LUMINARY MICRO™

Stellaris™ Driver Library

USER'S GUIDE

SW02034uvision-852

Legal Disclaimers and Trademark Information

Luminary Micro, Inc.
2499 South Capital of Texas Hwy, Suite A-100
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Revision Information

This is version 852 of this document, last updated on September 15, 2006.

Table of Contents

1 Introduction

The Stellaris Driver Library is a set of drivers for accessing the peripherals found on the Stellaris family of ARM® Cortex™-M3 microprocessors. While they are not drivers in the pure operating system sense (i.e. 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.

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 can not 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. Since 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.

2 Analog Comparator

2.1 Introduction

The comparator API provides a set of functions for dealing with the analog comparators. A comparator can compare a test voltage against 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 cause it to start capturing a sample sequence. The interrupt generation and ADC triggering logic is separate, so that an interrupt can be generated on a rising edge and the ADC triggered on a falling edge (for example).

2.2 API Functions

Functions

- void ComparatorConfigure (unsigned long ulBase, unsigned long ulComp, unsigned long ulConfig)
- void ComparatorIntClear (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntDisable (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntEnable (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntRegister (unsigned long ulBase, unsigned long ulComp, void(*pfnHandler)(void))
- tBoolean ComparatorIntStatus (unsigned long ulBase, unsigned long ulComp, tBoolean bMasked)
- void ComparatorIntUnregister (unsigned long ulBase, unsigned long ulComp)
- void ComparatorRefSet (unsigned long ulBase, unsigned long ulRef)
- tBoolean ComparatorValueGet (unsigned long ulBase, unsigned long ulComp)

2.2.1 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()).

2.2.2 Function Documentation

2.2.2.1 ComparatorConfigure

Configures a comparator.

Prototype:

```
void
ComparatorConfigure(unsigned long ulBase,
                    unsigned long ulComp,
                    unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator to configure.

ulConfig is the configuration of the comparator.

Description:

This function will configure a comparator. The *ulConfig* 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_NONE** to disable the output from the comparator to a device pin.
- **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.

2.2.2.2 ComparatorIntClear

Clears a comparator interrupt.

Prototype:
```
void
ComparatorIntClear(unsigned long ulBase,
                   unsigned long ulComp)
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

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

Returns:
None.

2.2.2.3 ComparatorIntDisable

Disables the comparator interrupt.

Prototype:
```
void
ComparatorIntDisable(unsigned long ulBase,
                     unsigned long ulComp)
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

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

Returns:
None.

2.2.2.4 ComparatorIntEnable

Enables the comparator interrupt.

Prototype:

```
void
ComparatorIntEnable(unsigned long ulBase,
                    unsigned long ulComp)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

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

Returns:

None.

2.2.2.5 ComparatorIntRegister

Registers an interrupt.handler for the comparator interrupt.

Prototype:

```
void
ComparatorIntRegister(unsigned long ulBase,
                      unsigned long ulComp,
                      void(*)(void) pfnHandler)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

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

Description:

This sets the handler to be called when the comparator interrupt occurs. This will enable 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.

2.2.2.6 ComparatorIntStatus

Gets the current interrupt status.

Prototype:

```
tBoolean
ComparatorIntStatus(unsigned long ulBase,
                    unsigned long ulComp,
                    tBoolean bMasked)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp 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 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.

2.2.2.7 ComparatorIntUnregister

Unregisters an interrupt.handler for a comparator interrupt.

Prototype:

```
void
ComparatorIntUnregister(unsigned long ulBase,
                        unsigned long ulComp)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

This function will clear the handler to be called when a comparator interrupt occurs. This will also mask 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.

2.2.2.8 ComparatorRefSet

Sets the internal reference voltage.

Prototype:

```
void
ComparatorRefSet(unsigned long ulBase,
                 unsigned long ulRef)
```

Parameters:

ulBase is the base address of the comparator module.

ulRef is the desired reference voltage.

Description:

This function will set 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.

2.2.2.9 ComparatorValueGet

Gets the current comparator output value.

Prototype:

```
tBoolean
ComparatorValueGet(unsigned long ulBase,
                   unsigned long ulComp)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp 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.

2.3 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 a comparator.
//
ComparatorConfigure(COMP_BASE, 0,
                    (COMP_TRIG_NONE | COMP_INT_BOTH |
                     COMP_ASRCP_REF | COMP_OUTPUT_NONE));

//
// Delay for some time...
//

//
// Read the comparator output value.
//
ComparatorValueGet(COMP_BASE, 0);
```

3 Analog to Digital Converter

3.1 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.

The ADC supports up to eight input channels plus an internal temperature sensor. Four sampling sequences, each with configurable trigger events, can be captured. The first sequence will capture up to eight samples, the second and third sequences will capture up to four samples, and the fourth sequence will capture a single sample. Each sample can be the same channel, different channels, or any combination in any order.

The sample sequences have configurable priorities that determine the order in which they are captured when multiple triggers occur simultaneously. The highest priority sequence that is currently triggered will be sampled. 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 sequences.

Software oversampling of the ADC data is available for improved accuracy. An oversampling factor of 2x, 4x, and 8x is supported, but reduces the depth of the sample sequences by a corresponding amount. For example, the first sample sequence will capture eight samples; in 4x oversampling mode it can only capture two samples since the first four samples are used over the first oversampled value and the second four samples are used for the second oversampled value.

A more sophisticated software oversampling can be used to eliminate the reduction of the sample sequence 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 sequence capability. In this case, an increase in the number of ADC triggers (and presumably ADC interrupts) is the consequence. Since this requires adjustments outside of the ADC driver itself, this 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.

3.2 API Functions

Functions

- void ADCIntClear (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntDisable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntEnable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntRegister (unsigned long ulBase, unsigned long ulSequenceNum, void(*pfnHandler)(void))
- unsigned long ADCIntStatus (unsigned long ulBase, unsigned long ulSequenceNum, tBoolean bMasked)

- void ADCIntUnregister (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCProcessorTrigger (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)
- long ADCSequenceDataGet (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long *pulBuffer)
- void ADCSequenceDisable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceEnable (unsigned long ulBase, unsigned long ulSequenceNum)
- long ADCSequenceOverflow (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
- long ADCSequenceUnderflow (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSoftwareOversampleConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulFactor)
- void ADCSoftwareOversampleDataGet (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long *pulBuffer, unsigned long ulCount)
- void ADCSoftwareOversampleStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)

3.2.1 Detailed Description

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

The sample sequences are configured with ADCSequenceConfigure() and ADCSequenceStepConfigure(). They are enabled and disabled with ADCSequenceEnable() and ADCSequenceDisable(). The captured data is obtained with ADCSequenceDataGet(), ADCSequenceOverflow(), and ADCSequenceUnderflow().

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 sequence interrupts are managed with ADCIntRegister() and ADCIntUnregister(). The sample sequence interrupt sources are managed with ADCIntDisable(), ADCIntEnable(), ADCIntStatus(), and ADCIntClear().

3.2.2 Function Documentation

3.2.2.1 ADCIntClear

Clears sample sequence interrupt source.

Prototype:

```
void
ADCIntClear(unsigned long ulBase,
            unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

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

Returns:

None.

3.2.2.2 ADCIntDisable

Disables a sample sequence interrupt.

Prototype:

```
void
ADCIntDisable(unsigned long ulBase,
              unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This function disables the requested sample sequence interrupt.

Returns:

None.

3.2.2.3 ADCIntEnable

Enables a sample sequence interrupt.

Prototype:

```
void
ADCIntEnable(unsigned long ulBase,
             unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum 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.

3.2.2.4 ADCIntRegister

Registers an interrupt.handler for an ADC interrupt.

Prototype:

```
void
ADCIntRegister(unsigned long ulBase,
               unsigned long ulSequenceNum,
               void(*)(void) pfnHandler)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum 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 will enable 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.

3.2.2.5 ADCIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
ADCIntStatus(unsigned long ulBase,
             unsigned long ulSequenceNum,
             tBoolean bMasked)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum 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 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.

3.2.2.6 ADCIntUnregister

Unregisters the interrupt.handler for an ADC interrupt.

Prototype:

```
void
ADCIntUnregister(unsigned long ulBase,
                 unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This function unregisters the interrupt.handler. This will disable 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.

3.2.2.7 ADCProcessorTrigger

Causes a processor trigger for a sample sequence.

Prototype:

```
void
ADCProcessorTrigger(unsigned long ulBase,
                    unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to ADC_TRIGGER_PROCESSOR.

Returns:

None.

3.2.2.8 ADCSequenceConfigure

Configures the trigger source and priority of a sample sequence.

Prototype:

```
void
ADCSequenceConfigure(unsigned long ulBase,
                     unsigned long ulSequenceNum,
                     unsigned long ulTrigger,
                     unsigned long ulPriority)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulTrigger is the trigger source that initiates the sample sequence; must be one of the **ADC_-TRIGGER_*** values.

ulPriority 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 sequences range from zero to three; sequence zero will capture up to eight samples, sequences one and two will capture up to four samples, and sequence three will capture a single sample. The trigger condition and priority (with respect to other sample sequence execution) is set.

The parameter **ulTrigger** 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.
- **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_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 Stellaris family members; consult the data sheet for the device in question to determine the availability of triggers.

The parameter **ulPriority** 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.

3.2.2.9 ADCSequenceDataGet

Gets the captured data for a sample sequence.

Prototype:

```
long
ADCSequenceDataGet(unsigned long ulBase,
                   unsigned long ulSequenceNum,
                   unsigned long *pulBuffer)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

pulBuffer is the address where the data is stored.

Description:

This function copies data from the specified sample sequence 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 will only return 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.

3.2.2.10 ADCSequenceDisable

Disables a sample sequence.

Prototype:

```
void
ADCSequenceDisable(unsigned long ulBase,
                   unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum 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.

3.2.2.11 ADCSequenceEnable

Enables a sample sequence.

Prototype:

```
void
ADCSequenceEnable(unsigned long ulBase,
                  unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum 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.

3.2.2.12 ADCSequenceOverflow

Determines if a sample sequence overflow occurred.

Prototype:

```
long
ADCSequenceOverflow(unsigned long ulBase,
                    unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This determines if a sample sequence overflow has occurred. This will happen 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.

3.2.2.13 ADCSequenceStepConfigure

Configure a step of the sample sequencer.

Prototype:

```
void
ADCSequenceStepConfigure(unsigned long ulBase,
                         unsigned long ulSequenceNum,
                         unsigned long ulStep,
                         unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulStep is the step to be configured.

ulConfig is the configuration of this step; must be a logical OR of **ADC_CTL_TS**, **ADC_CTL_-IE**, **ADC_CTL_END**, **ADC_CTL_D**, and one of the input channel selects (**ADC_CTL_CH0** through **ADC_CTL_CH7**).

Description:

This function will set the configuration of 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_CH7** 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). The configuration is used by the ADC at the appropriate time when the trigger for this sequence occurs.

The **ulStep** 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 sequence, from zero to three for the second and third sample sequence, and can only be zero for the fourth sample sequence.

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

3.2.2.14 ADCSequenceUnderflow

Determines if a sample sequence underflow occurred.

Prototype:

```
long
ADCSequenceUnderflow(unsigned long ulBase,
                     unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This determines if a sample sequence underflow has occurred. This will happen if too many samples are read from the FIFO.

Returns:

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

3.2.2.15 ADCSoftwareOversampleConfigure

Configures the software oversampling factor of the ADC.

Prototype:

```
void
ADCSoftwareOversampleConfigure(unsigned long ulBase,
                               unsigned long ulSequenceNum,
                               unsigned long ulFactor)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulFactor 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. 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 (i.e. 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.

3.2.2.16 ADCSoftwareOversampleDataGet

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

Prototype:

```
void
ADCSoftwareOversampleDataGet(unsigned long ulBase,
                             unsigned long ulSequenceNum,
                             unsigned long *pulBuffer,
                             unsigned long ulCount)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

pulBuffer is the address where the data is stored.

ulCount 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 will be 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.

3.2.2.17 ADCSoftwareOversampleStepConfigure

Configures a step of the software oversampled sequencer.

Prototype:

```
void
ADCSoftwareOversampleStepConfigure(unsigned long ulBase,
                                   unsigned long ulSequenceNum,
                                   unsigned long ulStep,
                                   unsigned long ulConfig)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.
ulStep is the step to be configured.
ulConfig 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 *ulConfig* is the same as defined for ADCSequenceStepConfigure().

Returns:
None.

3.3 Programming Example

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

```
unsigned long ulValue;

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

//
// Trigger the sample sequence.
//
ADCProcessorTrigger(ADC_BASE, 0);

//
// Wait until the sample sequence has completed.
//
```

```
while(!ADCIntStatus(ADC_BASE, 0, false))
{
}

//
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC_BASE, 0, &ulValue);
```

4 Flash

4.1 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, 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. 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 can be used to validate the operation of a program; the interrupt will keep invalid accesses from being 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). An interrupt can also be generated when an erase or programming operation has completed.

Depending upon the member of the Stellaris family used, the amount of available flash is 8 kB, 16 kB, 32 kB, or 64 kB.

4.2 API Functions

Functions

- long FlashErase (unsigned long ulAddress)
- void FlashIntClear (unsigned long ulIntFlags)
- void FlashIntDisable (unsigned long ulIntFlags)
- void FlashIntEnable (unsigned long ulIntFlags)
- unsigned long FlashIntGetStatus (tBoolean bMasked)
- void FlashIntRegister (void(∗pfnHandler)(void))

- void FlashIntUnregister (void)
- long FlashProgram (unsigned long *pulData, unsigned long ulAddress, unsigned long ulCount)
- tFlashProtection FlashProtectGet (unsigned long ulAddress)
- long FlashProtectSave (void)
- long FlashProtectSet (unsigned long ulAddress, tFlashProtection eProtect)
- unsigned long FlashUsecGet (void)
- void FlashUsecSet (unsigned long ulClocks)

4.2.1 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().

4.2.2 Function Documentation

4.2.2.1 FlashErase

Erases a block of flash.

Prototype:

```
long
FlashErase(unsigned long ulAddress)
```

Parameters:

ulAddress is the start address of the flash block to be erased.

Description:

This function will erase a 1 kB block of the on-chip flash. After erasing, the block will be filled with 0xFF bytes. Read-only and execute-only blocks cannot be erased.

This function will 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.

4.2.2.2 FlashIntClear

Clears flash controller interrupt sources.

Prototype:

```
void
FlashIntClear(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is the bit mask of the interrupt sources to be cleared. Can be any of the **FLASH_-FCMISC_PROGRAM** or **FLASH_FCMISC_ACCESS** values.

Description:

The specified flash controller interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt.handler to keep it from being called again immediately upon exit.

Returns:

None.

4.2.2.3 FlashIntDisable

Disables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntDisable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is a bit mask of the interrupt sources to be disabled. Can be any of the **FLASH_-FCIM_PROGRAM** or **FLASH_FCIM_ACCESS** values.

Description:

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.

4.2.2.4 FlashIntEnable

Enables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntEnable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **FLASH_-FCIM_PROGRAM** or **FLASH_FCIM_ACCESS** values.

Description:

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.

4.2.2.5 FlashIntGetStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
FlashIntGetStatus(tBoolean bMasked)
```

Parameters:

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

Description:

This 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_FCMISC_PROGRAM** and **FLASH_FCMISC_ACCESS**.

4.2.2.6 FlashIntRegister

Registers an interrupt.handler for the flash interrupt.

Prototype:

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

Parameters:

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

Description:

This 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 will be automatically enabled when the handler is registered.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

4.2.2.7 FlashIntUnregister

Unregisters the interrupt.handler for the flash interrupt.

Prototype:

```
void
FlashIntUnregister(void)
```

Description:

This function will clear the handler to be called when the flash interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt.handler is no longer called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

4.2.2.8 FlashProgram

Programs flash.

Prototype:

```
long
FlashProgram(unsigned long *pulData,
             unsigned long ulAddress,
             unsigned long ulCount)
```

Parameters:

pulData is a pointer to the data to be programmed.

ulAddress is the starting address in flash to be programmed. Must be a multiple of four.

ulCount is the number of bytes to be programmed. Must be a multiple of four.

Description:

This function will program a sequence of words into the on-chip flash. Programming each location consists of the result of an AND operation of the new data and the existing data; in other words bits that contain 1 can remain 1 or be changed to 0, but bits that are 0 cannot be changed to 1. Therefore, a word can be programmed multiple times as long as these rules are followed; if a program operation attempts to change a 0 bit to a 1 bit, that bit will not have its value changed.

Since 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 will not return until the data has been programmed.

Returns:

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

4.2.2.9 FlashProtectGet

Gets the protection setting for a block of flash.

Prototype:

```
tFlashProtection
FlashProtectGet(unsigned long ulAddress)
```

Parameters:

ulAddress is the start address of the flash block to be queried.

Description:

This function will get 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.

4.2.2.10 FlashProtectSave

Saves the flash protection settings.

Prototype:

```
long
FlashProtectSave(void)
```

Description:

This function will make the currently programmed flash protection settings permanent. This is a non-reversible operation; a chip reset or power cycle will not change the flash protection.

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

Returns:

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

4.2.2.11 FlashProtectSet

Sets the protection setting for a block of flash.

Prototype:

```
long
FlashProtectSet(unsigned long ulAddress,
                tFlashProtection eProtect)
```

Parameters:

ulAddress 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 will set the protection for the specified 2 kB block of flash. Blocks which are read/write can be made read-only or execute-only. Blocks which are read-only can be made execute-only. Blocks which are execute-only cannot have their protection modified. Attempts to make the block protection less stringent (i.e. read-only to read/write) will result in a failure (and be prevented by the hardware).

Changes to the flash protection are maintained only until the next reset. This 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.

4.2.2.12 FlashUsecGet

Gets the number of processor clocks per micro-second.

Prototype:
```
unsigned long
FlashUsecGet(void)
```

Description:
This function returns the number of clocks per micro-second, as presently known by the flash controller.

Returns:
Returns the number of processor clocks per micro-second.

4.2.2.13 FlashUsecSet

Sets the number of processor clocks per micro-second.

Prototype:
```
void
FlashUsecSet(unsigned long ulClocks)
```

Parameters:
ulClocks is the number of processor clocks per micro-second.

Description:
This function is used to tell the flash controller the number of processor clocks per micro-second. This value must be programmed correctly or the flash most likely will not program correctly; it has no affect on reading flash.

Returns:
None.

4.3 Programming Example

The following example shows how to use the flash API to erase a block of the flash and program a few words.

```
unsigned long pulData[2];

//
// Set the uSec value to 20, indicating that the processor is running at
// 20 MHz.
//
FlashUsecSet(20);
```

```
//
// Erase a block of the flash.
//
FlashErase(0x800);

//
// Program some data into the newly erased block of the flash.
//
pulData[0] = 0x12345678;
pulData[1] = 0x56789abc;
FlashProgram(pulData, 0x800, sizeof(pulData));
```

5 GPIO

5.1 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.
- 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 (i.e. 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; the GPIO pins whose corresponding bits in this parameter that are set will be affected (where pin 0 is in bit 0, pin 1 in bit 1, etc.). For example, if *ucPins* is 0x09, then pins 0 and 3 will be affected by the function.

This is most useful for the GPIOPinRead() and GPIOPinWrite() functions; a read will return only the value of the requested pins (with the other pin values masked out) and a write will affect the requested pins simultaneously (i.e. 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, this value specifies the pin number (i.e. 0 through 7).

5.2 API Functions

Functions

- unsigned long GPIODirModeGet (unsigned long ulPort, unsigned char ucPin)
- void GPIODirModeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)

- unsigned long GPIOIntTypeGet (unsigned long ulPort, unsigned char ucPin)
- void GPIOIntTypeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)
- void GPIOPadConfigGet (unsigned long ulPort, unsigned char ucPin, unsigned long ∗pulStrength, unsigned long ∗pulPinType)
- void GPIOPadConfigSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPinType)
- void GPIOPinIntClear (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinIntDisable (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinIntEnable (unsigned long ulPort, unsigned char ucPins)
- long GPIOPinIntStatus (unsigned long ulPort, tBoolean bMasked)
- long GPIOPinRead (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeComparator (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeI2C (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypePWM (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeQEI (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeSSI (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeTimer (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeUART (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinWrite (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
- void GPIOPortIntRegister (unsigned long ulPort, void(∗pfIntHandler)(void))
- void GPIOPortIntUnregister (unsigned long ulPort)

5.2.1 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() and GPIOPadConfigSet(). 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 GPIOPinTypeComparator(), GPIOPinTypeI2C(), GPIOPinTypePWM(), GPIOPinTypeQEI(), GPIOPinTypeSSI(), GPIOPinTypeTimer(), and GPIOPinTypeUART().

The GPIO interrupts are handled with GPIOIntTypeSet(), GPIOIntTypeGet(), GPIOPinIntEnable(), GPIOPinIntDisable(), GPIOPinIntStatus(), GPIOPinIntClear(), GPIOPortIntRegister(), and GPIOPortIntUnregister().

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

5.2.2 Function Documentation

5.2.2.1 GPIODirModeGet

Gets the direction and mode of a specified pin of the selected GPIO port.

Prototype:

```
unsigned long
GPIODirModeGet(unsigned long ulPort,
               unsigned char ucPin)
```

Parameters:

ulPort base address of the selected GPIO port

ucPin pin number of the specified pin, relative to the selected GPIO port.

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().

5.2.2.2 GPIODirModeSet

Sets the direction and mode of the specified pins of the selected GPIO port.

Prototype:

```
void
GPIODirModeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulPinIO)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

ulPinIO pin direction and/or mode

Description:

This function will set the specified pins on the selected GPIO port as either an input or output under software control, or it will set the pin to be under hardware control.

The parameter *ulPinIO* 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 will be programmed as a software controlled input, **GPIO_DIR_MODE_OUT** specifies that the pin will be programmed as a software controlled output, and **GPIO_DIR_MODE_HW** specifies that the pin will be placed under hardware control.

The pins 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, etc.

Returns:

None.

5.2.2.3 GPIOIntTypeGet

Gets the interrupt type for the specified pin of the selected GPIO port.

Prototype:

```
unsigned long
GPIOIntTypeGet(unsigned long ulPort,
               unsigned char ucPin)
```

Parameters:

ulPort base address of the selected GPIO port

ucPin pin number of the specified pin, relative to the selected GPIO port.

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 edge detected interrupt, or it can be configured as a low level or high level detected interrupt. The type of interrupt detection mechanism is returned as an enumerated data type.

Returns:

Returns one of the enumerated data types described for GPIOIntTypeSet().

5.2.2.4 GPIOIntTypeSet

Sets the interrupt type for the specified pins of the selected GPIO port.

Prototype:

```
void
GPIOIntTypeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulIntType)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

ulIntType specifies the type of interrupt trigger mechanism

Description:

This function sets up the various interrupt trigger mechanisms for the specified pins on the selected GPIO port.

The parameter *ulIntType* is an enumerated data type that can be one of the following values:

- **GPIO_FALLING_EDGE**
- **GPIO_RISING_EDGE**
- **GPIO_BOTH_EDGES**
- **GPIO_LOW_LEVEL**
- **GPIO_HIGH_LEVEL**

where the different values describe the interrupt detection mechanism (edge or level) and the particular triggering event (falling, rising, or both edges for edge detect, low or high for level detect).

The pins 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, etc.

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.

5.2.2.5 GPIOPadConfigGet

Gets the pad configuration for the specified pin of the selected GPIO port.

Prototype:
```
void
GPIOPadConfigGet(unsigned long ulPort,
                 unsigned char ucPin,
                 unsigned long *pulStrength,
                 unsigned long *pulPinType)
```

Parameters:
ulPort base address of the selected GPIO port
ucPin pin number of the specified pin, relative to the selected GPIO port.
pulStrength pointer to storage for the output drive strength
pulPinType 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 *eStrength* and *eOutType* correspond to the values used in GPIOPadConfig-Set(). This function also works for pins configured as input pins; however, the only meaningful data returned is whether the pin is terminated with a pull-up or down resistor.

Returns:
None

5.2.2.6 GPIOPadConfigSet

Sets the pad configuration for the specified pins of the selected GPIO port.

Prototype:
```
void
GPIOPadConfigSet(unsigned long ulPort,
                 unsigned char ucPins,
                 unsigned long ulStrength,
                 unsigned long ulPinType)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins bit-packed representation of the specified pins.
ulStrength specifies the output drive strength.
ulPinType specifies the pin type.

Description:

This function sets the drive strength and type for the specified pins on the selected GPIO port. For pins 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 *ulStrength* 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 *ulPinType* 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_OD_WPU**
- **GPIO_PIN_TYPE_OD_WPD**
- **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 (for the comparators).

The pins 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, etc.

Returns:

None.

5.2.2.7 GPIOPinIntClear

Clears the interrupt for the specified pins of the selected GPIO port.

Prototype:

```
void
GPIOPinIntClear(unsigned long ulPort,
                unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

Description:

Clears the interrupt for the specified pins.

The pins 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, etc.

Returns:
None.

5.2.2.8 GPIOPinIntDisable

Disables interrupts for the specified pins of the selected GPIO port.

Prototype:
```
void
GPIOPinIntDisable(unsigned long ulPort,
                  unsigned char ucPins)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins

Description:
Masks the interrupt for the specified pins.

The pins 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, etc.

Returns:
None.

5.2.2.9 GPIOPinIntEnable

Enables interrupts for the specified pins of the selected GPIO port.

Prototype:
```
void
GPIOPinIntEnable(unsigned long ulPort,
                 unsigned char ucPins)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins

Description:
Unmasks the interrupt for the specified pins.

The pins 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, etc.

Returns:
None.

5.2.2.10 GPIOPinIntStatus

Gets interrupt status for all the pins of the selected GPIO port.

Prototype:

```
long
GPIOPinIntStatus(unsigned long ulPort,
                 tBoolean bMasked)
```

Parameters:

ulPort base address of the selected 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 will be returned.

Returns:

Returns a bit-packed byte, where each bit that is set identifies an active masked or raw interrupt, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, etc. Bits 31:8 should be ignored.

5.2.2.11 GPIOPinRead

Reads the values present at the specified pins of the selected GPIO port.

Prototype:

```
long
GPIOPinRead(unsigned long ulPort,
            unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

Description:

The values at the specified pins are read, as specified by *ucPins*. Values are returned for both input and output pins, and the value for pins that are not specified by *ucPins* are set to 0.

The pins 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, etc.

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, etc. Any bit that is not specified by *ucPins* is returned as a 0. Bits 31:8 should be ignored.

5.2.2.12 GPIOPinTypeComparator

Configures pin(s) for use as an analog comparator input.

Prototype:

```
void
GPIOPinTypeComparator(unsigned long ulPort,
                      unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

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 pins.

Note:

This cannot be used to turn any pin into an analog comparator input; it only configures an analog comparator pin for proper operation.

Returns:

None.

5.2.2.13 GPIOPinTypeI2C

Configures pin(s) for use by the I2C peripheral.

Prototype:

```
void
GPIOPinTypeI2C(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

Description:

The I2C pins must be properly configured for the I2C peripheral to function correctly. This function provides the proper configuration for those pins.

Note:

This cannot be used to turn any pin into an I2C pin; it only configures an I2C pin for proper operation.

Returns:

None.

5.2.2.14 GPIOPinTypePWM

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

Prototype:

```
void
GPIOPinTypePWM(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins

Description:
The PWM pins must be properly configured for the PWM peripheral to function correctly. This function provides a typical configuration for those pins; other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

Note:
This cannot be used to turn any pin into a PWM pin; it only configures a PWM pin for proper operation.

Returns:
None.

5.2.2.15 GPIOPinTypeQEI

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

Prototype:
```
void
GPIOPinTypeQEI(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins

Description:
The QEI pins must be properly configured for the QEI peripheral to function correctly. This function provides a typical configuration for those pins; other configurations may work as well depending upon the board setup (for example, not using the on-chip pull-ups).

Note:
This cannot be used to turn any pin into a QEI pin; it only configures a QEI pin for proper operation.

Returns:
None.

5.2.2.16 GPIOPinTypeSSI

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

Prototype:
```
void
GPIOPinTypeSSI(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

Description:

The SSI pins must be properly configured for the SSI peripheral to function correctly. This function provides a typical configuration for those pins; other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

Note:

This cannot be used to turn any pin into a SSI pin; it only configures a SSI pin for proper operation.

Returns:

None.

5.2.2.17 GPIOPinTypeTimer

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

Prototype:

```
void
GPIOPinTypeTimer(unsigned long ulPort,
                 unsigned char ucPins)
```

Parameters:

ulPort base address of the selected GPIO port

ucPins bit-packed representation of the specified pins

Description:

The CCP pins must be properly configured for the timer peripheral to function correctly. This function provides a typical configuration for those pins; other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

Note:

This cannot be used to turn any pin into a timer pin; it only configures a timer pin for proper operation.

Returns:

None.

5.2.2.18 GPIOPinTypeUART

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

Prototype:

```
void
GPIOPinTypeUART(unsigned long ulPort,
                unsigned char ucPins)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins

Description:
The UART pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pins; other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

Note:
This cannot be used to turn any pin into a UART pin; it only configures a UART pin for proper operation.

Returns:
None.

5.2.2.19 GPIOPinWrite

Writes a value at the specified pins of the selected GPIO port.

Prototype:
```
void
GPIOPinWrite(unsigned long ulPort,
             unsigned char ucPins,
             unsigned char ucVal)
```

Parameters:
ulPort base address of the selected GPIO port
ucPins bit-packed representation of the specified pins
ucVal value to write to the specified pins

Description:
Writes the corresponding bit values to the output pins specified by *ucPins*. Writing to a pin configured as an input pin has no effect.

The pins 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, etc.

Returns:
None.

5.2.2.20 GPIOPortIntRegister

Registers an interrupt.handler for the selected GPIO port.

Prototype:
```
void
GPIOPortIntRegister(unsigned long ulPort,
                    void(*)(void) pfIntHandler)
```

Parameters:

ulPort base address of the selected GPIO port

pfIntHandler pointer to the GPIO port interrupt.handling function

Description:

This function will ensure that the interrupt.handler specified by *pfIntHandler* is called when an interrupt is detected from the selected GPIO port. This function will also enable the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with GPIOPinIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

5.2.2.21 GPIOPortIntUnregister

Removes an interrupt.handler for the selected GPIO port.

Prototype:

```
void
GPIOPortIntUnregister(unsigned long ulPort)
```

Parameters:

ulPort base address of the selected GPIO port

Description:

This function will unregister the interrupt.handler for the specified GPIO port. This function will also disable the corresponding GPIO port interrupt in the interrupt controller; individual GPIO interrupts and interrupt sources must be disabled with GPIOPinIntDisable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

5.3 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.

```
int iVal;

//
// Register the port-level interrupt handler. This handler is the
// first level interrupt handler for all the pin interrupts.
//
GPIOPortIntRegister(GPIO_PORTA_BASE, PortAIntHandler);
```

```
//
// Initialize the GPIO pin configuration.
//
// Set pins 2, 4, and 5 as input, SW controlled.
//
GPIODirModeSet(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5),
               GPIO_DIR_MODE_IN);

//
// Set pins 0 and 3 as output, SW controlled.
//
GPIODirModeSet(GPIO_PORTA_BASE, (GPIO_PIN_0 | GPIO_PIN_3),
               GPIO_DIR_MODE_OUT);

//
// 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.
//
iVal = 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 since 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),
             0xF4);

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

6 I2C

6.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the Stellaris 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 Stellaris 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 Stellaris I2C modules can operate at two speeds: Standard (100 kb/s) and Fast (400 kb/s).

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

6.1.1 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 I2CMasterInit(). That function will set the bus speed and enable 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 Stellaris 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 will result 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 will involve 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 I2CMaster-Control() 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 will occur when the master is no longer busy.

6.1.2 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 will enable the I2C slave module and initialize the slave's own address. After the initialization is complete, the user may poll the slave status using I2CSlave-Status() 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.

6.2 API Functions

Functions

- void I2CIntRegister (unsigned long ulBase, void($*$pfnHandler)(void))
- void I2CIntUnregister (unsigned long ulBase)
- tBoolean I2CMasterBusBusy (unsigned long ulBase)
- tBoolean I2CMasterBusy (unsigned long ulBase)
- void I2CMasterControl (unsigned long ulBase, unsigned long ulCmd)
- unsigned long I2CMasterDataGet (unsigned long ulBase)
- void I2CMasterDataPut (unsigned long ulBase, unsigned char ucData)
- void I2CMasterDisable (unsigned long ulBase)
- void I2CMasterEnable (unsigned long ulBase)
- unsigned long I2CMasterErr (unsigned long ulBase)
- void I2CMasterInit (unsigned long ulBase, tBoolean bFast)
- void I2CMasterIntClear (unsigned long ulBase)
- void I2CMasterIntDisable (unsigned long ulBase)
- void I2CMasterIntEnable (unsigned long ulBase)
- tBoolean I2CMasterIntStatus (unsigned long ulBase, tBoolean bMasked)
- void I2CMasterSlaveAddrSet (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)
- unsigned long I2CSlaveDataGet (unsigned long ulBase)
- void I2CSlaveDataPut (unsigned long ulBase, unsigned char ucData)

- void I2CSlaveDisable (unsigned long ulBase)
- void I2CSlaveEnable (unsigned long ulBase)
- void I2CSlaveInit (unsigned long ulBase, unsigned char ucSlaveAddr)
- void I2CSlaveIntClear (unsigned long ulBase)
- void I2CSlaveIntDisable (unsigned long ulBase)
- void I2CSlaveIntEnable (unsigned long ulBase)
- tBoolean I2CSlaveIntStatus (unsigned long ulBase, tBoolean bMasked)
- unsigned long I2CSlaveStatus (unsigned long ulBase)

6.2.1 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(), and I2CSlaveIntStatus() functions.

Status and initialization functions for the I2C modules are I2CMasterInit(), 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.

6.2.2 Function Documentation

6.2.2.1 I2CIntRegister

Registers an interrupt.handler for the I2C module

Prototype:
```
void
I2CIntRegister(unsigned long ulBase,
               void(*)(void) pfnHandler)
```

Parameters:

ulBase base address of the I2C module

pfnHandler is a pointer to the function to be called when the synchronous serial interface interrupt occurs.

Description:

This sets the handler to be called when an I2C interrupt occurs. This will enable 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.

6.2.2.2 I2CIntUnregister

Unregisters an interrupt.handler for the I2C module.

Prototype:
```
void
I2CIntUnregister(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C module

Description:
This function will clear the handler to be called when an I2C interrupt occurs. This will also mask 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.

6.2.2.3 I2CMasterBusBusy

Indicates whether or not the I2C bus is busy.

Prototype:
```
tBoolean
I2CMasterBusBusy(unsigned long ulBase)
```

Parameters:
ulBase 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**.

6.2.2.4 I2CMasterBusy

Indicates whether or not the I2C Master is busy.

Prototype:
```
tBoolean
I2CMasterBusy(unsigned long ulBase)
```

Parameters:

ulBase 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**.

6.2.2.5 I2CMasterControl

Controls the state of the I2C Master module.

Prototype:

```
void
I2CMasterControl(unsigned long ulBase,
                 unsigned long ulCmd)
```

Parameters:

ulBase base address of the I2C Master module

ulCmd 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 parameter *ucCmd* 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

Returns:

None.

6.2.2.6 I2CMasterDataGet

Receives a byte that has been sent to the I2C Master.

Prototype:

```
unsigned long
I2CMasterDataGet(unsigned long ulBase)
```

Parameters:

ulBase 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 unsigned long.

6.2.2.7 I2CMasterDataPut

Transmits a byte from the I2C Master.

Prototype:
```
void
I2CMasterDataPut(unsigned long ulBase,
                 unsigned char ucData)
```

Parameters:
ulBase base address of the I2C Master module
ucData data to be transmitted from the I2C Master

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

Returns:
None.

6.2.2.8 I2CMasterDisable

Disables the I2C master block.

Prototype:
```
void
I2CMasterDisable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Master module

Description:
This will disable operation of the I2C master block.

Returns:
None.

6.2.2.9 I2CMasterEnable

Enables the I2C Master block.

Prototype:
```
void
I2CMasterEnable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Master module

Description:
This will enable operation of the I2C Master block.

Returns:
None.

6.2.2.10 I2CMasterErr

Gets the error status of the I2C Master module.

Prototype:
```
unsigned long
I2CMasterErr(unsigned long ulBase)
```

Parameters:
ulBase 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. It returns one of the following values:

- I2C_MASTER_ERR_NONE
- I2C_MASTER_ERR_ADDR_ACK
- I2C_MASTER_ERR_DATA_ACK
- I2C_MASTER_ERR_ARB_LOST

Returns:
None.

6.2.2.11 I2CMasterInit

Initializes the I2C Master block.

Prototype:
```
void
I2CMasterInit(unsigned long ulBase,
              tBoolean bFast)
```

Parameters:
ulBase base address of the I2C Master module
bFast set up for fast data transfers

Description:
This function initializes operation of the I2C Master block. Upon successful initialization of the I2C block, this function will have set the bus speed for the master, and will have enabled the I2C Master block.

If the parameter *bFast* is **true**, then the master block will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

The I2C clocking is dependent upon the system clock rate returned by SysCtlClockGet(); if it does not return the correct system clock rate then the I2C clock rate will be incorrect.

Returns:
None.

6.2.2.12 I2CMasterIntClear

Clears I2C Master interrupt sources.

Prototype:
```
void
I2CMasterIntClear(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Master module

Description:
The I2C Master interrupt source is cleared, so that it no longer asserts. This must be done in the interrupt.handler to keep it from being called again immediately upon exit.

Returns:
None.

6.2.2.13 I2CMasterIntDisable

Disables the I2C Master interrupt.

Prototype:
```
void
I2CMasterIntDisable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Master module

Description:
Disables the I2C Master interrupt source.

Returns:
None.

6.2.2.14 I2CMasterIntEnable

Enables the I2C Master interrupt.

Prototype:
```
void
I2CMasterIntEnable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Master module

Description:
Enables the I2C Master interrupt source.

Returns:
None.

6.2.2.15 I2CMasterIntStatus

Gets the current I2C Master interrupt status.

Prototype:
```
tBoolean
I2CMasterIntStatus(unsigned long ulBase,
                   tBoolean bMasked)
```

Parameters:
ulBase 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 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.

6.2.2.16 I2CMasterSlaveAddrSet

Sets the address that the I2C Master will place on the bus.

Prototype:
```
void
I2CMasterSlaveAddrSet(unsigned long ulBase,
                      unsigned char ucSlaveAddr,
                      tBoolean bReceive)
```

Parameters:
ulBase base address of the I2C Master module
ucSlaveAddr 7-bit slave address
bReceive flag indicating the type of communication with the slave

Description:
This function will set the address that the I2C Master will place on the bus when initiating a transaction. When the parameter *bReceive* is set to **true**, the address will indicate that the I2C Master is initiating a read from the slave; otherwise the address will indicate that the I2C Master is initiating a write to the slave.

Returns:
None.

6.2.2.17 I2CSlaveDataGet

Receives a byte that has been sent to the I2C Slave.

Prototype:
```
unsigned long
I2CSlaveDataGet(unsigned long ulBase)
```

Parameters:
ulBase 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 unsigned long.

6.2.2.18 I2CSlaveDataPut

Transmits a byte from the I2C Slave.

Prototype:
```
void
I2CSlaveDataPut(unsigned long ulBase,
                unsigned char ucData)
```

Parameters:
ulBase base address of the I2C Slave module
ucData data to be transmitted from the I2C Slave

Description:
This function will place the supplied data into I2C Slave Data Register.

Returns:
None.

6.2.2.19 I2CSlaveDisable

Disables the I2C slave block.

Prototype:
```
void
I2CSlaveDisable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Slave module

Description:
This will disable operation of the I2C slave block.

Returns:
None.

6.2.2.20 I2CSlaveEnable

Enables the I2C Slave block.

Prototype:
```
void
I2CSlaveEnable(unsigned long ulBase)
```

Parameters:
ulBase base address of the I2C Slave module

Description:
This will enable operation of the I2C Slave block.

Returns:
None.

6.2.2.21 I2CSlaveInit

Initializes the I2C Slave block.

Prototype:
```
void
I2CSlaveInit(unsigned long ulBase,
             unsigned char ucSlaveAddr)
```

Parameters:
ulBase base address of the I2C Slave module
ucSlaveAddr 7-bit slave address

Description:
This function initializes operation of the I2C Slave block. Upon successful initialization of the I2C blocks, this function will have set the slave address and have enabled the I2C Slave block.

The parameter *ucSlaveAddr* is the value that will be compared against the slave address sent by an I2C master.

Returns:
None.

6.2.2.22 I2CSlaveIntClear

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClear(unsigned long ulBase)
```

Parameters:

ulBase base address of the I2C Slave module

Description:

The I2C Slave interrupt source is cleared, so that it no longer asserts. This must be done in the interrupt.handler to keep it from being called again immediately upon exit.

Returns:

None.

6.2.2.23 I2CSlaveIntDisable

Disables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntDisable(unsigned long ulBase)
```

Parameters:

ulBase base address of the I2C Slave module

Description:

Disables the I2C Slave interrupt source.

Returns:

None.

6.2.2.24 I2CSlaveIntEnable

Enables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntEnable(unsigned long ulBase)
```

Parameters:

ulBase base address of the I2C Slave module

Description:

Enables the I2C Slave interrupt source.

Returns:

None.

6.2.2.25 I2CSlaveIntStatus

Gets the current I2C Slave interrupt status.

Prototype:

```
tBoolean
I2CSlaveIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

Parameters:

ulBase 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 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.

6.2.2.26 I2CSlaveStatus

Gets the I2C Slave module status

Prototype:

```
unsigned long
I2CSlaveStatus(unsigned long ulBase)
```

Parameters:

ulBase base address of the I2C Slave module

Description:

This function will return the action requested from a master, if any. The possible values returned are:

- I2C_SLAVE_ACT_NONE
- I2C_SLAVE_ACT_RREQ
- I2C_SLAVE_ACT_TREQ

where I2C_SLAVE_ACT_NONE means that no action has been requested of the I2C Slave module, I2C_SLAVE_ACT_RREQ means that an I2C master has sent data to the I2C Slave module, and I2C_SLAVE_ACT_TREQ means that an I2C master has requested that the I2C Slave module send data.

Returns:

None.

6.3 Programming Example

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

```
//
// Initialize Master and Slave
//
I2CMasterInit(true);

//
// Specify slave address
//
I2CMasterSlaveAddrSet(0x3B, false);

//
// Place the character to be sent in the data register
//
I2CMasterDataPut('Q');

//
// Initiate send of character from Master to Slave
//
I2CMasterControl(I2C_MASTER_CMD_SINGLE_SEND);

//
// Delay until transmission completes
//
while(I2CMasterBusBusy())
{
}
```

7 Interrupt Controller

7.1 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. This version of the Stellaris family supports thirty-two 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-M3 microprocessor. When the processor responds to an interrupt, NVIC will supply the address of the function to handle the interrupt directly to the processor. This 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, NVIC can be configured (via software) for N - M bits of preemptable prioritization and M bits of subpriority. In this scheme, two interrupts with the same preemptable prioritization but different subpriorities will not cause a preemption; tail chaining will instead be used to process the two interrupts back-to-back.

If two interrupts with the same priority (and subpriority if so configured) are asserted at the same time, the one with the lower interrupt number will be processed first. 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 NVIC via IntEnable() before the processor will respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself).

Alternatively, interrupts can be configured at run-time using IntRegister() (or the analogue 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 call by the application.

Run-time configuration of interrupt handlers requires that the interrupt handler table be placed on a 1 kB boundary in SRAM (typically this would be at the beginning of SRAM). Failure to do so will result in an incorrect vector address being fetch in response to an interrupt. The vector table is in a section called “vtable” and should be placed appropriately with a linker script. Tools that do not support linker scripts (such as the evaluation version of RV-MDK) therefore do not support run-time configuration of interrupt handlers (though the full version of RV-MDK does).

7.2 API Functions

Functions

- __attribute__ ((section("vtable")))
- void IntDisable (unsigned long ulInterrupt)
- void IntEnable (unsigned long ulInterrupt)
- void IntMasterDisable (void)
- long IntPriorityGet (unsigned long ulInterrupt)
- unsigned long IntPriorityGroupingGet (void)
- void IntPriorityGroupingSet (unsigned long ulBits)
- void IntPrioritySet (unsigned long ulInterrupt, unsigned char ucPriority)
- void IntRegister (unsigned long ulInterrupt, void(∗pfnHandler)(void))
- void IntUnregister (unsigned long ulInterrupt)

7.2.1 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 utilized as a simple critical section (only NMI will interrupt the processor while the processor interrupt is disabled), though this will 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 Stellaris family, N is 3). This allows priorities to be defined without a real need to know the exact number of supported priorities; moving to a device with more or fewer priority bits will continue to treat the interrupt source with a similar level of priority. Smaller priority numbers correspond to higher interrupt priority, so 0 is the highest priority.

7.2.2 Function Documentation

7.2.2.1 __attribute__

Enables the processor interrupt.

Prototype:

Description:
Allows the processor to respond to interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Returns:
None.

7.2.2.2 IntDisable

Disables an interrupt.

Prototype:
```
void
IntDisable(unsigned long ulInterrupt)
```

Parameters:
ulInterrupt 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.

7.2.2.3 IntEnable

Enables an interrupt.

Prototype:
```
void
IntEnable(unsigned long ulInterrupt)
```

Parameters:
ulInterrupt 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.

7.2.2.4 IntMasterDisable

Disables the processor interrupt.

Prototype:
```
void
IntMasterDisable(void)
```

Description:
Prevents the processor from receiving interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Returns:
None.

7.2.2.5 IntPriorityGet

Gets the priority of an interrupt.

Prototype:
```
long
IntPriorityGet(unsigned long ulInterrupt)
```

Parameters:
ulInterrupt 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.

7.2.2.6 IntPriorityGroupingGet

Gets the priority grouping of the interrupt controller.

Prototype:
```
unsigned long
IntPriorityGroupingGet(void)
```

Description:
This function returns the split between preemptable priority levels and subpriority levels in the interrupt priority specification.

Returns:
The number of bits of preemptable priority.

7.2.2.7 IntPriorityGroupingSet

Sets the priority grouping of the interrupt controller.

Prototype:

```
void
IntPriorityGroupingSet(unsigned long ulBits)
```

Parameters:

ulBits specifies the number of bits of preemptable priority.

Description:

This function specifies the split between preemptable priority levels and subpriority levels in the interrupt priority specification. The range of the grouping values are dependent upon the hardware implementation; on the Stellaris family it can range from 0 to 3.

Returns:

None.

7.2.2.8 IntPrioritySet

Sets the priority of an interrupt.

Prototype:

```
void
IntPrioritySet(unsigned long ulInterrupt,
               unsigned char ucPriority)
```

Parameters:

ulInterrupt specifies the interrupt in question.

ucPriority 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 will only look at the upper N bits of the priority level (where N is 3 for the Stellaris 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.

7.2.2.9 IntRegister

Registers a function to be called when an interrupt occurs.

Prototype:
```
void
IntRegister(unsigned long ulInterrupt,
            void(*)(void) pfnHandler)
```

Parameters:

ulInterrupt 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 will be called in interrupt context. Since 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 NVIC will 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; some tool chains, such as the evaluation version of RV-MDK, do not support linker scripts and therefore will not produce a valid executable. See the discussion of compile-time versus run-time interrupt.handler registration in the introduction to this chapter.

Returns:

None.

7.2.2.10 IntUnregister

Unregisters the function to be called when an interrupt occurs.

Prototype:
```
void
IntUnregister(unsigned long ulInterrupt)
```

Parameters:

ulInterrupt specifies the interrupt in question.

Description:

This function is used to indicate that no handler should be called when the given interrupt is asserted to the processor. The interrupt source will be automatically disabled (via IntDisable()) if necessary.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

7.3 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();
```

8 Pulse Width Modulator

8.1 Introduction

Each instance of a Stellaris PWM module provides three 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 Stellaris PWM module are:

- Three 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

8.2 API Functions

Functions

- void PWMDeadBandDisable (unsigned long ulBase, unsigned long ulGen)
- void PWMDeadBandEnable (unsigned long ulBase, unsigned long ulGen, unsigned short usRise, unsigned short usFall)
- void PWMFaultIntClear (unsigned long ulBase)
- void PWMFaultIntRegister (unsigned long ulBase, void(∗pfIntHandler)(void))
- void PWMFaultIntUnregister (unsigned long ulBase)
- void PWMGenConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulConfig)
- void PWMGenDisable (unsigned long ulBase, unsigned long ulGen)
- void PWMGenEnable (unsigned long ulBase, unsigned long ulGen)
- void PWMGenIntClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulInts)
- void PWMGenIntRegister (unsigned long ulBase, unsigned long ulGen, void(∗pfIntHandler)(void))

- unsigned long PWMGenIntStatus (unsigned long ulBase, unsigned long ulGen, tBoolean bMasked)
- void PWMGenIntTrigDisable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- void PWMGenIntTrigEnable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- void PWMGenIntUnregister (unsigned long ulBase, unsigned long ulGen)
- unsigned long PWMGenPeriodGet (unsigned long ulBase, unsigned long ulGen)
- void PWMGenPeriodSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulPeriod)
- void PWMIntDisable (unsigned long ulBase, unsigned long ulGenFault)
- void PWMIntEnable (unsigned long ulBase, unsigned long ulGenFault)
- unsigned long PWMIntStatus (unsigned long ulBase, tBoolean bMasked)
- void PWMOutputFault (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bFaultKill)
- void PWMOutputInvert (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bInvert)
- void PWMOutputState (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bEnable)
- unsigned long PWMPulseWidthGet (unsigned long ulBase, unsigned long ulPWMOut)
- void PWMPulseWidthSet (unsigned long ulBase, unsigned long ulPWMOut, unsigned long ulWidth)
- void PWMSyncTimeBase (unsigned long ulBase, unsigned long ulGenBits)
- void PWMSyncUpdate (unsigned long ulBase, unsigned long ulGenBits)

8.2.1 Detailed Description

These are a group of functions for performing high-level operations on PWM modules. Although Stellaris only has one PWM module, these functions are defined to support using multiple instances of 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 it 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 three generator blocks are called **Gen0**, **Gen1**, and **Gen2**.
- The two PWM output signals associated with each generator block are called **OutA** and **OutB**.
- The six output signals are called **PWM0**, **PWM1**, **PWM2**, **PWM3**, **PWM4**, and **PWM5**.
- **PWM0** and **PWM1** are associated with **Gen0**, **PWM2** and **PWM3** are associated with **Gen1**, and **PWM4** and **PWM5** are associated with **Gen2**.

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**, and **PWM4**). In addition, comparator B is used exclusively for the odd numbered PWM outputs (**PWM1**, **PWM3**, **PWM5**).

8.2.2 Function Documentation

8.2.2.1 PWMDeadBandDisable

Disables the PWM dead band output.

Prototype:

```
void
PWMDeadBandDisable(unsigned long ulBase,
                   unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

Description:

This function disables the dead band mode for the specified PWM generator. Doing so decouples the **OutA** and **OutB** signals.

Returns:

None.

8.2.2.2 PWMDeadBandEnable

Enables the PWM dead band output, and sets the dead band delays.

Prototype:

```
void
PWMDeadBandEnable(unsigned long ulBase,
                  unsigned long ulGen,
                  unsigned short usRise,
                  unsigned short usFall)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

usRise specifies the width of delay from the rising edge.

usFall 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.

8.2.2.3 PWMFaultIntClear

Clears the fault interrupt for a PWM module.

Prototype:

```
void
PWMFaultIntClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the PWM module.

Description:

Clears the fault interrupt by writing to the appropriate bit of the interrupt status register for the selected PWM module.

Returns:

None.

8.2.2.4 PWMFaultIntRegister

Registers an interrupt.handler for a fault condition detected in a PWM module.

Prototype:

```
void
PWMFaultIntRegister(unsigned long ulBase,
                    void(*)(void) pfIntHandler)
```

Parameters:

ulBase is the base address of the PWM module.

pfIntHandler is a pointer to the function to be called when the PWM fault interrupt occurs.

Description:

This function will ensure that the interrupt.handler specified by *pfIntHandler* is called when a fault interrupt is detected for the selected PWM module. This function will also enable 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.

8.2.2.5 PWMFaultIntUnregister

Removes the PWM fault condition interrupt.handler.

Prototype:

```
void
PWMFaultIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the PWM module.

Description:

This function will remove the interrupt.handler for a PWM fault interrupt from the selected PWM module. This function will also disable 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.

8.2.2.6 PWMGenConfigure

Configures a PWM generator.

Prototype:

```
void
PWMGenConfigure(unsigned long ulBase,
                unsigned long ulGen,
                unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to configure. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

ulConfig 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 will count from a value down to zero, and then reset to the preset value. This will produce left-aligned PWM signals (i.e. the rising edge of the two PWM signals produced by the generator will occur at the same time). In count up/down mode, it will count up from zero to the preset value, count back down to zero, and then repeat the process. This will produce center-aligned PWM signals (i.e. the middle of the high/low period of the PWM signals produced by the generator will occur at the same time).

When the PWM generator parameters (period and pulse width) are modified, their affect on the output PWM signals can be delayed. In synchronous mode, the parameter updates are not applied until a synchronization event occurs. This allows multiple parameters to be modified and take affect 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 (i.e. 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 will continue to count until it reaches zero, at which point it will pause until the processor is restarted. If configured to continue running, it will keep counting as if nothing had happened.

The **ulConfig** 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 synchronization mode, and **PWM_GEN_MODE_DBG_RUN** or **PWM_GEN_MODE_DBG_STOP** to specify the debug behavior.

Note:

Changes to the counter mode will affect the period of the PWM signals produced. PWMGen-PeriodSet() and PWMPulseWidthSet() should be called after any changes to the counter mode of a generator.

Returns:

None.

8.2.2.7 PWMGenDisable

Disables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGenDisable(unsigned long ulBase,
              unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to be disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

Description:

This function blocks the **PWM** clock from driving the timer/counter for the specified generator block.

Returns:

None.

8.2.2.8 PWMGenEnable

Enables the timer/counter for a PWM generator block.

Prototype:

```
void
PWMGenEnable(unsigned long ulBase,
             unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to be enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

Description:
This function allows the **PWM** clock to drive the timer/counter for the specified generator block.

Returns:
None.

8.2.2.9 PWMGenIntClear

Clears the specified interrupt(s) for the specified PWM generator block.

Prototype:

```
void
PWMGenIntClear(unsigned long ulBase,
               unsigned long ulGen,
               unsigned long ulInts)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

ulInts specifies the interrupts to be cleared.

Description:
Clears the specified interrupt(s) by writing a 1 to the specified bits of the interrupt status register for the specified PWM generator. The defined values for the bits are as follows:

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD
- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD

Returns:
None.

8.2.2.10 PWMGenIntRegister

Registers an interrupt.handler for the specified PWM generator block.

Prototype:

```
void
PWMGenIntRegister(unsigned long ulBase,
                  unsigned long ulGen,
                  void(*)(void) pfIntHandler)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator in question.

pfIntHandler is a pointer to the function to be called when the PWM generator interrupt occurs.

Description:

This function will ensure that the interrupt.handler specified by *pfIntHandler* is called when an interrupt is detected for the specified PWM generator block. This function will also enable the corresponding PWM generator interrupt in the interrupt controller; individual generator interrupts and interrupt sources must be enabled with PWMIntEnable() and PWMGenIntTrig-Enable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

8.2.2.11 PWMGenIntStatus

Gets interrupt status for the specified PWM generator block.

Prototype:

```
unsigned long
PWMGenIntStatus(unsigned long ulBase,
                unsigned long ulGen,
                tBoolean bMasked)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

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 will be returned.

Returns:

Returns the contents of the interrupt status register, or the contents of the raw interrupt status register, for the specified PWM generator.

8.2.2.12 PWMGenIntTrigDisable

Disables interrupts for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigDisable(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulIntTrig)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

ulIntTrig specifies the interrupts and triggers to be disabled.

Description:

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 defined values for the bits are as follows:

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD
- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD
- PWM_TR_CNT_ZERO
- PWM_TR_CNT_LOAD
- PWM_TR_CMP_AU
- PWM_TR_CMP_AD
- PWM_TR_CMP_BU
- PWM_TR_CMP_BD

Returns:

None.

8.2.2.13 PWMGenIntTrigEnable

Enables interrupts and triggers for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigEnable(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulIntTrig)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

ulIntTrig specifies the interrupts and triggers to be enabled.

Description:

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 defined values for the bits are as follows:

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD

- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD
- PWM_TR_CNT_ZERO
- PWM_TR_CNT_LOAD
- PWM_TR_CMP_AU
- PWM_TR_CMP_AD
- PWM_TR_CMP_BU
- PWM_TR_CMP_BD

Returns:
None.

8.2.2.14 PWMGenIntUnregister

Removes an interrupt.handler for the specified PWM generator block.

Prototype:
```
void
PWMGenIntUnregister(unsigned long ulBase,
                    unsigned long ulGen)
```

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator in question.

Description:
This function will unregister the interrupt.handler for the specified PWM generator block. This function will also disable 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.

8.2.2.15 PWMGenPeriodGet

Gets the period of a PWM generator block.

Prototype:
```
unsigned long
PWMGenPeriodGet(unsigned long ulBase,
                unsigned long ulGen)
```

Parameters:
ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

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.

8.2.2.16 PWMGenPeriodSet

Set the period of a PWM generator.

Prototype:

```
void
PWMGenPeriodSet(unsigned long ulBase,
                unsigned long ulGen,
                unsigned long ulPeriod)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to be modified. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, or **PWM_GEN_2**.

ulPeriod 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 will cause the previous values to be overwritten.

Returns:

None.

8.2.2.17 PWMIntDisable

Disables generator and fault interrupts for a PWM module.

Prototype:

```
void
PWMIntDisable(unsigned long ulBase,
              unsigned long ulGenFault)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be disabled. Must be a logical OR of any of **PWM_INT_-GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, or **PWM_INT_FAULT**.

Description:

Masks the specified interrupt(s) by clearing the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

8.2.2.18 PWMIntEnable

Enables generator and fault interrupts for a PWM module.

Prototype:

```
void
PWMIntEnable(unsigned long ulBase,
             unsigned long ulGenFault)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be enabled. Must be a logical OR of any of **PWM_INT_-GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, or **PWM_INT_FAULT**.

Description:

Unmasks the specified interrupt(s) by setting the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

8.2.2.19 PWMIntStatus

Gets the interrupt status for a PWM module.

Prototype:

```
unsigned long
PWMIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:

ulBase 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 will be returned.

Returns:

The current interrupt status, enumerated as a bit field of **PWM_INT_GEN_0**, **PWM_INT_GEN_-1**, **PWM_INT_GEN_2**, and **PWM_INT_FAULT**.

8.2.2.20 PWMOutputFault

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

Prototype:

```
void
PWMOutputFault(unsigned long ulBase,
               unsigned long ulPWMOutBits,
               tBoolean bFaultKill)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. 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**, or **PWM_OUT_5_BIT**.

bFaultKill determines if the signal is killed or passed through during an active fault condition.

Description:

This function sets the fault handling characteristics of the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bFaultKill* determines the fault handling characteristics for the selected outputs. If *bFaultKill* is **true**, then the selected outputs will be made inactive. If *bFaultKill* is **false**, then the selected outputs are unaffected by the detected fault.

Returns:

None.

8.2.2.21 PWMOutputInvert

Selects the inversion mode for PWM outputs.

Prototype:

```
void
PWMOutputInvert(unsigned long ulBase,
                unsigned long ulPWMOutBits,
                tBoolean bInvert)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. 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**, or **PWM_OUT_5_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 *ulPWMOutBits*. The parameter *bInvert* determines the inversion mode for the selected outputs. If *bInvert* is **true**, this function will cause the specified PWM output signals to be inverted, or made active low. If *bInvert* is **false**, the specified output will be passed through as is, or be made active high.

Returns:

None.

8.2.2.22 PWMOutputState

Enables or disables PWM outputs.

Prototype:

```
void
PWMOutputState(unsigned long ulBase,
               unsigned long ulPWMOutBits,
               tBoolean bEnable)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. 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**, or **PWM_OUT_5_BIT**.

bEnable determines if the signal is enabled or disabled.

Description:

This function is used to enable or disable the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. 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.

8.2.2.23 PWMPulseWidthGet

Gets the pulse width of a PWM output.

Prototype:

```
unsigned long
PWMPulseWidthGet(unsigned long ulBase,
                 unsigned long ulPWMOut)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to query. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, or **PWM_OUT_5**.

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.

8.2.2.24 PWMPulseWidthSet

Sets the pulse width for the specified PWM output.

Prototype:

```
void
PWMPulseWidthSet(unsigned long ulBase,
                 unsigned long ulPWMOut,
                 unsigned long ulWidth)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to modify. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, or **PWM_OUT_5**.

ulWidth 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 will cause the previous values to be overwritten.

Returns:

None.

8.2.2.25 PWMSyncTimeBase

Synchronizes the counters in one or multiple PWM generator blocks.

Prototype:

```
void
PWMSyncTimeBase(unsigned long ulBase,
                unsigned long ulGenBits)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be synchronized. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, or **PWM_GEN_2_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.

8.2.2.26 PWMSyncUpdate

Synchronizes all pending updates.

Prototype:

```
void
PWMSyncUpdate(unsigned long ulBase,
                 unsigned long ulGenBits)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be updated. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, or **PWM_GEN_2_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.

8.3 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);
```

9 Quadrature Encoder

9.1 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; this allows wiring mistakes on the circuit board to be corrected without modifying the board.

The index pulse can be used to reset the position counter; this causes 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 will generate 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.

9.2 API Functions

Functions

- void QEIConfigure (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMax-Position)
- long QEIDirectionGet (unsigned long ulBase)
- void QEIDisable (unsigned long ulBase)

- void QEIEnable (unsigned long ulBase)
- tBoolean QEIErrorGet (unsigned long ulBase)
- void QEIIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntRegister (unsigned long ulBase, void(∗pfnHandler)(void))
- unsigned long QEIIntStatus (unsigned long ulBase, tBoolean bMasked)
- void QEIIntUnregister (unsigned long ulBase)
- unsigned long QEIPositionGet (unsigned long ulBase)
- void QEIPositionSet (unsigned long ulBase, unsigned long ulPosition)
- void QEIVelocityConfigure (unsigned long ulBase, unsigned long ulPreDiv, unsigned long ulPeriod)
- void QEIVelocityDisable (unsigned long ulBase)
- void QEIVelocityEnable (unsigned long ulBase)
- unsigned long QEIVelocityGet (unsigned long ulBase)

9.2.1 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().

9.2.2 Function Documentation

9.2.2.1 QEIConfigure

Configures the quadrature encoder.

Prototype:

```
void
QEIConfigure(unsigned long ulBase,
             unsigned long ulConfig,
             unsigned long ulMaxPosition)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulConfig is the configuration for the quadrature encoder. See below for a description of this parameter.

ulMaxPosition specifies the maximum position value.

Description:

This will configure the operation of the quadrature encoder. The *ulConfig* 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** to 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** to specify if the position integrator should be reset when the index pulse is detected.
- **QEI_CONFIG_QUADRATURE** or **QEI_CONFIG_CLOCK_DIR** to 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.

ulMaxPosition 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.

9.2.2.2 QEIDirectionGet

Gets the current direction of rotation.

Prototype:

```
long
QEIDirectionGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This 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:

1 if moving in the forward direction or -1 if moving in the reverse direction.

9.2.2.3 QEIDisable

Disables the quadrature encoder.

Prototype:

```
void
QEIDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:
This will disable operation of the quadrature encoder module.

Returns:
None.

9.2.2.4 QEIEnable

Enables the quadrature encoder.

Prototype:
```
void
QEIEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will enable operation of the quadrature encoder module. It must be configured before it is enabled.

See also:
QEIConfigure()

Returns:
None.

9.2.2.5 QEIErrorGet

Gets the encoder error indicator.

Prototype:
```
tBoolean
QEIErrorGet(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This 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:
true if an error has occurred and false otherwise.

9.2.2.6 QEIIntClear

Clears quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntClear(unsigned long ulBase,
            unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be cleared. 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 longer assert. This must be done in the interrupt.handler to keep it from being called again immediately upon exit.

Returns:

None.

9.2.2.7 QEIIntDisable

Disables individual quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be disabled. Can be any of the QEI_-INTERROR, QEI_INTDIR, QEI_INTTIMER, or QEI_INTINDEX values.

Description:

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.

9.2.2.8 QEIIntEnable

Enables individual quadrature encoder interrupt sources.

Prototype:

```
void
QEIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulIntFlags 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:
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.

9.2.2.9 QEIIntRegister

Registers an interrupt.handler for the quadrature encoder interrupt.

Prototype:
```
void
QEIIntRegister(unsigned long ulBase,
               void(*)(void) pfnHandler)
```

Parameters:
ulBase 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 sets the handler to be called when a quadrature encoder interrupt occurs. This will enable 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.

9.2.2.10 QEIIntStatus

Gets the current interrupt status.

Prototype:
```
unsigned long
QEIIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:
ulBase 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 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:

The current interrupt status, enumerated as a bit field of QEI_INTERROR, QEI_INTDIR, QEI_-INTTIMER, and QEI_INTINDEX.

9.2.2.11 QEIIntUnregister

Unregisters an interrupt.handler for the quadrature encoder interrupt.

Prototype:

```
void
QEIIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This function will clear the handler to be called when a quadrature encoder interrupt occurs. This will also mask 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.

9.2.2.12 QEIPositionGet

Gets the current encoder position.

Prototype:

```
unsigned long
QEIPositionGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This 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 (i.e. if in reset on index mode, if an index pulse has not been encountered, the position counter will not be aligned with the index pulse yet).

Returns:

The current position of the encoder.

9.2.2.13 QEIPositionSet

Sets the current encoder position.

Prototype:

```
void
QEIPositionSet(unsigned long ulBase,
               unsigned long ulPosition)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulPosition is the new position for the encoder.

Description:

This sets the current position of the encoder; the encoder position will then be measured relative to this value.

Returns:

None.

9.2.2.14 QEIVelocityConfigure

Configures the velocity capture.

Prototype:

```
void
QEIVelocityConfigure(unsigned long ulBase,
                     unsigned long ulPreDiv,
                     unsigned long ulPeriod)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulPreDiv 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.

ulPeriod specifies the number of clock ticks over which to measure the velocity; must be non-zero.

Description:

This will configure the operation of the velocity capture portion of the quadrature encoder. The position increment signal is predivided as specified by *ulPreDiv* before being accumulated by the velocity capture. The divided signal is accumulated over *ulPeriod* system clock before being saved and resetting the accumulator.

Returns:

None.

9.2.2.15 QEIVelocityDisable

Disables the velocity capture.

Prototype:

```
void
QEIVelocityDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This will disable operation of the velocity capture in the quadrature encoder module.

Returns:

None.

9.2.2.16 QEIVelocityEnable

Enables the velocity capture.

Prototype:

```
void
QEIVelocityEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This will enable operation of the velocity capture in the quadrature encoder module. It must be configured before it is enabled. Velocity capture will not occur if the quadrature encoder is not enabled.

See also:

QEIVelocityConfigure() and QEIEnable()

Returns:

None.

9.2.2.17 QEIVelocityGet

Gets the current encoder speed.

Prototype:

```
unsigned long
QEIVelocityGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This 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:
The number of pulses captured in the given time period.

9.3 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 since 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);
```

10 Synchronous Serial Interface

10.1 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. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

10.2 API Functions

Functions

- void SSIConfig (unsigned long ulBase, unsigned long ulProtocol, unsigned long ulMode, unsigned long ulBitRate, unsigned long ulDataWidth)
- void SSIDataGet (unsigned long ulBase, unsigned long ∗pulData)
- long SSIDataNonBlockingGet (unsigned long ulBase, unsigned long ∗ulData)
- long SSIDataNonBlockingPut (unsigned long ulBase, unsigned long ulData)
- void SSIDataPut (unsigned long ulBase, unsigned long ulData)
- void SSIDisable (unsigned long ulBase)
- void SSIEnable (unsigned long ulBase)
- void SSIIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void SSIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void SSIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void SSIIntRegister (unsigned long ulBase, void(∗pfnHandler)(void))
- unsigned long SSIIntStatus (unsigned long ulBase, tBoolean bMasked)
- void SSIIntUnregister (unsigned long ulBase)

10.2.1 Detailed Description

The SSI API is broken into 3 groups of functions: those that deal with configuration and state, those that handle data, and those that manage interrupts.

The configuration of the SSI module is managed by the SSIConfig() function, while state is managed by the SSIEnable() and SSIDisable() functions.

Data handling is performed by the SSIDataPut(), SSIDataNonBlockingPut(), SSIDataGet(), and SSIDataNonBlockingGet() functions.

Interrupts from the SSI module are managed using the SSIIntClear(), SSIIntDisable(), SSIInt-Enable(), SSIIntRegister(), SSIIntStatus(), and SSIIntUnregister() functions.

10.2.2 Function Documentation

10.2.2.1 SSIConfig

Configures the synchronous serial interface.

Prototype:

```
void
SSIConfig(unsigned long ulBase,
          unsigned long ulProtocol,
          unsigned long ulMode,
          unsigned long ulBitRate,
          unsigned long ulDataWidth)
```

Parameters:

ulBase specifies the SSI module base address.

ulProtocol specifies the data transfer protocol.

ulMode specifies the mode of operation.

ulBitRate specifies the clock rate.

ulDataWidth specifies number of bits transfered per frame.

Description:

This function configures the synchronous serial interface. It sets the SSI protocol, mode of operation, bit rate, and data width.

The parameter *ulProtocol* defines the data frame format. The parameter *ulProtocol* 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 imply 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 parameter *ulMode* defines the operating mode of the SSI module. The SSI module can operate as a master or slave; if a slave, the SSI can be configured to disable output on its serial output line. The parameter *ulMode* can be one of the following values: SSI_MODE_MASTER, SSI_MODE_SLAVE, or SSI_MODE_SLAVE_OD.

The parameter *ulBitRate* defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria:

- FSSI $>=$ 2 $*$ bit rate (master mode)
- FSSI $>=$ 12 $*$ bit rate (slave modes)

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

The parameter *ulDataWidth* defines the width of the data transfers. The parameter *ulDataWidth* can be a value between 4 and 16, inclusive.

The SSI clocking is dependent upon the system clock rate returned by SysCtlClockGet(); if it does not return the correct system clock rate then the SSI clock rate will be incorrect.

Returns:
None.

10.2.2.2 SSIDataGet

Gets a data element from the SSI receive FIFO.

Prototype:
```
void
SSIDataGet(unsigned long ulBase,
           unsigned long *pulData)
```

Parameters:
ulBase specifies the SSI module base address.
pulData pointer to a storage location for data that was received over the SSI interface.

Description:
This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *pulData* parameter.

Note:
Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by SSIConfig(). For example, if the interface is configured for 8 bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

Returns:
None.

10.2.2.3 SSIDataNonBlockingGet

Gets a data element from the SSI receive FIFO.

Prototype:
```
long
SSIDataNonBlockingGet(unsigned long ulBase,
                      unsigned long *ulData)
```

Parameters:

ulBase specifies the SSI module base address.

ulData pointer to a storage location for data that was received over the SSI interface.

Description:

This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *ulData* parameter. If there is no data in the FIFO, then this function will return a zero.

Note:

Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by SSIConfig(). For example, if the interface is configured for 8 bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

Returns:

Returns the number of elements read from the SSI receive FIFO.

10.2.2.4 SSIDataNonBlockingPut

Puts a data element into the SSI transmit FIFO.

Prototype:

```
long
SSIDataNonBlockingPut(unsigned long ulBase,
                      unsigned long ulData)
```

Parameters:

ulBase specifies the SSI module base address.

ulData data to be transmitted over the SSI interface.

Description:

This function will place the supplied data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function will return a zero.

Note:

The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by SSIConfig(). For example, if the interface is configured for 8 bit data width, the upper 24 bits of *ulData* will be discarded.

Returns:

Returns the number of elements written to the SSI transmit FIFO.

10.2.2.5 SSIDataPut

Puts a data element into the SSI transmit FIFO.

Prototype:

```
void
SSIDataPut(unsigned long ulBase,
           unsigned long ulData)
```

Parameters:

ulBase specifies the SSI module base address.

ulData data to be transmitted over the SSI interface.

Description:

This function will place the supplied data into the transmit FIFO of the specified SSI module.

Note:

The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by SSIConfig(). For example, if the interface is configured for 8 bit data width, the upper 24 bits of *ulData* will be discarded.

Returns:

None.

10.2.2.6 SSIDisable

Disables the synchronous serial interface.

Prototype:

```
void
SSIDisable(unsigned long ulBase)
```

Parameters:

ulBase specifies the SSI module base address.

Description:

This will disable operation of the synchronous serial interface.

Returns:

None.

10.2.2.7 SSIEnable

Enables the synchronous serial interface.

Prototype:

```
void
SSIEnable(unsigned long ulBase)
```

Parameters:

ulBase specifies the SSI module base address.

Description:

This will enable operation of the synchronous serial interface. It must be configured before it is enabled.

Returns:

None.

10.2.2.8 SSIIntClear

Clears SSI interrupt sources.

Prototype:

```
void
SSIIntClear(unsigned long ulBase,
            unsigned long ulIntFlags)
```

Parameters:

ulBase specifies the SSI module base address.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified SSI interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt.handler to keep it from being called again immediately upon exit. The parameter *ulIntFlags* can consist of either or both the SSI_RXTO and SSI_RXOR values.

Returns:

None.

10.2.2.9 SSIIntDisable

Disables individual SSI interrupt sources.

Prototype:

```
void
SSIIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase specifies the SSI module base address.

ulIntFlags is a bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated SSI interrupt sources. The parameter *ulIntFlags* Can be any of the SSI_TXFF, SSI_RXFF, SSI_RXTO, or SSI_RXOR values.

Returns:

None.

10.2.2.10 SSIIntEnable

Enables individual SSI interrupt sources.

Prototype:

```
void
SSIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase specifies the SSI module base address.

ulIntFlags is a bit mask of the interrupt sources to be enabled.

Description:

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 parameter *ulIntFlags* Can be any of the SSI_TXFF, SSI_RXFF, SSI_RXTO, or SSI_RXOR values.

Returns:

None.

10.2.2.11 SSIIntRegister

Registers an interrupt.handler for the synchronous serial interface.

Prototype:

```
void
SSIIntRegister(unsigned long ulBase,
               void(*)(void) pfnHandler)
```

Parameters:

ulBase 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 sets the handler to be called when an SSI interrupt occurs. This will enable 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 SSIInt-Clear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

10.2.2.12 SSIIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
SSIIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:

ulBase specifies the SSI module base address.

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

Description:

This 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.

10.2.2.13 SSIIntUnregister

Unregisters an interrupt.handler for the synchronous serial interface.

Prototype:

```
void
SSIIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase specifies the SSI module base address.

Description:

This function will clear the handler to be called when a SSI interrupt occurs. This will also mask 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.

10.3 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.";
long lIdx;

//
// Configure the SSI.
//
SSIConfig(SSI_BASE, SSI_FRF_MOTO_MODE0, SSI_MODE_MASTER, 2000000, 8);

//
// Enable the SSI module.
//
SSIEnable(SSI_BASE);

//
// Send some data.
//
lIdx = 0;
while(pcChars[lIdx])
{
    if(SSIDataPut(SSI_BASE, pcChars[lIdx]))
```

```
        {
            lIdx++;
        }
    }
```

11 System Control

11.1 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 Stellaris 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 will run on more than one member of the Stellaris family.

The device can be clocked from one of five sources: an external oscillator, the main oscillator, the internal oscillator, the internal oscillator divided by four, or the PLL. The PLL can use any of the four oscillators as its input. Since 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 between 3.579545 MHz and 8.192 MHz (i.e. the standard crystal frequencies in that range). 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 internal oscillator is 15 MHz, +/- 50%; its frequency will vary 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. The ADC and PWM blocks have their own clocks. In order to use the ADC, the PLL must be used; the PLL output will be used to create the clock required by the ADC. The PWM has its own optional divider from the system clock; this can be power of two divides between 1 and 64.

Three modes of operation are supported by the Stellaris 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 will return 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 on-chip 2.5 V power supply; 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, will 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, and a watchdog timeout. The properties of some of these events can be configured, and the reason for a reset can be determined from system control.

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, since 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) will not 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.

There are various system events that, when detected, will 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. Each of these interrupts can be individually enabled or disabled, and the sources must be cleared by the interrupt handler when they occur.

11.2 API Functions

Functions

- unsigned long SysCtlADCSpeedGet (void)
- void SysCtlADCSpeedSet (unsigned long ulSpeed)
- void SysCtlBrownOutConfigSet (unsigned long ulConfig, unsigned long ulDelay)
- void SysCtlClkVerificationClear (void)
- unsigned long SysCtlClockGet (void)
- void SysCtlClockSet (unsigned long ulConfig)
- void SysCtlDeepSleep (void)
- unsigned long SysCtlFlashSizeGet (void)
- void SysCtlIntClear (unsigned long ulInts)
- void SysCtlIntDisable (unsigned long ulInts)
- void SysCtlIntEnable (unsigned long ulInts)
- void SysCtlIntRegister (void(∗pfnHandler)(void))
- unsigned long SysCtlIntStatus (tBoolean bMasked)
- void SysCtlIntUnregister (void)
- void SysCtlIOSCVerificationSet (tBoolean bEnable)
- void SysCtlLDOConfigSet (unsigned long ulConfig)
- unsigned long SysCtlLDOGet (void)
- void SysCtlLDOSet (unsigned long ulVoltage)
- void SysCtlMOSCVerificationSet (tBoolean bEnable)
- void SysCtlPeripheralClockGating (tBoolean bEnable)
- void SysCtlPeripheralDeepSleepDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralDeepSleepEnable (unsigned long ulPeripheral)
- void SysCtlPeripheralDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralEnable (unsigned long ulPeripheral)

- tBoolean SysCtlPeripheralPresent (unsigned long ulPeripheral)
- void SysCtlPeripheralReset (unsigned long ulPeripheral)
- void SysCtlPeripheralSleepDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralSleepEnable (unsigned long ulPeripheral)
- tBoolean SysCtlPinPresent (unsigned long ulPin)
- void SysCtlPLLVerificationSet (tBoolean bEnable)
- unsigned long SysCtlPWMClockGet (void)
- void SysCtlPWMClockSet (unsigned long ulConfig)
- void SysCtlReset (void)
- void SysCtlResetCauseClear (unsigned long ulCauses)
- unsigned long SysCtlResetCauseGet (void)
- void SysCtlSleep (void)
- unsigned long SysCtlSRAMSizeGet (void)

11.2.1 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(), SysCtlPeripheralPresent(), and SysCtlPinPresent().

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().

11.2.2 Function Documentation

11.2.2.1 SysCtlADCSpeedGet

Gets the sample rate of the ADC.

Prototype:

```
unsigned long
SysCtlADCSpeedGet(void)
```

Description:
This function gets the current sample rate of the ADC.

Returns:
Returns the current ADC sample rate; will be one of **SYSCTL_ADCSPEED_-1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_-ADCSPEED_125KSPS**.

11.2.2.2 SysCtlADCSpeedSet

Sets the sample rate of the ADC.

Prototype:

```
void
SysCtlADCSpeedSet(unsigned long ulSpeed)
```

Parameters:
ulSpeed 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 sets 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() will return the actual speed in use.

Returns:
None.

11.2.2.3 SysCtlBrownOutConfigSet

Configures the brown-out control.

Prototype:

```
void
SysCtlBrownOutConfigSet(unsigned long ulConfig,
                        unsigned long ulDelay)
```

Parameters:
ulConfig is the desired configuration of the brown-out control. Must be the logical OR of **SYSCTL_BOR_RESET** and/or **SYSCTL_BOR_RESAMPLE**.

ulDelay is the number of internal oscillator cycles to wait before resampling an asserted brown-out signal. This value only has meaning when **SYSCTL_BOR_RESAMPLE** is set and must be less than 8192.

Description:

This function configures how the brown-out control operates. It can detect a brown-out by looking at only the brown-out output, or it can wait for it to be active for two consecutive samples separated by a configurable time. When it detects a brown-out condition, it can either reset the device or generate a processor interrupt.

Returns:

None.

11.2.2.4 SysCtlClkVerificationClear

Clears the clock verification status.

Prototype:

```
void
SysCtlClkVerificationClear(void)
```

Description:

This function clears the status of the clock verification timers, allowing them to assert another failure if detected.

Returns:

None.

11.2.2.5 SysCtlClockGet

Gets the processor clock rate.

Prototype:

```
unsigned long
SysCtlClockGet(void)
```

Description:

This function determines the clock rate of the processor clock. This is also the clock rate of all the peripheral modules (with the exception of PWM, which has its own clock divider).

Note:

This will not 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 later case, this function should be modified to directly return the correct system clock rate.

Returns:

The processor clock rate.

11.2.2.6 SysCtlClockSet

Sets the clocking of the device.

Prototype:

```
void
SysCtlClockSet(unsigned long ulConfig)
```

Parameters:

ulConfig 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 **ulConfig** 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_4**, **SYSCTL_SYSDIV_5**, **SYSCTL_SYSDIV_6**, **SYSCTL_SYSDIV_7**, **SYSCTL_SYSDIV_8**, **SYSCTL_SYSDIV_9**, **SYSCTL_SYSDIV_10**, **SYSCTL_SYSDIV_11**, **SYSCTL_SYSDIV_12**, **SYSCTL_SYSDIV_13**, **SYSCTL_SYSDIV_14**, **SYSCTL_SYSDIV_15**, or **SYSCTL_SYSDIV_16**.

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_3_57MHZ**, **SYSCTL_XTAL_3_68MHZ**, **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**, or **SYSCTL_XTAL_8_19MHZ**.

The oscillator source is chosen with one of the following values: **SYSCTL_OSC_MAIN**, **SYSCTL_OSC_INT**, or **SYSCTL_OSC_INT4**.

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 will be 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 (i.e. via **SYSCTL_USE_PLL**), this function will poll 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 will delay until its timeout has occurred instead of completing as soon as PLL lock is achieved.

Returns:

None.

11.2.2.7 SysCtlDeepSleep

Puts the processor into deep-sleep mode.

Prototype:

```
void
SysCtlDeepSleep(void)
```

Description:
This function places the processor into deep-sleep mode; it will 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.

11.2.2.8 SysCtlFlashSizeGet

Gets the size of the flash.

Prototype:

```
unsigned long
SysCtlFlashSizeGet(void)
```

Description:
This function determines the size of the flash on the Stellaris device.

Returns:
The total number of bytes of flash.

11.2.2.9 SysCtlIntClear

Clears system control interrupt sources.

Prototype:

```
void
SysCtlIntClear(unsigned long ulInts)
```

Parameters:
ulInts 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 must be done in the interrupt.handler to keep it from being called again immediately upon exit.

Returns:
None.

11.2.2.10 SysCtlIntDisable

Disables individual system control interrupt sources.

Prototype:

```
void
SysCtlIntDisable(unsigned long ulInts)
```

Parameters:

ulInts 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:

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.

Returns:

None.

11.2.2.11 SysCtlIntEnable

Enables individual system control interrupt sources.

Prototype:

```
void
SysCtlIntEnable(unsigned long ulInts)
```

Parameters:

ulInts 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:

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.

Returns:

None.

11.2.2.12 SysCtlIntRegister

Registers an interrupt.handler for the system control interrupt.

Prototype:

```
void
SysCtlIntRegister(void(*)(void) pfnHandler)
```

Parameters:

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

Description:

This sets the handler to be called when a system control interrupt occurs. This will enable 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.

Returns:

None.

11.2.2.13 SysCtlIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
SysCtlIntStatus(tBoolean bMasked)
```

Parameters:

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

Description:

This 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.

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**.

11.2.2.14 SysCtlIntUnregister

Unregisters the interrupt.handler for the system control interrupt.

Prototype:

```
void
SysCtlIntUnregister(void)
```

Description:

This function will clear the handler to be called when a system control interrupt occurs. This will also mask 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.

11.2.2.15 SysCtlIOSCVerificationSet

Configures the internal oscillator verification timer.

Prototype:
```
void
SysCtlIOSCVerificationSet(tBoolean bEnable)
```

Parameters:
bEnable is a boolean that is **true** if the internal oscillator verification timer should be enabled.

Description:
This function allows the internal oscillator verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the internal oscillator ceases to operate.

Note:
Both oscillators (main and internal) must be enabled for this verification timer to operate as the main oscillator will verify the internal oscillator.

Returns:
None.

11.2.2.16 SysCtlLDOConfigSet

Configures the LDO failure control.

Prototype:
```
void
SysCtlLDOConfigSet(unsigned long ulConfig)
```

Parameters:
ulConfig is the required LDO failure control setting; can be either **SYSCTL_LDOCFG_ARST** or **SYSCTL_LDOCFG_NORST**.

Description:
This function allows the LDO to be configured to cause a processor reset when the output voltage becomes unregulated.

Returns:
None.

11.2.2.17 SysCtlLDOGet

Gets the output voltage of the LDO.

Prototype:

```
unsigned long
SysCtlLDOGet(void)
```

Description:
This function determines the output voltage of the LDO, as specified by the control register.

Returns:
Returns the current voltage of the LDO; will be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

11.2.2.18 SysCtlLDOSet

Sets the output voltage of the LDO.

Prototype:

```
void
SysCtlLDOSet(unsigned long ulVoltage)
```

Parameters:
ulVoltage is the required output voltage from the LDO. Must be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

Description:
This function sets the output voltage of the LDO. The default voltage is 2.5 V; it can be adjusted +/- 10%.

Returns:
None.

11.2.2.19 SysCtlMOSCVerificationSet

Configures the main oscillator verification timer.

Prototype:

```
void
SysCtlMOSCVerificationSet(tBoolean bEnable)
```

Parameters:
bEnable is a boolean that is **true** if the main oscillator verification timer should be enabled.

Description:
This function allows the main oscillator verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the main oscillator ceases to operate.

Note:
Both oscillators (main and internal) must be enabled for this verification timer to operate as the internal oscillator will verify the main oscillator.

Returns:
None.

11.2.2.20 SysCtlPeripheralClockGating

Controls peripheral clock gating in sleep and deep-sleep mode.

Prototype:
```
void
SysCtlPeripheralClockGating(tBoolean 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 SysCtlPeripheralSleep-Enable(), SysCtlPeripheralSleepDisable(), SysCtlPeripheralDeepSleepEnable(), and SysCtl-PeripheralDeepSleepDisable().

Returns:
None.

11.2.2.21 SysCtlPeripheralDeepSleepDisable

Disables a peripheral in deep-sleep mode.

Prototype:
```
void
SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral 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 SysCtlPeripheral-Enable()), the peripheral will automatically resume 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 **ulPeripheral** argument must be one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.22 SysCtlPeripheralDeepSleepEnable

Enables a peripheral in deep-sleep mode.

Prototype:
```
void
SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral 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. Since the clocking configuration of the device may change, not all peripherals can safely continue operating while the processor is in sleep mode. Those that must run at a particular frequency (such as a UART) will 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 **ulPeripheral** argument must be one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.23 SysCtlPeripheralDisable

Disables a peripheral.

Prototype:
```
void
SysCtlPeripheralDisable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral to disable.

Description:
Peripherals are disabled with this function. Once disabled, they will not operate or respond to register reads/writes.

The **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.24 SysCtlPeripheralEnable

Enables a peripheral.

Prototype:
```
void
SysCtlPeripheralEnable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral to enable.

Description:
Peripherals are enabled with this function. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

The **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.25 SysCtlPeripheralPresent

Determines if a peripheral is present.

Prototype:
```
tBoolean
SysCtlPeripheralPresent(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral in question.

Description:
Determines if a particular peripheral is present in the device. Each member of the Stellaris family has a different peripheral set; this will determine which are present on this device.

The **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_MPU**, **SYSCTL_-PERIPH_TEMP**, or **SYSCTL_PERIPH_PLL**.

Returns:
Returns **true** if the specified peripheral is present and **false** if it is not.

11.2.2.26 SysCtlPeripheralReset

Performs a software reset of a peripheral.

Prototype:
```
void
SysCtlPeripheralReset(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral 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 deasserted, leaving the peripheral in a operating state but in its reset condition.

The **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_-PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_-PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_-TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_-COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.27 SysCtlPeripheralSleepDisable

Disables a peripheral in sleep mode.

Prototype:
```
void
SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral 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 will automatically resume 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 **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.28 SysCtlPeripheralSleepEnable

Enables a peripheral in sleep mode.

Prototype:
```
void
SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral to enable in sleep mode.

Description:
This function allows a peripheral to continue operating when the processor goes into sleep mode. Since 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 **ulPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_WDOG**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_SSI**, **SYSCTL_PERIPH_QEI**, **SYSCTL_PERIPH_I2C**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, or **SYSCTL_PERIPH_GPIOE**.

Returns:
None.

11.2.2.29 SysCtlPinPresent

Determines if a pin is present.

Prototype:
```
tBoolean
SysCtlPinPresent(unsigned long ulPin)
```

Parameters:
ulPin is the pin in question.

Description:
Determines if a particular pin is present in the device. The PWM, analog comparators, ADC, and timers have a varying number of pins across members of the Stellaris family; this will determine which are present on this device.

The **ulPin** argument must be only one of the following values: **SYSCTL_PIN_PWM0**, **SYSCTL_PIN_PWM1**, **SYSCTL_PIN_PWM2**, **SYSCTL_PIN_PWM3**, **SYSCTL_PIN_PWM4**, **SYSCTL_PIN_PWM5**, **SYSCTL_PIN_C0MINUS**, **SYSCTL_PIN_C0PLUS**, **SYSCTL_PIN_-C0O**, **SYSCTL_PIN_C1MINUS**, **SYSCTL_PIN_C1PLUS**, **SYSCTL_PIN_C1O**, **SYSCTL_-PIN_C2MINUS**, **SYSCTL_PIN_C2PLUS**, **SYSCTL_PIN_C2O**, **SYSCTL_PIN_ADC0**, **SYSCTL_PIN_ADC1**, **SYSCTL_PIN_ADC2**, **SYSCTL_PIN_ADC3**, **SYSCTL_PIN_ADC4**, **SYSCTL_PIN_ADC5**, **SYSCTL_PIN_ADC6**, **SYSCTL_PIN_ADC7**, **SYSCTL_PIN_CCP0**, **SYSCTL_PIN_CCP1**, **SYSCTL_PIN_CCP2**, **SYSCTL_PIN_CCP3**, **SYSCTL_PIN_CCP4**, **SYSCTL_PIN_CCP5**, **SYSCTL_PIN_CCP6**, **SYSCTL_PIN_CCP7**, or **SYSCTL_PIN_32KHZ**.

Returns:
Returns **true** if the specified pin is present and **false** if it is not.

11.2.2.30 SysCtlPLLVerificationSet

Configures the PLL verification timer.

Prototype:
```
void
SysCtlPLLVerificationSet(tBoolean bEnable)
```

Parameters:
bEnable is a boolean that is **true** if the PLL verification timer should be enabled.

Description:
This function allows the PLL verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the PLL ceases to operate.

Note:
The main oscillator must be enabled for this verification timer to operate as it is used to check the PLL. Also, the verification timer should be disabled while the PLL is being reconfigured via SysCtlClockSet().

Returns:
None.

11.2.2.31 SysCtlPWMClockGet

Gets the current PWM clock configuration.

Prototype:
```
unsigned long
SysCtlPWMClockGet(void)
```

Description:
This function returns the current PWM clock configuration.

Returns:
The current PWM clock configuration; will 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**.

11.2.2.32 SysCtlPWMClockSet

Sets the PWM clock configuration.

Prototype:
```
void
SysCtlPWMClockSet(unsigned long ulConfig)
```

Parameters:
ulConfig 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 sets 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 SysCtl-ClockSet().

Returns:
None.

11.2.2.33 SysCtlReset

Resets the device.

Prototype:

```
void
SysCtlReset(void)
```

Description:

This function will perform a software reset of the entire device. The processor and all peripherals will be reset and all device registers will return to their default values (with the exception of the reset cause register, which will maintain its current value but have the software reset bit set as well).

Returns:

This function does not return.

11.2.2.34 SysCtlResetCauseClear

Clears reset reasons.

Prototype:

```
void
SysCtlResetCauseClear(unsigned long ulCauses)
```

Parameters:

ulCauses 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.

11.2.2.35 SysCtlResetCauseGet

Gets the reason for a reset.

Prototype:

```
unsigned long
SysCtlResetCauseGet(void)
```

Description:

This function will return the reason(s) for a reset. Since the reset reasons are sticky until either cleared by software or an external reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason will 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**.

Returns:

The reason(s) for a reset.

11.2.2.36 SysCtlSleep

Puts the processor into sleep mode.

Prototype:
```
void
SysCtlSleep(void)
```

Description:
This function places the processor into sleep mode; it will 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 SysCtl-PeripheralClockGating(), otherwise all peripherals continue to operate).

Returns:
None.

11.2.2.37 SysCtlSRAMSizeGet

Gets the size of the SRAM.

Prototype:
```
unsigned long
SysCtlSRAMSizeGet(void)
```

Description:
This function determines the size of the SRAM on the Stellaris device.

Returns:
The total number of bytes of SRAM.

11.3 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);
```

12 SysTick

12.1 Introduction

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M3 microprocessor. Its intended purpose is to provide a periodic interrupt for a RTOS, but it can be used for other simple timing purposes.

12.2 API Functions

Functions

- void SysTickDisable (void)
- void SysTickEnable (void)
- void SysTickIntDisable (void)
- void SysTickIntEnable (void)
- void SysTickIntRegister (void(∗pfnHandler)(void))
- void SysTickIntUnregister (void)
- unsigned long SysTickPeriodGet (void)
- void SysTickPeriodSet (unsigned long ulPeriod)
- unsigned long SysTickValueGet (void)

12.2.1 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()).

12.2.2 Function Documentation

12.2.2.1 SysTickDisable

Disables the SysTick counter.

Prototype:

```
void
SysTickDisable(void)
```

Description:
This will stop the SysTick counter. If an interrupt.handler has been registered, it will no longer be called until SysTick is restarted.

Returns:
None.

12.2.2.2 SysTickEnable

Enables the SysTick counter.

Prototype:
```
void
SysTickEnable(void)
```

Description:
This will start the SysTick counter. If an interrupt.handler has been registered, it will be called when the SysTick counter rolls over.

Returns:
None.

12.2.2.3 SysTickIntDisable

Disables the SysTick interrupt.

Prototype:
```
void
SysTickIntDisable(void)
```

Description:
This function will disable the SysTick interrupt, preventing it from being reflected to the processor.

Returns:
None.

12.2.2.4 SysTickIntEnable

Enables the SysTick interrupt.

Prototype:
```
void
SysTickIntEnable(void)
```

Description:
This function will enable the SysTick interrupt, allowing it to be reflected to the processor.

Returns:
None.

12.2.2.5 SysTickIntRegister

Registers an interrupt.handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntRegister(void(*)(void) pfnHandler)
```

Parameters:

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

Description:

This sets the handler to be called when a SysTick interrupt occurs.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

12.2.2.6 SysTickIntUnregister

Unregisters the interrupt.handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntUnregister(void)
```

Description:

This function will clear the handler to be called when a SysTick interrupt occurs.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

12.2.2.7 SysTickPeriodGet

Gets the period of the SysTick counter.

Prototype:

```
unsigned long
SysTickPeriodGet(void)
```

Description:

This function returns the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Returns:

Returns the period of the SysTick counter.

12.2.2.8 SysTickPeriodSet

Sets the period of the SysTick counter.

Prototype:

```
void
SysTickPeriodSet(unsigned long ulPeriod)
```

Parameters:

ulPeriod is the number of clock ticks in each period of the SysTick counter; must be between 1 and 16,777,216, inclusive.

Description:

This function sets the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Returns:

None.

12.2.2.9 SysTickValueGet

Gets the current value of the SysTick counter.

Prototype:

```
unsigned long
SysTickValueGet(void)
```

Description:

This function returns the current value of the SysTick counter; this will be a value between the period - 1 and zero, inclusive.

Returns:

Returns the current value of the SysTick counter.

12.3 Programming Example

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

```
unsigned long ulValue;

//
// Configure and enable the SysTick counter.
//
SysTickPeriodSet(1000);
SysTickEnable();

//
// Delay for some time...
//

//
```

```
// Read the current SysTick value.
//
ulValue = SysTickValueGet();
```

13 Timer

13.1 Introduction

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

The timer module provides two 16-bit timer/counters that can be configured to operate independently as timers or event counters, or they can be configured to operate as one 32-bit timer or one 32-bit Real Time Clock (RTC). For the purpose of this API, the two timers provided by the timer are referred to as TimerA and TimerB.

When configured as either a 32-bit or 16-bit timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured as a one-shot timer, when it reaches zero the timer will cease counting. If configured as a continuous timer, when it reaches zero the timer will continue counting from a reloaded value. When configured as a 32-bit timer, the timer can also be configured to operate as an RTC. In that case, the timer expects to be driven by a 32 KHz external clock, which is divided down to produce 1 second clock ticks.

When in 16-bit 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 it can count 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 line used to capture events becomes an output line, and the timer is used to drive an edge-aligned pulse onto that line.

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 RTC matches a certain value.

13.2 API Functions

Functions

- void TimerConfigure (unsigned long ulBase, unsigned long ulConfig)
- void TimerControlEvent (unsigned long ulBase, unsigned long ulTimer, unsigned long ulEvent)
- void TimerControlLevel (unsigned long ulBase, unsigned long ulTimer, tBoolean bInvert)
- void TimerControlStall (unsigned long ulBase, unsigned long ulTimer, tBoolean bStall)
- void TimerControlTrigger (unsigned long ulBase, unsigned long ulTimer, tBoolean bEnable)
- void TimerDisable (unsigned long ulBase, unsigned long ulTimer)

- void TimerEnable (unsigned long ulBase, unsigned long ulTimer)
- void TimerIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntRegister (unsigned long ulBase, unsigned long ulTimer, void(∗pfnHandler)(void))
- unsigned long TimerIntStatus (unsigned long ulBase, tBoolean bMasked)
- void TimerIntUnregister (unsigned long ulBase, unsigned long ulTimer)
- unsigned long TimerLoadGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerLoadSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long TimerMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long TimerPrescaleGet (unsigned long ulBase, unsigned long ulTimer)
- unsigned long TimerPrescaleMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerPrescaleMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void TimerPrescaleSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void TimerQuiesce (unsigned long ulBase)
- void TimerRTCDisable (unsigned long ulBase)
- void TimerRTCEnable (unsigned long ulBase)
- unsigned long TimerValueGet (unsigned long ulBase, unsigned long ulTimer)

13.2.1 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 32- or 16-bit modes, and to select between PWM, capture, and timer operations. Timer control is performed by TimerEnable(), TimerDisable(), TimerControlLevel(), TimerControlTrigger(), TimerControlEvent(), TimerControlStall(), TimerRTCEnable(), TimerRTCDisable(), and TimerQuiesce().

Timer content is managed with TimerLoadSet(), TimerLoadGet(), TimerPrescaleSet(), TimerPrescaleGet(), TimerMatchSet(), TimerMatchGet(), TimerPrescaleMatchSet(), TimerPrescaleMatchGet(), and TimerValueGet().

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().

13.2.2 Function Documentation

13.2.2.1 TimerConfigure

Configures the timer(s).

Prototype:

```
void
```

```
TimerConfigure(unsigned long ulBase,
               unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the timer module.

ulConfig 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 configuration is specified in *ulConfig* as one of the following values:

- **TIMER_CFG_32_BIT_OS** - 32-bit one shot timer
- **TIMER_CFG_32_BIT_PER** - 32-bit periodic timer
- **TIMER_CFG_32_RTC** - 32-bit real time clock timer
- **TIMER_CFG_16_BIT_PAIR** - Two 16-bit timers

When configured for a pair of 16-bit timers, each timer is separately configured. The first timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the following values and *ulConfig:*

- **TIMER_CFG_A_ONE_SHOT** - 16-bit one shot timer
- **TIMER_CFG_A_PERIODIC** - 16-bit periodic timer
- **TIMER_CFG_A_CAP_COUNT** - 16-bit edge count capture
- **TIMER_CFG_A_CAP_TIME** - 16-bit edge time capture
- **TIMER_CFG_A_PWM** - 16-bit PWM output

Similarly, the second timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the corresponding **TIMER_CFG_B_**$*$ values and *ulConfig*.

Returns:

None.

13.2.2.2 TimerControlEvent

Controls the event type.

Prototype:

```
void
TimerControlEvent(unsigned long ulBase,
                  unsigned long ulTimer,
                  unsigned long ulEvent)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ulEvent 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 sets the signal edge(s) that will trigger the timer when in capture mode.

Returns:
None.

13.2.2.3 TimerControlLevel

Controls the output level.

Prototype:
```
void
TimerControlLevel(unsigned long ulBase,
                  unsigned long ulTimer,
                  tBoolean bInvert)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer 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 sets the PWM output level for the specified timer. If the parameter *bInvert* is **true**, then the timer's output will be made active low; otherwise, it will be made active high.

Returns:
None.

13.2.2.4 TimerControlStall

Controls the stall handling.

Prototype:
```
void
TimerControlStall(unsigned long ulBase,
                  unsigned long ulTimer,
                  tBoolean bStall)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer 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 parameter *bStall* is **true**, then the timer will stop counting if the processor enters debug mode; otherwise the timer will keep running while in debug mode.

Returns:
None.

13.2.2.5 TimerControlTrigger

Enables or disables the trigger output.

Prototype:

```
void
TimerControlTrigger(unsigned long ulBase,
                    unsigned long ulTimer,
                    tBoolean bEnable)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bEnable specifies the desired trigger state.

Description:

This function controls the trigger output for the specified timer. If the parameter *bEnable* is **true**, then the timer's output trigger is enabled; otherwise it is disabled.

Returns:

None.

13.2.2.6 TimerDisable

Disables the timer(s).

Prototype:

```
void
TimerDisable(unsigned long ulBase,
             unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to disable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_-BOTH**.

Description:

This will disable operation of the timer module.

Returns:

None.

13.2.2.7 TimerEnable

Enables the timer(s).

Prototype:

```
void
TimerEnable(unsigned long ulBase,
            unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to enable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_-BOTH**.

Description:

This will enable operation of the timer module. The timer must be configured before it is enabled.

Returns:

None.

13.2.2.8 TimerIntClear

Clears timer interrupt sources.

Prototype:

```
void
TimerIntClear(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the timer module.

ulIntFlags 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 must be done in the interrupt.handler to keep it from being called again immediately upon exit.

The parameter *ulIntFlags* has the same definition as the *ulIntFlags* parameter to TimerInt-Enable().

Returns:

None.

13.2.2.9 TimerIntDisable

Disables individual timer interrupt sources.

Prototype:

```
void
TimerIntDisable(unsigned long ulBase,
                unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the timer module.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

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 parameter *ulIntFlags* has the same definition as the *ulIntFlags* parameter to TimerInt-Enable().

Returns:
None.

13.2.2.10 TimerIntEnable

Enables individual timer interrupt sources.

Prototype:
```
void
TimerIntEnable(unsigned long ulBase,
               unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the timer module.
ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:
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 parameter *ulIntFlags* 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.

13.2.2.11 TimerIntRegister

Registers an interrupt.handler for the timer interrupt.

Prototype:
```
void
TimerIntRegister(unsigned long ulBase,
                 unsigned long ulTimer,
                 void(*)(void) pfnHandler)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer 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 sets the handler to be called when a timer interrupt occurs. This will enable 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.

13.2.2.12 TimerIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
TimerIntStatus(unsigned long ulBase,
               tBoolean bMasked)
```

Parameters:

ulBase 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 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().

13.2.2.13 TimerIntUnregister

Unregisters an interrupt.handler for the timer interrupt.

Prototype:

```
void
TimerIntUnregister(unsigned long ulBase,
                   unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This function will clear the handler to be called when a timer interrupt occurs. This will also mask 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.

13.2.2.14 TimerLoadGet

Gets the timer load value.

Prototype:
```
unsigned long
TimerLoadGet(unsigned long ulBase,
             unsigned long ulTimer)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:
This function gets the currently programmed interval load value for the specified timer.

Returns:
Returns the load value for the timer.

13.2.2.15 TimerLoadSet

Sets the timer load value.

Prototype:
```
void
TimerLoadSet(unsigned long ulBase,
             unsigned long ulTimer,
             unsigned long ulValue)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer 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 32-bit operation.
ulValue is the load value.

Description:
This function sets the timer load value; if the timer is running then the value will be immediately loaded into the timer.

Returns:
None.

13.2.2.16 TimerMatchGet

Gets the timer match value.

Prototype:

```
unsigned long
TimerMatchGet(unsigned long ulBase,
              unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:

This function gets the match value for the specified timer.

Returns:

Returns the match value for the timer.

13.2.2.17 TimerMatchSet

Sets the timer match value.

Prototype:

```
void
TimerMatchSet(unsigned long ulBase,
              unsigned long ulTimer,
              unsigned long ulValue)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer 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 32-bit operation.

ulValue is the match value.

Description:

This function sets the match value for a timer. This 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.

13.2.2.18 TimerPrescaleGet

Get the timer prescale value.

Prototype:

```
unsigned long
TimerPrescaleGet(unsigned long ulBase,
                 unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer 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 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:
The value of the timer prescaler.

13.2.2.19 TimerPrescaleMatchGet

Get the timer prescale match value.

Prototype:
```
unsigned long
TimerPrescaleMatchGet(unsigned long ulBase,
                      unsigned long ulTimer)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer 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 16-bit mode that uses the counter match (edge count or PWM), the prescale match effectively extends the range of the counter to 24-bits.

Returns:
The value of the timer prescale match.

13.2.2.20 TimerPrescaleMatchSet

Set the timer prescale match value.

Prototype:
```
void
TimerPrescaleMatchSet(unsigned long ulBase,
                      unsigned long ulTimer,
                      unsigned long ulValue)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_-BOTH**.
ulValue is the timer prescale match value; must be between 0 and 255, inclusive.

Description:
This function sets the value of the input clock prescaler match value. When in a 16-bit mode that uses the counter match (edge count or PWM), the prescale match effectively extends the range of the counter to 24-bits.

Returns:
None.

13.2.2.21 TimerPrescaleSet

Set the timer prescale value.

Prototype:
```
void
TimerPrescaleSet(unsigned long ulBase,
                 unsigned long ulTimer,
                 unsigned long ulValue)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_-BOTH**.
ulValue is the timer prescale value; must be between 0 and 255, inclusive.

Description:
This function sets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:
None.

13.2.2.22 TimerQuiesce

Puts the timer into its reset state.

Prototype:
```
void
TimerQuiesce(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the timer module.

Description:
The specified timer is disabled, and all its interrupts are disabled, cleared, and unregistered. Then the timer registers are set to their reset value.

Returns:
None.

13.2.2.23 TimerRTCDisable

Disable RTC counting.

Prototype:

```
void
TimerRTCDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the timer module.

Description:

This function causes the timer to stop counting when in RTC mode.

Returns:

None.

13.2.2.24 TimerRTCEnable

Enable RTC counting.

Prototype:

```
void
TimerRTCEnable(unsigned long ulBase)
```

Parameters:

ulBase 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 will do nothing.

Returns:

None.

13.2.2.25 TimerValueGet

Gets the current timer value.

Prototype:

```
unsigned long
TimerValueGet(unsigned long ulBase,
              unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:

This function reads the current value of the specified timer.

Returns:

Returns the current value of the timer.

13.3 Programming Example

The following example shows how to use the timer API to configure the timer as a 16-bit one shot timer and a 16-bit edge capture counter.

```
//
// Configure TimerA as a 16-bit one shot timer, and TimerB as a 16-bit edge
// capture counter.
//
TimerConfigure(TIMER0_BASE, (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_ONE_SHOT |
                             TIMER_CFG_B_CAP_COUNT));

//
// 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);
```

14 UART

14.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Stellaris 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 Stellaris 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 Stellaris 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

14.2 API Functions

Functions

- void UARTBreakCtl (unsigned long ulBase, tBoolean bBreakState)
- long UARTCharGet (unsigned long ulBase)
- long UARTCharNonBlockingGet (unsigned long ulBase)
- tBoolean UARTCharNonBlockingPut (unsigned long ulBase, unsigned char ucData)
- void UARTCharPut (unsigned long ulBase, unsigned char ucData)
- tBoolean UARTCharsAvail (unsigned long ulBase)
- void UARTConfigGet (unsigned long ulBase, unsigned long ∗pulBaud, unsigned long ∗pulConfig)
- void UARTConfigSet (unsigned long ulBase, unsigned long ulBaud, unsigned long ulConfig)
- void UARTDisable (unsigned long ulBase)
- void UARTEnable (unsigned long ulBase)
- void UARTIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void UARTIntDisable (unsigned long ulBase, unsigned long ulIntFlags)

- void UARTIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void UARTIntRegister (unsigned long ulBase, void(∗pfnHandler)(void))
- unsigned long UARTIntStatus (unsigned long ulBase, tBoolean bMasked)
- void UARTIntUnregister (unsigned long ulBase)
- unsigned long UARTParityModeGet (unsigned long ulBase)
- void UARTParityModeSet (unsigned long ulBase, unsigned long ulParity)
- tBoolean UARTSpaceAvail (unsigned long ulBase)

14.2.1 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 Stellaris 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.

Configuration and control of the UART are handled by the UARTConfigGet(), UARTConfigSet(), UARTDisable(), UARTEnable(), UARTParityModeGet(), and UARTParityModeSet() functions.

Sending and receiving data via the UART is handled by the UARTCharGet(), UARTCharNonBlockingGet(), UARTCharPut(), UARTCharNonBlockingPut(), UARTBreakCtl(), UARTCharsAvail(), and UARTSpaceAvail() functions.

Managing the UART interrupts is handled by the UARTIntClear(), UARTIntDisable(), UARTIntEnable(), UARTIntRegister(), UARTIntStatus(), and UARTIntUnregister() functions.

14.2.2 Function Documentation

14.2.2.1 UARTBreakCtl

Causes a BREAK to be sent.

Prototype:

```
void
UARTBreakCtl(unsigned long ulBase,
             tBoolean bBreakState)
```

Parameters:

ulBase is the base address of the UART port.

bBreakState controls the output level.

Description:

Calling this function with *bBreakState* set to **true** will assert a break condition on the UART. Calling this function with *bBreakState* set to **false** will remove the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

Returns:

None.

14.2.2.2 UARTCharGet

Waits for a character from the specified port.

Prototype:

```
long
UARTCharGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

Gets a character from the receive FIFO for the specified port. If there are no characters available, this function will wait until a character is received before returning.

Returns:

Returns the character read from the specified port, cast as an *int*.

14.2.2.3 UARTCharNonBlockingGet

Receives a character from the specified port.

Prototype:

```
long
UARTCharNonBlockingGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

Gets a character from the receive FIFO for the specified port.

Returns:

Returns the character read from the specified port, cast as a *long*. A **-1** will be returned if there are no characters present in the receive FIFO. The UARTCharsAvail() function should be called before attempting to call this function.

14.2.2.4 UARTCharNonBlockingPut

Sends a character to the specified port.

Prototype:

```
tBoolean
UARTCharNonBlockingPut(unsigned long ulBase,
                       unsigned char ucData)
```

Parameters:

ulBase is the base address of the UART port.

ucData is the character to be transmitted.

Description:

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 will have to retry the function later.

Returns:

Returns **true** if the character was successfully placed in the transmit FIFO, and **false** if there was no space available in the transmit FIFO.

14.2.2.5 UARTCharPut

Waits to send a character from the specified port.

Prototype:

```
void
UARTCharPut(unsigned long ulBase,
            unsigned char ucData)
```

Parameters:

ulBase is the base address of the UART port.

ucData is the character to be transmitted.

Description:

Sends the character *ucData* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function will wait until there is space available before returning.

Returns:

None.

14.2.2.6 UARTCharsAvail

Determines if there are any characters in the receive FIFO.

Prototype:

```
tBoolean
UARTCharsAvail(unsigned long ulBase)
```

Parameters:

ulBase 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, and **false** if there is no data in the receive FIFO.

14.2.2.7 UARTConfigGet

Gets the current configuration of a UART.

Prototype:

```
void
UARTConfigGet(unsigned long ulBase,
              unsigned long *pulBaud,
              unsigned long *pulConfig)
```

Parameters:

ulBase is the base address of the UART port.

pulBaud is a pointer to storage for the baud rate.

pulConfig is a pointer to storage for the data format.

Description:

The baud rate and data format for the UART is determined. 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 *pulConfig* is enumerated the same as the *ulConfig* parameter of UARTConfigSet().

The baud rate is dependent upon the system clock rate returned by SysCtlClockGet(); if it does not return the correct system clock rate then the baud rate will be computed incorrectly.

Returns:

None.

14.2.2.8 UARTConfigSet

Sets the configuration of a UART.

Prototype:

```
void
UARTConfigSet(unsigned long ulBase,
              unsigned long ulBaud,
              unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the UART port.

ulBaud is the desired baud rate.

ulConfig is the data format for the port (number of data bits, number of stop bits, and parity).

Description:

This function will configure the UART for operation in the specified data format. The baud rate is provided in the *ulBaud* parameter and the data format in the *ulConfig* parameter.

The *ulConfig* 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 baud rate is dependent upon the system clock rate returned by SysCtlClockGet(); if it does not return the correct system clock rate then the baud rate will be incorrect.

Returns:
None.

14.2.2.9 UARTDisable

Disables transmitting and receiving.

Prototype:
```
void
UARTDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Clears the UARTEN, TXE, and RXE bits, then waits for the end of transmission of the current character, and flushes the transmit FIFO.

Returns:
None.

14.2.2.10 UARTEnable

Enables transmitting and receiving.

Prototype:
```
void
UARTEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Sets the UARTEN, TXE, and RXE bits, and enables the transmit and receive FIFOs.

Returns:
None.

14.2.2.11 UARTIntClear

Clears UART interrupt sources.

Prototype:
```
void
UARTIntClear(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulIntFlags 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 must be done in the interrupt.handler to keep it from being called again immediately upon exit.

The parameter *ulIntFlags* has the same definition as the same parameter to UARTIntEnable().

Returns:

None.

14.2.2.12 UARTIntDisable

Disables individual UART interrupt sources.

Prototype:

```
void
UARTIntDisable(unsigned long ulBase,
               unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

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 parameter *ulIntFlags* has the same definition as the same parameter to UARTIntEnable().

Returns:

None.

14.2.2.13 UARTIntEnable

Enables individual UART interrupt sources.

Prototype:

```
void
UARTIntEnable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

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

Description:

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 parameter *ulIntFlags* is the logical OR of any of the following:

- 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

Returns:
None.

14.2.2.14 UARTIntRegister

Registers an interrupt.handler for a UART interrupt.

Prototype:
```
void
UARTIntRegister(unsigned long ulBase,
                void(*)(void) pfnHandler)
```

Parameters:
ulBase 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 will enable the global interrupt in the interrupt controller; specific UART interrupts must be enabled via UARTInt-Enable(). It is the interrupt handler's responsibility to clear the interrupt source.

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

Returns:
None.

14.2.2.15 UARTIntStatus

Gets the current interrupt status.

Prototype:
```
unsigned long
UARTIntStatus(unsigned long ulBase,
              tBoolean bMasked)
```

Parameters:
ulBase 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 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:

The current interrupt status, enumerated as a bit field of values described in UARTIntEnable().

14.2.2.16 UARTIntUnregister

Unregisters an interrupt.handler for a UART interrupt.

Prototype:

```
void
UARTIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function does the actual unregistering of the interrupt.handler. It will clear the handler to be called when a UART interrupt occurs. This will also mask 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.

14.2.2.17 UARTParityModeGet

Gets the type of parity currently being used.

Prototype:

```
unsigned long
UARTParityModeGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

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**.

14.2.2.18 void UARTParityModeSet (unsigned long *ulBase*, unsigned long *ulParity*)

Sets the type of parity.

Parameters:

ulBase is the base address of the UART port.

ulParity specifies the type of parity to use.

Description:

Sets the type of parity to use for transmitting and expect when receiving. The *ulParity* 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 allow direct control of the parity bit; it will always be either be one or zero based on the mode.

Returns:

None.

14.2.2.19 UARTSpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:

```
tBoolean
UARTSpaceAvail(unsigned long ulBase)
```

Parameters:

ulBase 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, and **false** if there is no space available in the transmit FIFO.

14.3 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.
//
UARTConfigSet(UART0_BASE, 38400, (UART_CONFIG_WLEN_8 |
                                  UART_CONFIG_STOP_ONE |
                                  UART_CONFIG_PAR_NONE));

//
// Enable the UART.
//
UARTEnable(UART0_BASE);

//
// Check for characters.  This will spin here until a character is placed
// into the receive FIFO.
//
while(!UARTCharsAvail(UART0_BASE))
{
}

//
// Get the character(s) in the receive FIFO.
//
while(UARTCharNonBlockingGet(UART0_BASE))
{
}

//
// Put a character in the output buffer.
//
UARTCharPut(UART0_BASE, 'c'));

//
// Disable the UART.
//
UARTDisable(UART0_BASE);
```

15 Watchdog Timer

15.1 Introduction

The Watchdog Timer API provides a set of functions for using the Stellaris watchdog timer modules. Functions are provided to deal with the watchdog timer interrupts, and to handle status and configuration of the watchdog timer.

The 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 upon its first timeout, and to generate a reset signal upon 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.

15.2 API Functions

Functions

- void WatchdogEnable (unsigned long ulBase)
- void WatchdogIntClear (unsigned long ulBase)
- void WatchdogIntEnable (unsigned long ulBase)
- void WatchdogIntRegister (unsigned long ulBase, void(∗pfnHandler)(void))
- unsigned long WatchdogIntStatus (unsigned long ulBase, tBoolean bMasked)
- void WatchdogIntUnregister (unsigned long ulBase)
- void WatchdogLock (unsigned long ulBase)
- tBoolean WatchdogLockState (unsigned long ulBase)
- unsigned long WatchdogReloadGet (unsigned long ulBase)
- void WatchdogReloadSet (unsigned long ulBase, unsigned long ulLoadVal)
- void WatchdogResetDisable (unsigned long ulBase)
- void WatchdogResetEnable (unsigned long ulBase)
- tBoolean WatchdogRunning (unsigned long ulBase)

- void WatchdogStallDisable (unsigned long ulBase)
- void WatchdogStallEnable (unsigned long ulBase)
- void WatchdogUnlock (unsigned long ulBase)
- unsigned long WatchdogValueGet (unsigned long ulBase)

15.2.1 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(), WatchdogInt-Unregister(), WatchdogIntEnable(), WatchdogIntClear(), and WatchdogIntStatus() functions.

Status and configuration functions for the Watchdog Timer module are WatchdogEnable(), WatchdogRunning(), WatchdogLock(), WatchdogUnlock(), WatchdogLockState(), Watchdog-ReloadSet(), WatchdogReloadGet(), WatchdogValueGet(), WatchdogResetEnable(), Watchdog-ResetDisable(), WatchdogStallEnable(), and WatchdogStallDisable().

15.2.2 Function Documentation

15.2.2.1 WatchdogEnable

Enables the watchdog timer.

Prototype:

```
void
WatchdogEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

This will enable the watchdog timer counter and interrupt.

Note:

This function will have no effect if the watchdog timer has been locked.

See also:

WatchdogLock(), WatchdogUnlock()

Returns:

None.

15.2.2.2 WatchdogIntClear

Clears the watchdog timer interrupt.

Prototype:

```
void
WatchdogIntClear(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
The watchdog timer interrupt source is cleared, so that it no longer asserts.

Returns:
None.

15.2.2.3 WatchdogIntEnable

Enables the watchdog timer interrupt.

Prototype:
```
void
WatchdogIntEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Enables the watchdog timer interrupt.

Note:
This function will have no effect if the watchdog timer has been locked.

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

Returns:
None.

15.2.2.4 WatchdogIntRegister

Registers an interrupt.handler for watchdog timer interrupt.

Prototype:
```
void
WatchdogIntRegister(unsigned long ulBase,
                    void(*)(void) pfnHandler)
```

Parameters:
ulBase 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 will enable the global interrupt in the interrupt controller; the watchdog timer interrupt must be enabled via Watchdog-Enable(). It is the interrupt handler's responsibility to clear the interrupt source via Watchdog-IntClear().

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

Returns:
None.

15.2.2.5 WatchdogIntStatus

Gets the current watchdog timer interrupt status.

Prototype:
```
unsigned long
WatchdogIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

Parameters:
ulBase 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 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:
The current interrupt status, where a 1 indicates that the watchdog interrupt is active, and a 0 indicates that it is not active.

15.2.2.6 WatchdogIntUnregister

Unregisters an interrupt.handler for the watchdog timer interrupt.

Prototype:
```
void
WatchdogIntUnregister(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This function does the actual unregistering of the interrupt.handler. This function will clear the handler to be called when a watchdog timer interrupt occurs. This will also mask 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.

15.2.2.7 WatchdogLock

Enables the watchdog timer lock mechanism.

Prototype:
```
void
WatchdogLock(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Locks out write access to the watchdog timer configuration registers.

Returns:
None.

15.2.2.8 WatchdogLockState

Gets the state of the watchdog timer lock mechanism.

Prototype:
```
tBoolean
WatchdogLockState(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
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.

15.2.2.9 WatchdogReloadGet

Gets the watchdog timer reload value.

Prototype:
```
unsigned long
WatchdogReloadGet(unsigned long ulBase)
```

Parameters:
ulBase 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.

15.2.2.10 WatchdogReloadSet

Sets the watchdog timer reload value.

Prototype:
```
void
WatchdogReloadSet(unsigned long ulBase,
                  unsigned long ulLoadVal)
```

Parameters:
ulBase is the base address of the watchdog timer module.
ulLoadVal is the load value for the watchdog timer.

Description:
This function sets 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 will be immediately loaded into the watchdog timer counter. If the parameter *ulLoadVal* is 0, then an interrupt is immediately generated.

Note:
This function will have no effect if the watchdog timer has been locked.

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

Returns:
None.

15.2.2.11 WatchdogResetDisable

Disables the watchdog timer reset.

Prototype:
```
void
WatchdogResetDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Disables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

15.2.2.12 WatchdogResetEnable

Enables the watchdog timer reset.

Prototype:
```
void
WatchdogResetEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Enables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

15.2.2.13 WatchdogRunning

Determines if the watchdog timer is enabled.

Prototype:
```
tBoolean
WatchdogRunning(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This will check to see if the watchdog timer is enabled.

Returns:
Returns **true** if the watchdog timer is enabled, and **false** if it is not.

15.2.2.14 WatchdogStallDisable

Disables stalling of the watchdog timer during debug events.

Prototype:
```
void
WatchdogStallDisable(unsigned long ulBase)
```

Parameters:
ulBase 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.

15.2.2.15 WatchdogStallEnable

Enables stalling of the watchdog timer during debug events.

Prototype:
```
void
WatchdogStallEnable(unsigned long ulBase)
```

Parameters:
ulBase 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 will instead expired 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.

15.2.2.16 WatchdogUnlock

Disables the watchdog timer lock mechanism.

Prototype:
```
void
WatchdogUnlock(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Enables write access to the watchdog timer configuration registers.

Returns:
None.

15.2.2.17 WatchdogValueGet

Gets the current watchdog timer value.

Prototype:

```
unsigned long
WatchdogValueGet(unsigned long ulBase)
```

Parameters:

ulBase 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.

15.3 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(WATCHDOG_BASE) == true)
{
    WatchdogUnlock(WATCHDOG_BASE);
}

//
// Initialize the watchdog timer.
//
WatchdogReloadSet(WATCHDOG_BASE, 0xFEEFEE);

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

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

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

16 Error Handling

Invalid arguments and error conditions are handled in a non-traditional manner in the 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, it would return an error code. The caller then has to check the return code from each invocation of the function to make sure that it succeeded.

This 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 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 `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 `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 will call 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 `debug.h` and must be provided by the application since it is the application's responsibility to deal with error conditions.

By setting a breakpoint on the `__error__` function, the debugger will immediately stop 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 will pinpoint the function that found an error, what it found to be a problem, and where it was called from. As an example:

```
void
SSIConfig(unsigned long ulBase, unsigned long ulProtocol,
          unsigned long ulMode, unsigned long ulBitRate,
          unsigned long ulDataWidth)
{
    //
    // Check the arguments.
    //
    ASSERT(ulBase == SSI_BASE);
    ASSERT((ulProtocol == SSI_FRF_MOTO_MODE_0) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_1) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_2) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_3) ||
           (ulProtocol == SSI_FRF_TI) ||
           (ulProtocol == SSI_FRF_NMW));
    ASSERT((ulMode == SSI_MODE_MASTER) ||
           (ulMode == SSI_MODE_SLAVE) ||
           (ulMode == SSI_MODE_SLAVE_OD));
    ASSERT((ulDataWidth >= 4) && (ulDataWidth <= 16));
```

Each argument is individually checked, so the line number of the failing `ASSERT` will indicate the argument that is invalid. The debugger will be 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 allows the problem to be quickly identified at the cost of a small amount of code.

17 DK-LM3Sxxx Example Applications

17.1 Introduction

The DK-LM3Sxxx example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the driver library. These applications are intended for demonstration and as a starting point for new applications; however some of them may be useful, such as the I2C application that reads and writes the on-board Atmel AT24C08A EEPROM.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

17.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)
- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

17.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

17.2.2 Function Documentation

17.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

Returns:

The current state of the DIP switches.

17.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

Returns:

The contents of the direction register.

17.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

Returns:
None.

17.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:
```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:
ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:
This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

Returns:
The contents of the data register.

17.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:
```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:
ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.
ucValue is the value to write to the GPIO data register.

Description:
This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

Returns:
None.

17.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:
```
void
PDCInit(void)
```

Description:
This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

Returns:
None.

17.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:
```
void
PDCLCDBacklightOff(void)
```

Description:
This function turns off the backlight on the LCD.

Returns:
None.

17.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:
```
void
PDCLCDBacklightOn(void)
```

Description:
This function turns on the backlight on the LCD.

Returns:
None.

17.2.2.9 PDCLCDClear

Clear the screen.

Prototype:
```
void
PDCLCDClear(void)
```

Description:
This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

Returns:
None.

17.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

Returns:

None.

17.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:

```
void
PDCLCDInit(void)
```

Description:

This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

Note:

The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:

None.

17.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:

ucX is the horizontal position. Valid values are zero through fifteen.

ucY is the vertical position.. Valid values are zero and one.

Description:

This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

Returns:

None.

17.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

Returns:

None.

17.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

Returns:

The value currently displayed by the LEDs.

17.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

Returns:

None.

17.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

Returns:

Returns the value read from the PDC.

17.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

Returns:
None.

17.3 Examples

Not all of these examples will run successfully on every Stellaris Family Development Kit board; for example, the DK-LM3S101 does not have an I2C interface, so the I2C example will not work on it. For board/example combinations that will not work, the example will detect the lack of a required peripheral and indicate its inability to execute (as opposed to failing for seemingly no reason).

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata will be fixed in the next revision, available by Q2'07.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a "roving eye" display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple "hello world" example. It simply displays "hello world" on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S101 Quickstart Application (qs_dk-lm3s101)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (i.e. the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

DK-LM3S102 Quickstart Application (qs_dk-lm3s102)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (i.e. the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

DK-LM3S301 Quickstart Application (qs_dk-lm3s301)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (i.e. the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

DK-LM3S801 Quickstart Application (qs_dk-lm3s801)

This example uses the potentiometer on the development board to vary the rate and frequency of a repetitive beep from the piezo buzzer. Turning the knob in one direction will result in slower beeps at lower frequency, while turning it the other direction will result in faster beeps at a higher frequency. The potentiometer setting along with the tone "note" is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still show the setting.

DK-LM3S811 Quickstart Application (qs_dk-lm3s811)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

DK-LM3S815 Quickstart Application (qs_dk-lm3s815)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

DK-LM3S828 Quickstart Application (qs_dk-lm3s828)

This example uses the potentiometer on the development board to vary the rate of a click sound from the piezo buzzer. Turning the knob in one direction will result in slower clicks while turning it in the other direction will result in faster clicks. The potentiometer setting is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the clicking noise on and off; when off the LCD and UART still provide the settings.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_out)

This example application utilizes the UART to display text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode and continuously display text. The text is transferred by servicing the interrupt from the UART; since it takes about 1 ms to drain half of the UART FIFO (causing an interrupt), this leaves plenty of time for other processing to occur during the transfer (though that time is not utilized by this example).

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

18 EV-LM3S811 Example Applications

18.1 Introduction

The example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the OSRAM 96x16 OLED graphical display on the Stellaris LM3S811 Evaluation Kit board.

18.2 API Functions

Functions

- void OSRAMClear (void)
- void OSRAMDisplayOff (void)
- void OSRAMDisplayOn (void)
- void OSRAMImageDraw (const unsigned char ∗pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void OSRAMInit (tBoolean bFast)
- void OSRAMStringDraw (const char ∗pcStr, unsigned long ulX, unsigned long ulY)

18.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

18.2.2 Function Documentation

18.2.2.1 OSRAMClear

Clears the OLED display.

Prototype:

```
void
OSRAMClear(void)
```

Description:
This function will clear the display. All pixels in the display will be turned off.

Returns:
None.

18.2.2.2 OSRAMDisplayOff

Turns off the OLED display.

Prototype:
```
void
OSRAMDisplayOff(void)
```

Description:
This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

Returns:
None.

18.2.2.3 OSRAMDisplayOn

Turns on the OLED display.

Prototype:
```
void
OSRAMDisplayOn(void)
```

Description:
This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

Returns:
None.

18.2.2.4 OSRAMImageDraw

Displays an image on the OLED display.

Prototype:
```
void
OSRAMImageDraw(const unsigned char *pucImage,
               unsigned long ulX,
               unsigned long ulY,
               unsigned long ulWidth,
               unsigned long ulHeight)
```

Parameters:
pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in eight scan line blocks from the top of the display (i.e. only 0 and 1 are valid).

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in eight row blocks (i.e. only 1 and 2 are valid).

Description:

This function will display a bitmap graphic on the display. The image to be displayed must be a multiple of eight scan lines high (i.e. one row) and will be drawn at a vertical position that is a multiple of eight scan lines (i.e. scan line zero or scan line eight, corresponding to row zero or row one).

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for the eight scan lines of the column, with the top scan line being in the least significant bit of the byte and the bottom scan line being in the most significant bit of the byte.

For example, an image four columns wide and sixteen scan lines tall would be arranged as follows (showing how the eight bytes of the image would appear on the display):

```
+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 0 | 6 |  | 1 | 6 |  | 2 | 6 |  | 3 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
+-------+  +-------+  +-------+  +-------+

+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 4 | 6 |  | 5 | 6 |  | 6 | 6 |  | 7 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
+-------+  +-------+  +-------+  +-------+
```

Returns:

None.

18.2.2.5 OSRAMInit

Initialize the OLED display.

Prototype:

```
void
OSRAMInit(tBoolean bFast)
```

Parameters:

bFast is a boolean that is *true* if the I2C interface should be run at 400 kbps and *false* if it should be run at 100 kbps.

Description:

This function initializes the I2C interface to the OLED display and configures the SSD0303 controller on the panel.

Returns:

None.

18.2.2.6 OSRAMStringDraw

Displays a string on the OLED display.

Prototype:

```
void
OSRAMStringDraw(const char *pcStr,
                unsigned long ulX,
                unsigned long ulY)
```

Parameters:

pcStr is a pointer to the string to display.

ulX is the horizontal position to display the string, specified in columns from the left edge of the display.

ulY is the vertical position to display the string, specified in eight scan line blocks from the top of the display (i.e. only 0 and 1 are valid).

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as "i" and "l" have more white space around them than characters such as "m" or "w".

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is "too long" to display.

Returns:

None.

18.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata will be fixed in the next revision, available by Q2'07.

Hello World (hello)

A very simple "hello world" example. It simply displays "hello world" on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; GPIO pins D0 through D2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EV-LM3S811 Quickstart Application (qs_ev-lm3s811)

A game in which a ship is navigated through an endless tunnel. The potentiometer is used to move the ship up and down, and the user push button is used to fire a missile to destroy obstacles in the tunnel. Score accumulates for survival and for destroying obstacles. The game lasts for only one ship; the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (i.e. it will never come on during game play). An implementation of the Game of Life is run with a field of random data as the seed value.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (Game of Life or blank display) will be exited by pressing the user push button. The button will then need to be pressed again to start the game.

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_out)

This example application utilizes the UART to display text. The first UART (connected to the FTDI virtual serial port on the Stellaris LM3S811 Evaluation Board) will be configured in 115,200 baud, 8-n-1 mode and continuously display text. The text is transferred by servicing the interrupt from the UART; since it takes about 1 ms to drain half of the UART FIFO (causing an interrupt), this leaves plenty of time for other processing to occur during the transfer (though that time is not utilized by this example).

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port C5 is inverted so that it is easy to see that it is being fed, which occurs once every second.

Company Information

Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3 based microcontrollers for use in embedded applications within the industrial, commercial, and consumer markets. Luminary Micro is ARM's lead partner in the implementation of the Cortex-M3 core. Please contact us if you are interested in obtaining further information about our company or our products.

Luminary Micro, Inc.
2499 South Capital of Texas Hwy, Suite A-100
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com
sales@luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext 3