.21 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.

26.2.3.22 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.

26.2.3.23 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 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.

Returns:

None.

26.2.3.24 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.

26.2.3.25 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.

26.2.3.26 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.

26.3 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);
```

27 USB Controller

27.1 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, USBEndpoint, and USBFIFO. The APIs in the USBDev group are only used with microcontrollers that have a USB device controller. The APIs in the USBHost group can only be used with microcontrollers that have a USB host controller. The USBOTG APIs are used by microcontrollers with an OTG interface. With USB OTG controllers, once the mode of the USB controller is configured, the device or host APIs should be 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 while the USBFIFO APIs are used to configure the size and location of the FIFOs.

27.2 Using USB with the uDMA Controller

The USB controller can be used with the uDMA 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 USBDevEndpointConfigSet() 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** should be 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 should call DMAChannelAttributeClear(). Then the application should call 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** should be 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 should normally be 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 could also have been caused 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.
    //
    ...
}
```

27.3 API Functions

Functions

- 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 *pui32MaxPacketSize, uint32_t *pui32Flags)
- 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 USBEndpointDataAvail (uint32_t ui32Base, uint32_t ui32Endpoint)
- int32_t USBEndpointDataGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t *pui8Data, uint32_t *pui32Size)

- 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)
- 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 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)
- 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)

27.3.1 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 and IN and OUT transactions. For example: In device mode, endpoint 1 could 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 could 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 three, seven, or fifteen IN endpoints and three, seven, or fifteen OUT endpoints, depending on the total number of endpoints on the Tiva device.

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);
```

27.3.2 Function Documentation

27.3.2.1 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.

27.3.2.2 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.

27.3.2.3 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 USBDevDisconnect() to remove the USB device from the bus.

Note:
This function must only be called in device mode.

Returns:
None.

27.3.2.4 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.

27.3.2.5 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.

27.3.2.6 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 given 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" section for more information on DMA configuration.

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 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 in DMA mode.

Note:
This function must only be called in device mode.

Returns:
None.

27.3.2.7 USBDevEndpointDataAck

Acknowledge that data was read from the given 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.

27.3.2.8 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.

27.3.2.9 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.

27.3.2.10 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.

27.3.2.11 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 and have the DEVMODOTG bit in the USBGPCS register.

Returns:
None.

27.3.2.12 USBEndpointDataAvail

Determine the number of bytes of data available in a given 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 given 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 given endpoint FIFO.

27.3.2.13 USBEndpointDataGet

Retrieves data from the given 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 given 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.

27.3.2.14 USBEndpointDataPut

Puts data into the given 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.

27.3.2.15 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 given 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.

27.3.2.16 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.

27.3.2.17 USBEndpointDMAChannel

Sets the DMA channel to use for a given 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 given 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.

27.3.2.18 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 given 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.

27.3.2.19 USBEndpointDMADisable

Disable DMA on a given 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 given 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.

27.3.2.20 USBEndpointDMAEnable

Enable DMA on a given 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 given 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.

27.3.2.21 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.

27.3.2.22 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 given 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 given 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.

27.3.2.23 USBFIFOAddrGet

Returns the absolute FIFO address for a given 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 given endpoint.

Returns:

None.

27.3.2.24 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 given 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.

27.3.2.25 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 given 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.

27.3.2.26 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.

27.3.2.27 USBFrameNumberGet

Get 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.

27.3.2.28 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.

27.3.2.29 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.

27.3.2.30 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 the 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 enable the type of DMA used to access the endpoint's 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" section for more information on DMA configuration.

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().

Note:
This function must only be called in host mode.

Returns:
None.

27.3.2.31 USBHostEndpointDataAck

Acknowledge that data was read from the given 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.

27.3.2.32 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.

27.3.2.33 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.

27.3.2.34 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.

27.3.2.35 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.

27.3.2.36 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 and have the DEVMODOTG bit in the USBGPCS register.

Returns:
None.

27.3.2.37 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.

On devices that support 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.

Returns:
None.

27.3.2.38 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.

27.3.2.39 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.

27.3.2.40 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.

27.3.2.41 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.

27.3.2.42 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.

27.3.2.43 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.

27.3.2.44 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.

Returns:
None.

27.3.2.45 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.

27.3.2.46 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.

Returns:

None.

27.3.2.47 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**, or **USB_UNDEF_SPEED**.

27.3.2.48 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.

27.3.2.49 USBIntDisableControl

Disables control interrupts on a given 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.

27.3.2.50 USBIntDisableEndpoint

Disables endpoint interrupts on a given 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.

27.3.2.51 USBIntEnableControl

Enables control interrupts on a given 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.

27.3.2.52 USBIntEnableEndpoint

Enables endpoint interrupts on a given 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.

27.3.2.53 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.

27.3.2.54 USBIntStatusControl

Returns the control interrupt status on a given 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.

27.3.2.55 USBIntStatusEndpoint

Returns the endpoint interrupt status on a given 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.

27.3.2.56 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.

27.3.2.57 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**, **USB_OTG_MODE_ASIDE_DEV**, **USB_OTG_MODE_BSIDE_HOST**, **USB_OTG_MODE_BSIDE_DEV**, **USB_OTG_MODE_NONE**.

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**, **USB_DUAL_MODE_DEVICE**, or **USB_DUAL_MODE_NONE**.

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**.

27.3.2.58 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.

27.3.2.59 USBOTGMode

Change 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.

27.3.2.60 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.

27.3.2.61 USBPHYPowerOff

Powers off the USB PHY.

Prototype:

```
void
USBPHYPowerOff(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function powers off the USB PHY, reducing the current consuption of the device. While in the powered-off state, the USB controller is unable to operate.

Returns:

None.

27.3.2.62 USBPHYPowerOn

Powers on the USB PHY.

Prototype:

```
void
USBPHYPowerOn(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function powers on the 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.

27.4 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 USBFIFOConfig() 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.
//
USBFIFOConfig(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);
```

28 Watchdog Timer

28.1 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 definitions for use by applications.

28.2 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)

28.2.1 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().

28.2.2 Function Documentation

28.2.2.1 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.

See also:

WatchdogLock(), WatchdogUnlock()

Returns:

None.

28.2.2.2 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).

Returns:

None.

28.2.2.3 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.

See also:

WatchdogLock(), WatchdogUnlock(), WatchdogEnable()

Returns:

None.

28.2.2.4 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.

28.2.2.5 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.

28.2.2.6 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.

Returns:

None.

28.2.2.7 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.

28.2.2.8 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 configuration registers.

Returns:
None.

28.2.2.9 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.

28.2.2.10 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.

See also:
WatchdogReloadSet()

Returns:
None.

28.2.2.11 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.

See also:
WatchdogLock(), WatchdogUnlock(), WatchdogReloadGet()

Returns:
None.

28.2.2.12 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.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

28.2.2.13 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.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

28.2.2.14 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.

28.2.2.15 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.

Returns:

None.

28.2.2.16 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).

Returns:

None.

28.2.2.17 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 configuration registers.

Returns:
None.

28.2.2.18 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.

28.3 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, 0xFEEFEE);

//
// Enable the reset.
//
WatchdogResetEnable(WATCHDOG0_BASE);

//
// Enable the watchdog timer.
//
WatchdogEnable(WATCHDOG0_BASE);

//
// Wait for the reset to occur.
//
while(1)
```

```
{
}
```

29 Using the ROM

29.1 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.

29.2 Direct ROM Calls

In order to call the ROM, the following steps must be performed:

- The device on which the application will be 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 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_DUSTDEVIL_RA0`	The application is being built to run on a DustDevil-class device, silicon revision A0.
`TARGET_IS_TEMPEST_RB1`	The application is being built to run on a Tempest-class device, silicon revision B1.
`TARGET_IS_TEMPEST_RC1`	The application is being built to run on a Tempest-class device, silicon revision C1.
`TARGET_IS_TEMPEST_RC3`	The application is being built to run on a Tempest-class device, silicon revision C3.
`TARGET_IS_TEMPEST_RC5`	The application is being built to run on a Tempest-class device, silicon revision C5.

TARGET_IS_FIRESTORM_RA2	The application is being built to run on a Firestorm-class device, silicon revision A2.
TARGET_IS_BLIZZARD_RA1	The application is being built to run on a Blizzard-class device, silicon revision A1.

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_DUSTDEVIL_RA0
#include "driverlib/rom.h"
#include "driverlib/systick.h"

int
main(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();

    // ...
}
```

29.3 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();
}
```

29.4 Firmware Update

Functions

- void ROM_UpdateI2C (void)
- void ROM_UpdateSSI (void)
- void ROM_UpdateUART (void)

29.4.1 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.

29.4.2 Function Documentation

29.4.2.1 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.

29.4.2.2 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.

29.4.2.3 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.

30 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		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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