Biricha Digital Power Ltd

Chip Support
Library API for Piccolo B (C2803x)
Version: v1.6
Date: 18/01/2010

© Biricha Digital Power Limited
Parkway Drive
Sonning
Reading
RG4 6XG
UNITED KINGDOM

ISSUE HISTORY

Author	Changes	Version	Date
Dr Chris Hossack	First draft	V1.0	02/12/2009
Dr Chris Hossack	Add CLA	V1.5	09/12/2009
Dr Chris Hossack	Added capture module	V1.6	18/01/2010

REFERENCES

Id	Version	Title
1.		
2.		
3.		
4.		

Contents

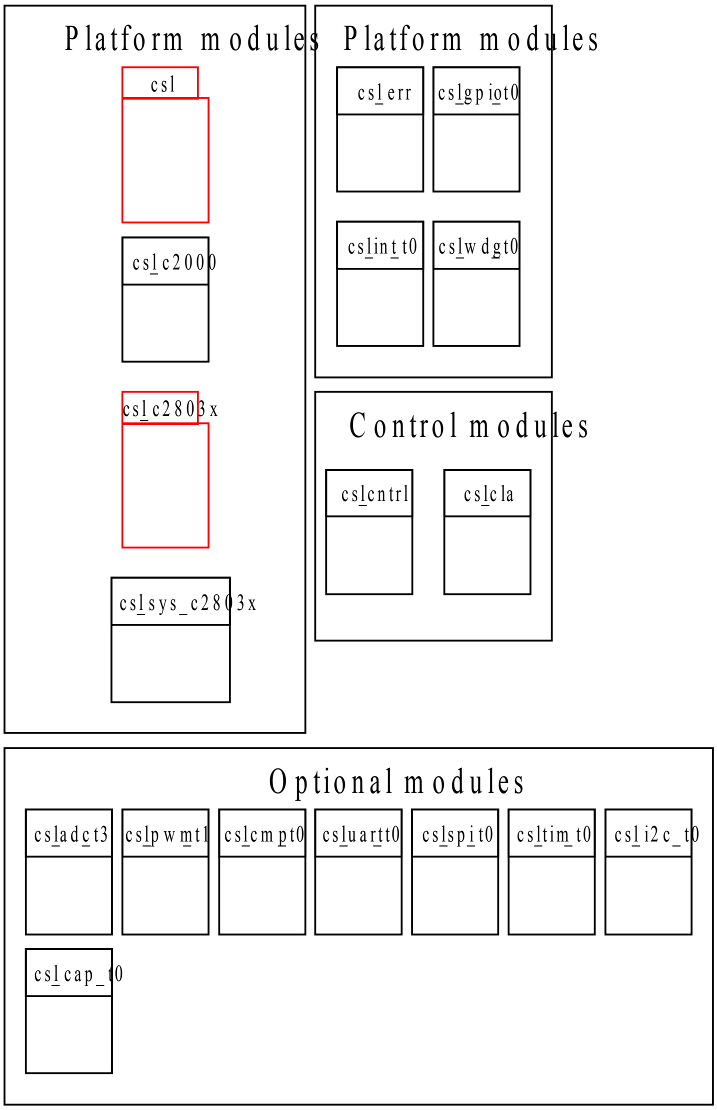
1 Introduction

This document covers the Chip Support Library (CSL). The follow devices are supported.

- TMS320x28035

The following modules are supported at the moment within the CSL.

Module Name	Peripheral Description
ADC	Analog to Digital Converter
CNTRL	A 3p3z controller
GPIO	General Purpose Input Output Pins
I2C	Inter-Integrated Circuit
INT	Interrupts
PWM	Pulse Width Modulations
SPI	Serial Peripheral Interface
SYS	System
TIMER	Timers
UART	RS232
WDG	Watch Dog
ERR	Error Interface
CMP	Analog comparator
CLA	Control Law Accelerator
CAP	Capture Module


Platform modules
csl
csl_c2000
csl_c2803x
csl_sys_c2803x
Platform modules
csl_err
csl_gpio_t0
csl_int_t0
csl_wdg_t0
Control modules
csl_cntrl
csl_cla
Optional modules
csl_adc_t3
csl_pwm_t1
csl_cmp_t0
csl_uart_t0
csl_spi_t0
csl_tim_t0
csl_i2c_t0
csl_cap_t0

2 Installation

You need to have installed F2803x C/C++ Header Files and Peripheral Examples (SPRC892) before installing this free library.

Run the self-extracting zip file and the files will be extracted to C:\tidcs\c28\CSL_C2803x.

2.1 File Structure

When you install the CSL library it creates aCSL_C2803x directory next to the DSP2803x directory.

C:\tidcs\c28\CSL_C2803x\v100

- common
  - cmd
    - csl_28035.cmd
    - csl_28035_RAM_lnk.cmd
  - include
    - csl.h
    - csl_adc_t3_Pub.h
    - csl_c2000_Pub.h
    - csl_c2803x.h
    - csl_cmp_t0_Pub.h
    - csl_cntrl_Pub.h
    - csl_err_Pub.h
    - csl_gpio_t0_Pub.h
    - csl_i2c_t0_Pub.h
    - csl_int_t0_Pub.h
    - csl_pwm_t1_Pub.h
    - csl_spi_t0_Pub.h
    - csl_stdbool.h
    - csl_stdint.h
    - csl_sys_c2803x_Pub.h
    - csl_tim_t0_Pub.h
    - csl_uart_t0_Pub.h
    - csl_wdg_t0_Pub.h
  - lib
    - csl2803x_ml.lib
- doc
  - CSL_C2803x.pdf

3 Example

This example sets up a single ADC conversion every PWM period. The value is read within an interrupt service routine.

```
#include "csl.h"

uint16_t period = PWM_freqToTicks(100000);
uint16_t duty   = PWM_freqToTicks(100000)/4;
uint16_t Vout;

interrupt void IsrAdc( void )
{
	GPIO_set(GPIO_34);

	/* ack the ADC and get a sample */
	ADC_ackInt(ADC_INT_6);
	Vout  = ADC_getValue(ADC_MOD_1);
	Vout += ADC_getValue(ADC_MOD_2);

	/* set a new pwm value */
	PWM_setDutyA(PWM_MOD_2, duty );

	GPIO_clr(GPIO_34);
}

void main ( void )
{
	SYS_init();
	ADC_init();

	/* set up a user IO pin */
	GPIO_config( GPIO_34, GPIO_DIR_OUT, false);

	/* set up a single PWM output */
	PWM_config( PWM_MOD_2, period, PWM_COUNT_UP );
	PWM_pin( PWM_MOD_2, PWM_CH_A, GPIO_NON_INVERT );
	PWM_setAdcSoc( PWM_MOD_2, PWM_CH_A, PWM_INT_ZERO );

	/* set up the ADC to sample every period */
	ADC_config(ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_B5, ADC_TRIG_EPWM2_SOCA );
	ADC_config(ADC_MOD_2, ADC_SH_WIDTH_7, ADC_CH_B6, ADC_TRIG_EPWM2_SOCA );
	ADC_setCallback(ADC_MOD_2,IsrAdc, ADC_INT_6);

	INT_enableGlobal(true);
	while(1);/* endless loop*/
}
```

3.1 CLA

This example sets up the CLA to perform a 3p3z controller using ADC_MOD_1 (ADC_CH_A0), PWM_MOD_3.

```
#include "csl.h"

CLA_3p3zCode( PsuTask, 1, 3,
              1.64135759, -0.44965862, -0.19169897,
              1.90042237, -1.83342509, -1.89984373,  1.83400372,
              0.48, 0.0, 240.0 );

interrupt void IsrFunc()
{
    ADC_clrInt(ADC_INT_7);
    CLA_ackInt(CLA_MOD_7);
}

void main ( void )
{
    SYS_init();
    ADC_init();

    /* Setup PWM and SoC of the ADC */
    PWM_config( PWM_MOD_3, PWM_freqToTicks(500000), PWM_COUNT_DOWN );
    PWM_pin( PWM_MOD_3, PWM_CH_A, GPIO_NON_INVERT );
    PWM_setAdcSoc( PWM_MOD_3, PWM_CH_A, PWM_INT_ZERO );

    /* Set up the ADC to sample Vo and jump to the ISR when sampling is
    *  complete
    */
    ADC_setEarlyInterrupt(1);
    ADC_config(ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM3_SOCA );
    ADC_setCallback(ADC_MOD_1, 0, ADC_INT_7);

      CLA_setRef( CLA_getCtrlPtr(PsuTask),2048);
    CLA_config( CLA_MOD_7, &PsuTask, CLA_INT_ADC );
    CLA_setCallback( CLA_MOD_7, IsrFunc );


    INT_enableGlobal(1);

    while(1)
    {
    }
}
```

4 csl

This is the main header file for the csl library. It pulls in all of the csl modules' header files and the DSP280x header files.

Before you use any API from the csl library you must call SYS_int() which performs the initialisation.

EXAMPLES

```
void main ( void )
{
 SYS_init();

 //your code
}
```

LINKS
file:C:\tidcs\c28\CSL_C280x\v100\doc\CSL_C280x.doc

4.1 Types

5 csl_c2000_

5.1.1.1 Description

```
This files contains all functions that are common to the C2000
microcontroller.
```

5.2 Api

SYS_checkStack()
SYS_getStackUnused()
SYS_setTideMarker()
SYS_dummyRamFuncs()
SYS_usDelay()
SYS_msDelay()

5.2.1 SYS_checkStack

void SYS_checkStack(void)

where:

5.2.1.1 Description

Checks for a stack overflow and raises an assertion if one is detected.

Prerequisite; Immediately following power up the SYS_init() function must be called. SYS_init() calls SYS_setTideMarker() to write a known value to the all of the stack that is currently unused.

This SYS_checkStack() function makes sure the last 8 locations of the stack are still equal to this value. If they are no longer equal to this initial value then it is assumed that the stack has been completely filled and an overflow may have occurred. In this case an assertion is raised by the function.

5.2.1.2 Examples

Call this function within your main idle loop to continually check the status of the stack in-between any interrupt routines.

```
void main ( void )
{
    SYS_init();
    while(1)
    {
        // Do idle loop code
        SYS_checkStack();
    }
}
```

5.2.2 SYS_getStackUnused

uint16_t SYS_getStackUnused(void)

where:

5.2.2.1 Description

Returns the number of unused bytes on the stack.

Prerequisite; Immediately following power up the SYS_init() function must be called. SYS_init() calls SYS_setTideMarker() to write a known value to the all of the stack that is currently unused.

This function counts the number of bytes which contain this known value starting from the stack end.

An application of this function would be to determine the amount of spare stack during the development process.

5.2.2.2 Examples

This returns the number unused bytes on the stack.

```
uint16_t ui_stackSpaceSpare = SYS_getStackUnused();
```

5.2.3 SYS_setTideMarker

void SYS_setTideMarker(void)

where:

5.2.3.1 Description

Writes the stack marker, a known value, from the current stack position to the end of the stack.

For development use only. Initializing the stack with this known value allows the user to determine how much free stack space remains and when a stack overflow may have occurred. These functions can be performed by using SYS_getStackUnused() and SYS_checkStack().

The initialization function SYS_init() includes a call to SYS_setTideMarker(). Therefore SYS_setTideMarker() will not need to be called again later in the code as SYS_init() should have already been called.

An instance where SYS_setTideMarker() might be called again by the user would be to approximately determine how much stack space a particular function has used.

5.2.3.2 Examples

This example approximately determines the number of bytes that a particular function used on the stack. The number is approximate as the SYS_getStackUnused() function call will require some stack space. Thus this example would only be appropriate if MY_func() were a large function.

```
SYS_setTideMarker();
stack_before = SYS_getStackUnused();
MY_func();
stack_free = SYS_getStackUnused();
```

5.2.4 SYS_dummyRamFuncs

void SYS_dummyRamFuncs(void)

where:

5.2.4.1 Description

```
(Private)
For internal use only.
This does nothing except create &RamfuncsLoadStart, &RamfuncsLoadEnd,
&RamfuncsRunStart that the SYS_init() can then use.
```

5.2.5 SYS_usDelay

void SYS_usDelay(uint16_t Delay)

where:
Delay - The number of microseconds.

5.2.5.1 Description

Waits for a specified number of microseconds.

The system clock frequency is unknown at compile time. This function assumes that a 100MHz system clock is used (after PPL and pre-scaling) and will produce a delay of at least the time specified given a 100MHz clock frequency or less.

5.2.5.2 Examples

Waits for five milliseconds.

```
SYS_usDelay( 5000 );
```

NOTES

5.2.6 SYS_msDelay

void SYS_msDelay(uint16_t Delay)

where:
Delay -

5.2.6.1 Description

Waits for a specified number of milliseconds.

The system clock frequency is unknown at compile time. This function assumes that a 100MHz system clock is used (after PPL and pre-scaling) and will produce a delay of at least the time specified given a 100MHz clock frequency or less.

5.2.6.2 Examples

Waits for five seconds.

```
SYS_msDelay( 5000 );
```

NOTES

5.3 Types

6 csl_c2803x

6.1 Types

6.1.1 MOD_COUNT

```
#if 1
#define PWM_MOD_COUNT     7
#define UART_MOD_COUNT    1
#define SPI_MOD_COUNT     2
#define TIM_MOD_COUNT     3
#define I2C_MOD_COUNT     1
#define CMP_MOD_COUNT     3
#define CAP_MOD_COUNT     1
#endif
```

6.1.1.1 Description

This can be use to determine the number of modules per peripheral type used by the csl.

6.1.2 UART_FIFO_DEPTH

```
#define UART_FIFO_DEPTH (4)
```

6.1.2.1 Description

This is the size of the FIFO used by the module.

6.1.3 SPI_FIFO_DEPTH

```
#define SPI_FIFO_DEPTH (4)
```

6.1.3.1 Description

This is the size of the FIFO used by the module.

7 csl_err_

7.1.1.1 Description

Protected functions used for error handling.

When an error happens within the csl code you can look at the value of ERR_Value to what went wrong.

7.2 Api

7.3 Types

7.3.1 ERR_Id

```
enum ERR_Id
{
    ERR_ERR_OK,
    ERR_ADC_MOD_1_2_INVALID,            /* You can not use ADC_MOD_1/2 since you
are using ADC_MOD_3 */
    ERR_ADC_MOD_3_INVALID,              /* You can not use ADC_MOD_3 since you
are using ADC_MOD_1/2 */
    ERR_ADC_MOD_X_INVALID,              /* Invalid ADC_MOD_X */
    ERR_ADC_MOD_1_BUSY,                 /* ADC_MOD_1 is busy */
    ERR_ADC_MOD_2_BUSY,                 /* ADC_MOD_1 is busy */
    ERR_ADC_CONV_GR_8,                  /* Only 8 samples per ADC module in non-
casade mode */
    ERR_ADC_CONV_GR_16,                 /* Only 16 samples per ADC module in
non-casade mode */
    ERR_ADC_CHAN_INVALID,               /* The channelOnly 16 samples per ADC
module in non-casade mode */
    ERR_GPIO_PIN_INVALID,               /* The GPIO pin is invalid */
    ERR_GPIO_NOT_ACQUIRED,              /* The GPIO pin is being used before it
has been acquired */
    ERR_GPIO_ALREADY_ACQUIRED,          /* The GPIO pin is already in use */
    ERR_GPIO_LIMIT_REACHED,             /* The free version has acquire too many
GPIO pins. Please buy the full version */
    ERR_I2C_MOD_X_INVALID,              /* Invalid I2C_MOD_x */
    ERR_INT_ISR_DEFAULT,                /* The default ISR has been called */
    ERR_INT_GROUP_INVALID,              /* The group is is not valid */
    ERR_PWM_ALREADY_ACQUIRED,           /* The PWM is already in use */
    ERR_PWM_COUNT_MODE_INVALID,         /* The PWM cound mode is invalid */
    ERR_PWM_NOT_ACQUIRED,               /* The PWM is being used before it has
been acquired */
    ERR_SPI_MOD_X_INVALID,              /* Invalid SPI_MOD_X */
    ERR_SPI_TICK_INVALID,               /* An invalid value for the spi baud
rate has been used */
    ERR_SYS_STACK_OVERFLOW,             /* A overflow in the stack has been
detected */
    ERR_SYS_PERIPHERAL_INVALID,         /* The required Peripheral not supported
on this device */
    ERR_UART_RX_PIN_INVALID,            /* the selected GPIO pin is not valid
for RX */
    ERR_UART_TX_PIN_INVALID,            /* the selected GPIO pin is not valid
for TX */
    ERR_UART_MOD_X_INVALID,             /* Invalid UART_MOD_X */
    ERR_WDG_ISR_DEFAULT,                /* The default ISR for the watch dog has
been called */
    ERR_RESERVED_INTERNAL_1,            /* Reserved for internal errors */
    ERR_ADC_NOT_INIT,                   /* the ad isn't set up yet */
    ERR_HIRES_CABILRATION               /* something went wrong calibrating the
HiRes PWM */
};
```

7.3.1.1 Description

These are a list of error conditions provided by the CSL.

8 csl_gpio_t0_

8.1.1.1 Description

Contains functions to acquire the GPIO pins for use by the user and other modules such as the ePWM and UART.

Once a pin has been acquired no other module can use it. If another module attempts to acquire an already acquired pin an assertion is raised.

8.1.1.2 Examples

Configures two GPIO pins, one as an input and the other as an output. The input pin is read and the output pin is set to the same value.

```
GPIO_config( GPIO_32, GPIO_DIR_IN,  false );
GPIO_config( GPIO_31, GPIO_DIR_OUT, false );

 while ( 1 )
 {
    GPIO_setValue( GPIO_31, GPIO_get( GPIO_32 ) );
 }
```

8.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru712f/spru712f.pdf

8.2 Api

GPIO_acquire()
GPIO_config()
GPIO_setMux()
GPIO_reConfig()
GPIO_setValue()
GPIO_set()
GPIO_clr()
GPIO_tog()
GPIO_get()

8.2.1 GPIO_acquire

void GPIO_acquire(GPIO_Pin Pin)

where:
Pin - Selects a gpio pin.

8.2.1.1 Description

Acquires a GPIO pin and allows it to be configured for a particular use.

The GPIO_config() function automatically calls this acquire function before configuring the pin. However GPIO_reConfig() does not call the acquire function. If a GPIO pin is not acquired before calling GPIO_reConfig() an assertion will be raised.

Once a pin has been acquired it should not be re-acquired.

8.2.1.2 Examples

Acquires the GPIO pin and then sets up the internal GPIO mux.

```
  GPIO_acquire(  GPIO_0 );
  GPIO_reConfig( GPIO_0, GPIO_DIR_IN, false, GPIO_MUX_ALT1, GPIO_SYNCHRONIZE
);
.
```

8.2.1.3 Notes

Use GPIO_setMux() to only specify the mux.

8.2.2 GPIO_config

void GPIO_config(GPIO_Pin Pin,GPIO_Direction Direction,bool PullUp)

where:
Pin - Selects a gpio pin.
Direction - Selects the direction of the pin.
PullUp - Enables internal pull up.

8.2.2.1 Description

Acquires and sets up the specified GPIO pin as an input or output pin depending on the direction specified.

Pull ups are only functional when the pin is configured as an input.

The input is synchronized to the system clock by default. To change this use the GPIO_reConfig() function and specify the input mode as required.

To change the functionality of this pin at a later point use GPIO_reConfig(). The GPIO_reConfig() function will not re-acquire the pin.

Pins must be acquired before they can be initially configured. The GPIO_config() function automatically acquires pins by calling GPIO_acquire(). Therefore any sub-sequent reconfigurations for the same pin using GPIO_reConfig() will not need to be re-acquired.

8.2.2.2 Examples

This sets up GPIO34 as an output pin with pull-ups disabled and then sets the pin high.

```
GPIO_config( GPIO_34, GPIO_DIR_OUT, false);
GPIO_set( GPIO_34 );
```

8.2.3 GPIO_setMux

void GPIO_setMux(GPIO_Pin Pin,GPIO_Multiplex Mux)

where:
Pin - Selects a gpio pin.
Mux - This sets the GPIO mux.

8.2.3.1 Description

```
Acquires and sets up the multiplexer on the specified GPIO pin for one of
the hardware modules.

It sets the MUX to the value specified. The direction of the pin and pull ups
are set to their default values (input and pull ups disabled).

If a pin has already been configured using either GPIO_config() or
GPIO_reConfig() then it will already have been acquired. This function attempts
to acquire the pin that has been specified. If it has already been acquired an
assertion will be raised.

Thus to re-configure a pin that has already been configured or acquired use the
function GPIO_reConfig() and specify the 'Mux' parameter using that function
instead.
```

8.2.3.2 Examples

```
This acquires GPIO_3 and sets it to use mux 1.

  GPIO_setMux( GPIO_3, GPIO_MUX_ALT1 );
```

8.2.4 GPIO_reConfig

void GPIO_reConfig(GPIO_Pin PinNumber,GPIO_Direction Direction,bool PullUp,GPIO_Multiplex Mux,GPIO_InputMode InputMode)

where:
PinNumber - Selects a gpio pin.
Direction - Selects the direction of the pin.
PullUp - Enables internal pull-up.
Mux - This sets the GPIO mux.
InputMode -

8.2.4.1 Description

Updates the GPIO pin assignment for the specified pin.

This function allows the pin to be changed from IO to one of the alternative assignments using the multiplexer.

When the 'Mux' parameter is set as IO the pin will be configured as an input or output pin depending on the direction specified.

Pull ups are only functional when the pin is configured as an input.

The 'InputMode' allows a pin, configured as an input, to be sampled at specific windows.

The pin must be acquired before this function is called. Use the GPIO_acquire() function to acquire the pin.

If the pin has previously been configured using GPIP_config() then it has already been acquired. If this is the case then GPIO_acquire() should not be called before GPIO_reConfig().

8.2.4.2 Examples

This acquires GPIO_1 as an input without pull-ups and uses 6 samples. Each sampling window is 510*SYSCLKOUT.

```
GPIO_acquire( GPIO_1 );
GPIO_reConfig( GPIO_1, GPIO_DIR_IN, false, GPIO_MUX_GPIO, GPIO_SAMPLES_6 );
```

8.2.5 GPIO_setValue

void GPIO_setValue(GPIO_Pin Pin,int Value)

where:
Pin - Selects a gpio pin.
Value - The value to set the GPIO pin to.

8.2.5.1 Description

Sets the value of the GPIO pin to the logic value specified.

If the argument 'Value' is non-zero the pin will be set to logic 1. Otherwise the pin will be set to logic 0.

The GPIO pin must be set up and configured as an output pin using GPIO_config() or equivalent prior to using this function.

This function has no effect if the pin is not configured as an output.

8.2.5.2 Examples

This sets the values of the GPIO_33 and GPIO_34 pins to logic 1.

```
GPIO_setValue(GPIO_33, 1);
GPIO_setValue(GPIO_34, true);
```

8.2.5.3 Notes

For time critical applications use

```
GPIO_set()
GPIO_clr()
```

8.2.6 GPIO_set

void GPIO_set(GPIO_Pin Pin)

where:
Pin - Selects a gpio pin.

8.2.6.1 Description

Sets the value of the specified GPIO pin to logic 1.

The GPIO pin must be set up and configured as an output pin using GPIO_config() or equivalent prior to using this function.

This function has no effect if the pin is not configured as an output.

8.2.6.2 Examples

This sets the value of the GPIO_34 pin to logic 1.

```
GPIO_set(GPIO_34);
```

NOTES

8.2.7 GPIO_clr

void GPIO_clr(GPIO_Pin Pin)

where:
Pin - Selects a gpio pin.

8.2.7.1 Description

Sets the GPIO pin to logic 0.

The GPIO pin must be set up and configured as an output pin using GPIO_config() or equivalent prior to using this function.

This function has no effect if the pin is not configured as an output.

8.2.7.2 Examples

This clears the value of the GPIO_34 pin to logic 0.

```
GPIO_clr(GPIO_34);
```

NOTES

8.2.8 GPIO_tog

void GPIO_tog(GPIO_Pin Pin)

where:
Pin - Selects a gpio pin.

8.2.8.1 Description

Toggles the output value of the specified GPIO pin.

The GPIO pin must be set up and configured as an output pin using GPIO_config() or equivalent prior to using this function.

This function has no effect if the pin is not configured as an output.

8.2.8.2 Examples

This toggles the value of the GPIO_34 pin.

```
GPIO_tog(GPIO_34);
```

NOTES

8.2.9 GPIO_get

uint16_t GPIO_get(GPIO_Pin Pin)

where:
Pin - Selects a gpio pin.

8.2.9.1 Description

Reads the current logical value from the specified GPIO pin.

The GPIO pin must be set up using GPIO_config() or equivalent prior to using this function.

If the pin under test is high then a non-zero value is returned, specifically 2^(GPIO_Pin_Number & 0xF). Otherwise zero is returned.

8.2.9.2 Examples

This returns a logical value for GPIO_34.

```
GPIO_get(GPIO_34);
```

If GPIO_34 is high then ((uint16_t) 4) will be returned.
If GPIO_34 is low then zero will be returned.

NOTES

8.3 Types

8.3.1 GPIO_Pin

```
enum GPIO_Pin
{
    GPIO_0,
    GPIO_1,
    GPIO_2,
    GPIO_3,
    GPIO_4,
    GPIO_5,
    GPIO_6,
    GPIO_7,
    GPIO_8,
    GPIO_9,
    GPIO_10,
    GPIO_11,
    GPIO_12,
    GPIO_13,
    GPIO_14,
    GPIO_15,
    GPIO_16,
    GPIO_17,
    GPIO_18,
    GPIO_19,
    GPIO_20,
    GPIO_21,
    GPIO_22,
    GPIO_23,
    GPIO_24,
    GPIO_25,
    GPIO_26,
    GPIO_27,
    GPIO_28,
    GPIO_29,
    GPIO_30,
    GPIO_31,
    GPIO_32,
    GPIO_33,
    GPIO_34,          /* last 2808 pin */
    GPIO_35,
    GPIO_36,
    GPIO_37,
    GPIO_38,
    GPIO_39,
    GPIO_40,
    GPIO_41,
    GPIO_42,
    GPIO_43,
    GPIO_44,
    GPIO_45,
    GPIO_46,
    GPIO_47,
    GPIO_48,
    GPIO_49,
    GPIO_50,
    GPIO_51,
    GPIO_52,
    GPIO_53,
```

```
    GPIO_54,
    GPIO_55,
    GPIO_56,
    GPIO_57,
    GPIO_58,
    GPIO_59,
    GPIO_60,
    GPIO_61,
    GPIO_62,
    GPIO_63,
    Gpio_MAX
};
```

8.3.1.1 Description

These are the GPIO pins.

8.3.2 GPIO_Direction

```
enum GPIO_Direction
{
    GPIO_DIR_IN      = 0,     /* input */
    GPIO_DIR_OUT     = 1      /* output */
};
```

8.3.2.1 Description

This specifies the GPIO as an input or output pin.

8.3.3 GPIO_Multiplex

```
enum GPIO_Multiplex
{
    GPIO_MUX_GPIO    = 0,
    GPIO_MUX_ALT1    = 1,
    GPIO_MUX_ALT2    = 2,
    GPIO_MUX_ALT3    = 3
};
```

8.3.3.1 Description

Each GPIO pin can have different functionality. This is selected by specifing the correct mux for each pin.

8.3.4 GPIO_Level

```
enum GPIO_Level
{
    GPIO_NON_INVERT,          /* non-invert output */
    GPIO_INVERT               /* invert output */
};
```

8.3.4.1 Description

This is used to indicate the polarity of the pin.

8.3.5 GPIO_TriState

```
enum GPIO_TriState
{
    GPIO_FLOAT,          /* pin set to high impedance */
    GPIO_SET,            /* pin set to logic 1 */
    GPIO_CLR,            /* pin set to logic 0 */
    GPIO_NO_ACTION       /* no action taken */
};
```

8.3.5.1 Description

This is used to indicate the tri state value of the pin.

8.3.6 GPIO_InputMode

```
enum GPIO_InputMode
{
    GPIO_SYNCHRONIZE,        /* Synchronize to SYSCLKOUT only. Valid for both
peripheral and GPIO pins */
    GPIO_SAMPLE_3,           /* Qualification using 3 samples */
    GPIO_SAMPLES_6,          /* Qualification using 6 samples */
    GPIO_ASYNCHRONOUS        /* Asynchronous. This option applies to pins
configured as peripherals only */
};
```

8.3.6.1 Description

Selects input qualification type for GPIO pins.

9 csl_int_t0_

9.1.1.1 Description

Contains functions to handle and configure the interrupts associated with the Peripheral Interrupt Expansion controller.

The CPU is limited to 12 interrupt signals. Therefore to service all the interrupts from the modules and external sources a Peripheral Interrupt Expansion controller (PIE) is used.

The PIE is a complicated multiplexing interrupt module. These functions attempts to simplify the use of the PIE controller.

For more information about the PIE read "TMS320x280x, 2801x, 2804x DSP System Control and Interrupts" and familiarize yourself with the terminology used and how the controller functions.

The functions from this module can be used to set up an interrupt service routine (ISR) to be called when an interrupt is raised by another module. The example below shows how this is possible.

Please check to see if a customized interrupt configuration function is available for the specific module that is being used. Check by browsing the functions associated with that module. For example, when using the ADC module the function ADC_setCallaback() can be used to assign an ISR to the ADC interrupt event. The same result can be achieved using the functions provided in this module.

9.1.1.2 Examples

Assigns the interrupt service routine function to the interrupt vector associated with the ePWM module 1 interrupt. Enables the interrupt within the PIE controller and also enables the global interrupt flag.

```
INT_setCallback( INT_pieIdToVectorId( INT_ID_EPWM1 ), isr_pwm1 );
INT_enablePieId( INT_ID_EPWM1, true );
INT_enableGlobal( true );
```

The interrupt service routine must acknowledge the correct interrupt and clear the peripheral interrupt flag.

```
interrupt void isr_pwm1( void )
{
   // Acknowledge PIE interrupt group and clear peripheral interrupt flag
   INT_ackPieId(INT_ID_EPWM1);
}
```

9.1.1.3 Notes

INT_setCallback() and INT_enablePieId() should only be called when global interrupts are disabled.

Convert a INT_PieId to a GROUP_Id using INT_pieIdToGroup().

Convert a INT_PieId to a bit index using INT_pieIdToIndex().

9.1.1.4 Links

```
file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru712f/spru712f.pdf
```

9.2 Api

INT_setCallback()
INT_enableGlobal()
INT_enableInt()
INT_enablePieIndex()
INT_enablePieGroup()
INT_enablePieId()
INT_ackInt()
INT_ackPieIndex()
INT_ackGroup()
INT_ackPieId()
INT_pieIdToVectorId()
INT_pieIdToGroup()
INT_pieIdToIndex()
INT_ackPieGroup()

9.2.1 INT_setCallback

void INT_setCallback(INT_VectorId VectorId,INT_IsrAddr Func)

where:
VectorId - The index into the vector table for the required interrupt.
Func - The pointer to the interrupt function.

9.2.1.1 Description

Assigns a function, an interrupt service routine (ISR), to the specified interrupt vector.

This function need not normally be called directly as each module has its own XXX_setCallback function. These functions are unique to each module and call this main INT_setCallback() function with the correct INT_VectorId.

Each module has a matching interrupt vector, e.g.

```
 Module        INT_VectorId
 PWM_MOD_1     INT_pieIdToVectorId(INT_ID_EPWM1)
     :               :
 PWM_MOD_6     INT_pieIdToVectorId(INT_ID_EPWM6)
 ADC_MOD_1     INT_pieIdToVectorId(INT_ID_SEQ1)
 ADC_MOD_2     INT_pieIdToVectorId(INT_ID_SEQ2)
 ADC_MOD_3     INT_pieIdToVectorId(INT_ID_SEQ1)
```

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the INT_setCallback() function as the address of the ISR is used in the function call.

PIE controller interrupts are not enabled by this function and must be done so using the INT_enablePieIndex() function.

No interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

To allow the ISR to be called after the first interrupt the PIE group acknowledgement bit must be cleared using INT_ackGroup() or INT_ackPieGroup().

9.2.1.2 Examples

The interrupt function isr_xint1() will be called when an external interrupt is generated.

```
  INT_setCallback( INT_pieIdToVectorId( INT_ID_XINT2 ), isr_xint2 );

  // Enable the ID within the PIE group for the external interrupts
  INT_enablePieIndex( INT_ID_XINT1, true );
  INT_enablePieIndex( INT_ID_XINT2, true );

  // Enable the interrupt that XINT1 and XINT2 are part of
  INT_enableInt( 0 );
```

```
// Enable global interrupts
INT_enableGlobal( true );


interrupt void isr_xint1( void )
{
   // Acknowledge the PIE group interrupt
   INT_ackPieGroup( INT_ID_XINT1 );
}
```

9.2.1.3 Notes

There is not always a one-to-one mapping of modules to interrupt IDs.

9.2.2 INT_enableGlobal

void INT_enableGlobal(int Enable)

where:
Enable - Enables global interrupts.

9.2.2.1 Description

Enables/disables global interrupts.

Until global interrupts are enabled all interrupts are blocked.

If it is not already done so, individual interrupts will need to be enabled within the interrupt enable register and the PIE group enable registers. This can be performed using the functions INT_enableInt() and INT_enablePieIndex() respectively.

However, the XXX_setCallback() functions associated with the peripheral modules will enable the required interrupts - apart from the global interrupt which this this function sets.

9.2.2.2 Examples

Enables global interrupts.

```
INT_enableGlobal(true);
```

9.2.3 INT_enableInt

void INT_enableInt(int IntId)

where:
IntId - Index of interrupt.

9.2.3.1 Description

Enables one of the 12 DSP interrupts.

This function need not normally be called directly if the XXX_setCallback() function associated with a peripheral module is being used. The XXX_setCallback() functions are unique to each module and will call this INT_enableInt() function with the correct index of the interrupt.

9.2.3.2 Examples

This enables INT5.

```
INT_enableInt(4);
```

9.2.3.3 Notes

There are INT1 to INT12 interrupts, e.g. 0..11

9.2.4 INT_enablePieIndex

void INT_enablePieIndex(INT_PieId PieId,int Value)

where:
PieId - Selects the peripheral interrupt ID.
Value - Enable value.

9.2.4.1 Description

Enables/disables the interrupt bit associated with each PIE group for the specified interrupt ID.

Up to 8 PieIds can belong to one PIE group. The PIE group is extracted from the PieId and the required bit within the PIE group enable register is changed.

Unless it is already done so, PIE interrupts must be enabled using this function before any interrupt requests will be serviced by the PIE controller.

Before an interrupt can be actioned the interrupt group must be enabled using INT_enablePieGroup().

This function need not normally be called directly if the XXX_setCallback() function associated with a peripheral module is being used. The XXX_setCallback() functions are unique to each module and will call this INT_enablePieIndex() function with the correct interrupt ID.

9.2.4.2 Examples

This enables bit 0 of PIE Group 1, since INT_ID_EPWM1 belongs to PIE Group 1.

```
INT_enablePieIndex(INT_ID_EPWM1, 1);
```

NOTES

9.2.5 INT_enablePieGroup

void INT_enablePieGroup(INT_PieId PieId,int Value)

where:
PieId - Selects the peripheral interrupt ID.
Value - Enable value.

9.2.5.1 Description

Enables/disables interrupts for the entire PIE group containing the interrupt ID.

The function sets or clears the bit within the interrupt enable register associated with a particular PIE group. The PIE group is the one which contains the interrupt ID that has been passed as an argument to this function.

Before an individual interrupt can be actioned it must be enabled using INT_enablePieIndex().

This function need not normally be called directly if the XXX_setCallback() function associated with a peripheral module is being used. The XXX_setCallback() functions are unique to each module and will enable the correct interrupts associated with the interrupt ID of the peripheral.

9.2.5.2 Examples

Since INT_ID_EPWM1 belongs to PIE Group 1, this enables INT1 which contains the interrupts for the EPWMs.

```
INT_enablePieGroup( INT_ID_EPWM1, true );
```

NOTES

9.2.6 INT_enablePieId

void INT_enablePieId(INT_PieId PieId,int Value)

where:
PieId - Selects the peripheral interrupt id.
Value - Enable value.

9.2.6.1 Description

Enables/disables both sets of interrupt enable registers by calling INT_enablePieIndex() and INT_enablePieGroup().

This enables/disables interrupts for the entire PIE group within the CPU registers and enables/disables the interrupt associated with the interrupt ID within the correct PIE group register.

This function need not normally be called directly if the XXX_setCallback() function associated with a peripheral module is being used. The XXX_setCallback() functions are unique to each module and will enable all of the correct interrupts associated with the interrupt ID of the peripheral.

9.2.6.2 Examples

This enables the interrupt for the PIE Group that INT_ID_EPWM1 belongs to and also sets the bit which enables interrupts for INT_ID_EPWM1 within the PIE group enable register.

```
INT_enablePieId(INT_ID_EPWM1, 1);
```

9.2.7 INT_ackInt

void INT_ackInt(int IntId)

where:
IntId - Interrupt index.

9.2.7.1 Description

Clears the interrupt flag for the (IntId+1) interrupt.

There are twelve possible interrupts that can be cleared,

```
Interrupts INT1 to INT12
Indexes    0    to 11
```

9.2.7.2 Examples

Clears the INT3 flag.

```
   INT_ackInt(2);
```

NOTES

9.2.8 INT_ackPieIndex

void INT_ackPieIndex(INT_PieId PieId)

where:
PieId - Selects the peripheral interrupt id.

9.2.8.1 Description

Clears the interrupt flag within the corresponding PIE group interrupt register for the INT_PieId specified.

This does not acknowledge the PIE group and therefore any subsequent interrupt calls will not be serviced until the previous PIE interrupt has been acknowledged. This can be achieved using INT_ackPieGroup().

Both actions can be performed using INT_ackPieId().

9.2.8.2 Examples

Clears bit 0 in the PIE Group 1 interrupt flag register.

```
INT_ackPieIndex(INT_ID_EPWM1);
```

9.2.9 INT_ackGroup

void INT_ackGroup(INT_PieGroup GroupId)

where:
GroupId - Selects the peripheral group id.

9.2.9.1 Description

Clears the bit corresponding to the PIE group within the PIE acknowledgement register.

This must be called after an interrupt service routine (ISR) has been entered. If the PIE acknowledgement flag for this PIE group is not cleared then any subsequent interrupts associated with this group will not be serviced by the PIE controller.

This should only be called from inside a interrupt handler or when global interrupts are disabled to avoid missing other interrupts within the group.

9.2.9.2 Examples

This clears bit 1 within the PIE acknowledgement register.

```
  INT_ackGroup( INT_GROUP_1 );
```

The same result can be achieved in using INT_ackPieGroup() if the PIE ID associated with this group is known. Alternatively, the INT_pieIdToGroup() function can be used to convert the known PIE ID to a Group ID.

This clears bit 1 within the PIE acknowledgement register.

```
  INT_ackGroup( INT_pieIdToGroup( INT_ID_XINT1 ) );
```

NOTES

9.2.10 INT_ackPieId

void INT_ackPieId(INT_PieIndex PieId)

where:
PieId - Selects the peripheral interrupt id.

9.2.10.1 Description

Clears both the acknowledgement bit and the interrupt flag for the specified INT_PieIndex.

The function calls both INT_ackPieGroup() and INT_ackPieIndex().

This must be called after an interrupt service routine (ISR) has been entered. It will clear both the acknowledgement bit and interrupt flag allowing any subsequent interrupt flags to be serviced by PIE controller.

This should only be called from inside a interrupt handler or when global interrupts are disabled to avoid missing other interrupts within the group.

9.2.10.2 Examples

Clears bit 1 in the PIE acknowledgement register. Clears bit 4 in the PIE Group 1 interrupt flag register.

```
  INT_ackPieId( INT_ID_XINT1 );
```

9.2.11 INT_pieIdToVectorId

INT_VectorId INT_pieIdToVectorId(INT_PieIndex PieId)

where:
PieId - Selects the peripheral interrupt id.

9.2.11.1 Description

Converts the INT_PieIndex to the corresponding vector ID (INT_VectorId).

This can be used to assign an interrupt function to the interrupt vector.

9.2.11.2 Examples

Assigns the interrupt function for INT_ID_TZINT1.

```
INT_setCallback( INT_pieIdToVectorId( INT_ID_TZINT1 ), epwm1_tzint_isr );
```

9.2.12 INT_pieIdToGroup

INT_PieGroup INT_pieIdToGroup(INT_PieIndex PieId)

where:
PieId - Selects the peripheral interrupt ID.

9.2.12.1 Description

Converts the interrupt ID (INT_PieIndex) to the corresponding PIE interrupt group (INT_PieGroup).

The format of the PIE multiplexed interrupts are,

```
  INTx.y
```

Where up to eight interrupts, y, are mapped on to the twelve system interrupts, x, INT1 to INT12. This function returns the 'x' value.

9.2.12.2 Examples

Returns INT_GROUP_3 (2) as the bit index of INT_ID_EPWM1 within its interrupt group.

```
  PieGroup = INT_pieIdToGroup( INT_ID_EPWM1 );
```

9.2.13 INT_pieIdToIndex

INT_PieIndex INT_pieIdToIndex(INT_PieIndex PieId)

where:
PieId - Selects the peripheral interrupt ID.

9.2.13.1 Description

```
Converts the interrupt ID (INT_PieIndex) to the bit index (INT_PieIndex) of the
multiplexed interrupt within the PIE interrupt group.

The format of the PIE multiplexed interrupts are,

  INTx.y

Where up to eight interrupts, y, are mapped on to the twelve system
interrupts, x, INT1 to INT12. This function returns the 'y' value.
```

9.2.13.2 Examples

```
Returns INT_INDEX_1 (0) as the bit index of INT_ID_EPWM1 within its interrupt
group.

  PieIndex = INT_pieIdToIndex( INT_ID_EPWM1 );
```

9.2.14 INT_ackPieGroup

void INT_ackPieGroup(INT_PieIndex PieId)

where:
PieId - Selects the peripheral interrupt id.

9.2.14.1 Description

Clears the bit corresponding of the PIE group within the PIE acknowledgement register for the specified PIE ID.

This must be called after an interrupt service routine (ISR) has been entered. If the PIE acknowledgement flag for this PIE group is not cleared then any subsequent interrupts will not be serviced by the PIE controller.

This should only be called from inside a interrupt handler or when global interrupts are disabled to avoid missing other interrupts within the group.

9.2.14.2 Examples

Clears bit 1 in the PIE acknowledgement register.

```
INT_ackPieGroup( INT_ID_XINT1 );
```

The same result can be achieved by using INT_ackGroup() if the group ID associated with the device generating an interrupt is known.

9.3 Types

9.3.1 INT_VectorId

```
enum INT_VectorId
{
    INT_VECT_RESET      = 0,
    INT_VECT_INT_13     = 13,   /* TIM_MOD_2 */
    INT_VECT_INT_14     = 14    /* TIM_MOD_3 */
};
```

9.3.1.1 Description

This is a short list of PIE vectors. There are a lot more not listed below, but these are generally generated from a INT_PieId value and INT_pieIdToVectorId().

9.3.2 INT_PieGroup

```
enum INT_PieGroup
{
    INT_GROUP_1,
    INT_GROUP_2,
    INT_GROUP_3,
    INT_GROUP_4,
    INT_GROUP_5,
    INT_GROUP_6,
    INT_GROUP_7,
    INT_GROUP_8,
    INT_GROUP_9,
    INT_GROUP_10,
    INT_GROUP_11,
    INT_GROUP_12
};
```

9.3.2.1 Description

This is the PIE groups 1 to 12. Each PIE group can have up to 8 different interrupt sources.

9.3.3 INT_PieIndex

```
enum INT_PieIndex
{
    INT_INDEX_1,
    INT_INDEX_2,
    INT_INDEX_3,
    INT_INDEX_4,
    INT_INDEX_5,
    INT_INDEX_6,
    INT_INDEX_7,
    INT_INDEX_8
};
```

9.3.3.1 Description

This is the index to each PIE Group.

9.3.4 INT_IsrAddr

```
typedef interrupt void(*INT_IsrAddr)(void);
```

9.3.4.1 Description

This is the prototype for the interrupt service functions.

10 csl_wdg_t0_

10.1.1.1 Description

Contains functions to configure and use the watch dog module.

By default the watch dog timer is disabled.

10.1.1.2 Examples

Configures the watch dog timer to cause a reset if it expires. The main idle loop kicks the watch dog to prevent this from occurring.

```
  WDG_config( WDG_RESET, NULL );

  while(1)
  {
     WDG_kick();
  }
```

10.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru712f/spru712f.pdf

10.2 Api

`WDG_config()`
`WDG_kick()`
`WDG_ackInt()`

10.2.1 WDG_config

void WDG_config(WDG_Mode Mode,INT_IsrAddr Func)

where:
Mode - Selects the watch dog mode.
Func - The pointer to the interrupt function.

10.2.1.1 Description

Enables/disables the watch dog and sets the action to be taken when the watch dog expires.

The watch dog can be set in either reset or interrupt mode or it can be disabled altogether.

If the interrupt mode is selected global interrupts must be enabled using INT_enableGlobal(true) for the interrupt to be serviced. The associated PIE group interrupt and the individual PIE interrupt are both enabled by this function.

The interrupt service routine (ISR) assigned to the watch dog interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the WDG_config() function as the address of the ISR is used in the function call.

The reset mode configures the watch dog to reset the device by pulling !XRS low. In this case no ISR function will be needed and 'NULL' can be passed as the argument to this function.

The watch dog event will be triggered when the watch dog timer expires. The timer must be reset before it expires using WDG_kick().

By default the watch dog time out is set to USR_CLK_IN_HZ/(512*64).

10.2.1.2 Examples

Causes the watch dog to reset the device if it is not kicked within a certain time period.

```
WDG_config( WDG_RESET, NULL );
```

Causes the watch dog to call the ISR function isr_wdg() if it is not kicked within a certain time period.

```
 WDG_config( WDG_INTERRUPT, isr_wdg );

interrupt void isr_wdgr( void )
{
   WDG_ackInt();
}
```

10.2.2 WDG_kick

void WDG_kick(void)

where:

10.2.2.1 Description

Kicks the watch dog and prevents it from expiring.

If the watch dog is not kicked the action defined by the call to WDG_config() will be taken when the timer expires.

10.2.2.2 Examples

Kicks the watch dog, which resets its timer, and prevents it from expiring.

```
  WDG_kick();
```

10.2.3 WDG_ackInt

void WDG_ackInt(void)

where:

10.2.3.1 Description

Used within an interrupt service routine (ISR) to clear the watch dog PIE group flag.

10.2.3.2 Examples

The watch dog is configured to call the ISR isr_wdg() if its timer expires. Inside the ISR the interrupt flags are cleared to allow future watch dog interrupt functions to be called.

```
 WDG_config( WDG_INTERRUPT, isr_wdg );

interrupt void isr_wdgr( void )
{
   WDG_ackInt();
}
```

10.3 Types

10.3.1 WDG_Mode

```
enum WDG_Mode
{
    WDG_NONE,               /* disable watchdog */
    WDG_RESET,              /* cause a reset */
    WDG_RESET_LOCK,         /* cause a reset, can't be changed */
    WDG_INTERRUPT,          /* cause an interrupt */
    WDG_INTERRUPT_LOCK      /* cause an interrupt, can't be changed */
};
```

10.3.1.1 Description

This is used to select the different watch dog modes of operation.

11 csl_adc_t3_

11.1.1.1 Description

Controls the ADC modules. Use the functions to configure an ADC module to sample and convert an input channel according to a trigger source and then read the result.

ADC_init() must be called before any of the API functions are called.

There are 16 start of conversion registers which are referred to within this library as ADC modules. Therefore there are 16 ADC modules which can be configured using ADC_config() to sample any combination of the analog input channels. There is only one ADC core which performs the conversions. Therefore only one module can be serviced at a time.

A trigger will cause the start of conversion for each module. As there is only one ADC core, a round robin arbitration process determines which module will be serviced by the ADC core.

As stated in <http://focus.ti.com/lit/ug/spruge5a/spruge5a.pdf> it takes 13 ADC clock ticks to convert a sample. The minimum number of clock ticks required to sample a channel is 7. Therefore the minimum time for a conversion is 20 clock ticks. The ADC module clock ticks are derived directly from the system clock. Therefore if the system clock is 60MHz the minimum time for a complete conversion would be,

```
Tmin = 20 cycles x 16.6ns = 333ns
```

When using a ePWM SOC you need to allow for the following timings

```
ADC SOC to sample (2xsys clk)        33.2ns
ADC sample & Hold (7xsys clk)       116.2ns
           total time to 1st sample 150ns
```

Once a sample has been taken, it takes 600ns to enter the ISR routine using late interrupt pulse.
If you use Early interrupt pulse then this is reduce to 380ns.

The ADC module, channels to sample, and how the conversion is triggered must be configured in order to use the ADC.

11.1.1.2 Examples

Configures the channels A0 to A4 to be sampled using ADC modules 1 to 5. The function isr_adc_hall is set as the interrupt service routine that is called when the last ADC module has completed its conversion. Global interrupts are enabled and the ePWM module 1 SOC A is used to generate the start of conversion pulse.

```
void main( void )
{
   ADC_config( ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM1_SOCA );
   ADC_config( ADC_MOD_2, ADC_SH_WIDTH_7, ADC_CH_A1, ADC_TRIG_EPWM1_SOCA );
   ADC_config( ADC_MOD_3, ADC_SH_WIDTH_7, ADC_CH_A2, ADC_TRIG_EPWM1_SOCA );
   ADC_config( ADC_MOD_4, ADC_SH_WIDTH_7, ADC_CH_A3, ADC_TRIG_EPWM1_SOCA );
   ADC_config( ADC_MOD_5, ADC_SH_WIDTH_7, ADC_CH_A4, ADC_TRIG_EPWM1_SOCA );
```

```
    ADC_setCallback( ADC_MOD_5, isr_adc_hall, ADC_INT_1 );

    PWM_setAdcSoc( PWM_MOD_1, PWM_CH_A, PWM_INT_ZERO );

    INT_enableGlobal( true );
}

interrupt void isr_adc_hall( void )
{
    // Acknowledge interrupt
    ADC_ackInt( ADC_INT_1 );
    r1 = ADC_getValue( ADC_MOD_1 );
    r2 = ADC_getValue( ADC_MOD_1 );
    r3 = ADC_getValue( ADC_MOD_1 );
    r4 = ADC_getValue( ADC_MOD_1 );
    r5 = ADC_getValue( ADC_MOD_1 );
    ...
}
```

11.1.1.3 Notes

On some devices ADC_CH_A5 and ADC_CH_B5 are not implemented in hardware. Check the device specific datasheet for further details.

11.1.1.4 Links

file:///C:/tidcs/c28/CSL_C2802x/v100/doc/CSL_C2802x.pdf
http://www.ti.com/litv/pdf/spru716c

11.2 Api

ADC_init()
ADC_setEarlyInterrupt()
ADC_setCallback()
ADC_startConversion()
ADC_clrInt()
ADC_getValue()
ADC_setPriority()
ADC_ackInt()
ADC_socSoftware()
ADC_isReady()
ADC_config()
ADC_getIndex()
ADC_setExternalRefernce()
ADC_getPieId()
ADC_getIqValueMult()

11.2.1 ADC_init

void ADC_init(void)

where:

11.2.1.1 Description

Initializes the ADC module. This function must be called before any of the API functions are called.

By default the ADC is configured to use,

Internal bandgap reference.
Interrupt flags are configured to be set one clock cycle before the result is stored in result registers. However interrupts are disabled by default.

This function will delay for approximately five milliseconds whilst the ADC initializes.

11.2.1.2 Examples

Initializes the ADC module.

```
ADC_init();
```

NOTES

11.2.2 ADC_setEarlyInterrupt

void ADC_setEarlyInterrupt(int Enable)

where:
Enable -

11.2.2.1 Description

This changes when the inettrupt is generated for a EOC pulse.

Early;
INT pulse generation occurs when ADC begins conversion. You not read the ADC value for another 12 ADC clkc (eg 13*16.6ns = 214ns)

Late;
INT pulse generation occurs 1 cycle prior to ADC result latching into its result register

11.2.2.2 Examples

Initializes the ADC module to use early interrupt pulse.

```
  ADC_setEarlyInterrupt(true);
```

NOTES

11.2.3 ADC_setCallback

void ADC_setCallback(ADC_Module Mod,INT_IsrAddr Func,ADC_Interrupt AdcInt)

where:
Mod - Selects the ADC module.
Func - The pointer to the interrupt function.
AdcInt -

11.2.3.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt specified in the function call and configures the interrupt to be set by the specified ADC module.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the ADC_setCallback() function as the address of the ISR is used in the function call.

If the address of the ISR passed is NULL then only the interrupt specified in the function call is configured to be set by the ADC module. This can then be used to determine when a conversion is complete for the ADC module using the function ADC_isReady(). In this instance no ISR would be called when the interrupt flag is set.

If the address of the ISR passed to the function is not NULL then the PIE controller interrupts are enabled automatically by this function.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The ADC interrupt flags are cleared by the function however the PIE group flag is not. To allow the ISR to be called after the first interrupt the ADC module interrupt flag and PIE group flag must be cleared using ADC_ackInt().

11.2.3.2 Examples

The interrupt function isr_adc_int6() will be called when the ADC module 1, ADC_MOD_1, completes its conversion. The interrupt function another_isr() will be called when the ADC module 2, ADC_MOD_2, complete its conversion.

```
interrupt void isr_adc_int6( void )
{
   ADC_ackInt( ADC_INT_6 );
}

interrupt void another_isr( void )
{
   ADC_ackInt( ADC_INT_5 );
}

ADC_setCallback( ADC_MOD_1, isr_adc_int6, ADC_INT_6 );
ADC_setCallback( ADC_MOD_2, another_isr,  ADC_INT_5 );
```

```
  INT_enableGlobal( true );
```

11.2.3.3 Notes

If a NULL pointer is passed as the function pointer, then the interrupt will be enable for the module, but not within the PIE module.

11.2.4 ADC_startConversion

uint16_t ADC_startConversion(ADC_Module Mod,ADC_Interrupt AdcInt)

where:
Mod - Selects the ADC module.
AdcInt - Selects the ADC interrupt to assign to the ADC module.

11.2.4.1 Description

Performs an ADC conversion and then waits until the conversion is complete.

The interrupt passed to the function will be assigned to the ADC module. A start of conversion pulse will be generated for the module specified and the function will wait until the interrupt flag is set signifying the end of the conversion.

The conversion result is returned by the function. This result can also be read from the ADC result registers using ADC_getValue().

11.2.4.2 Examples

Sets up ADC_MOD_1 to sample A0 and then converts and reads the ADC in a continuous loop.

```
ADC_config( ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_NONE );

while(1)
{
   v = ADC_startConversion(ADC_MOD_1, ADC_INT_4);
}
```

11.2.5 ADC_clrInt

void ADC_clrInt(ADC_Interrupt AdcInt)

where:
AdcInt - The ADC interrupt assigned to an ADC module.

11.2.5.1 Description

Clears the ADC interrupt flag only. Does not clear the PIE group flag.

The ADC interrupt is assigned to an ADC module using the function ADC_setCallback().

After entering an interrupt service routine the ADC interrupt flag and PIE group flag must be cleared. If only the ADC interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

11.2.5.2 Examples

This only clears the ADC interrupt flag 1.

```
ADC_clrInt( ADC_INT_1 );
```

11.2.5.3 Notes

ADC_ackInt() clears both the ADC interrupt flag and PIE group flag.

11.2.6 ADC_getValue

uint16_t ADC_getValue(ADC_Module Mod)

where:
Mod - Selects the ADC module.

11.2.6.1 Description

Returns the result from the specified ADC module.

11.2.6.2 Examples

Reads the conversion result from ADC module 2.

```
  ui_Result = ADC_getValue( ADC_MOD_2 );
```

NOTES

11.2.7 ADC_setPriority

void ADC_setPriority(ADC_Module Mod)

where:
Mod - Selects the ADC module.

11.2.7.1 Description

Determines the highest cut-off point for priority mode and round robin arbitration.

When a module is configured as high priority, a start of conversion request for that module will interrupt the round robin sequence after the current conversion has completed and the high priority module will be the next module to be serviced by the ADC core.

If two high priority ADC modules are triggered at the same time then the ADC module with the lower number has the overall priority.

Following a reset no modules are configured as high priority.
sample,

This function specifies the highest module that should be given priority. The modules below this are all configured as high priority.

11.2.7.2 Examples

Set ADC_MOD_1 to ADC_MOD_5 as high priority.

```
ADC_setPriority( ADC_MOD_5 );
```

NOTES

11.2.8 ADC_ackInt

void ADC_ackInt(ADC_Interrupt AdcInt)

where:
AdcInt - The ADC interrupt assigned to an ADC module.

11.2.8.1 Description

Used within an interrupt service routine to clear both the ADC interrupt flag and the PIE group flag.

11.2.8.2 Examples

This clears the ADC interrupt flag and the PIE group flag for the specified interrupt.

```
interrupt void isr_adc_int6( void )
{
   ADC_ackInt( ADC_INT_6 );
}
```

11.2.9 ADC_socSoftware

void ADC_socSoftware(ADC_Module Mod)

where:
Mod - Selects the ADC module.

11.2.9.1 Description

```
Generates a start of conversion pulse which forces the ADC module to begin
the sampling and conversion process.

The software must detect when the conversion is complete before reading the
values from the result registers. This can be achieved using ADC_isReady().
An interrupt must be assigned to the ADC module before the end of conversion can
be detected. See ADC_isReady() for details.

EXAMPES
A SOC pulse is generated for ADC module 1.

  ADC_socSoftware( ADC_MOD_1 );

NOTES
```

11.2.10 ADC_isReady

int ADC_isReady(ADC_Interrupt AdcInt)

where:
AdcInt - The ADC interrupt assigned to an ADC module.

11.2.10.1 Description

Returns the status of the ADC interrupt flag.

A non-zero value indicates that the ADC has finished and the results are ready to be read.

If this function returns a non-zero value the ADC interrupt flag will have been set, indicating a conversion sequence is complete. This flag must be cleared using ADC_clrInt() or ADC_ackInt() before calling ADC_isReady() again.

An interrupt must be assigned to the ADC module before calling this function using ADC_setCallback().

11.2.10.2 Examples

After the ADC module 1 has been configured a conversion is started by generating a start of conversion pulse. This function is used to check whether the conversion is complete.

```
// Clear the interrupt flag ready for conversion
ADC_clrInt(AdcInt);

// Assign an ADC interrupt to be triggered when the conversion finishes
ADC_setCallback( ADC_MOD_1, NULL, ADC_INT_6 );

// Start the conversion
ADC_socSoftware( ADC_MOD_1 );

// Wait for the conversion to be complete
while ( ADC_isReady( ADC_INT_6 ) == false );
```

11.2.11 ADC_config

void ADC_config(ADC_Module Mod,ADC_SampleHoldWidth SH,ADC_Channel Chan,ADC_TriggerSelect TrigSel)

where:
Mod - Selects the ADC module.
SH - The duration of sampling as a number of clock cycles.
Chan - The channel to sample.
TrigSel - The trigger to start the sampling and conversion process.

11.2.11.1 Description

Configures an ADC module to sample an analog input channel for a specified number of clock cycles and start the sample/conversion process based on a trigger event.
numbers.

There are 16 ADC modules, ADC_MOD_1 to ADC_MOD_16, that can be configured to sample any combination of analog input channels. The same channel can be assigned to more than one ADC module.

The minimum sample width is 7 clock cycles, ADC_SH_WIDTH_7.

The trigger will cause the start of conversion. As there is only one ADC core, a round robin arbitration process determines which module will be serviced by the ADC core at any one time. Therefore multiple ADC modules can be triggered by the same event. The module that is serviced first will depend upon the last serviced module and any high priority ADC module set using ADC_setPriority().

11.2.11.2 Examples

Configures ADC modules 1 to 4 to sample analog input channel A0. The sampling window will last for 7 clock cycles. The sampling and conversion process is started by the SOC pulse generated from PWM module 1. All four modules are triggered by the same event. The round robin pointer will begin at ADC module 1 and increment to ADC module 4.

An ADC interrupt, ADC_INT_6, is assigned to ADC_MOD_4 and will be triggered when ADC module 4 finishes its conversion. The interrupt service routine (ISR), named isr_adc_int6, is assigned to this ADC interrupt. This ISR will be called when ADC module 4 finishes its conversion.

```
ADC_config( ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM1_SOCA );
ADC_config( ADC_MOD_2, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM1_SOCA );
ADC_config( ADC_MOD_3, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM1_SOCA );
ADC_config( ADC_MOD_4, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM1_SOCA );

ADC_setCallback( ADC_MOD_4, isr_adc_int6, ADC_INT_6 );
```

NOTES

11.2.12 ADC_getIndex

int ADC_getIndex(ADC_Module Mod)

where:
Mod - Selects the ADC module.

11.2.12.1 Description

Returns the index of the ADC module.

ADC_isReady()

11.2.12.2 Examples

Returns 0 for ADC_MOD_1

```
  intIdx = ADC_getIndex( ADC_MOD_1 );
```

NOTES

11.2.13 ADC_setExternalRefernce

void ADC_setExternalRefernce(int Enable)

where:
Enable - Enable or disable the external reference.

11.2.13.1 Description

Enables or disables the external reference voltage for all ADC modules.

By default the internal reference is set up and used when ADC_init() is called. The function delays for 5ms whilst the internal reference voltage settles.

11.2.13.2 Examples

The internal reference is disabled and an external reference voltage is used from pins VREFLO and VREFHI.

```
ADC_setExternalRefernce( true );
```

11.2.14 ADC_getPieId

INT_PieId ADC_getPieId(ADC_Interrupt AdcInt)

where:
AdcInt - The ADC interrupt assigned to an ADC module.

11.2.14.1 Description

Returns the PIE Id associated with the ADC interrupt.

This can be used when configuring the PIE controller manually using the interrupt functions from the INT CSL library.

This function does not normally need to be called as the interrupts are configured automatically using ADC_setCallback().

11.2.14.2 Examples

The function will return INT_ID_ADCINT2 when the ADC interrupt ADC_INT_2 is passed as the argument.

```
PieIdAdc = ADC_getPieId( ADC_INT_2 );
```

11.2.15 ADC_getIqValueMult

void ADC_getIqValueMult(ADC_Module Mod,_iq Mult)

where:
Mod - Selects the ADC sequencer.
Mult - IQ multiplier.

11.2.15.1 Description

Returns the ADC value as an _iq number after multiplying it with the _iq number passed as an argument to the function.

The ADC value is read from the result registers at the index value of the ADC module specified in the function call.

11.2.15.2 Examples

Reads the value from index 2 and multiples it by 3.14.

```
iq_result = ADC_getIqValueMult(ADC_MOD_1, 2, _IQ(3.14) );
```

If the result stored in the ADC mirror register ADCRESULT2 is 0x9 and an _iq value of _iq(3.14) is passed as an argument to this function, the _iq value returned will be 102891 assuming a GLOBAL_Q of 12 is defined.

In this case the ADC result, 0x9, is divided by 4096 (left-shifted 12 places) and multiplied by the _iq version of 3.14.

11.2.15.3 Notes

The GLOBAL_Q must be greater than or equal to the number of ADC bits, e.g. 12.

11.3 Types

11.3.1 ADC_TYPE_3

```
#define ADC_TYPE_3
```

11.3.1.1 Description

This can be use to determine the peripheral type used by the csl.

11.3.2 ADC_Module

```
enum ADC_Module
{
    ADC_MOD_1,          /* ADC SOC 0 */
    ADC_MOD_2,          /* ADC SOC 1 */
    ADC_MOD_3,          /* ADC SOC 2 */
    ADC_MOD_4,          /* ADC SOC 3 */
    ADC_MOD_5,          /* ADC SOC 4 */
    ADC_MOD_6,          /* ADC SOC 5 */
    ADC_MOD_7,          /* ADC SOC 6 */
    ADC_MOD_8,          /* ADC SOC 7 */
    ADC_MOD_9,          /* ADC SOC 8 */
    ADC_MOD_10,         /* ADC SOC 9 */
    ADC_MOD_11,         /* ADC SOC 10 */
    ADC_MOD_12,         /* ADC SOC 11 */
    ADC_MOD_13,         /* ADC SOC 12 */
    ADC_MOD_14,         /* ADC SOC 13 */
    ADC_MOD_15,         /* ADC SOC 14 */
    ADC_MOD_16          /* ADC SOC 15 */
};
```

11.3.2.1 Description

This uses the same style as the rest of the CSL, referring to the ADC sequencer 1 & 2 as ADC_MOD_1/2.
For the special case of the cascaded sequencer ADC_MOD_3 should be used.

The naming convention should be ADC_MODULE, but we are trying to make it look like a normal pwm, i2c, etc module.

11.3.3 ADC_Channel

```
enum ADC_Channel
{
    ADC_CH_A0   = 0,
    ADC_CH_A1   = 1,
    ADC_CH_A2   = 2,
    ADC_CH_A3   = 3,
    ADC_CH_A4   = 4,
    ADC_CH_A5   = 5,
    ADC_CH_A6   = 6,
    ADC_CH_A7   = 7,
    ADC_CH_B0   = 8,
    ADC_CH_B1   = 9,
```

```
    ADC_CH_B2   = 10,
    ADC_CH_B3   = 11,
    ADC_CH_B4   = 12,
    ADC_CH_B5   = 13,
    ADC_CH_B6   = 14,
    ADC_CH_B7   = 15
};
```

11.3.3.1 Description

The analog ADC channels that can be sampled and converted to a digital value. Refer to the device datasheet for the equivalent device pins.

11.3.4 ADC_SampleHoldWidth

```
enum ADC_SampleHoldWidth
{
    ADC_SH_WIDTH_7       = 6,
    ADC_SH_WIDTH_8,
    ADC_SH_WIDTH_9,
    ADC_SH_WIDTH_10,
    ADC_SH_WIDTH_11,
    ADC_SH_WIDTH_12,
    ADC_SH_WIDTH_13,
    ADC_SH_WIDTH_14,
    ADC_SH_WIDTH_15,
    ADC_SH_WIDTH_16,
    ADC_SH_WIDTH_17,
    ADC_SH_WIDTH_18,
    ADC_SH_WIDTH_19,
    ADC_SH_WIDTH_20,
    ADC_SH_WIDTH_21,
    ADC_SH_WIDTH_22,
    ADC_SH_WIDTH_23,
    ADC_SH_WIDTH_24,
    ADC_SH_WIDTH_55      = 54,
    ADC_SH_WIDTH_56,
    ADC_SH_WIDTH_57,
    ADC_SH_WIDTH_58,
    ADC_SH_WIDTH_59,
    ADC_SH_WIDTH_60,
    ADC_SH_WIDTH_61,
    ADC_SH_WIDTH_62,
    ADC_SH_WIDTH_63,
    ADC_SH_WIDTH_64
};
```

11.3.4.1 Description

These are the values used to specify the sample and hold width.

11.3.5 ADC_TriggerSelect

```
enum ADC_TriggerSelect
{
    ADC_TRIG_NONE,                  /* Software trigger only */
```

```
    ADC_TRIG_TIMER_0,
    ADC_TRIG_TIMER_1,
    ADC_TRIG_TIMER_2,
    ADC_TRIG_XINT,
    ADC_TRIG_EPWM1_SOCA,
    ADC_TRIG_EPWM1_SOCB,
    ADC_TRIG_EPWM2_SOCA,
    ADC_TRIG_EPWM2_SOCB,
    ADC_TRIG_EPWM3_SOCA,
    ADC_TRIG_EPWM3_SOCB,
    ADC_TRIG_EPWM4_SOCA,
    ADC_TRIG_EPWM4_SOCB,
    ADC_TRIG_EPWM5_SOCA,
    ADC_TRIG_EPWM5_SOCB,
    ADC_TRIG_EPWM6_SOCA,
    ADC_TRIG_EPWM6_SOCB,
    ADC_TRIG_EPWM7_SOCA,
    ADC_TRIG_EPWM7_SOCB,
    ADC_TRIG_ADCINT1    = 0x81,     /* special trigger from another EOC using
ADCINT1 */
    ADC_TRIG_ADCINT2    = 0x82      /* special trigger from another EOC using
ADCINT2 */
};
```

11.3.5.1 Description

11.3.6 ADC_Interrupt

```
enum ADC_Interrupt
{
    ADC_INT_1    = SYS_LIT( 0, INT_ID_ADCINT1),       /* Group 10 PIE */
    ADC_INT_2    = SYS_LIT( 1, INT_ID_ADCINT2),       /* Group 10 PIE */
    ADC_INT_3    = SYS_LIT( 2, INT_ID_ADCINT3),       /* Group 10 PIE */
    ADC_INT_4    = SYS_LIT( 3, INT_ID_ADCINT4),       /* Group 10 PIE */
    ADC_INT_5    = SYS_LIT( 4, INT_ID_ADCINT5),       /* Group 10 PIE */
    ADC_INT_6    = SYS_LIT( 5, INT_ID_ADCINT6),       /* Group 10 PIE */
    ADC_INT_7    = SYS_LIT( 6, INT_ID_ADCINT7),       /* Group 10 PIE */
    ADC_INT_8    = SYS_LIT( 7, INT_ID_ADCINT8),       /* Group 10 PIE */
    ADC_INT_9H   = SYS_LIT( 8, INT_ID_ADCINT9H),      /* Group 1 PIE */
    ADC_INT_1H   = SYS_LIT( 0, INT_ID_ADCINT1H),      /* Group 1 PIE */
    ADC_INT_2H   = SYS_LIT( 1, INT_ID_ADCINT2H)       /* Group 1 PIE */
};
```

11.3.6.1 Description

Each ADC Module can generate 1 of 9 interrupts.
When you configure the ADC module for a call back you need to specif which interrupt it will generate.

ADCINT9H/1H/2H are high prioritys.

NOTES

11.3.7 ADC_VrefMax

```
#define ADC_VrefMax (3.3)
```

11.3.7.1 Description

This is the ADC internal max reference voltage.

11.3.8 ADC_ValueMax

```
#define ADC_ValueMax (4095)
```

11.3.8.1 Description

This is the ADC max value returned from the ADC.

12 csl_pwm_t1_

12.1.1.1 Description

Contains functions for configuring the ePWM modules.

PWM_configClocks() or PWM_config() must be called before using any of the ePWM API functions. The ePWM hardware contains 6 ePWM modules which are accessed using the following pointers,

```
PWM_MOD_1
PWM_MOD_2
PWM_MOD_3
PWM_MOD_4
```

Each ePWM module has two duty counters which are referred to as channels.

```
PWM_CH_A
PWM_CH_B
```

To control an ePWM duty the ePWM module must be specified along with the channel to be set.

All timing is measured in ePWM ticks. The duration of an ePWM tick is dependent on the system clock speed. The conversion functions provided can be used to convert from frequency and ns to ePWM ticks.

This module is also capable of performing high resolution PWM using the micro edge positioner (MEP) logic on channel A. See PWM_setDutyHiRes() for more information.

12.1.1.2 Examples

This example configures ePWM module 1 with a 100kHz frequency and channel A of that module with a 25% duty cycle.

```
PWM_config( PWM_MOD_1,
                  PWM_freqToTicks(100000),
                  PWM_COUNT_UP );
PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_INVERT );
PWM_setDutyA( PWM_MOD_1,
              PWM_freqToTicks(100000)/4 );
```

12.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru791e/spru791e.pdf

12.2 Api

PWM_getIndex()
PWM_getPieId()
PWM_ackInt()
PWM_freqToTicks()
PWM_freqToTicksClocks()
PWM_configClocks()
PWM_config()
PWM_pin()
PWM_setDuty()
PWM_setTripZone()
PWM_setCallback()
PWM_setDeadBand()
PWM_setAdcSoc()
PWM_nsToTicks()
PWM_nsToTicksClocks()
PWM_isInt()
PWM_clrInt()
PWM_getPeriod()
PWM_setPeriod()
PWM_setDutyA()
PWM_setDutyHiRes()
PWM_setDutyB()
PWM_softwareSync()
PWM_setPhase()
PWM_setSyncOutSelect()
PWM_getDuty()
PWM_enableTpzInt()
PWM_clrTpzInt()
PWM_ackTpzInt()
PWM_getTzPieId()
PWM_setTripState()
PWM_getMod()
PWM_setDeadBandHalfBridge()
PWM_calibrateMep()
PWM_configBlanking()
PWM_setBlankingOffset()
PWM_setBlankingWindow()
PWM_getGpioPinA()
PWM_getGpioPinB()

12.2.1 PWM_getIndex

int PWM_getIndex(PWM_Module Mod)

where:
Mod -

12.2.1.1 Description

Returns the index value for each ePWM module, e.g.

```
PWM_MOD_1 = 0
PWM_MOD_2 = 1
     :
PWM_MOD_6 = 5
```

12.2.1.2 Examples

Returns 2 for ePWM module 3.

```
ui_PWMModule = PWM_getIndex( PWM_MOD_3 );
```

12.2.2 PWM_getPieId

INT_PieId PWM_getPieId(PWM_Module Mod)

where:
Mod - Selects the ePWM module.

12.2.2.1 Description

Returns the INT_PieId literal for each module, e.g.

```
PWM_MOD_1 = INT_ID_EPWM1
PWM_MOD_2 = INT_ID_EPWM2
     :
PWM_MOD_6 = INT_ID_EPWM6
```

This can be used when configuring interrupts using the functions INT_setCallback() and INT_enablePieId().

12.2.2.2 Examples

Returns INT_ID_EPWM3 for ePWM 3.

```
INT_ID_EPWMx = PWM_getPieId( PWM_MOD_3 );
```

12.2.3 PWM_ackInt

void PWM_ackInt(PWM_Module Mod)

where:
Mod - Selects the ePWM module.

12.2.3.1 Description

Used within an interrupt service routine to clear both the ePWM interrupt flag and the PIE group flag.

12.2.3.2 Examples

Clears the ePWM interrupt flag and the PIE group flag after the ePWM module 2 generates an interrupt and the PIE controller calls this ISR.

```
interrupt void isr_pwm2(void)
{
   PWM_ackInt( PWM_MOD_2 );
}
```

12.2.4 PWM_freqToTicks

uint16_t PWM_freqToTicks(uint32_t freq)

where:
freq - The frequency in Hz.

12.2.4.1 Description

Returns the number of ePWM ticks required to generate the frequency value (in Hertz) passed to the function.

The duration of one ePWM tick is calculated using the default divisor values, PWM_HSP_DIV_1 and PWM_DIV_1, which are used within the ePWM time base module.

This function may be used when the ePWM module is configured using PWM_config().

If the ePWM module is configured using PWM_configClocks() using non-default values for the divisors then PWM_freqToTicksClocks() should be used in-place of this function.

12.2.4.2 Examples

Returns the number of ePWM ticks required for 100kHz

```
  ui_100kHzInTicks = PWM_freqToTicks(100000);
```

For a device with a 100MHz system clock each ePWM tick would last for 10ns. Therefore the function would return a value of 1000 ePWM ticks. This would be the value that the ePWM module counter must count up to in order to generate a PWM output signal with a frequency of 100kHz using a 100MHz system clock.

Similarly, for a device with a 80MHz system clock each ePWM tick would last for 12.5ns. Therefore the function would return a value of 800 ePWM ticks.
Again this is the value that the ePWM counter must count up to in order to generate a PWM output signal with a frequency of 100kHz using an 80MHz system clock.

NOTES

12.2.5 PWM_freqToTicksClocks

uint16_t PWM_freqToTicksClocks(uint32_t freq,PWM_HspClkDiv HspClkDiv,PWM_ClkDiv ClkDiv)

where:
freq - The frequency in Hz.
HspClkDiv - The high speed divider.
ClkDiv - The low speed divider.

12.2.5.1 Description

Returns the number of ePWM ticks required to generate the frequency value (in Hertz) passed to the function.

The duration of one ePWM tick is calculated using the divisor values passed as parameters to the function. These values, HspClkDiv and ClkDiv are used within the ePWM time base module.

This function should be used when the ePWM module is configured with PWM_configClocks() using non-default values for the divisors (HspClkDiv and ClkDiv).

If the ePWM module is configured using PWM_config() then PWM_freqToTicks() may be used in-place of this function.

12.2.5.2 Examples

Returns the number of ePWM ticks required for 100kHz

```
ui_100kHzInTicks = PWM_freqToTicksClocks( 100000,
                                          PWM_HSP_DIV_2,
                                          PWM_DIV_1);
```

For a device with a 100MHz system clock each ePWM tick would last for 20ns using the ePWM time base module divisor settings of 2 and 1. Therefore this function would return a value of 500 ePWM ticks. This would be the value that the ePWM module counter must count up to in order to generate a PWM output signal with a frequency of 100kHz using a 100MHz system clock and the specified divisor settings.

Similarly, for a device with a 80MHz system clock each ePWM tick would last for 25ns using the ePWM time base module divisor settings of 2 and 1.
Therefore the function would return a value of 400 ePWM ticks. Again this is the value that the ePWM counter must count up to in order to generate a PWM output signal with a frequency of 100kHz using an 80MHz system clock and the specified divisor settings.

NOTES

12.2.6 PWM_configClocks

void PWM_configClocks(PWM_Module Module,uint16_t Ticks,PWM_HspClkDiv HspClkDiv,PWM_ClkDiv ClkDiv,PWM_CountMode CountMode)

where:
Module - Selects the ePWM module.
Ticks - Sets the period in ePWM ticks.
HspClkDiv - The high speed divisor value.
ClkDiv - The low speed divisor value.
CountMode - Sets the direction of the duty count.

12.2.6.1 Description

Configures and acquires an ePWM module and sets its frequency and count mode.

The period is set as a number of ePWM ticks. A value of 5 using a 10ns tick would give a period of 50ns.

The ePWM tick is dependent on the ePWM time base module which is configured by setting the high speed and low speed divisors values. In the above example a system clock of 100MHz with a high speed divisor of 1, PWM_HSP_DIV_1, and a low speed divisor of 1, PWM_DIV_1, would give a 10ns clock tick.

The 'HspClkDiv' and 'ClkDiv' arguments determine the ePWM time base as follows
 ePWMTick = 1 / (SYSCLKOUT / (HspClkDiv * ClkDiv))

Where SYSCLKOUT is the system clock frequency.

The function PWM_freqToTicks() should not be used if values other than PWM_HSP_DIV_1 and PWM_DIV_1 are used for HspClkDiv and ClkDiv. Instead use PWM_freqToTicksClocks() as per the example below.

The 'CountMode' argument sets the direction the duty register.

E.g. For a non inverting output,

```
 CountMode          Duty ____                      High Res acts on
 PWM_COUNT_UP       25%      |_____________        Falling edge
                                          ____
 PWM_COUNT_DOWN     25%  _____________|            Rising edge
                                    ____
 PWM_COUNT_UP_DOWN  25%  ______|       |______     Both edges
```

The high resolution mode can be used alongside the standard resolution functions. The effect of setting a high resolution duty will depend upon the 'CountMode' that the module is in as per the diagram above.

By default the ePWM module is not connected to an external pin. PWM_pin() must be called to acquire a pin and configure the multiplexer.

The duty cycle will need to be configured using the PWM_setDuty(), PWM_setDutyA(), or PWM_setDutyB() functions.

12.2.6.2 Examples

Sets the ePWM module 1 to a frequency of 100kHz

```
  PWM_configClocks( PWM_MOD_1,
```

```
PWM_freqToTicksClocks( 100000, PWM_HSP_DIV_1, PWM_DIV_1 ),
PWM_HSP_DIV_1,
PWM_DIV_1,
PWM_COUNT_UP );
```

12.2.6.3 Notes

When using PWM_COUNT_UP_DOWN the ePWM module first counts up and then counts down. Therefore the actual value stored in the period hardware register is divided by two. This means that any value set for the duty when using PWM_setDutyA/B() or PWM_setDuty() should be halved manually.

When using PWM_COUNT_UP_DOWN the HiRes only affects the phase of the signal and not the duty.

By default the synchronization input is enabled but the synchronization output is disabled. To enable and set the synchronization output call PWM_setSyncOutSelect().

The shadow registers are enabled and new values will be loaded when the counter value of the ePWM module equals zero.

Once an ePWM module has been acquired by this function it cannot be acquired again.

12.2.7 PWM_config

void PWM_config(PWM_Module Module,uint16_t Ticks,PWM_CountMode CountMode)

where:
Module - Selects the ePWM module.
Ticks - Sets the period in ePWM ticks.
CountMode - Sets the direction of the duty count.

12.2.7.1 Description

Configures and acquires an ePWM module and sets its frequency and count mode.

The period is set as a number of ePWM ticks. A value of 5 using a 10ns tick would give a period of 50ns.

The 'CountMode' argument sets the direction the duty register.

```
E.g. For a non inverting output,
 CountMode          Duty ____                         High Res acts on
 PWM_COUNT_UP       25%      |_____________           Falling edge
                                          ____
 PWM_COUNT_DOWN     25%  _____________|               Rising edge
                                   ____
 PWM_COUNT_UP_DOWN  25%  ______|      |______         Both edges
```

The high resolution mode can be used alongside the standard resolution functions. The effect of setting a high resolution duty will depend upon the 'CountMode' that the module is in as per the diagram above.

By default the ePWM module is not connected to an external pin. PWM_pin() must be called to acquire a pin and configure the multiplexer.

The duty cycle will need to be configured using the PWM_setDuty(), PWM_setDutyA(), or PWM_setDutyB() functions.

12.2.7.2 Examples

Sets the ePWM module 1 to a frequency of 100kHz

```
 PWM_config( PWM_MOD_1,
             PWM_freqToTicks(100000),
             PWM_COUNT_UP );
```

12.2.7.3 Notes

When using PWM_COUNT_UP_DOWN the ePWM module first counts up and then counts down. Therefore the actual value stored in the period hardware register is divided by two. This means that any value set for the duty when using PWM_setDutyA/B() or PWM_setDuty() should be halved manually.

By default the synchronization input is enabled but the synchronization output is disabled. To enable and set the synchronization output call PWM_setSyncOutSelect().

The shadow registers are enabled and new values will be loaded when the counter value of the ePWM module equals zero.

The HSPCLKDIV and CLKDIV parameters are set to 1. Therefore the system clock is not divided down before in order to generate the ePWM time base module clock.

Once an ePWM module has been acquired by this function it cannot be acquired again.

12.2.8 PWM_pin

void PWM_pin(PWM_Module Module,PWM_ModuleChannel Channel,GPIO_Level Invert)

where:
Module - Selects the ePWM module.
Channel - Selects the ePWM channel A or B.
Invert - Sets the pin polarity.

12.2.8.1 Description

Connects the ePWM channel to a pre-defined GPIO pin.

Each ePWM channel is hardwired to a GPIO pin. This function acquires the pin associated with the ePWM channel specified and sets the multiplexer for ePWM control. If the pin has already been acquired by another section of the code an assertion will be raised.

The output can be inverted by setting the 'Invert' argument.

12.2.8.2 Examples

This sets ePWM module 1 channel A GPIO pin to a non-inverting output.

```
PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_NON_INVERT );
```

12.2.8.3 Notes

ePWM	Channel A	Channel B
1	GPIO_0	GPIO_1
2	GPIO_2	GPIO_3
3	GPIO_4	GPIO_5
4	GPIO_6	GPIO_7
5	GPIO_8	GPIO_9
6	GPIO_10	GPIO_11

12.2.9 PWM_setDuty

void PWM_setDuty(PWM_Module Module,PWM_ModuleChannel Channel,uint16_t Ticks)

where:
Module - Selects the ePWM module.
Channel - Selects the ePWM channel A or B.
Ticks - Sets the duty in ePWM ticks.

12.2.9.1 Description

Sets the duty for ePWM module channel A or B in terms of the number of ePWM ticks.

This is the value that is continuously compared to the ePWM time base counter value. When the two values are equal an event is generated.

The duty value must be less than the period value, which is the maximum value of the time base counter, for this ePWM module. The duty value can be different for each channel, A or B, within the same ePWM module.

12.2.9.2 Examples

The ePWM output is high until the ePWM counter reaches 10.
Assuming that the period for this module is set to 100 this would give a 10% duty cycle on pin A in ePWM module 1.

```
PWM_setDuty( PWM_MOD_1, PWM_CH_A, 10 );
```

12.2.9.3 Notes

Where speed is required PWM_setDutyA() and PWM_setDutyB() should be used instead.
For PWM_COUNT_UP mode you can acheive a 0 to period (100%) duty cycle.
For PWM_COUNT_DOWN mode you can acheive a 0 to period-1 duty cycle.

12.2.10 PWM_setTripZone

void PWM_setTripZone(PWM_Module Module,uint16_t Mask,PWM_TpzMode Mode)

where:
Module - Selects the ePWM module.
Mask - This is an OR-ed mask of the possible trip zone pins.
Mode - Sets the action taken when a trip occurs.

12.2.10.1 Description

Enables the required trip zone pins for a selected PWM module.

The trip zone pins are passed to this function as a bit mask. The specific trip zone pins to enable for this ePWM module are logically OR-ed together from the possible GPIO pins (PWM_TZ1 to PWM_TZ3) to create the mask.

The function configures the selected IO as a trip pin. A low on this pin will enable the trip. Pull-ups are not enabled by default on trip pins. The state that is applied to the ePWM output pins following a trip is set using PWM_setTripState().

This function can be called twice for each ePWM module to configure different behaviors, either a one-shot trip (PWM_TPZ_ONE_SHOT) or cycle-by-cycle tripping (PWM_TPZ_CYCLE_BY_CYCLE), for different trip pins. This is set with the 'Mode' argument.

With cycle-by-cycle tripping the ePWM output pin is set to the condition specified by PWM_setTripState() for the current cycle only. This is commonly used for current limiting operation.

For one-shot trips the ePWM output pin is set to the condition specified by PWM_setTripState() until the trip is cleared using PWM_clrTpzInt() or PWM_ackTpzInt(). This is commonly used for short circuit and over-current protection.

12.2.10.2 Examples

Trips the ePWM module 2 output when either of the TZ1 or TZ2 GPIO pins are taken low. Upon this condition the PWM output is taken to the state indicated by PWM_setTripState(). In this example it is low:

```
PWM_setTripZone(  PWM_MOD_2, PWM_TZ1|PWM_TZ2, PWM_TPZ_CYCLE_BY_CYCLE );
PWM_setTripState( PWM_MOD_2, PWM_CH_A, GPIO_CLR );
PWM_setTripState( PWM_MOD_2, PWM_CH_B, GPIO_CLR );
```

NOTES

12.2.11 PWM_setCallback

void PWM_setCallback(PWM_Module Module,INT_IsrAddr Func,PWM_IntMode Mode,PWM_IntPrd Prd)

where:
Module - Selects the ePWM module.
Func - The pointer to the interrupt function.
Mode - When the interrupt function will be executed.
Prd - The number of periods before the interrupt is executed.

12.2.11.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the ePWM module.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the PWM_setCallback() function as the address of the ISR is used in the function call.

PIE controller interrupts are enabled automatically by this function for the specified ePWM module.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

12.2.11.2 Examples

In this example the function isr_pwm1() will be called each time the counter value of the ePWM module 1 is equal to zero.

```
interrupt void isr_pwm1( void )
{
  // The next line clears the ePWM and PIE group flags
  PWM_ackInt(PWM_MOD_1);

  // User code here
}

PWM_setCallback( PWM_MOD_1, isr_pwm1, PWM_INT_ZERO, PWM_INT_PRD_1 );
INT_enableGlobal( true );
```

12.2.11.3 Notes

If a NULL pointer is passed as the function pointer, then the interrupt will be enable for the module, but not within the PIE module.

12.2.12 PWM_setDeadBand

void PWM_setDeadBand(PWM_Module Module,uint16_t Ticks,GPIO_Level InvertA,GPIO_Level InvertB)

where:
Module - Selects the ePWM module.
Ticks - Sets the deadband gap in ePWM ticks.
InvertA - Sets the pin polarity.
InvertB - Sets the pin polarity.

12.2.12.1 Description

Enables the ePWM dead-band module to generate a pair of PWM signals which are related but have a specified dead-band between the two signals.

The signals from the original output of ePWM module channel A are used as an input to the dead-band module.

Therefore, using the original ePWM module channel A as the input, the outputs on channel A and channel B are related as follows,

```
             ____________
 Org A      |            |
 -----------              ------------
            .            .
            .            .
            .   _________.
 A          .  |         |      (chan A non-inverted)
 --------------           ------------
            .            .
            .            .
 ___________             .  __________
 B          |              |    (chan B inverted)
             --------------
```

The illustration above shows how the original ePWM channel A is the source for both the falling-edge and rising-edge delay.

While this sets up the dead-band values, it does not connect the ePWM modules outputs A and B to the GPIO pins.

Therefore the function PWM_pin() must be called to set up the output for both channels A and B.

The logic level set for channel B does not have any effect since the output for channel B is generated from channel A.

12.2.12.2 Examples

Produces the waveforms seen above.

```
  PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_NON_INVERT );
  // The last parameter for channel B is ignored when using the deadband
  PWM_pin( PWM_MOD_1, PWM_CH_B, GPIO_NON_INVERT );

// Set the dead-band module for ePWM module 1, invert channel B
  PWM_setDeadBand( PWM_MOD_1, 10, GPIO_NON_INVERT, GPIO_INVERT );
```

12.2.12.3 Notes

When using the dead-band module the PWM_pin() function for PWM_CH_B is only used to connect the GPIO pin. Therefore the invert level is ignored in the PWM_pin() function call. Furthermore PWM_setDutyB() does not have any effect when using dead-band.

NN To configure deadband for a half bridge with two N-type FETs.
```
  PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_NON_INVERT );
  PWM_pin( PWM_MOD_1, PWM_CH_B, GPIO_INVERT );
  PWM_setDeadBand( PWM_MOD_1, DeadCount, GPIO_NON_INVERT, GPIO_INVERT );
```

PP To configure deadband for a half bridge with two P-type FETs.
```
  PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_INVERT );
  PWM_pin( PWM_MOD_1, PWM_CH_B, GPIO_INVERT );
  PWM_setDeadBand( PWM_MOD_1, DeadCount, GPIO_INVERT, GPIO_NON_INVERT );
```

PN To configure deadband for a half bridge with a P-type FET on the high side and a N-type FET on the low side.
```
  PWM_pin( PWM_MOD_1, PWM_CH_A, GPIO_INVERT );
  PWM_pin( PWM_MOD_1, PWM_CH_B, GPIO_INVERT );
  PWM_setDeadBand( PWM_MOD_1, DeadCount, GPIO_INVERT, GPIO_INVERT );
```

Alternatively, the PWM_setDeadBandHalfBridge() function can be called with the required half bridge topology passed as an argument.

12.2.13 PWM_setAdcSoc

void PWM_setAdcSoc(PWM_Module Module,PWM_ModuleChannel Ch,PWM_IntMode Mode)

where:
Module - Selects the ePWM module.
Ch - Selects the ePWM channel A or B.
Mode - This is used to indicate when the ADC module is started.

12.2.13.1 Description

```
Enables the start of conversion (SOC) pulse allowing the ADC module to be
triggered by the ePWM module.

The SOC pulse will always be generated once enabled, even when the
ETFLG[SOCA] flag is already set. Therefore the flag does not need to be cleared.

The SOC pulse is generated from the ePWM module when the condition
specified by the 'Mode' argument is met. The condition compares the counter
value of the ePWM module to either zero, the period register or the duty
register of either channels A or B as specified in the 'Ch' argument.

The frequency of the ePWM module must allow sufficient time between SOC pulses
to allow the ADC module to sample the signal.

It takes 160ns*2.5 = 400ns for the first sample to be captured.

The 160ns value is the ADC clock which depends on the system clock and the
parameters set using,

  ADC_setClkDiv( ADC_CLK_DIV_4, ADC_CLK_PRE_DIV_1, ADC_SH_WIDTH_1 );

The 160ns figure is obtained when using a 100MHz system clock.
```

12.2.13.2 Examples

```
The ADC sequence will start when the ePWM period counter matches
channel A duty.

  PWM_setAdcSoc(PWM_MOD_1, PWM_CH_A, PWM_INT_CMPA_UP);

NOTES
```

12.2.14 PWM_nsToTicks

uint16_t PWM_nsToTicks(uint32_t Ns)

where:
Ns - Nano seconds.

12.2.14.1 Description

Returns the number of ePWM ticks required for the time value, in nanoseconds, passed to the function.

The duration of one ePWM tick is calculated using the default divisor values, PWM_HSP_DIV_1 and PWM_DIV_1, which are used within the ePWM time base module.

This function may be used when the ePWM module is configured using PWM_config().

If the ePWM module is configured using PWM_configClocks() using non-default values for the divisors then PWM_nsToTicksClocks() should be used in-place of this function.

12.2.14.2 Examples

Returns the number of ePWM ticks required for 100ns

```
ui_100nsInTicks = PWM_nsToTicks(100);
```

For a device with a 100MHz system clock each ePWM tick would last for 10ns. Therefore the function would return a value of 10 ePWM ticks. This would be the value that the ePWM module counter must count up to in order to generate a PWM output signal with a period of 100ns using a 100MHz system clock.

Similarly, for a device with a 80MHz system clock each ePWM tick would last for 12.5ns. Therefore the function would return a value of 8 ePWM ticks.
Again this is the value that the ePWM counter must count up to in order to generate a PWM output signal with a period of 100ns using an 80MHz system clock.

NOTES

12.2.15 PWM_nsToTicksClocks

uint16_t PWM_nsToTicksClocks(uint32_t Ns,PWM_HspClkDiv HspClkDiv,PWM_ClkDiv ClkDiv)

where:
Ns - Nano seconds.
HspClkDiv - The high speed divider.
ClkDiv - The low speed divider.

12.2.15.1 Description

Returns the number of ePWM ticks required for the time value, in nanoseconds, passed to the function.

The duration of one ePWM tick is calculated using the divisor values passed as parameters to the function. These values, HspClkDiv and ClkDiv are used within the PWM time base module.

This function should be used when the ePWM module is configured with PWM_configClocks() using non-default values for the divsors (HspClkDiv and ClkDiv).

If the ePWM module is configured using PWM_config() then PWM_nsToTicks() may be used in-place of this function.

12.2.15.2 Examples

Returns the number of ePWM ticks required for 100ns

```
ui_100nsInTicks = PWM_nsToTicksClocks( 100, PWM_HSP_DIV_2, PWM_DIV_1 );
```

For a device with a 100MHz system clock each ePWM tick would last for 20ns using the ePWM time base module divisor settings of 2 and 1. Therefore this function would return a value of 5 ePWM ticks. This would be the value that the ePWM module counter must count up to in order to generate a PWM output signal with a period of 100ns using a 100MHz system clock and the specified divisor settings.

Similarly, for a device with a 80MHz system clock each ePWM tick would last for 25ns using the ePWM time base module divisor settings of 2 and 1. Therefore the function would return a value of 4 ePWM ticks. Again this is the value that the ePWM counter must count up to in order to generate a PWM output signal with a period of 100ns using an 80MHz system clock and the specified divisor settings.

NOTES

12.2.16 PWM_isInt

int PWM_isInt(PWM_Module Module)

where:
Module -

12.2.16.1 Description

This returns non-zero if the ePWM interrupt flag is set. You can use this with PWM_clrInt() if you want to use the ePWM within the idle loop instead of using interrupts.

12.2.16.2 Examples

This creates a ePWM running at 100,000kHz and then waits in the idle loop untill the period reaches zero and then runs your user code from within the idle loop.

```
  PWM_config( PWM_MOD_5, PWM_freqToTicks( 100000 ),PWM_COUNT_UP );
 PWM_setCallback( PWM_MOD_5, 0, PWM_INT_ZERO, PWM_INT_PRD_1 );

  while ( 1 )
 {
 while ( PWM_isInt( PWM_MOD_5 )==0 );
  PWM_clrInt(PWM_MOD_5);
       // call your code every 100,000kHz
}
```

12.2.17 PWM_clrInt

void PWM_clrInt(PWM_Module Module)

where:
Module - Selects the ePWM module.

12.2.17.1 Description

Clears the ePWM interrupt flag only. Does not clear the PIE group flag.

If an ePWM interrupt occurs and the ePWM interrupt flag is set when the PIE group is not enabled, then the ePWM interrupt flag will remain set. Therefore the ePWM interrupt flag may need to be cleared before enabling the PIE group as any set flags will be serviced by the PIE controller when it is enabled.

After entering an interrupt service routine the ePWM interrupt flag and PIE group flag must be cleared. If only the ePWM interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

12.2.17.2 Examples

Clears the interrupt flag for ePWM module 1.

```
PWM_clrInt( PWM_MOD_1 );
```

12.2.17.3 Notes

PWM_ackInt() clears both the ePWM interrupt flag and PIE group flag.

12.2.18 PWM_getPeriod

uint16_t PWM_getPeriod(PWM_Module Module)

where:
Module - Selects the ePWM module.

12.2.18.1 Description

Returns the period of the ePWM module as the number of ePWM ticks.

The ePWM module counts from 0 to this value. The frequency of the ePWM module can be calculated using the time taken for one ePWM clock tick.

12.2.18.2 Examples

The number of clock ePWM ticks in one period is returned from the function for ePWM module 1.

```
ui_PWMPeriod = PWM_getPeriod(PWM_MOD_1);
```

12.2.19 PWM_setPeriod

void PWM_setPeriod(PWM_Module Module,uint16_t Ticks)

where:
Module - Selects the ePWM module.
Ticks - Ticks is calculated by the user based on the clock frequency and count mode.

12.2.19.1 Description

Sets the duty for ePWM module as a number of ePWM ticks.

The ePWM module counts from 0 to the period value. Thus the frequency of the ePWM module can be determined using the time taken for one ePWM tick. Use the function PWM_freqToTicks() to convert directly from a frequency value to a number of ePWM ticks.

PWM_freqToTicks() should only be used if the module has been configured using the default time base module divisor values which are automatically set using PWM_config(). If the ePWM module has been configured using PWM_configClock() using non-default divisor values then PWM_freqToTicksClocks() must be used to calculate the correct number of ePWM ticks.

12.2.19.2 Examples

A period of 1us (using 10ns ePWM ticks) is set by passing 100-1 = 99 ticks. The module will count from 0 to 99.

```
  PWM_setPeriod( PWM_MOD_1, 99 );
```

NOTES

12.2.20 PWM_setDutyA

void PWM_setDutyA(PWM_Module Module,uint16_t Ticks)

where:
Module - Selects the ePWM module.
Ticks - Duty in ePWM ticks.

12.2.20.1 Description

Sets the duty for ePWM module channel A as a number of ePWM ticks.

This is the value that is continuously compared to the ePWM time base counter value. When the two values are equal an event is generated.

The duty value must be less than the period value, which is the maximum value of the time base counter, for this ePWM module.

12.2.20.2 Examples

The PWM output is high until the ePWM counter reaches 10.
Assuming that the period for this module is set to 100 this would give a 10% duty cycle on pin A in ePWM module 1.

```
  PWM_setDutyA( PWM_MOD_1, 10 );
```

NOTES

12.2.21 PWM_setDutyHiRes

void PWM_setDutyHiRes(PWM_Module Module,uint32_t Ticks)

where:
Module - Selects the ePWM module.
Ticks - Duty in ePWM ticks.

12.2.21.1 Description

Sets the duty and high resolution value for channel A of the ePWM module specified.

The 'Ticks' argument to the function is a combination of both the standard duty and the high resolution value controlled by the micro edge positioner (MEP) logic of this module.

This 'Ticks' argument is a 32-bit number of which the lower 8 bits are unused. The 32-bit number is made up as follows,

```
  Bits 31 . . . 24 23 . . . 16 15 . . . 8 7 . . . 0
       |     standardDuty    |  | hiRes | |un-used|
```

The standard duty is the value that is continuously compared to the ePWM time base counter value. This can be any value up to the period of the ePWM module which has been set using the PWM_setPeriod() function. The high resolution value is appended to either the rising, falling or both rising and falling edges of the standard duty output depending on the mode of the ePWM module.

Based on the information given above, the Ticks argument to the function is calculated by first left-shifting the 16-bit period value of the ePWM module, which has been set using PWM_setPeriod(), by 16-bits. This new 32-bit value provides the range required to encompass both the standard duty and the high resolution component as per the diagram above. It will be referred to as the 32-bit period. The Ticks argument can be obtained by multiplying this 32-bit period by the required duty value as a fraction in the range of 0 to 1.

For example, if a duty of 40.5% is required and the period register has been set to 80 ticks (i.e. 800ns for a 10ns clock tick giving a frequency of 1.25MHz) then,

```
uiPeriod32 = uiPeriod16 * 65536   // Left-shift by 16-bits
uiPeriod32 = 80         * 65536
```

Get 40.5% (0.405) of uiPeriod32,

```
uiTicks32 = Duty  * uiPeriod32
uiTicks32 = 0.405 * uiPeriod32   // This would require a floating point
                                 // calculation and can be easily avoided.
uiTicks32 = (uint32_t) (0.405 * 1000.0) * (uiPeriod32 / 1000L)
                                 // The compiler would calculate
                                 // 0.405 * 1000.0 = 405 at compile time
                                 // and cast this to a 32-bit unsigned
                                 // integer. Or it can be done by the
                                 // user at the time of writing the code.
uiTicks32 = 405   * (5242880L / 1000L)
          = 405   * 5242
          = 2123010
```

To accurately set duty for the high resolution module, the MEP needs to have been calibrated by calling the function PWM_calibrateMep() before this function is called. The high resolution time base will vary slightly as time elapses. Therefore it is important to re-calibrate the MEP at regular intervals throughout your code.

EXMAPLES
Using the calculations above a duty of 40.5% is set for channel A of ePWM module 2 which is configured to have a period of 80 ticks giving a frequency of 1.25MHz.

```
  PWM_config( PWM_MOD_2,
              80,
              PWM_COUNT_UP );
  PWM_pin( PWM_MOD_2, PWM_CH_A, GPIO_NON_INVERT );
  PWM_calibrateMep();
  PWM_setDutyHiRes( PWM_MOD_2,
                    2123010 );  // Duty of 40.5% given an 80ns period
```

12.2.21.2 Notes

When using PWM_COUNT_DOWN you must invert the lower 16 bits of the duty. This is a feature of the chip.

```
   PWM_setDutyHiRes( PWM_MOD_2, duty ); // count up
   PWM_setDutyHiRes( PWM_MOD_2, duty^0x0000FFFF ); // count down
```

12.2.22 PWM_setDutyB

void PWM_setDutyB(PWM_Module Module,uint16_t Ticks)

where:
Module - Selects the ePWM module.
Ticks - Duty in ePWM ticks.

12.2.22.1 Description

Sets the duty for ePWM module channel B as a number of ePWM ticks.

This is the value that is continuously compared to the ePWM time base counter value. When the two values are equal an event is generated.

The duty value must be less than the period value, which is the maximum value of the time base counter, for this ePWM module.

12.2.22.2 Examples

The PWM output is high until the ePWM counter reaches 10.
Assuming that the period for this module is set to 100 this would give a 10% duty cycle on pin B in ePWM module 1.

```
  PWM_setDutyB( PWM_MOD_1, 10 );
```

NOTES

12.2.23 PWM_softwareSync

void PWM_softwareSync(PWM_Module Module)

where:
Module - Selects the ePWM module.

12.2.23.1 Description

The counter of the ePWM module specified is loaded with the phase offset value. This is a number of ePWM ticks set with PWM_setPhase().

If the phase value is set to zero, which it is by default, the counter value will be reset to zero. This allows the ePWM module to be synchronized to other modules or interrupts.

12.2.23.2 Examples

This resets the counter value for ePWM module 1 (the phase value is zero).

```
  PWM_softwareSync( PWM_MOD_1 );
```

NOTES

12.2.24 PWM_setPhase

void PWM_setPhase(PWM_Module Module,uint16_t Phase)

where:
Module - Selects the ePWM module.
Phase - Phase in ePWM ticks.

12.2.24.1 Description

Sets the phase offset value to be loaded in to the ePWM counter when a synchronization input pulse is received.

The value is loaded in to the counter on the next valid ePWM time base clock pulse.

The synchronization input pulse is generated from either a forced software synchronization performed by calling PWM_softwareSync() or from the synchronization output pulse generated by another ePWM module.

By default the synchronization input is enabled for each module but the synchronization output is disabled. To enable and set the synchronization output call PWM_setSyncOutSelect(). An external input or another ePWM module will be responsible for generating the synchronization pulse that will load this phase value in to the ePWM counter specified with this function.

12.2.24.2 Examples

A phase of 100ns is added to ePWM module 2 (if a 10ns ePWM tick is used).

For a C280x device the synchronization input for ePWM module 2 comes from ePWM module 1. Therefore the PWM_setSyncOutSelect() function will also need to be called to determine the event which will generate this synchronization pulse from ePWM module 1 if a forced software synchronization is not used.

```
PWM_setPhase( PWM_MOD_2, 100 );
```

NOTES

12.2.25 PWM_setSyncOutSelect

void PWM_setSyncOutSelect(PWM_Module Module,PWM_SyncOutSelect Mode)

where:
Module - Selects the ePWM module.
Mode - Select when the Sync-out is generated,

12.2.25.1 Description

Sets the event that generates the synchronization output pulse for the specified ePWM module.

A synchronization pulse can be generated when,

A sync-in pulse is detected,
The ePWM counter value is zero,
The ePWM counter value is equal to the period value,
Or it can be disabled.

The synchronization output is connected to the synchronization input of another ePWM module. See the appropriate reference manual for the device in use to determine which output is connected to which input. For the C280x Sync-out 1 (ePWM module 1) is connected to Sync-in 2 (ePWM module 2) and so on.

When the subsequent PWM module receives the synchronization input from the previous module the phase value set for the subsequent ePWM module is loaded in to the ePWM counter on the next valid ePWM time base clock pulse. This allows a multi-phase PWM system to be implemented. By default the synchronization input is enabled for all modules.

The phase value for an ePWM module can be set using PWM_setPhase(). However the modules will not be synchronized until PWM_setSyncOutSelect() has been configured.

Upon initial configuration each ePWM module has a value of zero phase offset set. In this case the ePWM counter value would be loaded with zero at the next valid ePWM time base clock pulse following the synchronization pulse.

12.2.25.2 Examples

Connects the synchronization output of ePWM module 1 to the synchronization input of ePWM module 2. A pulse will be generated when the ePWM counter value of module 1 is zero. There is no phase offset for ePWM module 2.

Effectively this ensures that ePWM modules 1 and 2 are precisely synchronized. The counter value of ePWM module 2 will be reset to zero one ePWM clock tick after the ePWM module 1 counter reaches zero.

```
PWM_setSyncOutSelect( PWM_MOD_1, PWM_SYNCOSEL_ZERO );
```

12.2.25.3 Notes

A synchronization pulse can also be generated manually using the function PWM_softwareSync().

12.2.26 PWM_getDuty

uint16_t PWM_getDuty(PWM_Module Mod,PWM_ModuleChannel Channel)

where:
Mod - Selects the ePWM module.
Channel - Selects the ePWM channel A or B.

12.2.26.1 Description

Returns the duty, in the number of ePWM ticks, for the ePWM module and channel.

This is the value that is continuously compared to the time base counter value.
When the two values are equal an event is generated.

The period of the time base counter can be obtained using the PWM_getPeriod() function. Thus the duty cycle can be calculated by dividing the duty value returned from this function by the period returned from PWM_getPeriod().

12.2.26.2 Examples

Returns the duty for ePWM module 1 channel A.

```
ui_Duty = PWM_getDuty( PWM_MOD_1, PWM_CH_A );
```

12.2.27 PWM_enableTpzInt

void PWM_enableTpzInt(PWM_Module Mod,PWM_TpzMode Mode,int Enable)

where:
Mod - Selects the ePWM module.
Mode - Selects what causes the interrupt to occur.
Enable - Enables the interrupt.

12.2.27.1 Description

Allows an interrupt to be generated when a trip occurs of a specific type for the selected ePWM module.

Trip zone pins can be configured for different modes, one-shot or cycle-by-cycle. The particular mode which will cause the interrupt must be specified.

An interrupt service routine must be set up following this function call for the trip zone pins associated with this ePWM module using the function INT_setCallback().

Finally the interrupts will need to be enabled within the PIE controller using INT_enablePieId().

12.2.27.2 Examples

This sets an interrupt service routine to be called when the trip zone pin 1 is taken low.

The ISR will be called by any ePWM modules that have trip zone interrupts enabled and their trip zone mode configured to one-shot with the trip zone pin set to trip pin 1.

Therefore within the ISR the trip zone flag and PIE group flag must be cleared for all of the ePWM modules that call the same ISR following a trip zone interrupt.

For example, if ePWM module 1 and module 2 both have trip zone interrupts enabled, with one-shot mode and the trip zone pin set to pin 1 then PWM_ackTpzInt() will need to be called twice within the ISR to clear the flag for ePWM module 1 and ePWM module 2 at the same time as clearing the PIE group flags.

```
// Configure the trip pins, trip pin 1, and trip mode, one-shot,
// for ePWM module 1
  PWM_setTripZone( PWM_MOD_1, PWM_TZ1, PWM_TPZ_ONE_SHOT );

// Set the state that the ePWM output pins will be forced to during a trip.
// Channel A and B will be forced low when a trip occurs on ePWM module 1
  PWM_setTripState( PWM_MOD_1, PWM_CH_A, GPIO_CLR );
  PWM_setTripState( PWM_MOD_1, PWM_CH_B, GPIO_CLR );

// Enable trip zone interrupts for ePWM module 1
  PWM_enableTpzInt( PWM_MOD_1, PWM_TPZ_ONE_SHOT, true );

  // Set up the PIE call back function
  INT_setCallback( INT_pieIdToVectorId( INT_ID_TZINT1 ), isr_tzint_epwm1 );

  // Enable the PIE flags
```

```
INT_enablePieId( INT_ID_TZINT1, true );
```

12.2.28 PWM_clrTpzInt

void PWM_clrTpzInt(PWM_Module Mod,PWM_TpzMode Mode)

where:
Mod - Selects the ePWM module.
Mode - Trip zone mode.

12.2.28.1 Description

Clears the trip zone flag for the specified ePWM module and the selected trip mode.

Each ePWM module can have a trip zone configured for either one-shot or cycle-by-cycle tripping modes. This is configured with the PWM_setTripZone() function.

With one-shot trip mode the outputs of the ePWM module will be re-enabled when the trip zone flag is cleared using this function provided that there are no current trips.

Cycle-by-cycle tripping will re-enable the output automatically when the ePWM module counter value is zero. However the cycle-by-cycle trip flag will remain set until it is manually cleared.

After entering an interrupt service routine the trip zone flag for the current mode and the PIE group flag must be cleared. If only the trip zone flag is cleared, as is the case with this function, then any subsequent interrupts will not be serviced by the PIE controller.

12.2.28.2 Examples

Clears the one shot flag for ePWM module 1. The ePWM module outputs, channel A and channel B, will be re-enabled if the active trip pins are currently high.

```
PWM_clrTpzInt( PWM_MOD_1, PWM_TPZ_ONE_SHOT );
```

12.2.28.3 Notes

PWM_ackTpzInt() clears both the trip zone flag and PIE group flag.

12.2.29 PWM_ackTpzInt

void PWM_ackTpzInt(PWM_Module Mod,PWM_TpzMode Mode)

where:
Mod - Selects the ePWM module.
Mode - Trip zone mode.

12.2.29.1 Description

Clears the trip zone flag for the specified ePWM module and the associated PIE group flag given the selected trip mode.

Each ePWM module can have a trip zone configured for either one-shot or cycle-by-cycle tripping modes. This is configured with the PWM_setTripZone() function.

With one-shot trip mode the outputs of the ePWM module will be re-enabled when the trip zone flag is cleared using this function provided that there are no current trips.

Cycle-by-cycle tripping will re-enable the output automatically when the ePWM module counter value is zero. However the cycle-by-cycle trip flag will remain set until it is manually cleared.

The function clears both the trip zone flag for the specified mode and the PIE group flag. This function should be called upon entering an interrupt service routine to clear both flags otherwise any subsequent interrupts will not be serviced by the PIE controller.

12.2.29.2 Examples

Clears the one shot flag for ePWM module 1. The ePWM module outputs, channel A and channel B will be re-enabled if the active trip pins are currently high. The PIE group flag will also be cleared.

```
PWM_ackTpzInt( PWM_MOD_1, PWM_TPZ_ONE_SHOT );
```

NOTES

12.2.30 PWM_getTzPieId

INT_PieId PWM_getTzPieId(PWM_TripZone Tz)

where:
Tz - Trip zone mode.

12.2.30.1 Description

Returns the INT_PieId literal for each trip zone value, e.g.

```
  PWM_TZ1 = INT_ID_TZINT1
  PWM_TZ2 = INT_ID_TZINT2
       :
  PWM_TZ5 = INT_ID_TZINT6
```

This can be used when configuring interrupts using the functions INT_setCallback() and INT_enablePieId().

12.2.30.2 Examples

Returns INT_ID_TZINT1 for trip zone 1.

```
  INT_ID_TZINTx = PWM_getTzPieId(PWM_TZ1);
```

NOTES

12.2.31 PWM_setTripState

void PWM_setTripState(PWM_Module Module,PWM_ModuleChannel Channel,GPIO_TriState TripState)

where:
Module - Selects the ePWM module.
Channel - Selects the ePWM channel A or B.
TripState - Sets the state of the pin when the trip occurs.

12.2.31.1 Description

Sets the state of the ePWM output channel to be applied when a trip occurs.

Allows the ePWM output channel to be set as either high, low, high-impedance or for no action to be taken when a trip occurs.

The function will need to be called for both channels of the same ePWM module separately if required.

The pins which cause a trip event for this particular ePWM module must be set first by calling the function PWM_setTripZone().

12.2.31.2 Examples

When a trip event occurs the ePWM module 2 outputs are taken high for both channel A and channel B.

```
PWM_setTripState( PWM_MOD_2, PWM_CH_A, GPIO_SET );
PWM_setTripState( PWM_MOD_2, PWM_CH_B, GPIO_SET );
```

NOTES

12.2.32 PWM_getMod

PWM_Module PWM_getMod(int Index)

where:
Index - ePWM module index.

12.2.32.1 Description

Returns the ePWM module address for the specified index.

The indexes start from 0, therefore,

```
  0 = PWM_MOD_1
  1 = PWM_MOD_2
       :
  5 = PWM_MOD_6
```

12.2.32.2 Examples

This returns PWM_MOD_2.

```
  PWM_MOD_X = PWM_getMod(1);
```

12.2.33 PWM_setDeadBandHalfBridge

void PWM_setDeadBandHalfBridge(PWM_Module Module,uint16_t Ticks,PWM_Half_Bridge HalfBridge)

where:
Module - Selects the ePWM module.
Ticks - Sets the deadband gap in ePWM ticks.
HalfBridge -

12.2.33.1 Description

Enables the ePWM dead-band module to generate a pair of PWM signals for a specific half bridge topology (NN, PP, PN).

The two signals are related but have a specified dead-band between the signals and use the original ePWM module channel A as an input.

This functions differs from PWM_setDeadBand() as it also connects the ePWM modules outputs, channel A and channel B, to GPIO pins associated with this ePWM module. The correct inverting or non-inverting output is configured on each channel as required for the half bridge topology specified.

Therefore the PWM_pin() function must not be called. Otherwise an assertion will be raised.

Using the original ePWM module channel A as the input, the outputs on channel A and channel B are related as follows,

```
             ____________
 Org A      |            |
 -----------              ------------
            .            .
            .            .
            .   __________.
 A          .  |          |      (chan A non-inverted)
 --------------            ------------
            .            .
            .            .
 ___________             .  __________
 B          |              |    (chan B inverted)
             --------------
```

The illustration above shows how the original ePWM channel A is the source for both the falling-edge and rising-edge delay.

While this sets up the dead-band values, it does not connect the ePWM modules outputs A and B to the GPIO pins.

Therefore the function PWM_pin() must be called to set up the output for both channels A and B.

The logic level set for channel B does not have any effect since the output for channel B is generated from channel A.

12.2.33.2 Examples

Produces the waveforms seen above.

```
  PWM_setDeadBandHalfBridge( PWM_MOD_2, 10, HALF_BRIDGE_PP );
```

12.2.33.3 Notes

PWM_setDutyB() does not have any effect when using dead-band.

12.2.34 PWM_calibrateMep

uint16_t PWM_calibrateMep(void)

where:

12.2.34.1 Description

```
Calibrates the MEP scale factor to account for variables in the system
parameters.

This function should be called in the background to ensure that the MEP scale
factor value is up to date. A convenient place would be within any idle
loops.
```

12.2.35 PWM_configBlanking

void PWM_configBlanking(PWM_Module Mod,PWM_CmpSelect Select,GPIO_Level Level,bool Async)

where:
Mod - Selects the ePWM module.
Select - Selects the input for the digital compare unit.
Level - Inverts the PWM_CmpSelect input to the blanking block.
Async -

12.2.35.1 Description

This allows you to add a blanking window to the comparator outputs trip zones (TZ1, TZ2, TZ3) in the digital compare sub-module of your selected PWM module. The output of the digital compare unit is PWM_DCEVT.
During the blanking window the specified trips pins are disabled. This is used to stop noise on these pins from false tripping the system.

12.2.35.2 Examples

This sets up a blanking window for trip zone 1 and then configures the trip zone to use PWM_DCEVT which causes the ePWM output to zero when the trip zone occurs outside the blanking area.

```
  PWM_configBlanking(PWM_MOD_1, PWM_CMP_TPZ1, GPIO_INVERT, false );
  PWM_setBlankingOffset(PWM_MOD_1,50);
  PWM_setBlankingWindow(PWM_MOD_1,100);

PWM_setTripZone(  PWM_MOD_1, PWM_DCEVT, PWM_TPZ_CYCLE_BY_CYCLE );
  PWM_setTripState( PWM_MOD_1, PWM_CH_A, GPIO_CLR );
```

12.2.35.3 Notes

For faster response you should set Async to true.

12.2.36 PWM_setBlankingOffset

void PWM_setBlankingOffset(PWM_Module Mod,uint16_t Value)

where:
Mod - Selects the ePWM module.
Value - Sets the blanking offset in ePWM ticks.

12.2.36.1 Description

This sets the offset to the start of the blanking window period. During this period the trip zone selected by PWM_configBlanking() is ignored.

The offset can cause the blanking window to overlap in to the next period.

12.2.36.2 Examples

This sets the blanking window offset to 50 ePWM ticks.

```
  PWM_setBlankingOffset( PWM_MOD_1, 50 );
```

12.2.37 PWM_setBlankingWindow

void PWM_setBlankingWindow(PWM_Module Mod,uint8_t Value)

where:
Mod - Selects the ePWM module.
Value - Sets the blanking window in ePWM ticks.

12.2.37.1 Description

This sets the blanking window period. During this period the trip zone selected by PWM_configBlanking() is ignored.

The offset and window can cause the blanking window to overlap in to the next period.

12.2.37.2 Examples

This sets the blanking window to 50 ePWM ticks.

```
  PWM_setBlankingWindow( PWM_MOD_1, 50 );
```

12.2.38 PWM_getGpioPinA

void PWM_getGpioPinA(PWM_Module Mod)

where:
Mod -

12.2.38.1 Description

Returns the GPIO pin associated with the ePWM module channel A.

```
GPIO_2 = PWM_getGpioPinA( PWM_MOD_2 )
```

12.2.39 PWM_getGpioPinB

void PWM_getGpioPinB(PWM_Module Mod)

where:
Mod -

12.2.39.1 Description

Returns the GPIO pin associated with the ePWM module channel B.

```
GPIO_3 = PWM_getGpioPinA( PWM_MOD_2 )
```

12.3 Types

12.3.1 PWM_TYPE_1

```
#define PWM_TYPE_1
```

12.3.1.1 Description

This can be use to determine the peripheral type used by the csl.

12.3.2 PWM_MODULE_X

```
#if 1
#define PWM_MOD_1 (&EPwm1Regs)
#define PWM_MOD_2 (&EPwm2Regs)
#define PWM_MOD_3 (&EPwm3Regs)
#define PWM_MOD_4 (&EPwm4Regs)
#define PWM_MOD_5 (&EPwm5Regs)
#define PWM_MOD_6 (&EPwm6Regs)
#define PWM_MOD_7 (&EPwm7Regs)
#endif
```

12.3.2.1 Description

These values are used to specify the ePWM module.

12.3.3 PWM_Module

```
typedef volatile struct EPWM_REGS* PWM_Module;
```

12.3.3.1 Description

This is used to map hardware register values to PWM_Module.

12.3.4 PWM_ModuleChannel

```
enum PWM_ModuleChannel
{
    PWM_CH_A,
    PWM_CH_B
};
```

12.3.4.1 Description

This is used to select channel A or B for each ePWM module.

12.3.5 PWM_CountMode

```
enum PWM_CountMode
{
```

```
    PWM_COUNT_UP,            /* counts up to period */
    PWM_COUNT_DOWN,          /* counts down from period */
    PWM_COUNT_UP_DOWN        /* counts up to period and then down */
};
```

12.3.5.1 Description

This is used to control the direction of the duty count.

12.3.6 PWM_TripZone

```
enum PWM_TripZone
{
    PWM_TZ1      = SYS_LIT(0, (1<<0)),
    PWM_TZ2      = SYS_LIT(1, (1<<1)),
    PWM_TZ3      = SYS_LIT(2, (1<<2)),
    PWM_TZ4      = SYS_LIT(3, (1<<3)),
    PWM_TZ5      = SYS_LIT(4, (1<<4)),
    PWM_TZ6      = SYS_LIT(5, (1<<5)),
    PWM_DCEVT    = SYS_LIT(6, (1<<6))     /* output from the digital compare unit
*/
};
```

12.3.6.1 Description

These are used for setting the trip zone values.

12.3.7 PWM_IntMode

```
enum PWM_IntMode
{
    PWM_INT_ZERO         = 1,     /* interrupt when counter is equal to zero */
    PWM_INT_PERIOD       = 2,     /* interrupt when counter is equal to period
register */
    PWM_INT_CMPA_UP      = 4,     /* interrupt when counter is equal to counter
compare register A on up-count */
    PWM_INT_CMPA_DOWN    = 5,     /* interrupt when counter is equal to counter
compare register A on down-count */
    PWM_INT_CMPB_UP      = 6,     /* interrupt when counter is equal to counter
compare register B on up-count */
    PWM_INT_CMPB_DOWN    = 7      /* interrupt when counter is equal to counter
compare register B on down-count */
};
```

12.3.7.1 Description

This is used to indicate when the ePWM interrupt occurs.

Only the following interrupt modes are valid when operating in PWM_COUNT_UP mode.

- PWM_INT_ZERO
- PWM_INT_PERIOD
- PWM_INT_CMPA_UP

```
  PWM_INT_CMPB_UP
```

Only the following interrupt modes are valid when operating in PWM_COUNT_DOWN mode.

```
  PWM_INT_ZERO
  PWM_INT_PERIOD
  PWM_INT_CMPA_DOWN
  PWM_INT_CMPB_DOWN
```

Any of the interrupt modes are valid when operating in PWM_COUNT_UP_DOWN mode.

12.3.8 PWM_TpzMode

```
enum PWM_TpzMode
{
    PWM_TPZ_CYCLE_BY_CYCLE  = (1<<1),   /* limits output for one cycle */
    PWM_TPZ_ONE_SHOT        = (1<<2)    /* limits output forever */
};
```

12.3.8.1 Description

This defines the different trip zone modes.

12.3.9 PWM_SyncOutSelect

```
enum PWM_SyncOutSelect
{
    PWM_SYNCOSEL_IN         = 0,    /* connect sync in to sync out */
    PWM_SYNCOSEL_ZERO       = 1,    /* connect period==zero to sync out */
    PWM_SYNCOSEL_CMPB       = 2,    /* connect period=dutyB to sync out */
    PWM_SYNCOSEL_DISBALE    = 3     /* disable sync out */
};
```

12.3.9.1 Description

This is used to allow one ePWM module to sync to another.

12.3.10 PWM_HspClkDiv

```
enum PWM_HspClkDiv
{
    PWM_HSP_DIV_1    = SYS_LIT( 1,    0 ),
    PWM_HSP_DIV_2    = SYS_LIT( 2,    1 ),
    PWM_HSP_DIV_4    = SYS_LIT( 4,    2 ),
    PWM_HSP_DIV_6    = SYS_LIT( 6,    3 ),
    PWM_HSP_DIV_8    = SYS_LIT( 8,    4 ),
    PWM_HSP_DIV_10   = SYS_LIT( 10,   5 ),
    PWM_HSP_DIV_12   = SYS_LIT( 12,   6 ),
    PWM_HSP_DIV_14   = SYS_LIT( 14,   7 )
};
```

12.3.10.1 Description

This is used to set the high speed ePWM divider.

12.3.11 PWM_ClkDiv

```
enum PWM_ClkDiv
{
    PWM_DIV_1        = SYS_LIT( 1,    0 ),
    PWM_DIV_2        = SYS_LIT( 2,    1 ),
    PWM_DIV_4        = SYS_LIT( 4,    2 ),
    PWM_DIV_8        = SYS_LIT( 8,    3 ),
    PWM_DIV_16       = SYS_LIT( 16,   4 ),
    PWM_DIV_32       = SYS_LIT( 32,   5 ),
    PWM_DIV_64       = SYS_LIT( 64,   6 ),
    PWM_DIV_128      = SYS_LIT( 128, 7 )
};
```

12.3.11.1 Description

This is used to set the ePWM divider.

12.3.12 PWM_IntPrd

```
enum PWM_IntPrd
{
    PWM_INT_PRD_1    = SYS_LIT( 1, 1 ),   /* cause interrupt on every occurance */
    PWM_INT_PRD_2    = SYS_LIT( 2, 2 ),   /* cause interrupt on every 2nd
occurance */
    PWM_INT_PRD_3    = SYS_LIT( 3, 3 )    /* cause interrupt on every 3rd
occurance */
};
```

12.3.12.1 Description

This is used to set when the ePWM interrupt will occur.

12.3.13 PWM_Half_Bridge

```
enum PWM_Half_Bridge
{
    HALF_BRIDGE_NN,
    HALF_BRIDGE_PP,
    HALF_BRIDGE_PN
};
```

12.3.13.1 Description

12.3.14 PWM_CmpSelect

```
enum PWM_CmpSelect
{
    PWM_CMP_TPZ1     = 0,
    PWM_CMP_TPZ2     = 1,
```

```
    PWM_CMP_TPZ3     = 2,
    PWM_CMP_COMP1    = 8,
    PWM_CMP_COMP2    = 9,
    PWM_CMP_COMP3    = 10    /* piccolo B only */
};
```

12.3.14.1 Description

13 csl_cmp_t0_

13.1.1.1 Description

Provides the functions necessary for configuring the comparator modules.

CMP_config() must be called before using any of the comparator API functions. The comparator hardware contains 2 comparator modules which are accessed using the following pointers,

```
CMP_MOD_1
CMP_MOD_2
```

The output of the comparator can be fed in to the ePWM module as PWM_CMP_COMP1 or PWM_CMP_COMP2.

13.2 Api

CMP_config()
CMP_pin()
CMP_getIndex()
CMP_getMod()
CMP_mVtoDacValue()
CMP_setDac()
CMP_getGpioPin()

13.2.1 CMP_config

void CMP_config(CMP_Module Mod,CMP_Sample Sample,GPIO_Level Level,CMP_Source Source)

where:
Mod -
Sample -
Level -
Source -

13.2.1.1 Description

Configure the analog comparator that is built in to the Piccolo.

Both inputs can be connected to external analog pins. Alternatively, the inverting input can be connected to the built-in DAC.

There is also an option to specify the number of continuous samples before the comparator output changes state.

The comparator output can be inverted and attached to a digital GPIO pin using CMP_pin().

13.2.1.2 Examples

Sets up comparator 2 to wait for 1 valid sample. A non-inverted output is generated on the GPIO pin associated with CMP_MOD_2. The DAC is used for the inverting input.

```
CMP_config( CMP_MOD_2, CMP_SAMPLE_1, GPIO_NON_INVERT, CMP_DAC );
CMP_pin( CMP_MOD_2 );
CMP_setDac( CMP_MOD_2, CMP_mVtoDacValue( 1500 ) ); //1.5v
```

13.2.1.3 Notes

The output is fed in to the ePWM as PWM_CMP_COMP1 and PWM_CMP_COMP2. The comparator output can also be used by the ePWM module as a trip zone input.

13.2.2 CMP_pin

void CMP_pin(CMP_Module Mod)

where:
Mod -

13.2.2.1 Description

Connects the output of the comparator to a digital GPIO pin.

13.2.2.2 Examples

This connects the output from comparator 1 to GPIO_1.

```
  CMP_pin( CMP_MOD_1);
```

13.2.3 CMP_getIndex

int CMP_getIndex(CMP_Module Mod)

where:
Mod -

13.2.3.1 Description

Returns the index value for each comparator module, e.g.

```
CMP_MOD_1 = 0
CMP_MOD_2 = 1
```

13.2.3.2 Examples

Returns 1 for Comparator module 2.

```
1 == CMP_getIndex( CMP_MOD_2 );
```

13.2.4 CMP_getMod

CMP_Module CMP_getMod(int Index)

where:
Index -

13.2.4.1 Description

Returns the Comparator module address for the specified index.

The indexes start from 0, therefore,

```
0 = CMP_MOD_1
1 = CMP_MOD_2
```

13.2.4.2 Examples

This returns CMP_MOD_2.

```
CMP_MOD_2 = CMP_getMod(1);
```

13.2.5 CMP_mVtoDacValue

uint16_t CMP_mVtoDacValue(uint16_t mVolts)

where:
mVolts -

13.2.5.1 Description

Returns a value for the DAC that is the equivalent of the analog voltage passed to the function in milli-volts.

13.2.5.2 Examples

Returns the DAC value required to set the inverting input to 1.5 volts.

```
465 == CMP_mVtoDacValue( 1500 );
```

13.2.6 CMP_setDac

void CMP_setDac(CMP_Module Mod,uint16_t Value)

where:
Mod -
Value -

13.2.6.1 Description

Sets the DAC of the analog comparator module to the value passed to this function.

Use the CMP_mVtoDacValue() function within this function call to set the DAC to an equivalent voltage value.

13.2.6.2 Examples

Sets the voltage on the inverting input of the comparator to 1.5V using the DAC.

```
CMP_setDac( CMP_MOD_2, CMP_mVtoDacValue( 1500 ) ); //1.5v
```

13.2.7 CMP_getGpioPin

void CMP_getGpioPin(PWM_Module Mod)

where:
Mod -

13.2.7.1 Description

Returns the GPIO pin associated with the CMP module channel A.

```
  GPIO_3 = CMP_getGpioPin( CMP_MOD_2 )
```

13.3 Types

13.3.1 CMP_TYPE_0

```
#define CMP_TYPE_0
```

13.3.1.1 Description

This can be use to determine the peripheral type used by the CSL.

13.3.2 CMP_MODULE_X

```
#if 1
#define CMP_MOD_1 (&Comp1Regs)
#define CMP_MOD_2 (&Comp2Regs)
#define CMP_MOD_3 (&Comp3Regs)
#endif
```

13.3.2.1 Description

These values are used to specify the Comparator module.

13.3.3 CMP_Module

```
typedef volatile struct COMP_REGS* CMP_Module;
```

13.3.3.1 Description

This is used to map hardware register values to CMP_Module.

13.3.4 CMP_Sample

```
enum CMP_Sample
{
    CMP_SAMPLE_1     = 0,
    CMP_SAMPLE_2,
    CMP_SAMPLE_3,
    CMP_SAMPLE_4,
    CMP_SAMPLE_5,
    CMP_SAMPLE_6,
    CMP_SAMPLE_7,
    CMP_SAMPLE_8,
    CMP_SAMPLE_9,
    CMP_SAMPLE_10,
    CMP_SAMPLE_11,
    CMP_SAMPLE_12,
    CMP_SAMPLE_13,
    CMP_SAMPLE_14,
    CMP_SAMPLE_15,
    CMP_SAMPLE_16,
    CMP_ASYNC        = 0xFF
};
```

13.3.4.1 Description

13.3.5 CMP_Source

```
enum CMP_Source
{
    CMP_DAC      = 0,     /* Comparator inverting input sourced from internal DAC
*/
    CMP_GPIO              /* Comparator inverting input sourced from input pin */
};
```

13.3.5.1 Description

13.3.6 CMP_ValueMax

```
#define CMP_ValueMax (1023)
```

13.3.6.1 Description

14 csl_uart_t0_

14.1.1.1 Description

Contains functions for configuring the UART module of the DSP.

The UART_config() function must be called before using any of the API functions.

By default the module will always use the Rx and Tx FIFOs.

14.1.1.2 Examples

Configures and opens a serial port. The code then enters a loop which will echo back any received characters.

```
UART_config( UART_MOD_1, GPIO_28, GPIO_29,
             UART_baudToTicks(115200),
             UART_DATA_8, UART_PARITY_NONE, UART_STOP_2 );
UART_setLoopback( true );

for(;;)
{
   // Wait for incoming character to be received
   while(UART_getRxCount(UART_MOD_1)==0 );
   UART_putc(UART_MOD_1, UART_getc(UART_MOD_1) );
}
```

14.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru051b/spru051b.pdf

14.2 Api

UART_config()
UART_flush()
UART_flushRx()
UART_flushTx()
UART_putc()
UART_puts()
UART_getRxCount()
UART_getc()
UART_isRxOverFlow()
UART_clrRxOverFlow()
UART_baudToTicks()
UART_setRxCallback()
UART_ackRxInt()
UART_getIndex()
UART_getRxPieId()
UART_setTxCallback()
UART_enableRxInt()
UART_enableTxInt()
UART_getTxPieId()
UART_ackTxInt()
UART_setTicks()
UART_setLoopback()
UART_clrRxInt()
UART_clrTxInt()

14.2.1 UART_config

void UART_config(UART_Module Mod,GPIO_Pin Rx,GPIO_Pin Tx,uint16_t Ticks,UART_DataBits DataBits,UART_Parity Parity,UART_StopBits StopBits)

where:
Mod - Selects the UART module.
Rx - Selects the GPIO pin to use for the RX pin.
Tx - Selects the GPIO pin to use for the TX pin.
Ticks - Baud rate in UART ticks.
DataBits - Number of data bits.
Parity - Selects the type of parity used.
StopBits - Selects the number of stop bits.

14.2.1.1 Description

Configures the UART module with the required baud rate, the number of stop bits, the parity used and selects the GPIO pins used for the port.

The UART module is generated using the serial communications interface (SCI) within the DSP. The SCI supports return-to-zero communications such as that used by the UART.

There are two SCI modules, A and B, which can be configured as UART_MOD_1 and UART_MOD_2. There are several possible combinations of transmit and receive pins for each UART module.

Only the following GPIO pin combinations are valid for each UART module,

	Rx	Tx
UART_MOD_1	GPIO_28(SCI-A)	GPIO_29(SCI-A)
UART_MOD_2	GPIO_11(SCI-B)	GPIO_9 (SCI-B)
UART_MOD_2	GPIO_15(SCI-B)	GPIO_14(SCI-B)
UART_MOD_2	GPIO_19(SCI-B)	GPIO_18(SCI-B)
UART_MOD_2	GPIO_23(SCI-B)	GPIO_22(SCI-B)

Please refer to the datasheet of the specific DSP in order to confirm these combinations.

The baud rate is set as a number of UART clock ticks which are derived from the low-speed peripheral clock. The UART_baudToTicks() function can be used to directly convert from baud to clock ticks.

14.2.1.2 Examples

Configures UART module 1 for 9600 baud, 8 data bits, no parity, 1 stop bit and Rx/Tx pins to GPIO_28/29.

```
UART_config( UART_MOD_1,
    GPIO_28,
    GPIO_29,
    UART_baudToTicks(9600),
    UART_DATA_8,
    UART_PARITY_NONE,
    UART_STOP_1 );
```

14.2.2 UART_flush

void UART_flush(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.2.1 Description

Flushes the UART Rx and Tx FIFOs.

The FIFOs are emptied one character at a time. The FIFOs will continue to be flushed until they are empty.

14.2.2.2 Examples

Flushes the FIFOs for UART module 1.

```
UART_flush( UART_MOD_1 );
```

14.2.3 UART_flushRx

void UART_flushRx(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.3.1 Description

Flushes the UART Rx FIFO.

The FIFO is emptied one character at a time. The FIFO will continue being flushed until it is empty.

14.2.3.2 Examples

Flushes the RX FIFO for UART 1.

```
UART_flushRx( UART_MOD_1 );
```

14.2.4 UART_flushTx

void UART_flushTx(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.4.1 Description

Flushes the UART Tx FIFO.

The FIFO is emptied one character at a time. The FIFO will continue being flushed until it is empty.

14.2.4.2 Examples

Flushes the TX FIFO for UART 1.

```
UART_flushTx( UART_MOD_1 );
```

14.2.5 UART_putc

void UART_putc(UART_Module Mod,int a)

where:
Mod - Selects the UART module.
a - Symbol to write.

14.2.5.1 Description

Writes a character to the Tx FIFO.

If the FIFO is full the function will suspend until there is room in the FIFO buffer.

14.2.5.2 Examples

Writes the character 'A' to the serial port.

```
  UART_putc( UART_MOD_1, 'A' );
```

NOTE

14.2.6 UART_puts

void UART_puts(UART_Module Mod,const char* str)

where:
Mod - Selects the UART module.
str - Null terminated string.

14.2.6.1 Description

Writes a string to the Tx FIFO.

If the FIFO is full the function will suspend until there is room in the FIFO buffer.

A new line character, '\n', is automatically appended to the end of the string.

14.2.6.2 Examples

This writes the string "hello" to the serial port.

```
UART_putc( UART_MOD_1, "hello" );
```

NOTES

14.2.7 UART_getRxCount

int UART_getRxCount(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.7.1 Description

Returns the number of characters in the Rx FIFO.

14.2.7.2 Examples

Returns the number of received characters in the UART module 1 Rx FIFO.

```
int count = UART_getRxCount( UART_MOD_1 );
```

14.2.8 UART_getc

char UART_getc(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.8.1 Description

Returns a character from the Rx FIFO.

This function will suspend until there is a character in the Rx FIFO. It will then remove the character from the FIFO and return its value.

This suspend may have complications if the function is being called from an interrupt service routine. The function will hang inside the ISR until a character is received on the buffer. Therefore no other interrupts would be serviced. Consider using UART_getRxCount() to check if there is data on the buffer before calling the function from the ISR.

The function also services the watchdog periodically as it may be suspending for a considerable period of time.

14.2.8.2 Examples

Removes the character from UART module 1 Rx FIFO.

```
char ch = UART_getc( UART_MOD_1 );
```

14.2.9 UART_isRxOverFlow

int UART_isRxOverFlow(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.9.1 Description

Returns a non-zero value if the Rx FIFO has overflowed.

14.2.9.2 Examples

Returns a value to indicate if the RX FIFO has overflowed.

```
int overflow = UART_isRxOverFlow( UART_MOD_1 );
```

14.2.10 UART_clrRxOverFlow

void UART_clrRxOverFlow(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.10.1 Description

Clears the Rx FIFO overflow flag.

14.2.10.2 Examples

Clears the UART module 1 overflow flag.

```
UART_clrRxOverFlow( UART_MOD_1 );
```

14.2.11 UART_baudToTicks

uint16_t UART_baudToTicks(int baud)

where:
baud - Selects the baud rate.

14.2.11.1 Description

Converts the required baud rate to the number of UART clock ticks using the following formula.

```
                     SYS_CLK_HZ
Ticks =  ------------------------------  - 1
           USR_PER_LSP_DIV * baud x 8
```

14.2.11.2 Examples

Returns the UART ticks required for a baud rate of 115200.

```
  uint16_t BaudTicks = UART_baudToTicks( 115200 );
```

14.2.11.3 Notes

When USR_PER_LSP_DIV and baud are large the result Ticks can be small, which due to rounding errors can give the incorrect value

```
 SYS_setPerhiperalClk(SYS_PER_CLK_DIV_2, SYS_PER_CLK_DIV_14);
 6 = UART_baudToTicks( 115200 ); instead of 6.750
```

For a 100Mhz system clock the value of 6 fails to work, while a value of 7 works correctly.
The code has therefore been changed to add 0.5 to the result before converting to an integer value.

14.2.12 UART_setRxCallback

void UART_setRxCallback(UART_Module Mod,INT_IsrAddr Func,int RxLevel)

where:
Mod - Selects the UART module.
Func - The pointer to the interrupt function.
RxLevel - The number of characters in the Rx FIFO before the interrupt is called.

14.2.12.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the Rx interrupt. The interrupt flag will be raised when the number of characters in the Rx FIFO >= RxLevel.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the UART_setRxCallback() function as the address of the ISR is used in the function call.

The Rx interrupt and associated PIE controller interrupt are enabled automatically by this function.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The Rx interrupt flag and PIE group acknowledgement flag are not cleared by the function. Therefore an Rx interrupt may be entered immediately after calling this function if the flags are not cleared beforehand using UART_ackRxInt().

Inside the ISR, UART_ackRxInt() must be called after reading the characters from the FIFO. Otherwise the interrupt flag will automatically be raised again since the cause of the interrupt is still valid.

14.2.12.2 Examples

The interrupt function isr_uart1_rx() will be called when a single character is received by UART module 1. The ISR is called which checks to make sure there are characters to be read and then reads these characters from the Rx FIFO. Finally the interrupt flag is cleared and the PIE group is acknowledged.

```
interrupt void isr_uart1_rx( void )
{
   char ch;

   if (UART_getRxCount( UART_MOD_1 ))
   {
      ch = UART_getc( UART_MOD_1 );
   }

   // Acknowledge interrupt
   UART_ackRxInt( UART_MOD_1 );
}
```

```
  UART_setRxCallback( UART_MOD_1, isr_uart1_rx, 1 );
  INT_enableGlobal( true );
```

NOTES

14.2.13 UART_ackRxInt

void UART_ackRxInt(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.13.1 Description

Used within an interrupt service routine to clear both the Rx interrupt flag and the PIE group acknowledgment flag.

14.2.13.2 Examples

Clears the Rx interrupt flag and the PIE group acknowledgment flag after the UART module 1 generates an Rx interrupt and the PIE controller calls the ISR. First the character is read from the Rx FIFO buffer and then the flags are cleared.

```
interrupt void isr_uart1_rx( void )
{
    // Get character
    ch = UART_getc( UART_MOD_1 );

    // Clear flags
    UART_ackRxInt( UART_MOD_1 );
}
```

14.2.14 UART_getIndex

int UART_getIndex(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.14.1 Description

Returns an index value for the UART module.

14.2.14.2 Examples

Returns the index 0 for UART 1.

```
int index = UART_getIndex(UART_MOD_1);
```

14.2.14.3 Notes

```
UART_MOD_1     0
   UART_MOD_2     1
```

14.2.15 UART_getRxPieId

INT_PieId UART_getRxPieId(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.15.1 Description

Returns the PIE Id for the Rx UART module.

14.2.15.2 Examples

Returns the PieId for UART 1.

```
INT_PieId id = UART_getRxPieId( UART_MOD_1 );
```

14.2.15.3 Notes

```
UART_MOD_1      INT_ID_SCIRXINTA
   UART_MOD_2      INT_ID_SCIRXINTB
```

14.2.16 UART_setTxCallback

void UART_setTxCallback(UART_Module Mod,INT_IsrAddr Func,int TxLevel)

where:
Mod - Selects the UART module.
Func - The pointer to the interrupt function.
TxLevel - The number of characters in the Tx FIFO before the interrupt is called.

14.2.16.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the Tx interrupt. The interrupt flag will be raised when the Tx FIFO is <= TxLevel.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the UART_setTxCallback() function as the address of the ISR is used in the function call.

The Tx interrupt and associated PIE controller interrupt are enabled automatically by this function.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The Tx interrupt flag and PIE group acknowledgement flag are not cleared by the function. Therefore a Tx interrupt may be entered immediately after calling this function if the flags are not cleared beforehand using UART_ackTxInt().

Inside the ISR, UART_ackTxInt() must be called after transferring the characters on to the Tx FIFO. Otherwise the interrupt flag will automatically be raised again since the cause of the interrupt is still valid.

14.2.16.2 Examples

The interrupt function isr_uart1_tx() will be called when the number of characters in the Tx FIFO is zero. The ISR is called which adds another character to the Tx FIFO. Finally the interrupt flag is cleared and the PIE group is acknowledged.

```
interrupt void isr_uart1_tx( void )
{
   UART_putc( UART_MOD_1, 'a' );

   // Acknowledge interrupt
   UART_ackTxInt( UART_MOD_1 );
}

UART_setTxCallback( UART_MOD_1, isr_uart1_tx, 0 );
INT_enableGlobal( true );
```

NOTES

14.2.17 UART_enableRxInt

void UART_enableRxInt(UART_Module Mod,int Enable)

where:
Mod - Selects the UART module.
Enable - Enables the interrupt.

14.2.17.1 Description

Enables/disables the Rx interrupt.

The interrupt is automatically enabled when UART_setRxCallback() is called.

No interrupts will be serviced until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

14.2.17.2 Examples

Enables the Rx interrupt for UART module 1.

```
UART_enableRxInt( UART_MOD_1, true );
```

NOTES

14.2.18 UART_enableTxInt

void UART_enableTxInt(UART_Module Mod,int Enable)

where:
Mod - Selects the UART module.
Enable - Enables the interrupt.

14.2.18.1 Description

Enables/disables the Tx interrupt.

The interrupt is automatically enabled when UART_setTxCallback() is called.

No interrupts will be serviced until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

14.2.18.2 Examples

Enables the Tx interrupt for UART module 1.

```
UART_enableTxInt( UART_MOD_1, true );
```

14.2.19 UART_getTxPieId

INT_PieId UART_getTxPieId(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.19.1 Description

Returns the PIE Id for the Tx UART module.

14.2.19.2 Examples

Returns INT_ID_SCITXINTA for UART 1.

```
  INT_PieId id = UART_getTxPieId( UART_MOD_1 );
```

14.2.19.3 Notes

```
UART_MOD_1      INT_ID_SCITXINTA
   UART_MOD_2      INT_ID_SCITXINTB
```

14.2.20 UART_ackTxInt

void UART_ackTxInt(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.20.1 Description

Used within an interrupt service routine to clear both the Tx interrupt flag and the PIE group acknowledgment flag.

14.2.20.2 Examples

Clears the Tx interrupt flag and the PIE group acknowledgment flag after the UART module 1 generates a Tx interrupt and the PIE controller calls the ISR. First a character is written to the Tx FIFO buffer and then the flags are cleared.

```
interrupt void isr_uart1_tx( void )
{
    // Write character
    UART_putc( UART_MOD_1, 'a');

    // Clear flags
    UART_ackTxInt( UART_MOD_1 );
}
```

14.2.21 UART_setTicks

void UART_setTicks(UART_Module Mod,uint16_t Ticks)

where:
Mod - Selects the UART module.
Ticks - The baud rate in UART ticks.

14.2.21.1 Description

Sets baud rate of the UART module as a number of UART clock ticks.

A baud rate can be converted to a number of UART clock ticks using the function UART_baudToTicks().

14.2.21.2 Examples

Sets the baud rate of UART module 1 to 115200 baud.

```
UART_setTicks( UART_MOD_1, UART_baudToTicks( 115200 ) );
```

14.2.22 UART_setLoopback

void UART_setLoopback(UART_Module Mod,int Value)

where:
Mod - Selects the UART module.
Value -

14.2.22.1 Description

```
Enables/disables loopback, a test mode, within the UART module.

The Rx and Tx pins are internally connected allowing data the transmitted data
from the Tx pin to be read in and stored on the Rx FIFO as data received
from the Rx pin.
```

14.2.22.2 Examples

```
Enables loop back mode. Any character transmitted on the UART module will
be received by the same UART module.

  UART_setLoopback( UART_MOD_1, true);
```

14.2.23 UART_clrRxInt

void UART_clrRxInt(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.23.1 Description

Clears the Rx interrupt flag only. Does not clear the PIE group acknowledgment flag.

If an Rx interrupt occurs and the Rx interrupt flag is set when the PIE group is not enabled, then the Rx interrupt flag will remain set. Therefore the Rx interrupt flag may need to be cleared before enabling the PIE group as any set flags will be serviced by the PIE controller when they are enabled.

After entering an interrupt service routine the Rx interrupt flag and PIE group acknowledgment flag must be cleared. If only the Rx interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

14.2.23.2 Examples

Clears the Rx interrupt flag for UART module 1.

```
UART_clrRxInt( UART_MOD_1 );
```

14.2.23.3 Notes

UART_ackRxInt() clears both the Rx interrupt flag and PIE group acknowledgement flag.

14.2.24 UART_clrTxInt

void UART_clrTxInt(UART_Module Mod)

where:
Mod - Selects the UART module.

14.2.24.1 Description

Clears the Tx interrupt flag only. Does not clear the PIE group acknowledgment flag.

If an Tx interrupt occurs and the Tx interrupt flag is set when the PIE group is not enabled, then the Tx interrupt flag will remain set. Therefore the Tx interrupt flag may need to be cleared before enabling the PIE group as any set flags will be serviced by the PIE controller when they are enabled.

After entering an interrupt service routine the Tx interrupt flag and PIE group acknowledgment flag must be cleared. If only the Tx interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

14.2.24.2 Examples

Clears the Tx interrupt flag for UART module 1.

```
UART_clrTxInt( UART_MOD_1 );
```

14.2.24.3 Notes

UART_ackTxInt() clears both the Tx interrupt flag and PIE group acknowledgement flag.

14.3 Types

14.3.1 UART_MOD_X

```
#if 1
#define UART_MOD_1 (&SciaRegs)
#define UART_MOD_2 (&ScibRegs)
#endif
```

14.3.1.1 Description

These values are used to specify the UART module.

14.3.2 UART_Module

```
typedef volatile struct SCI_REGS* UART_Module;
```

14.3.2.1 Description

This is used to map hardware register values to UART_Module.

14.3.3 UART_DataBits

```
enum UART_DataBits
{
    UART_DATA_5     = 4,
    UART_DATA_6     = 5,
    UART_DATA_7     = 6,
    UART_DATA_8     = 7
};
```

14.3.3.1 Description

These are the values for the number of data bits sent.

14.3.4 UART_Parity

```
enum UART_Parity
{
    UART_PARITY_NONE    = 0,
    UART_PARITY_ODD     = 2,
    UART_PARITY_EVEN    = 3
};
```

14.3.4.1 Description

These are the values for the type of parity used.

14.3.5 UART_StopBits

```
enum UART_StopBits
{
    UART_STOP_1     = 0,
    UART_STOP_2     = 1
};
```

14.3.5.1 Description

These are the values for the stops bits.

15 csl_spi_t0_

15.1.1.1 Description

Contains functions to set up the SPI module.

SPI_config() must be called before any of the API functions can be called. The functions allow SPI module to be configured as a master using the FIFOs to transmit and receive data.

15.1.1.2 Examples

Sets up the SPI module and enables the internal loopback mode. The module transmits 0x55 and then the code waits until this is received back.

```
SPI_config( SPI_MOD_1, 4, SPI_DO_POS_DI_NEG );
SPI_setLoopback(SPI_MOD_1, 1);

for(;;)
{
   // Transmit data
   SPI_write(SPI_MOD_1,0x55);

   // Wait until data is received
   while(SPI_getRxCount(SPI_MOD_1) == 0);
}
```

15.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru059e/spru059e.pdf
http://focus.ti.com/lit/ug/sprug71b/sprug71b.pdf (2802x, 2803x)

15.2 Api

SPI_config()
SPI_setTxCallback()
SPI_flush()
SPI_reset()
SPI_clrTxInt()
SPI_write()
SPI_getRxCount()
SPI_read()
SPI_setLoopback()
SPI_getIndex()
SPI_getTxPieId()
SPI_ackTxInt()
SPI_setRxCallback()
SPI_clrRxInt()
SPI_getRxPieId()
SPI_ackRxInt()
SPI_baudToTicks()

15.2.1 SPI_config

void SPI_config(SPI_Module Spi,uint16_t BRR,SPI_ClockEdge Mode)

where:
Spi - Selects the SPI module.
BRR - The baud rate for the SPI module.
Mode -

15.2.1.1 Description

```
Initializes the specified SPI module as an SPI master device using 16 bit words
and with a SPI_FIFO_DEPTH word deep Rx/Tx FIFO.

The low speed peripheral clock (LSPCLK) is used to generate the SPI module baud
rate. Unless it has been changed using SYS_setPerhiperalClk(), the default value
for the LSPCLK is SYSCLK/4. Therefore, using a 100MHz system clock and the
default settings would give a LSPCLK of 25MHz.

The baud rate of the SPI module is calculated as follows,

For BBR = 0 to 2                             LSPCLK
                    SPI baud Rate = ----------
                                                4

For BBR = 3 to 127                           LSPCLK
                    SPI baud Rate = ----------
                                              BBR+1

Therefore the maximum baud rate for the SPI module is LSPCLK/4. With the
settings described above this would give,

Maximum SPI baud rate = LSPCLK/4 = 25*10^6 / 4 = 6.25*10^6 bps

It is not recommended that the user run the SPI module at the maximum
baud rate as more errors will occur during transmission and greater
error detection and correction facilities may be required. Consider reducing the
baud rate if unexpected results are received from the slave devices as
not all devices will be able to communicate at the maximum baud rate.

The master SPI module is the only module responsible for generating the clock
used by any slave devices connected to the master device. Therefore all
slaves must be capable of operating at this baud rate.

SPI can be thought of as a data exchange protocol. Data is transmitted
within the same clock period that data is received. The edges of the clock
signal which result in data being transmitted and received must be specified.
There are two combinations,

Data is transmitted on the positive clock edge and received on the following
negative clock edge, or,

Data is transmitted on the negative clock edge and received on the following
positive clock edge.


                            _____       _____       _____
  SPI_DO_POS_DI_NEG        |     |_____|     |_____|     |_____|


                                  _____       _____       _____
```

```
  SPI_DO_NEG_DI_POS     |_____|     |_____|     |_____|     |

  SPISIMO / SPISOMI     |<databit 2>|<databit 1>|<databit 0>|
```

Where SPISIMO is the slave-in-master-out line and SPISOMI is the slave- out-master-in line.

The slave device should be configured with the same clock polarity as the master.

Refer to the device datasheet for the GPIO pins which connect to the specified SPI module. For C280x devices the GPIO pins are configured as follows,

	SPI Module	MOD_1 (A)	MOD_2 (B)	MOD_3 (C)	MOD_4 (D)
Pins	SPISIMO	GPIO16	GPIO24	GPIO20	GPIO1
	SPISOMI	GPIO17	GPIO25	GPIO21	GPIO3
	SPICLK	GPIO18	GPIO26	GPIO22	GPIO5
	SPISTE	GPIO19	GPIO27	GPIO23	GPIO7

This functions configures all of the necessary pins except for the slave transmit enable pin (SPISTE) which must be configured manually if it is required.

15.2.1.2 Examples

This sets up SPI module 1 with a baud rate of 6.25Mbps (assuming a system clock of 100MHz and a LSPCLK divider of 4). The data is transmitted on the negative clock edge and the received pin is sampled on the positive clock edge.

```
    SPI_config( SPI_MOD_1,
                3,
                SPI_DO_NEG_DI_POS );
```

NOTES

15.2.2 SPI_setTxCallback

void SPI_setTxCallback(SPI_Module Spi,INT_IsrAddr Func,uint16_t TxLevel)

where:
Spi - Selects the SPI module.
Func - The pointer to the interrupt function.
TxLevel - The tx fifo level to cause an interrupt.

15.2.2.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the Tx interrupt. The interrupt flag will be raised when the Tx FIFO is <= TxLevel.

The ISR will be called continuously until the words in the Tx FIFO > TxLevel.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the SPI_setTxCallback() function as the address of the ISR is used in the function call.

The Tx interrupt and associated PIE controller interrupt are enabled automatically by this function.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The Tx interrupt flag is cleared by the function however the PIE group acknowledgement flag is not.

Inside the ISR SPI_ackTxInt() must be called after transferring data on to the Tx FIFO. Otherwise the interrupt flag will automatically be raised again since the cause of the interrupt is still valid.

15.2.2.2 Examples

The interrupt function isr_uart1_tx() will be called when the number of words in the Tx FIFO is less than or equal to five. Within the ISR the interrupt flag is cleared and the PIE group is acknowledged.

```
interrupt void isr_spi1_tx( void )
{
   // User code

   // Acknowledge interrupt
   SPI_ackTxInt( SPI_MOD_1 );
}

SPI_setTxCallback( SPI_MOD_1, isr_spi1_tx, 5 );
INT_enableGlobal( true );
```

NOTES

15.2.3 SPI_flush

void SPI_flush(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.3.1 Description

Empties both the Rx and Tx SPI FIFOs.

15.2.3.2 Examples

Flushes the SPI module 1 Rx and Tx FIFOs.

```
SPI_flush(SPI_MOD_1);
```

15.2.4 SPI_reset

void SPI_reset(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.4.1 Description

Resets the SPI module and then resets the FIFOs.

Resetting the FIFOs ensures that there is no data remaining from the previous module run.

15.2.4.2 Examples

Resets SPI module 1.

```
  SPI_reset(SPI_MOD_1);
```

15.2.5 SPI_clrTxInt

void SPI_clrTxInt(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.5.1 Description

```
Clears the Tx interrupt flag only. Does not clear the PIE group
acknowledgment flag.

If an Tx interrupt occurs and the Tx interrupt flag is set when the PIE
group is not enabled, then the Tx interrupt flag will remain set. Therefore the
Tx interrupt flag may need to be cleared before enabling the PIE group
as any set flags will be serviced by the PIE controller when they are enabled.

After entering an interrupt service routine the Tx interrupt flag and
PIE group acknowledgment flag must be cleared. If only the Tx interrupt flag is
cleared then any subsequent interrupts will not be serviced by the PIE
controller.
```

15.2.5.2 Examples

```
Clears the Tx interrupt flag for SPI module 1.

  SPI_clrTxInt( SPI_MOD_1 );
```

15.2.5.3 Notes

```
SPI_ackTxInt() clears both the Tx interrupt flag and PIE group
acknowledgement flag.
```

15.2.6 SPI_write

void SPI_write(SPI_Module Spi,uint16_t Value)

where:
Spi - Selects the SPI module.
Value -

15.2.6.1 Description

Writes one word, 16 bits, to the Tx FIFO of the specified SPI module.

15.2.6.2 Examples

Writes 0x1234 to the Tx FIFO of SPI module 1.

```
SPI_write( SPI_MOD_1, 0x1234 );
```

15.2.7 SPI_getRxCount

uint16_t SPI_getRxCount(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.7.1 Description

Returns the number of items in the SPI Rx FIFO for the specified SPI module.

15.2.7.2 Examples

Returns the number of values in the SPI module 1 Rx FIFO.

```
uint16_t RxCountVal = SPI_getRxCount( SPI_MOD_1 );
```

15.2.8 SPI_read

uint16_t SPI_read(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.8.1 Description

Returns the first word, a 16 bit value, from the Rx FIFO for the specified SPI module.

15.2.8.2 Examples

Removes the 1st item from the FIFO and returns it to the user.

```
uint16_t SPIRead = SPI_read( SPI_MOD_1 );
```

15.2.9 SPI_setLoopback

void SPI_setLoopback(SPI_Module Mod,int Value)

where:
Mod - Selects the SPI module.
Value - Enables loopback.

15.2.9.1 Description

```
Enables/disables loopback, a test mode, within the SPI module.

The Rx and Tx pins are internally connected allowing the data transmitted
from the Tx pin to be read in and stored on the Rx FIFO as data received
from the Rx pin.
```

15.2.9.2 Examples

```
Enables loop back mode. Any words transmitted on the SPI module will
be received by the same SPI module.

  SPI_setLoopback( SPI_MOD_1, true );
```

15.2.10 SPI_getIndex

uint16_t SPI_getIndex(SPI_Module Mod)

where:
Mod - Selects the SPI module.

15.2.10.1 Description

Returns an index value for each SPI module.

The possibilities are as follows,

```
SPI_MOD_1 = 0
SPI_MOD_2 = 1
SPI_MOD_3 = 2
SPI_MOD_4 = 3
```

15.2.10.2 Examples

Returns 1 for SPI 2.

```
uint16_t SPIIndex = SPI_getIndex( SPI_MOD_2 );
```

NOTES

15.2.11 SPI_getTxPieId

INT_PieId SPI_getTxPieId(SPI_Module Mod)

where:
Mod - Selects the SPI module.

15.2.11.1 Description

Returns the PIE Id for the Tx part of the specified SPI module.

The possibilities are as follows,

```
SPI_MOD_1      INT_ID_SPITXA
SPI_MOD_2      INT_ID_SPITXB
SPI_MOD_3      INT_ID_SPITXC
SPI_MOD_4      INT_ID_SPITXD
```

15.2.11.2 Examples

Returns INT_ID_SCITXINTA for SPI module 1.

```
INT_PieId id = SPI_getTxPieId( SPI_MOD_1 );
```

15.2.12 SPI_ackTxInt

void SPI_ackTxInt(SPI_Module Mod)

where:
Mod - Selects the SPI module.

15.2.12.1 Description

Used within an interrupt service routine to clear both the Tx interrupt flag and the PIE group acknowledgment flag.

15.2.12.2 Examples

Clears the Tx interrupt flag and the PIE group acknowledgment flag after the SPI module 1 generates a Tx interrupt and the PIE controller calls the ISR.

```
interrupt void isr_spi1_tx( void )
{
   // User code

   // Clear flags
   SPI_ackTxInt( SPI_MOD_1 );
}
```

15.2.13 SPI_setRxCallback

void SPI_setRxCallback(SPI_Module Spi,INT_IsrAddr Func,uint16_t RxLevel)

where:
Spi - Selects the SPI module.
Func -
RxLevel - The number of values in the Rx FIFO before the interrupt is called.

15.2.13.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the Rx interrupt. The interrupt flag will be raised when the number of characters in the Rx FIFO >= RxLevel.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the SPI_setRxCallback() function as the address of the ISR is used in the function call.

The Rx interrupt and associated PIE controller interrupt are enabled automatically by this function.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The Rx interrupt flag is cleared by the function however the PIE group acknowledgement flag is not.

Inside the ISR, SPI_ackRxInt() must be called after reading the words from the FIFO. Otherwise the interrupt flag will automatically be raised again since the cause of the interrupt is still valid.

15.2.13.2 Examples

The interrupt function isr_spi1_rx() will be called when a single word is received by SPI module 1. The ISR is called which checks to make sure there are words to be read and then reads these words from the Rx FIFO.
Finally the interrupt flag is cleared and the PIE group is acknowledged.

```
interrupt void isr_spi1_rx( void )
{
   uint16_t word;

   if (SPI_getRxCount( SPI_MOD_1 ))
   {
      word = SPI_read( SPI_MOD_1 );
   }

   // Acknowledge interrupt
   SPI_ackRxInt( SPI_MOD_1 );
}

SPI_setRxCallback( SPI_MOD_1, isr_spi1_rx, 1 );
INT_enableGlobal( true );
```

NOTES

15.2.14 SPI_clrRxInt

void SPI_clrRxInt(SPI_Module Spi)

where:
Spi - Selects the SPI module.

15.2.14.1 Description

Clears the Rx interrupt flag only. Does not clear the PIE group acknowledgment flag.

If an Rx interrupt occurs and the Rx interrupt flag is set when the PIE group is not enabled, then the Rx interrupt flag will remain set. Therefore the Rx interrupt flag may need to be cleared before enabling the PIE group as any set flags will be serviced by the PIE controller when they are enabled.

After entering an interrupt service routine the Rx interrupt flag and PIE group acknowledgment flag must be cleared. If only the Rx interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

15.2.14.2 Examples

Clears the Rx interrupt flag for SPI module 1.

```
SPI_clrRxInt( SPI_MOD_1 );
```

15.2.14.3 Notes

SPI_ackRxInt() clears both the Rx interrupt flag and PIE group acknowledgement flag.

15.2.15 SPI_getRxPieId

void SPI_getRxPieId(SPI_Module Mod)

where:
Mod - Selects the SPI module.

15.2.15.1 Description

```
Returns the PIE Id for the Rx part of the specified SPI module.

The possibilities are as follows,

   SPI_MOD_1     INT_ID_SPIRXA
   SPI_MOD_2     INT_ID_SPIRXB
   SPI_MOD_3     INT_ID_SPIRXC
   SPI_MOD_4     INT_ID_SPIRXD
```

15.2.15.2 Examples

```
Returns INT_ID_SCIRXINTA for SPI module 1.

  INT_PieId id = SPI_getRxPieId( SPI_MOD_1 );
```

15.2.16 SPI_ackRxInt

void SPI_ackRxInt(SPI_Module Mod)

where:
Mod - Selects the SPI module.

15.2.16.1 Description

Used within an interrupt service routine to clear both the Rx interrupt flag and the PIE group acknowledgment flag.

15.2.16.2 Examples

Clears the Rx interrupt flag and the PIE group acknowledgment flag after the SPI module 1 generates an Rx interrupt and the PIE controller calls the ISR. The word is read from the Rx FIFO buffer first and then the flags are cleared.

```
interrupt void isr_spi1_rx( void )
{
   // Get character
   word = SPI_read( SPI_MOD_1 );

   // Clear flags
   SPI_ackRxInt( SPI_MOD_1 );
}
```

15.2.17 SPI_baudToTicks

uint16_t SPI_baudToTicks(int baud)

where:
baud - Selects the baud rate.

15.2.17.1 Description

Converts the required baud rate to the number of SPI clock ticks using the following formula.

```
                     SYS_CLK_HZ
Ticks =  ------------------------------ - 1
              USR_PER_LSP_DIV * baud
```

15.2.17.2 Examples

Returns the SPI ticks required for a baud rate of 115200.

```
  uint16_t BaudTicks = SPI_baudToTicks( 115200 );
```

15.2.17.3 Notes

This function only works if the value returned is not 0,1 or 2 due to the forumal below
 for Ticks = 0, 1, or 2

```
                     SYS_CLK_HZ
baud =  ------------------------------
              USR_PER_LSP_DIV * 4
```

15.3 Types

15.3.1 SPI_MOD_X

```
#if 1
#define SPI_MOD_1 (&SpiaRegs)
#define SPI_MOD_2 (&SpibRegs)
#define SPI_MOD_3 (&SpicRegs)
#define SPI_MOD_4 (&SpidRegs)
#endif
```

15.3.1.1 Description

These values are used to specify the SPI module.

15.3.2 SPI_Module

```
typedef volatile struct SPI_REGS* SPI_Module;
```

15.3.2.1 Description

This is used to map hardware register values to SPI_Module.

15.3.3 SPI_ClockEdge

```
enum SPI_ClockEdge
{
    SPI_DO_POS_DI_NEG   = 0,    /* data out on positive clock edge and in on
following negative clock edge */
    SPI_DO_NEG_DI_POS   = 1     /* data out on negative clock edge and in on
following positive clock edge */
};
```

15.3.3.1 Description

This defines the different sampling/transmitting options for the SPI module.

16 csl_tim_t0_

16.1.1.1 Description

Contains functions for configuring and manipulating the CPU timer modules.

All three timer modules can be used since the CSL library does not use BIOS.

All timing is measured in TIM clock ticks. The duration of a TIM clock tick is dependant on the system clock speed. The conversion functions provided can be used to convert from frequency and ns to TIM clock ticks.

16.1.1.2 Examples

Creates a timer with a system clock divider of 7. The timer counter register is loaded with a number of TIM clock ticks which will cause the timer to expire with a frequency of 1Hz using the function TIM_freqToTicks(). An interrupt service routine is configured to be executed when the timer module interrupt flag is raised.

```
TIM_config( TIM_MOD_1,
            TIM_freqToTicks( 1.0, 7 ),
            7 );
TIM_setCallback( TIM_MOD_1, isr_tim1 );

interrupt void isr_tim1(void)
{
   // Acknowledge timer interrupt
   TIM_ackInt(TIM_MOD);
   // User code
}
```

16.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru712f/spru712f.pdf

16.2 Api

TIM_setPeriod()
TIM_clrInt()
TIM_isInt()
TIM_nsToTicks()
TIM_reloadPeriod()
TIM_stop()
TIM_setPrecaler()
TIM_config()
TIM_setCallback()
TIM_getPrescaler()
TIM_freqToTicks()
TIM_getIndex()
TIM_ackInt()

16.2.1 TIM_setPeriod

void TIM_setPeriod(TIM_Module Mod,uint32_t Ticks)

where:
Mod - Selects the TIM module.
Ticks - Ticks in TIM module ticks.

16.2.1.1 Description

```
Sets the current period value of the timer.

The period is defined as a number of TIM clock ticks. With each TIM clock tick
lasting for the following number of seconds,

  TimTick = 1 / ( SYSCLKOUT / ( Prescaler ) )

Therefore if the function is called with an argument of 5 TIM clock ticks
and one TIM clock tick lasts for 10ns this would equate to a period of 50ns.

In this case the timer would expire after 50ns and an interrupt flag would
be raised.

The new period value is loaded in to the count register of the timer.
Therefore the counter will begin counting down from this new period value
immediately after the function call. The timer will expire when the count value
reaches zero.
```

16.2.1.2 Examples

```
Sets the period value for timer module 1 to 1000ns. This assumes that the clock
divider is set to 100 for timer module 1.

   TIM_setPeriod( TIM_MOD_1, TIM_nsToTicks( 1000, 100 ) );
```

16.2.2 TIM_clrInt

void TIM_clrInt(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.2.1 Description

Clears the TIM interrupt flag only. Does not clear the PIE group flag.

If a TIM interrupt occurs and the TIM interrupt flag is set when the PIE group is not enabled, then the TIM interrupt flag will remain set. Therefore the TIM interrupt flag may need to be cleared before enabling the PIE group as any set flags will be serviced by the PIE controller when it is enabled.

After entering an interrupt service routine the TIM interrupt flag and PIE group flag must be cleared. If only the TIM interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

16.2.2.2 Examples

Clears the interrupt flag for TIM module 1.

```
TIM_clrInt( TIM_MOD_1 );
```

16.2.2.3 Notes

TIM_ackInt() clears both the TIM interrupt flag and PIE group flag.

16.2.3 TIM_isInt

uint16_t TIM_isInt(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.3.1 Description

Returns the status of the timer interrupt flag.

If the returned value is non-zero then it means that the timer has expired.

If this function is being used inside a loop to detect if a timer has expired then the interrupt flag for the timer must be cleared after it has been detected as being set.

This will allow the function to detect that the timer has expired again.
This function will not clear the interrupt flag; use TIM_clrInt() to do so.

16.2.3.2 Examples

Sets timer module 1 to expire every 1Hz. The main loop waits until the timer has expired and clears the interrupt flag ready for the next cycle.

```
TIM_config( TIM_MOD_1,
            TIM_freqToTicks( 1.0, 7 ),
            7 );
while(1)
{
   while ( !TIM_isInt(TIM_MOD_1) ); //wait until expired
   TIM_clrInt(TIM_MOD_1); //clear int flag for next sequence
   // User code here
}
```

16.2.4 TIM_nsToTicks

uint32_t TIM_nsToTicks(uint32_t Ns,uint16_t Prescaler)

where:
Ns -
Prescaler -

16.2.4.1 Description

Returns the number of TIM ticks required for the time value, in nanoseconds, passed to the function.

The duration of one TIM tick is calculated using the Prescaler value passed as an argument to the function.

This function could be used when the TIM module is configured with TIM_config() to set the period count register and thus the length of time that the timer counts for.

16.2.4.2 Examples

Returns the number of TIM ticks required for 100ns with a prescaler of 2.

```
ui32_100nszInTicks = TIM_nsToTicks( 100, TIM_getPrescaler( TIM_MOD_1 ) );
```

For a device with a 100MHz system clock each TIM tick would last for 20ns using the Prescaler value of 2. Therefore this function would return a value of 5 TIM ticks. This would be the value that the TIM module counter must count down from in order to generate an interrupt with a period of 100ns.

Similarly, for a device with a 80MHz system clock each TIM tick would last for 25ns using the Prescaler value of 2. Therefore this function would return a value of 4 TIM ticks. Again this would be the value that the TIM module counter must count down from in order to generate an interrupt with a period of 100ns.

16.2.4.3 Notes

The maxium value is 1,000,000 (1ms) before the integer operation result is out of range.

16.2.5 TIM_reloadPeriod

void TIM_reloadPeriod(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.5.1 Description

Causes the count to be restarted for the specified timer module.

The count value is reloaded with the period register value. At every TIM clock tick this count value is decremented. When the count value reaches zero the timer expires.

No interrupt flag is set as a result of this function being called.

16.2.5.2 Examples

Causes the current timer value for TIM 1 to be reset back to the period value.

```
TIM_reloadPeriod( TIM_MOD_1 );
```

16.2.6 TIM_stop

void TIM_stop(TIM_Module Mod,int Value)

where:
Mod - Selects the TIM module.
Value - Enable/disables the TIM module.

16.2.6.1 Description

```
Stops or starts the timer.

When the timer is stopped (when Value == true) the timer count value is not
decremented. Therefore the timer will not expire and an interrupt will not
be generated as the time will not reach zero.

When the timer is running (when Value == false) the timer count value is
decremented by the timer module clock. Therefore the timer will expire when the
count value reaches zero. An interrupt will be generated if it is enabled within
the module.

EXAMPLE
Stops timer module 1 from counting down.

  TIM_stop( TIM_MOD_1, true );
```

16.2.7 TIM_setPrecaler

void TIM_setPrecaler(TIM_Module Mod,uint16_t Value)

where:
Mod - Selects the TIM module.
Value - System clock divider.

16.2.7.1 Description

Sets the timer prescaler value.

Each CPU timer module generates its own clock derived from the system clock using this prescaler value.

A CPU timer tick is generated from the system clock as follows,

```
  TimTick = 1 / ( SYSCLKOUT / ( Prescaler ) )
```

Where the 'Prescaler' argument is a 16-bit value which is used to divide down the system clock. The 'Value' argument to the function, the prescaler value, must be greater than zero.

16.2.7.2 Examples

Sets the timer module 1 system clock divider to 100.

```
  TIM_setPrecaler( TIM_MOD_1, 100 );
```

16.2.7.3 Notes

A value of zero will produce unexpected results.

16.2.8 TIM_config

void TIM_config(TIM_Module Mod,uint32_t Ticks,uint16_t Prescale)

where:
Mod - Selects the TIM module.
Ticks - System clock divider.
Prescale -

16.2.8.1 Description

```
Configures a CPU timer module, sets its period count and prescaler values.

Each timer module has a 32 bit period count register. This is the number of CPU
timer clock ticks (TIM ticks) that must be counted before an interrupt flag is
raised.

Therefore the period count register is set as a number of TIM ticks. A value of
5 using a 10ns tick would give a period of 50ns (or a frequency of 20MHz).

A CPU timer tick is generated from the system clock as follows,

  TimTick = 1 / ( SYSCLKOUT / ( Prescaler ) )

The 'Prescale' argument is a 16-bit value which is used to divide down the
system clock.

Given the Prescaler argument, the number of TIM ticks require to generate a
timer with a particular frequency can be calculated using the function
TIM_freqToTicks().
```

16.2.8.2 Examples

```
Starts a 3Hz timer, using timer module 1, with a prescaler of 2.

  TIM_config( TIM_MOD_1 ,
              TIM_freqToTicks(3, 2),
              2 );

NOTES
```

16.2.9 TIM_setCallback

void TIM_setCallback(TIM_Module Mod,INT_IsrAddr Func)

where:
Mod - Selects the TIM module.
Func - The pointer to the interrupt function.

16.2.9.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the timer module which will be called when the timer expires if interrupts are enabled.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the TIM_setCallback() function as the address of the ISR is used in the function call.

PIE controller interrupts and timer interrupts are enabled automatically by this function for the specified TIM module.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

The TIM interrupt flags are cleared by the function however the PIE group flag is not. To allow the ISR to be called after the first interrupt the TIM interrupt flag and PIE group flag must be cleared using TIM_ackInt().

16.2.9.2 Examples

The interrupt function isr_tim1() will be called when timer module 1 expires.

```
interrupt void isr_tim1( void )
{
   TIM_ackInt( TIM_MOD_1 );
}

TIM_setCallback( TIM_MOD_1, isr_tim1 );
INT_enableGlobal( true );
```

NOTES

16.2.10 TIM_getPrescaler

uint16_t TIM_getPrescaler(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.10.1 Description

Returns the current prescaler value for the specified timer module.

The prescaler is used to generate the CPU timer clock. A CPU timer tick (TIM tick) is generated from the system clock as follows,

```
TimTick = 1 / ( SYSCLKOUT / ( Prescaler ) )
```

16.2.10.2 Examples

Returns the current prescaler value for timer module 1.

```
ui_Prescaler = TIM_getPrescaler( TIM_MOD_1 );
```

16.2.10.3 Notes

The Prescaler value can be set using TIM_setPrescaler().

16.2.11 TIM_freqToTicks

uint32_t TIM_freqToTicks(uint16_t Fs_Hz,void Prescaler)

where:
Fs_Hz -
Prescaler -

16.2.11.1 Description

Returns the number of TIM ticks required to generate the frequency value (in Hertz) passed to the function.

The duration of one TIM tick is calculated using the Prescaler value passed as an argument to the function.

This function could be used when the TIM module is configured with TIM_config() to set the period count register and thus frequency of the timer.

16.2.11.2 Examples

Returns the number of TIM ticks required for 100kHz with a prescaler of 2.

```
  ui32_100kHzInTicks = TIM_freqToTicks( 100000, 2 );
```

For a device with a 100MHz system clock each TIM tick would last for 20ns using the Prescaler value of 2. Therefore this function would return a value of 500 TIM ticks. This would be the value that the TIM module counter must count down from in order to generate an interrupt at a frequency of 100kHz.

Similarly, for a device with a 80MHz system clock each TIM tick would last for 25ns using the Prescaler value of 2. Therefore this function would return a value of 400 TIM ticks. Again this would be the value that the TIM module counter must count down from in order to generate an interrupt at a frequency of 100kHz.

NOTES

16.2.12 TIM_getIndex

void TIM_getIndex(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.12.1 Description

Returns the index value for each TIM module, e.g.

```
TIM_MOD_1 = 0
TIM_MOD_2 = 1
     :
TIM_MOD_n = n-1
```

16.2.12.2 Examples

Returns 2 for TIM module 3.

```
TIM_getIndex( TIM_MOD_3 );
```

16.2.13 TIM_ackInt

void TIM_ackInt(TIM_Module Mod)

where:
Mod - Selects the TIM module.

16.2.13.1 Description

Used within an interrupt service routine to clear both the TIM interrupt flag and the PIE group flag.

16.2.13.2 Examples

Clears the TIM interrupt flag and the PIE group flag after the TIM module 1 generates an interrupt and the PIE controller calls this ISR.

```
interrupt void isr_tim1( void )
{
   TIM_ackInt( TIM_MOD_1 );
}
```

16.3 Types

16.3.1 TIM_MOD_X

```
#if 1
#define TIM_MOD_1 (&CpuTimer0Regs)
#define TIM_MOD_2 (&CpuTimer1Regs)
#define TIM_MOD_3 (&CpuTimer2Regs)
#endif
```

16.3.1.1 Description

These values are used to specify the TIM module.

16.3.2 TIM_Module

```
typedef volatile struct CPUTIMER_REGS* TIM_Module;
```

16.3.2.1 Description

This is used to map hardware register values to the TIM Module.

17 csl_i2c_t0_

17.1.1.1 Description

Sets up the I2C module. The module is currently under development and has limited functionality.

It can be configured as an 8-bit master only. The FIFOs are always used and interrupts are not yet supported.

17.1.1.2 Examples

Sets up the I2C module 1 and writes two bytes, 0x55 and 0x66, to the device at address Addr. Following this, five bytes are read back and stored in the array 'data'. The process is repeated.

```
I2C_config( I2C_MOD_1, 9, 10, 5 );

for(;;)
{
   uint8_t data[5] = { 0x55, 0x66 };
   I2C_write( I2C_MOD_1, Addr, 2, data );
   I2C_read( I2C_MOD_1, Addr, 5, data );
}
```

17.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru721a/spru721a.pdf

17.2 Api

I2C_config()
I2C_write()
I2C_read()
I2C_writeAddr()
I2C_writeData()
I2C_writeEnd()

17.2.1 I2C_config

void I2C_config(I2C_Module Mod,uint16_t IPSC,uint16_t ICCL,uint16_t ICCH)

where:
Mod - Selects the I2C module.
IPSC - Prescaler for the system low speed clock.
ICCL - Low byte for the I2C module clock divider.
ICCH - High byte for the I2C module clock divider.

17.2.1.1 Description

Initializes the I2C module. This must be called before any of the API functions are called.

The I2C module must be clocked between 7 and 12MHz. The input clock source to the I2C clocking module is equivalent to the system clock. The argument IPSC is the value that this input clock is divided by to provide the I2C module clock.

Therefore if a 100MHz system clock were being used,

```
I2C_CLOCK = ( sysClkHz / (IPSC+1) )
          = ( 100MHz   / (9    +1) ) =  10MHz   // Between 7 and 12Mhz
```

If a 60MHz system clock were being used,

```
I2C_CLOCK = ( sysClkHz / (IPSC+1) )
          = ( 60MHz    / (5    +1) ) =  10MHz   // Between 7 and 12Mhz
```

This I2C module clock is divided down further to provide the master clock which appears on the SCL pin when the I2C module is transmitting data. The arguments ICCL and ICCH determine the master clock period as follows,

```
Tmaster = ( (ICCL+d) + (ICCH+d) ) / I2C_CLOCK
```

Where 'd' depends on the value of IPSC as follows,

IPSC	d
0	7
1	6
>1	5

17.2.1.2 Examples

Assuming a 100MHz system clock, the I2C module 1 is configured with a I2C module clock of 10MHz and an appropriate master clock value.

```
I2C_config( I2C_MOD_1, 9, 10, 5 );
```

Assuming a 60MHz system clock, the I2C module 1 is configured with a I2C module clock of 10MHz and an appropriate master clock value.

```
I2C_config( I2C_MOD_1, 5, 10, 5 );
```

17.2.1.3 Notes

For C280x devices the I2C pins are as follows,
  SDAA - GPIO_32

```
  SCLA - GPIO_33
```

This module is still under development.

17.2.2 I2C_write

I2C_Status I2C_write(I2C_Module Mod,uint16_t Address,uint16_t Argc,const uint8_t* Argi)

where:
Mod - Selects the I2C module.
Address - Address of I2C device on bus.
Argc - The number of bytes to write.
Argi - A pointer to an array of bytes to write.

17.2.2.1 Description

```
Writes Argc bytes from the data pointer Argi to the requested device at the
address specified on the I2C bus.

A return value of I2C_STATUS_SUCCESS indicates a successful operation.

EXAMPLE
Writes 2 bytes to the I2C address 0x11.

  uint8_t data[2] = { 0x55, 0x66 };
  I2C_write( I2C_MOD_1, 0x11, 2, data );
```

17.2.2.2 Notes

```
This module is still under development.
```

17.2.3 I2C_read

I2C_Status I2C_read(I2C_Module Mod,uint16_t Address,uint16_t Argc,uint8_t* Argi)

where:
Mod - Selects the I2C module.
Address - The I2C device number.
Argc - The number of bytes to read.
Argi - The destination pointer.

17.2.3.1 Description

Reads Argc bytes into the data pointer Argi from the requested device at the address specified on the I2C bus.

The return value of I2C_STATUS_SUCCESS indicates a successful operation.
EXAMPLE
Reads 2 bytes from the I2C address 0x11.

```
uint8_t data[];
I2C_read( I2C_MOD_1, 0x11, 2, data );
```

17.2.3.2 Notes

This module is still under development.

17.2.4 I2C_writeAddr

I2C_Status I2C_writeAddr(I2C_Module Mod,uint16_t Address,uint16_t Argc)

where:
Mod - Selects the I2C module.
Address - The I2C device number.
Argc - The number of bytes to write.

17.2.4.1 Description

Starts an I2C write session by sending the I2C device address and setting the number of bytes to write for the I2C module.

A return value of I2C_STATUS_SUCCESS indicates a successful operation.

EXAMPLE
Writes 100 bytes to the I2C address 0x11.

```
uint8_t data[1];
I2C_writeAddr( I2C_MOD_1, 0x11, 100 );
for ( i = 0 ; i < 100 ; i++ )
{
   data[1] = i;
   I2C_writeData( I2C_MOD_1, 1, data );
}
I2C_writeEnd( I2C_MOD_1 );
```

17.2.4.2 Notes

This module is still under development.

17.2.5 I2C_writeData

I2C_Status I2C_writeData(I2C_Module Mod,uint16_t Argc,const uint8_t* Argi)

where:
Mod - Selects the I2C module.
Argc - The number of bytes to write.
Argi - A pointer to an array of bytes to write.

17.2.5.1 Description

Writes Argc bytes from the data pointer Argi to the requested device at the address in the I2CSAR register on the I2C bus.

The total number of bytes written from all calls to this function must not exceed the maximum specified in the call to I2C_writeAddr(). The address that is written to, I2CSAR, is also set in the call to I2C_writeAddr().

A return value of I2C_STATUS_SUCCESS indicates a successful operation.

EXAMPLE
Writes 100 bytes to the I2C address 0x11.

```
uint8_t data[1];
I2C_writeAddr( I2C_MOD_1, 0x11, 100 );
for ( i = 0 ; i < 100 ; i++ )
{
  data[1] = i;
  I2C_writeData( I2C_MOD_1, 1, data );
}
I2C_writeEnd(I2C_MOD_1);
```

17.2.5.2 Notes

This module is still under development.

17.2.6 I2C_writeEnd

I2C_Status I2C_writeEnd(I2C_Module Mod)

where:
Mod - Selects the I2C module.

17.2.6.1 Description

Closes the current I2C write session by waiting for a stop bit or NACK to be sent.

17.2.6.2 Examples

Closes the I2C session open on module 1.

```
I2C_writeEnd( I2C_MOD_1 );
```

17.2.6.3 Notes

This module is still under development.

17.3 Types

17.3.1 I2C_Status

```
enum I2C_Status
{
    I2C_STATUS_ERROR                  = (int)0xFFFE,
    I2C_STATUS_ARB_LOST_ERROR         = 0x0001,
    I2C_STATUS_NACK_ERROR             = 0x0002,
    I2C_STATUS_TIME_OUT               = 0x0003,
    I2C_STATUS_BUS_BUSY_ERROR         = 0x1000,
    I2C_STATUS_STP_NOT_READY_ERROR    = 0x5555,
    I2C_STATUS_NO_FLAGS               = (int)0xAAAA,
    I2C_STATUS_SUCCESS                = 0x0000
};
```

17.3.1.1 Description

These are the status values returned from the I2C module.

17.3.2 I2C_MOD_1

```
#define I2C_MOD_1 (&I2caRegs)
```

17.3.2.1 Description

These values are used to specify the I2C module.

17.3.3 I2C_Module

```
typedef volatile struct I2C_REGS* I2C_Module;
```

17.3.3.1 Description

This is used to map hardware register values to I2C Module.

18 csl_cap_t0_

18.1.1.1 Description

The capture module allows you to take 4 samples of the 32 bit counter.
Each module has 4 event channels that can be configured to store the counter value on either a rising or falling edge of the input capture pin.
On each capture the counter can be reset to allow the difference between events to be stored.
After the last event channel has been store the event channel can wrap around in continuous mode or stop in single shot mode.

The counter is clocked by the system clock.

18.1.1.2 Examples

This sets up CAP_MOD_4 to capture the counter on event channel 1 on the negative edge and reset the counter. Then the event channel 2 is triggered on the next rising edge. This causes an interrupt and the positive width is stored in pos_wdith.

```
CAP_config( CAP_MOD_4, CAP_DIV_1, CAP_CONTINUOUS,
                CAP_MOD_4_PIN_27, GPIO_SAMPLE_3 );

CAP_setCapture( CAP_MOD_4, CAP_CH_1, CAP_CTR_DIF, CAP_EVENT_POS );
CAP_setCapture( CAP_MOD_4, CAP_CH_2, CAP_CTR_ABS, CAP_EVENT_NEG );
CAP_setCallback( CAP_MOD_4, Irs, CAP_INT_CEVT2 );
CAP_stop(CAP_MOD_4, true);

INT_enableGlobal(1);

   interrupt void Irs( void )
   {
    CAP_ackInt( CAP_MOD_4, CAP_INT_CEVT2 );
    pos_wdith = CAP_getValue( CAP_MOD_4, CAP_CH_2 );
   }
```

18.2 Api

CAP_config()
CAP_enableLoad()
CAP_getValue()
CAP_setReArm()
CAP_setMaxChannel()
CAP_stop()
CAP_setCapture()
CAP_getIndex()
CAP_getPieId()
CAP_enableInt()
CAP_setCallback()
CAP_getIntFlags()
CAP_clrInt()
CAP_ackInt()
CAP_setMode()
CAP_softwareStart()
CAP_setCounter()
CAP_nsToTicks()
CAP_usToTicks()

18.2.1 CAP_config

void CAP_config(CAP_Module Mod,CAP_PreScale PreScale,CAP_Mode Mode,CAP_Pin Pin,GPIO_InputMode InputMode)

where:
Mod - Selects the eCAP module.
PreScale - This prescale the input trigger.
Mode - Single shot or continuous mode.
Pin - Selects the pin to use for triggering.
InputMode - Number of average samples taken for the trigger input.

18.2.1.1 Description

Initializes the CAP module. All 4 capture channels are set for absolute mode and rising edge polarity.

The presale can be used when the events are very close together or you want to count how many events occur before the counter value is stored in the event channel registers.

The trigger pin can be located on various different pins which are specified by CAP_Pin.

To avoid spikes causing false triggers you can specify the number of average sample the trigger pin uses before being used as a trigger to the module.

18.2.1.2 Examples

This configures CAP_MOD_4 module wait for a single trigger event per event channel register.
The module is in continuous mode so after the last event channel is stored the module starts again at event channel 1.
The trigger pin is connected to GPIO_27 which is averaged by 3 samples before being used by the module.

```
  CAP_config( CAP_MOD_4, CAP_DIV_1, CAP_CONTINUOUS,
                   CAP_MOD_4_PIN_27, GPIO_SAMPLE_3 );
```

18.2.1.3 Notes

You need to call CAP_setCapture to set up each event channel.

18.2.2 CAP_enableLoad

void CAP_enableLoad(CAP_Module Mod,int Enable)

where:
Mod - Selects the eCAP module.
Enable - Enables the module to change the event channel registers.

18.2.2.1 Description

This enables the module to update the CAP1-4 channel registers on a capture event.
When an event is detected the current event channel is updated with the current count value if the channel registers are enabled.

18.2.2.2 Examples

This enables CAP_MOD_1 module to write to the event channel registers.

```
CAP_enableLoad(CAP_MOD_1, true);
```

18.2.2.3 Notes

By default the event channel registers are enabled after calling CAP_config().

18.2.3 CAP_getValue

uint32_t CAP_getValue(CAP_Module Mod,CAP_ModuleChannel Ch)

where:
Mod - Selects the eCAP module.
Ch - Selected the event channel register.

18.2.3.1 Description

This function reads the value stored in the event channel register.
These are 32 bits in size.

18.2.3.2 Examples

This reads the 32 bit value from CAP_MOD_1 event channel 2.

```
uint32_t c = CAP_getValue(CAP_MOD_1, CAP_CH_2);
```

18.2.4 CAP_setReArm

void CAP_setReArm(CAP_Module Mod)

where:
Mod - Selects the eCAP module.

18.2.4.1 Description

When the module is configured for single shot the module stops sampling after the last event is captured (see CAP_setMaxChannel).
To re-arm the module to start a new sequence you can call this function.

18.2.4.2 Examples

This re-arms the CAP_MOD_1 module to re-capture a sequence of events.

```
CAP_setReArm(CAP_MOD_1);
```

18.2.5 CAP_setMaxChannel

void CAP_setMaxChannel(CAP_Module Mod,CAP_ModuleChannel Ch)

where:
Mod - Selects the eCAP module.
Ch - Selected the event channel register.

18.2.5.1 Description

This sets the number of events to capture before either wrapping around in continuous mode or stopping in single shot mode.

18.2.5.2 Examples

This sets CAP_MOD_1 module to sample 3 event channels before wrapping back to the first event channel again.

```
CAP_setMode(CAP_MOD_1, CAP_CONTINUOUS);
CAP_setMaxChannel(CAP_MOD_1, CAP_CH_3);
```

18.2.5.3 Notes

In single shot mode you need to call CAP_setReArm() to re-capture a sequence of events.

18.2.6 CAP_stop

void CAP_stop(CAP_Module Mod,int Disable)

where:
Mod - Selects the eCAP module.
Disable -

18.2.6.1 Description

This stops or starts the module 32 bit counter from free running.
This 32 bit value is stored in the event channel registers when the correct event is detected.

18.2.6.2 Examples

This sets CAP_MOD_1 module to sample 3 event channels before wrapping back to the first event channel again.

```
CAP_setMode(CAP_MOD_1, CAP_CONTINUOUS);
CAP_setMaxChannel(CAP_MOD_1, CAP_CH_3);
```

18.2.6.3 Notes

The 32 bit counter wraps around automatically in free running mode.
You can change the counter value with CAP_setCounter().

18.2.7 CAP_setCapture

void CAP_setCapture(CAP_Module Mod,CAP_ModuleChannel Channel,CAP_CounterReset Reset,CAP_EventPolarity Polarity)

where:
Mod - Selects the eCAP module.
Channel - Selected the event channel register.
Reset - Resets the counter after storing the counter in the event register.
Polarity - Capture the counter value on the rising or falling edge.

18.2.7.1 Description

After calling CAP_config() you need to set the how each event register will store the counter value. The event can be triggered on the falling or rising edge of the input.
After the event is triggered you can reset the counter value. This is can be used to measure the relative time between events.

18.2.7.2 Examples

This sets CAP_MOD_4 module to sample 2 event channels continuous and enables the counter to free run. The first event waits for a negative edge before storing the counter value. Then the second event waits for a rising edge before storing the next counter value. The module then repeats this series of capture events.

```
  CAP_config( CAP_MOD_4, CAP_DIV_1, CAP_CONTINUOUS,
                CAP_MOD_4_PIN_27, GPIO_SAMPLE_3 );
  CAP_stop(Mod, false);
CAP_setCapture( CAP_MOD_4, CAP_CH_1, CAP_CTR_ABS, CAP_EVENT_NEG );
CAP_setCapture( CAP_MOD_4, CAP_CH_2, CAP_CTR_ABS, CAP_EVENT_POS );
```

18.2.7.3 Notes

This function also calls CAP_setMaxChannel() using the Channel as it's augment.

18.2.8 CAP_getIndex

int CAP_getIndex(CAP_Module Mod)

where:
Mod - Selects the eCAP module.

18.2.8.1 Description

Returns the index value for each capture module, e.g.

```
CAP_MOD_1 = 0
CAP_MOD_2 = 1
```

18.2.8.2 Examples

Returns 1 for capture module 2.

```
1 == CAP_getIndex( CMP_MOD_2 );
```

18.2.9 CAP_getPieId

INT_PieId CAP_getPieId(CAP_Module Mod)

where:
Mod - Selects the eCAP module.

18.2.9.1 Description

Returns the PIE Id associated with the CAP interrupt.

This can be used when configuring the PIE controller manually using the interrupt functions from the INT CSL library.

This function does not normally need to be called as the interrupts are configured automatically using CAP_setCallback().

18.2.9.2 Examples

The function will return INT_ID_ECAP1 when the module CAP_MOD_1 is passed as the argument.

```
PieId = CAP_getPieId( CAP_MOD_1 );
```

18.2.10 CAP_enableInt

void CAP_enableInt(CAP_Module Mod,int Mask)

where:
Mod - Selects the eCAP module.
Mask - A bit mask of CAP_IntMode used to repersent a collection of interrupts.

18.2.10.1 Description

The module can generate an interrupt when any of the capture events are triggered or when the counter wraps around.

18.2.10.2 Examples

This enable interrupts for CAP_MOD_2 when event channel 2 is triggered or the counter wraps around.

```
CAP_enableInt( CMP_MOD_2, CAP_INT_CEVT2 | CAP_INT_CTR_OVF );
```

18.2.10.3 Notes

This is called automatically when CAP_setCallback() is used.

18.2.11 CAP_setCallback

void CAP_setCallback(CAP_Module Mod,INT_IsrAddr Func,int Mask)

where:
Mod - Selects the eCAP module.
Func -
Mask - A bit mask of CAP_IntMode used to repersent a collection of interrupts.

18.2.11.1 Description

Assigns a function, an interrupt service routine (ISR), to the interrupt vector of the eCAP module.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the CAP_setCallback() function as the address of the ISR is used in the function call.

PIE controller interrupts are enabled automatically by this function for the specified eCAP module.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

18.2.11.2 Examples

In this example the function isr_cap1() will be called each time the counter value wraps around or event channel 2 is triggered.

```
interrupt void isr_cap1( void )
{
  // The next line clears the eCAP and PIE group flags
  CAP_ackInt(CAP_MOD_1, CAP_INT_CEVT2 | CAP_INT_CTR_OVF);

  // User code here
}

CAP_setCallback( CAP_MOD_1, isr_cap1, CAP_INT_CEVT2 | CAP_INT_CTR_OVF );
INT_enableGlobal( true );
```

18.2.11.3 Notes

If a NULL pointer is passed as the function pointer, then the interrupt will be enable for the module, but not within the PIE module.

18.2.12 CAP_getIntFlags

uint16_t CAP_getIntFlags(CAP_Module Mod)

where:
Mod - Selects the eCAP module.

18.2.12.1 Description

This returns the bit mask of the events that caused an interrupt to occur. Since multiple events can cause an interrupt this function you can use this function to determine which event caused the interrupt.

18.2.12.2 Examples

This enable interrupts for CAP_MOD_2 when event channel 2 is triggered or the counter wraps around.

```
if      ( CAP_getIntFlags( CMP_MOD_2 ) && CAP_INT_CTR_OVF )
    //user code for counter wrap
else if ( CAP_getIntFlags( CMP_MOD_2 ) && CAP_INT_CEVT2 )
    //user code for handling event channel 2 trigger
```

18.2.13 CAP_clrInt

void CAP_clrInt(CAP_Module Mod,int Mask)

where:
Mod - Selects the eCAP module.
Mask - A bit mask used to repersent a collection of interrupts.

18.2.13.1 Description

Clears the CAP interrupt masks. Does not clear the PIE group flag.

The CAP interrupt is assigned to an CAP module using the function CAP_setCallback().

After entering an interrupt service routine the CAP interrupt mask and PIE group flag must be cleared. If only the CAP interrupt flag is cleared then any subsequent interrupts will not be serviced by the PIE controller.

18.2.13.2 Examples

This only clears the CAP interrupt masks for event channel 2 trigger and counter wrap around.

```
CAP_clrInt( CMP_MOD_2, CAP_INT_CEVT2 | CAP_INT_CTR_OVF );
```

18.2.13.3 Notes

When any event triggers an interrupt the bit CAP_INT is also set, which this function automatically clears.

18.2.14 CAP_ackInt

void CAP_ackInt(CAP_Module Mod,int Mask)

where:
Mod - Selects the eCAP module.
Mask - A bit mask used to repersent a collection of interrupts.

18.2.14.1 Description

Used within an interrupt service routine to clear both the CAP interrupt mask and the PIE group flag.

18.2.14.2 Examples

This clears the CAP interrupt mask for event channel 2 trigger and counter wrap around.
Also the PIE group flag for the specified interrupt.

```
interrupt void isr_cap1( void )
{
   CAP_ackInt( CMP_MOD_2, CAP_INT_CEVT2 | CAP_INT_CTR_OVF );
}
```

18.2.15 CAP_setMode

void CAP_setMode(CAP_Module Mod,CAP_Mode Mode)

where:
Mod - Selects the eCAP module.
Mode - Single shot or continuous mode.

18.2.15.1 Description

```
This sets the module to use single shot or continuous mode.
In single shot the module stops sampling after the last event is captured.
In continuous the module wraps back to event channel 1 after the last event is
triggered.
```

18.2.15.2 Examples

```
This set the module to use single shot.

     CAP_setMode( CMP_MOD_2, CAP_ONE_SHOT );
```

18.2.16 CAP_softwareStart

void CAP_softwareStart(CAP_Module Mod,int Mask)

where:
Mod - Selects the eCAP module.
Mask - A bit mask used to repersent a collection of interrupts.

18.2.16.1 Description

This forces the event mask for the module, which will cause an event to be generated.

18.2.16.2 Examples

This set the module CMP_MOD_2 events to think the event channel 2 register has occurred.

```
CAP_softwareStart( CMP_MOD_2, CAP_INT_CEVT2 );
```

18.2.17 CAP_setCounter

void CAP_setCounter(CAP_Module Mod,uint32_t Value)

where:
Mod - Selects the eCAP module.
Value - The new 32 bit value for the counter.

18.2.17.1 Description

This sets the module 32 bit counter with the new value.

18.2.17.2 Examples

This set the module CMP_MOD_2 counter to overflow in 10ms.

```
CAP_setCounter( CMP_MOD_2, 0xFFFFFFFF-CAP_usToTicks(10000) );
```

18.2.18 CAP_nsToTicks

uint32_t CAP_nsToTicks(uint64_t Ns)

where:
Ns - Number of nanoseconds.

18.2.18.1 Description

Returns the number of eCAP ticks required for the time value, in nanoseconds, passed to the function.

The duration of one eCAP tick is calculated using the system clock.

18.2.18.2 Examples

Returns the number of eCAP ticks required for 100ns

```
ui_100nsInTicks = CAP_nsToTicks(100);
```

For a device with a 100MHz system clock each eCAP tick would last for 10ns. Therefore the function would return a value of 10 eCAP ticks.

18.2.19 CAP_usToTicks

uint32_t CAP_usToTicks(uint32_t Us)

where:
Us - Number of microseconds.

18.2.19.1 Description

Returns the number of eCAP ticks required for the time value, in microseconds, passed to the function.

The duration of one eCAP tick is calculated using the system clock.

18.2.19.2 Examples

Returns the number of eCAP ticks required for 10ms

```
ui_10msInTicks = CAP_nsToTicks(10000);
```

For a device with a 100MHz system clock each eCAP tick would last for 10ns. Therefore the function would return a value of 1000 eCAP ticks.

18.3 Types

18.3.1 CAP_MOD_X

```
#if 1
#define CAP_MOD_1 (&ECap1Regs)
#define CAP_MOD_2 (&ECap2Regs)
#define CAP_MOD_3 (&ECap3Regs)
#define CAP_MOD_4 (&ECap4Regs)
#define CAP_MOD_5 (&ECap5Regs)
#define CAP_MOD_6 (&ECap6Regs)
#endif
```

18.3.1.1 Description

These values are used to specify the Capture module.

18.3.2 CAP_Module

```
typedef volatile struct ECAP_REGS* CAP_Module;
```

18.3.2.1 Description

This is used to map hardware register values to CAP_Module.

18.3.3 CAP_ModuleChannel

```
enum CAP_ModuleChannel
{
    CAP_CH_1,
    CAP_CH_2,
    CAP_CH_3,
    CAP_CH_4
};
```

18.3.3.1 Description

Each capture module has 4 capture channels.

18.3.4 CAP_PreScale

```
enum CAP_PreScale
{
    CAP_DIV_1    = SYS_LIT( 1,   0 ),
    CAP_DIV_2    = SYS_LIT( 2,   1 ),
    CAP_DIV_4    = SYS_LIT( 4,   2 ),
    CAP_DIV_6    = SYS_LIT( 6,   3 ),
    CAP_DIV_8    = SYS_LIT( 8,   4 ),
    CAP_DIV_10   = SYS_LIT( 10,  5 ),
    CAP_DIV_12   = SYS_LIT( 12,  6 ),
    CAP_DIV_14   = SYS_LIT( 14,  7 ),
    CAP_DIV_16   = SYS_LIT( 16,  8 ),
    CAP_DIV_18   = SYS_LIT( 18,  9 ),
```

```
    CAP_DIV_20   = SYS_LIT( 20,   10 ),
    CAP_DIV_22   = SYS_LIT( 22,   11 ),
    CAP_DIV_24   = SYS_LIT( 24,   12 ),
    CAP_DIV_26   = SYS_LIT( 26,   13 ),
    CAP_DIV_28   = SYS_LIT( 28,   14 ),
    CAP_DIV_30   = SYS_LIT( 30,   15 ),
    CAP_DIV_32   = SYS_LIT( 32,   16 ),
    CAP_DIV_34   = SYS_LIT( 34,   17 ),
    CAP_DIV_36   = SYS_LIT( 36,   18 ),
    CAP_DIV_38   = SYS_LIT( 38,   19 ),
    CAP_DIV_40   = SYS_LIT( 40,   20 ),
    CAP_DIV_42   = SYS_LIT( 42,   21 ),
    CAP_DIV_44   = SYS_LIT( 44,   22 ),
    CAP_DIV_46   = SYS_LIT( 46,   23 ),
    CAP_DIV_48   = SYS_LIT( 48,   24 ),
    CAP_DIV_50   = SYS_LIT( 50,   25 ),
    CAP_DIV_52   = SYS_LIT( 52,   26 ),
    CAP_DIV_54   = SYS_LIT( 54,   27 ),
    CAP_DIV_56   = SYS_LIT( 56,   28 ),
    CAP_DIV_58   = SYS_LIT( 58,   29 ),
    CAP_DIV_60   = SYS_LIT( 60,   30 ),
    CAP_DIV_62   = SYS_LIT( 62,   31 )
};
```

18.3.4.1 Description

This is used to set the pre scalar to the capture module.

18.3.5 CAP_CounterReset

```
enum CAP_CounterReset
{
    CAP_CTR_ABS      = 0,     /* Do not reset counter on Capture Event (absolute
time stamp operation) */
    CAP_CTR_DIF      = 1      /* Reset counter after Capture Event time-stamp has
been captured */
};
```

18.3.5.1 Description

This is used to set what happens to the counter when a capture event occurs.

18.3.6 CAP_EventPolarity

```
enum CAP_EventPolarity
{
    CAP_EVENT_POS    = 0,     /* Capture Event triggered on a rising edge */
    CAP_EVENT_NEG    = 1      /* Capture Event triggered on a falling edge */
};
```

18.3.6.1 Description

This is used to set the edge polarity that causes a capture event.

18.3.7 CAP_Mode

```
enum CAP_Mode
{
    CAP_CONTINUOUS   = 0,     /* Operate in continuous mode */
    CAP_ONE_SHOT     = 1      /* Operate in one-Shot mode */
};
```

18.3.7.1 Description

Continuous or one-shot mode control.

18.3.8 CAP_IntMode

```
enum CAP_IntMode
{
    CAP_INT                = (1<<0),    /* Global Interrupt Flag */
    CAP_INT_CEVT1          = (1<<1),    /* Capture Event channel 1 Interrupt Enable
*/
    CAP_INT_CEVT2          = (1<<2),    /* Capture Event channel 2 Interrupt Enable
*/
    CAP_INT_CEVT3          = (1<<3),    /* Capture Event channel 3 Interrupt Enable
*/
    CAP_INT_CEVT4          = (1<<4),    /* Capture Event channel 4 Interrupt Enable
*/
    CAP_INT_CTR_OVF        = (1<<5),    /* Counter Overflow Interrupt Enable */
    CAP_INT_CTR_PRD        = (1<<6),    /* Counter Equal Period Interrupt Enable */
    CAP_INT_CTR_CMP        = (1<<7),    /* Counter Equal Compare Interrupt Enable */
    CAP_INT_ALL            = 0xFF       /* All interrupt mask */
};
```

18.3.8.1 Description

This is used to indicate when the Capture interrupt occurs.

18.3.9 CAP_Pin

```
enum CAP_Pin
{
    CAP_MOD_6_PIN_1        = GPIO_ASSIGN_LIT(6, 1,  2),
    CAP_MOD_5_PIN_3        = GPIO_ASSIGN_LIT(5, 3,  2),
    CAP_MOD_1_PIN_5        = GPIO_ASSIGN_LIT(1, 5,  3),
    CAP_MOD_2_PIN_7        = GPIO_ASSIGN_LIT(2, 7,  3),
    CAP_MOD_3_PIN_9        = GPIO_ASSIGN_LIT(3, 9,  3),
    CAP_MOD_4_PIN_11       = GPIO_ASSIGN_LIT(4, 11, 3),
    CAP_MOD_1_PIN_24       = GPIO_ASSIGN_LIT(1, 24, 1),
    CAP_MOD_2_PIN_25       = GPIO_ASSIGN_LIT(2, 25, 1),
    CAP_MOD_3_PIN_26       = GPIO_ASSIGN_LIT(3, 26, 1),
    CAP_MOD_4_PIN_27       = GPIO_ASSIGN_LIT(4, 27, 1),
    CAP_MOD_1_PIN_34       = GPIO_ASSIGN_LIT(1, 34, 1),
    CAP_MOD_2_PIN_37       = GPIO_ASSIGN_LIT(2, 37, 1),
    CAP_MOD_5_PIN_48       = GPIO_ASSIGN_LIT(5, 48, 1),
    CAP_MOD_6_PIN_49       = GPIO_ASSIGN_LIT(6, 49, 1)
};
```

18.3.9.1 Description

This defines which pins each one the module can use and the required multiplexer.

```
GPIO_ASSIGN_LIT(Module_Index, GPIO_Pin,  GPIO_Mux)
```

19 csl_cntrl_

19.1.1.1 Description

Contains functions to execute a digital 3 pole 3 zero (3p3z) and 2 pole 2 zero (2p2z) algorithm for use in the control of switch mode power supplies (SMPS).

The control structure must be declared and then initialized using CNTRL_3p3zInit()/CNTRL_2p2zInit before the control function is run.

The control function has been optimized in Assembler for maximum speed. In standard C a 3p3z algorithm can take circa 170 cycles (1.7us based on a 10ns system clock).

CNTRL_3p3z()	71	.71us
CNTRL_3p3zInline()	53	.53us
CNTRL_2p2z()	64	.64us
CNTRL_2p2zInline()	44	.44us

The C wrapper contains a small time penalty when compared to pure assembly but it has the advantage that no knowledge of assembly is required.

The values for the 3p3z algorithm must be determined through control theory analysis of the system. The poles and zeros in the analogue frequency domain can be converted to the digital domain using the tool provided on the Biricha Digital Power website <http://www.biricha.com/resources/converter.php?type=4>

The arguments are passed as _iq26 numbers. The limits of these arguments are,

Value	Limit
A0-A2, B0-B3	-32 < value < 31.999999985
REF, MIN, MAX	0 < value < 1

The argument REF is the value that is compared to the feedback value from the system under control. The user code reads the feedback value from the system and stores it within the structure during each cycle of the control loop. The CNTRL_3p3z() function is used to update the output value based on REF and this feedback value.

19.1.1.2 Examples

Initializes the 3p3z structure with the correct coefficients. It then sets the m_IQ feedback value to the IQ value FDBK. The output value is then updated by running the control algorithm. Note that it is also possible to set the feedback value as an integer using the m_Int property of the structure.

```
CNTRL_3p3zInit(&CNTL_3P3Z_1,            // Structure
               REF                      // Ref
               A0,A1,A2                 // a0,a1,a2
               B0,B1,B2,B3              // b0,b1,b2,b3
               K,MIN,MAX                // K, min, max
               );

// Control
CNTL_3P3Z_1.Fdbk.m_IQ = _IQ15(FDBK);    // Set feedback value
CNTRL_3p3z(&CNTL_3P3Z_1 );              // Update
```

19.1.1.3 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf

NOTES
Due to a compiler bug in v5 you must enable "No DP Load optimizations" -md when you use #pragma DATA_ALIGN.

19.2 Api

CNTRL_3p3zInit()
CNTRL_3p3z()
CNTRL_softStartConfig()
CNTRL_softStartUpdate()
CNTRL_softStartDirection()
CNTRL_2p2zInit()
CNTRL_2p2z()
CNTRL_2p2zSoftStartConfig()
CNTRL_2p2zSoftStartUpdate()
CNTRL_2p2zSoftStartDirection()
CNTRL_3p3zFloatInit()
CNTRL_3p3zFloat()

19.2.1 CNTRL_3p3zInit

void CNTRL_3p3zInit(CNTRL_3p3zData* Ptr,_iq15 Ref,_iq26 A1,_iq26 A2,_iq26 A3,_iq26 B0,_iq26 B1,_iq26 B2,_iq26 B3,_iq23 K,_iq15 Min,_iq15 Max)

where:
Ptr -
Ref -
A1 -
A2 -
A3 -
B0 -
B1 -
B2 -
B3 -
K -
Min -
Max -

19.2.1.1 Description

Initializes the 3 pole 3 zero (3p3z) structure with the required coefficients.

A structure, of type CNTRL_3p3zData, must be declared and passed as a reference to this function. This is the location where the function will store the parameters. It will be used later on by the CNTRL_3p3z() function within the control loop.

This structure, the CNTRL_3p3zData* Ptr, must be aligned to 64 words,

```
// Structure is aligned to 64 words
#pragma DATA_ALIGN ( Cntrl3p3z , 64 );
CNTRL_3p3zData Cntrl3p3z;
```

The coefficients A1-A3 and B0-B3 are passed as _iq26 numbers. Therefore the coefficients must be within the range -32 < A < 31.999999985. Where A is the coefficient.

The argument REF is the value that is compared to the feedback value within the structure. This feedback value will most likely come from the ADC, which returns a value between 0 and 0.1249694824 (i.e. a 12 bit value 0xFFF stored as a _iq15).
REF is also stored as a _iq15 value. Therefore it is recommended that the REF argument is set with the desired return value of the ADC.

For example, if a 3.5V output value is required, then using an ADC that has a range of 0 (0V) to 4095 (3.3V) and a 1/2x prescaler (a potential divider) on the input to the ADC pin,

```
_IQ15val = ( (REFval * Prescaler) * (ADCmax / ADCmaxV)
_IQ15val = ( (3.5    * 0.5      ) * (4095   / 3.3    ) ) = 2172
```

Therefore the argument REF can be passed as 2172 or _IQ15(0.06628417969).

The control algorithm will attempt to keep the output of the ADC feedback value around 2172 (out of the 4095 range in this case).

Min and Max are also stored as _IQ15 numbers in a 16 bit value. Therefore their range is also limited to >= 0.0 and < 1.0 and follow the same principle as above.

The parameter K is the scaling factor. It is deteremined using the following equation,

```
K = ( 1 / Prescaler ) * ( ADCmaxV / ADCmax ) * ( PWMperiod )
  = ( 1 /    0.5    ) * (   3.3   /  4095   ) * (    500    )
  = _IQ23(0.80586)
```

Where Prescaler is the potential diviver scaling factor on the input to the ADC pin. PWMperiod is the period of the PWM signal as a number of PWM ticks. This can be obtained from the function PWM_freqToTicks(). The value of K is an _IQ23 number between 0 and 1.

19.2.1.2 Examples

Initializes the CNTL_3P3Z_1 structure with A1..A3, B0..B3, reference, min and max values.

```
#pragma DATA_ALIGN ( CNTL_3P3Z_1 , 64 );
CNTRL_3p3zInit(&CNTL_3P3Z_1
       ,REF                        // Ref
       ,A1,A2,A3                   // a1,a2,a3
       ,B0,B1,B2,B3                // b0,b1,b2,b3
       ,_IQ23(1.0)                 // K
       ,_IQ23(0.0),_IQ23(0.9999)   // min, max
       );
```

NOTES

19.2.2 CNTRL_3p3z

void CNTRL_3p3z(CNTRL_3p3zData* Ptr)

where:
Ptr - Pointer to a 3p3z control struture.

19.2.2.1 Description

Performs the 3 pole 3 zero (3p3z) control algorithm using the information stored within the 3p3z control structure that is passed as a pointer to this function.

The structure should already have been declared and populated with coefficients using the function CNTRL_3p3zInit().

The feedback value from the system being controlled must be read and stored within the 3p3z structure before this function is called.

The result of the control algorithm is also stored within the structure in the Out.m_Int property.

This function is a "C" wrapper around an assembly function. This gives faster execution time without requiring any assembly knowledge.

19.2.2.2 Examples

Reads the feedback value from the ADC, which will be >=0.0 and < 1.0 and calls the 3p3z control algorithm. The ePWM module 1 duty for channel A is updated using the output of the control algorithm.

```
// Control
CNTL_3P3Z_1.Fdbk.m_Int = ADC_getValue(ADC_MOD_1,3);  // Read feedback
CNTRL_3p3z(&CNTL_3P3Z_1 );                           // Run algorithm
PWM_setDutyA(PWM_MOD_1, CNTL_3P3Z_1.Out.m_Int );     // Set new output
```

19.2.3 CNTRL_softStartConfig

void CNTRL_softStartConfig(CNTRL_3p3zData* Ptr,uint32_t RampMs,uint32_t UpdatePeriodNs)

where:
Ptr -
RampMs -
UpdatePeriodNs -

19.2.3.1 Description

```
Configures and enables a soft start for the 3p3z control code.

The 'RampMs' argument is the time in milli-seconds for the reference to reach
its steady state value. The period of execution for the update function,
CNTRL_softStartUpdate(), is specified by the argument 'UpdatePeriodNs'.

After configuring the soft start using this function the soft start is executed
by calling the update function CNTRL_softStartUpdate() at the frequency
determined by the period argument 'UpdatePeriodNs'. The update function should
preferably be called from inside an idle loop.

EXMPLES
This sets the soft start for 2 seconds with an update rate of 200kHz
(T=5000ns).

  CNTRL_softStartConfig( &Cntrl3p3z, 2000, 5000 );

NOTES
```

19.2.4 CNTRL_softStartUpdate

void CNTRL_softStartUpdate(CNTRL_3p3zData* Ptr)

where:
Ptr -

19.2.4.1 Description

Performs an update of the 3p3z reference value according to the soft start parameters set using CNTRL_softStartConfig().

The reference value is updated with a value initially calculated within the function CNTRL_softStartConfig().

This function must be called at the frequency determined by the period argument of the CNTRL_softStartConfig() function call. The update function should be called from within an idle loop.

EXMPLES
Updates the current 3p3z reference with the soft ramp delta value from within the main idle loop. A delay is generated which last for the period specified in the configuration parameters.

```
while ( 1 )
{
   CNTRL_softStartUpdate( &Cntrl3p3z );
   SYS_usDelay( 5 ); // Delay for 5000ns
}
```

NOTES

19.2.5 CNTRL_softStartDirection

void CNTRL_softStartDirection(CNTRL_3p3zData* Ptr,int PowerUp)

where:
Ptr -
PowerUp -

19.2.5.1 Description

```
Converts the soft start to a soft stop or vice versa.

After a soft start has been configured the user may require a soft stop. This
function will reverse the ramp value allowing for the CNTRL_softStartUpdate()
function to generate a soft stop.

The ramp value may be reversed again to generate a soft start. The soft start or
soft stop is determined by the 'PowerUp' argument. If this is true the update
function will generate a soft start. If this is false the update function will
generate a soft stop. This parameter could be read from an
input pin allowing the end user to generate a soft start or soft stop.

EXMPLES
This configures the controller to perform a soft stop.
 CNTRL_softStartDirection( &Cntrl3p3z, false );
```

19.2.6 CNTRL_2p2zInit

void CNTRL_2p2zInit(CNTRL_2p2zData* Ptr,_iq15 Ref,_iq26 A1,_iq26 A2,_iq26 B0,_iq26 B1,_iq26 B2,_iq23 K,_iq15 Min,_iq15 Max)

where:
Ptr -
Ref -
A1 -
A2 -
B0 -
B1 -
B2 -
K -
Min -
Max -

19.2.6.1 Description

Initializes the 2 pole 2 zero (2p2z) structure with the required coefficients.

A structure, of type CNTRL_2p2zData, must be declared and passed as a reference to this function. This is the location where the function will store the parameters. It will be used later on by the CNTRL_2p2z() function within the control loop.

This structure, the CNTRL_2p2zData* Ptr, must be aligned to 64 words,

```
// Structure is aligned to 64 words
#pragma DATA_ALIGN ( Cntrl2p2z , 64 );
CNTRL_2p2zData Cntrl2p2z;
```

The coefficients A1-A2 and B0-B3 are passed as _iq26 numbers. Therefore the coefficients must be within the range -32 < A < 31.999999985. Where A is the coefficient.

The argument REF is the value that is compared to the feedback value within the structure. This feedback value will most likely come from the ADC, which returns a value between 0 and 0.1249694824 (i.e. a 12 bit value 0xFFF stored as a _iq15).
REF is also stored as a _iq15 value. Therefore it is recommended that the REF argument is set with the desired return value of the ADC.

For example, if a 3.5V output value is required, then using an ADC that has a range of 0 (0V) to 4095 (3.3V) and a 1/2x prescaler (a potential divider) on the input to the ADC pin,

```
_IQ15val = ( (REFval * Prescaler) * (ADCmax / ADCmaxV)
_IQ15val = ( (3.5    * 0.5      ) * (4095   / 3.3     ) ) = 2172
```

Therefore the argument REF can be passed as 2172 or _IQ15(0.06628417969).

The control algorithm will attempt to keep the output of the ADC feedback value around 2172 (out of the 4095 range in this case).

Min and Max are also stored as _IQ15 numbers in a 16 bit value. Therefore their range is also limited to >= 0.0 and < 1.0 and follow the same principle as above.

The parameter K is the scaling factor. It is deteremined using the following equation,

```
K = ( 1 / Prescaler ) * ( ADCmaxV / ADCmax ) * ( PWMperiod )
  = ( 1 /    0.5    ) * (   3.3   /  4095   ) * (    500    )
  = _IQ23(0.80586)
```

Where Prescaler is the potential diviver scaling factor on the input to the ADC pin. PWMperiod is the period of the PWM signal as a number of PWM ticks. This can be obtained from the function PWM_freqToTicks(). The value of K is an _IQ23 number between 0 and 1.

19.2.6.2 Examples

Initializes the CNTL_2P2Z_1 structure with A1..A3, B0..B3, reference, min and max values.

```
#pragma DATA_ALIGN ( CNTL_2P2Z_1 , 64 );
CNTRL_2p2zInit(&CNTL_2P2Z_1
      ,REF                          // Ref
      ,A2,A3                        // a1,a2
      ,B0,B1,B2                     // b0,b1,b2
      ,_IQ23(1.0)                   // K
      ,_IQ23(0.0),_IQ23(0.9999)     // min, max
      );
```

NOTES

19.2.7 CNTRL_2p2z

void CNTRL_2p2z(CNTRL_2p2zData* Ptr)

where:
Ptr - Pointer to a 2p2z control struture.

19.2.7.1 Description

Performs the 2 pole 2 zero (2p2z) control algorithm using the information stored within the 2p2z control structure that is passed as a pointer to this function.

The structure should already have been declared and populated with coefficients using the function CNTRL_2p2zInit().

The feedback value from the system being controlled must be read and stored within the 2p2z structure before this function is called.

The result of the control algorithm is also stored within the structure in the Out.m_Int property.

This function is a "C" wrapper around an assembly function. This gives faster execution time without requiring any assembly knowledge.

19.2.7.2 Examples

Reads the feedback value from the ADC, which will be >=0.0 and < 1.0 and calls the 3p3z control algorithm. The ePWM module 1 duty for channel A is updated using the output of the control algorithm.

```
// Control
CNTL_2P2Z_1.Fdbk.m_Int = ADC_getValue(ADC_MOD_1,3);  // Read feedback
CNTRL_2p2z(&CNTL_2P2Z_1 );                            // Run algorithm
PWM_setDutyA(PWM_MOD_1, CNTL_2P2Z_1.Out.m_Int );     // Set new output
```

19.2.8 CNTRL_2p2zSoftStartConfig

void CNTRL_2p2zSoftStartConfig(CNTRL_2p2zData* Ptr,uint32_t RampMs,uint32_t UpdatePeriodNs)

where:
Ptr -
RampMs -
UpdatePeriodNs -

19.2.8.1 Description

```
Configures and enables a soft start for the 2p2z control code.

The 'RampMs' argument is the time in milli-seconds for the reference to reach
its steady state value. The period of execution for the update function,
CNTRL_2p2zSoftStartUpdate(), is specified by the argument 'UpdatePeriodNs'.

After configuring the soft start using this function the soft start is executed
by calling the update function CNTRL_2p2zSoftStartUpdate() at the frequency
determined by the period argument 'UpdatePeriodNs'. The update function should
preferably be called from inside an idle loop.

EXMPLES
This sets the soft start for 2 seconds with an update rate of 200kHz
(T=5000ns).

  CNTRL_2p2zSoftStartConfig( &Cntrl2p2z, 2000, 5000 );

NOTES
```

19.2.9 CNTRL_2p2zSoftStartUpdate

void CNTRL_2p2zSoftStartUpdate(CNTRL_2p2zData* Ptr)

where:
Ptr -

19.2.9.1 Description

Performs an update of the 2p2z reference value according to the soft start parameters set using CNTRL_2p2zSoftStartConfig().

The reference value is updated with a value initially calculated within the function CNTRL_2p2zSoftStartConfig().

This function must be called at the frequency determined by the period argument of the CNTRL_2p2zSoftStartConfig() function call. The update function should be called from within an idle loop.

EXMPLES
Updates the current 2p2z reference with the soft ramp delta value from within the main idle loop. A delay is generated which last for the period specified in the configuration parameters.

```
while ( 1 )
{
   CNTRL_softStartUpdate( &Cntrl2p2z );
   SYS_usDelay( 5 ); // Delay for 5000ns
}
```

NOTES

19.2.10 CNTRL_2p2zSoftStartDirection

void CNTRL_2p2zSoftStartDirection(CNTRL_2p2zData* Ptr,int PowerUp)

where:
Ptr -
PowerUp -

19.2.10.1 Description

Converts the soft start to a soft stop or vice versa.

After a soft start has been configured the user may require a soft stop. This function will reverse the ramp value allowing for the CNTRL_2p2zSoftStartUpdate() function to generate a soft stop.

The ramp value may be reversed again to generate a soft start. The soft start or soft stop is determined by the 'PowerUp' argument. If this is true the update function will generate a soft start. If this is false the update function will generate a soft stop. This parameter could be read from an input pin allowing the end user to generate a soft start or soft stop.

EXMPLES
This configures the controller to perform a soft stop.

```
CNTRL_2p2zSoftStartDirection( &Cntrl2p2z, false );
```

19.2.11 CNTRL_3p3zFloatInit

void CNTRL_3p3zFloatInit(CNTRL_3p3zDataFloat* Ptr,uint16_t Ref,float a1,float a2,float a3,float b0,float b1,float b2,float b3,float k,uint16_t Min,uint16_t Max)

where:
Ptr -
Ref -
a1 -
a2 -
a3 -
b0 -
b1 -
b2 -
b3 -
k -
Min -
Max -

19.2.11.1 Description

```
Initializes the 3 pole 3 zero (3p3z) structure with the required
coefficients.

A structure, of type CNTRL_3p3zDataFloat, must be declared and passed as a
reference to this function. This is the location where the function will
store the parameters. It will be used later on by the CNTRL_3p3zFloat() function
within the control loop.

The coefficients A1-A3 and B0-B3 are passed as floats numbers.

The argument REF is stored as a uint16_t value that is compared to the feedback
value within the structure. This feedback value will most likely
come from the ADC, which returns a value between 0 and 0xFFF (i.e. a 12 bit
value).

REF is also stored as a uint16_t value. Therefore it is recommended that the REF
argument is set with the desired return value of the ADC.

For example, if a 3.5V output value is required, then using an ADC that has a
range of 0 (0V) to 4095 (3.3V) and a 1/2x prescaler (a potential divider)
on the input to the ADC pin,

Ref = ( (REFval * Prescaler) * (ADCmax / ADCmaxV)
Ref = ( (3.5    * 0.5      ) * (4095   / 3.3    ) ) = 2172

Therefore the argument REF can be passed as 2172.

The control algorithm will attempt to keep the output of the ADC feedback value
around 2172 (out of the 4095 range in this case).

Min and Max are also stored as uint16_t. Therefore
their range is also limited to 0 and 0xFFFF and follow the same principle
as above.

The parameter K is the scaling factor. It is deteremined using the following
equation,
```

```
K = ( 1 / Prescaler ) * ( ADCmaxV / ADCmax ) * ( PWMperiod )
  = ( 1 /    0.5    ) * (   3.3   /  4095   ) * (    500    )
  = 0.80586
```

Where Prescaler is the potential diviver scaling factor on the input to the ADC pin. PWMperiod is the period of the PWM signal as a number of PWM ticks. This can be obtained from the function PWM_freqToTicks(). The value of K is float.

19.2.11.2 Examples

Initializes the CNTL_3P3Z_f structure with A1..A3, B0..B3, reference, min and max values.

```
  CNTRL_3p3zFloatInit(&CNTL_3P3Z_f
        ,REF                          // Ref
        ,A1,A2,A3                     // a1,a2,a3
        ,B0,B1,B2,B3                  // b0,b1,b2,b3
        ,1.0                          // K
        ,0, 500                       // min, max
        );
```

19.2.12 CNTRL_3p3zFloat

void CNTRL_3p3zFloat(CNTRL_3p3zDataFloat* Ptr)

where:
Ptr -

19.2.12.1 Description

```
Performs the 3 pole 3 zero (3p3z) control algorithm using the information stored
within the 3p3z control structure that is passed as a pointer to this function.

The structure should already have been declared and populated with coefficients
using the function CNTRL_3p3zFloatInit().

The feedback value from the system being controlled must be read and stored
within the 3p3z structure before this function is called.

The result of the control algorithm is also stored within the structure
in the m_U[0] property.
```

19.2.12.2 Examples

```
Reads the feedback value from the ADC, which will be 0 and 0xFFF and
calls the 3p3z control algorithm. The ePWM module 1 duty for channel A is
updated using the output of the control algorithm.
```

```
// Control
CNTL_3P3Z_f.m_Ref = ADC_getValue(ADC_MOD_1,3);  // Read feedback
CNTRL_3p3z(&CNTL_3P3Z_f );                      // Run algorithm
PWM_setDutyA(PWM_MOD_1, CNTL_3P3Z_f.Out );      // Set new output
```

19.3 Types

19.3.1 CNTRL_ARG

```
union CNTRL_ARG
{
	_iq15 m_IQ;
	int   m_Int;
};
```

19.3.1.1 Description

Allows a int to be written directly in to a _iq varaible.

19.3.2 CNTRL_3p3zData

```
struct CNTRL_3p3zData
{
	CNTRL_ARG   Ref;  /* +0 This is a range of +1 */
	CNTRL_ARG   Fdbk; /* +2 This is a range of +1 */
	CNTRL_ARG   Out;  /* +4 This is a range of +1 */
	long  temp; /* +6 */
	_iq24 m_U1; /* +8 */
	_iq24 m_U2; /* +10 */
	_iq24 m_U3; /* +12 */
	_iq31 m_E0; /* +14 */
	_iq31 m_E1; /* +16 */
	_iq31 m_E2; /* +18 */
	_iq31 m_E3; /* +20 */
	_iq26 m_B3; /* +22 */
	_iq26 m_B2; /* +24 */
	_iq26 m_B1; /* +26 */
	_iq26 m_B0; /* +28 */
	_iq26 m_A3; /* +30 */
	_iq26 m_A2; /* +32 */
	_iq26 m_A1; /* +34 */
	_iq23 m_K;  /* +36 */
	_iq15 m_max;      /* +38 */
	_iq15 m_min;      /* +40 */
	int   m_PeriodCount;
	long  m_SoftRamp;
	long  m_SoftRef;
	long  m_SoftMax;
};
```

19.3.2.1 Description

This is the 3 pole 3 zero control structure.

19.3.3 UNKNOWN_B23853E2

```
#if 1
#define CNTRL_inlineContextSave() \
asm("        PUSH    XAR7"\
    "\t\n    PUSH    XT"\
```

```
    "\t\n    PUSH    ACC"\
    )
#endif
```

19.3.3.1 Description

Stores the registers used by the 3p2z/2p2z inline function.

19.3.4 UNKNOWN_AFF11ED4

```
#if 1
#define CNTRL_inlineContextRestore() \
asm("         POP     ACC"\
    "\t\n     POP     XT"\
    "\t\n     POP     XAR7"\
    )
#endif
```

19.3.4.1 Description

Restores the registers used by the 3p2z/2p2z inline function.

19.3.5 CNTRL_3p3zInline

```
#if 1
#define CNTRL_3p3zInline(x) \
asm("         MOVW    DP, #_"#x"+0          ;CNTRL_3p3z"\
    "\t\n     MOVL    XAR7,#_"#x"+22        ;(COEFF) Local coefficient pointer
(XAR7)"\
\
    "\t\n     SETC    SXM,OVM"\
    "\t\n     MOV     ACC,@0                ;(Ref)Q15"\
    "\t\n     SUB     ACC,@2                ;(Fdbk)Q15"\
    "\t\n     LSL     ACC,#16               ;Q31"\
\
    "\t\n     ; Diff equation"\
    "\t\n     MOVL    @8+6,ACC              ;(DBUFF+6)"\
    "\t\n     MOVL    XT,@8+12              ;(DBUFF+12) XT=e(n-3),Q31"\
    "\t\n     QMPYL   ACC,XT,*XAR7++        ; b3*e(n-3),Q26*Q31(64-bit result)"\
\
    "\t\n     MOVDL   XT,@8+10              ;(DBUFF+10) XT=e(n-2), e(n-3)=e(n-2)"\
    "\t\n     QMPYL   P,XT,*XAR7++          ; ACC=b3*e(n-3)+b2*e(n-2) P=b1*e(n-
1),Q26*Q31(64-bit result)"\
    "\t\n     ADDL    ACC,P                 ; 64-bit result in Q57, So ACC is in
Q25"\
\
    "\t\n     MOVDL   XT,@8+8               ;(DBUFF+10) XT=e(n-1), e(n-2)=e(n-1)"\
    "\t\n     QMPYL   P,XT,*XAR7++          ; ACC=b2*e(n-2) P=b1*e(n-1),Q26*Q31(64-
bit result)"\
    "\t\n     ADDL    ACC,P                 ; 64-bit result in Q57, So ACC is in
Q25"\
\
    "\t\n     MOVDL   XT,@8+6               ;(DBUFF+6) XT=e(n), e(n-1)=e(n)"\
    "\t\n     QMPYL   P,XT,*XAR7++          ; ACC=b3*e(n-3)+b2*e(n-2)+b1*e(n-1),
P=b0*e(n),Q26*Q31(64-bit result)"\
```

```
    "\t\n                                   ; 64-bit result in Q57, So ACC is in
Q25"\
    "\t\n    ADDL    ACC,P                  ; ACC=b3*e(n-3)+b2*e(n-2)+b1*e(n-
1)+b0*e(n), Q25"\
    "\t\n    SFR     ACC,#1"\
    "\t\n    MOVL    @6,ACC                 ;(temp) Q24"\
\
    "\t\n    MOVL    XT,@8+4                ;(DBUFF+4) XT=u(n-3),Q24"\
    "\t\n    QMPYL   P,XT,*XAR7++           ; P=a3*u(n-3), Q26*Q24(64-bit result)"\
\
    "\t\n    MOVDL   XT,@8+2                ;(DBUFF+2) XT=u(n-2), u(n-3)=u(n-
2),Q24"\
    "\t\n    QMPYL   ACC,XT,*XAR7++         ; ACC=a2*u(n-2)"\
    "\t\n                                   ; 64-bit result in Q50, So ACC is in
Q18"\
    "\t\n    ADDL    ACC,P                  ; ACC=a1*u(n-1)+a2*u(n-2)+a3*u(n-3),ACC
in Q18"\
\
    "\t\n    MOVDL   XT,@8+0                ;(DBUFF+0) XT=u(n-1), u(n-2)=u(n-
1),Q24"\
    "\t\n    QMPYL   P,XT,*XAR7++           ; P=a2*u(n-2)"\
    "\t\n                                   ; 64-bit result in Q50, So ACC is in
Q18"\
    "\t\n    ADDL    ACC,P                  ; ACC=a1*u(n-1)+a2*u(n-2)+a3*u(n-3),ACC
in Q18"\
\
    "\t\n    LSL     ACC,#5                 ; Q23"\
    "\t\n    ADDL    ACC,ACC                ; Q24"\
    "\t\n    ADDL    ACC,@6                 ;(temp) Q24,ACC=a1*u(n-1)+a2*u(n-
2)+b2*e(n-2)+b1*e(n-1)+b0*e(n)"\
    "\t\n    MOVL    @8+0,ACC               ; (DBUFF+0) ACC=u(n)(Q24)"\
\
    "\t\n    MOVL    XT,ACC                 ; XT = ACC iq24"\
    "\t\n    QMPYL   ACC,XT,*XAR7++         ; ACC = XT * K(23) >> 32 => iq15"\
\
    "\t\n    MINL    ACC,*XAR7++            ; Saturate the result [0,1]"\
    "\t\n    MAXL    ACC,*XAR7++"\
\
    "\t\n    MOV     @4, AL ;(Out)")

/*end of code macro*/
#endif
```

19.3.5.1 Description

Performs the 3 pole 3 zero (3p3z) control algorithm using the information stored within the 2p2z control structure that is passed as a structure to this function.

The structure should already have been declared and populated with coefficients using the function CNTRL_3p3zInit().

The feedback value from the system being controlled must be read and stored within the 3p3z structure before this function is called.

The result of the control algorithm is also stored within the structure in the Out.m_Int property.

This function is inline assembly code and it is the responsability of the user to make sure the "C" context is saved between calls.

There are CNTRL_inlineContextSave() and CNTRL_inlineContextRestore() which saves/restores the required context.

EXAMPLES
Reads the feedback value from the ADC, which will be >=0.0 and < 1.0 and calls the 3p3z control algorithm. The ePWM module 1 duty for channel A is updated using the output of the control algorithm.

```
// Control
CNTL_3P3Z_1.Fdbk.m_Int = ADC_getValue(ADC_MOD_1,3);  // Read feedback
CNTRL_inlineContextSave();
CNTRL_3p3zInline(CNTL_3P3Z_1 );                      // Run algorithm
CNTRL_inlineContextRestore();
PWM_setDutyA(PWM_MOD_1, CNTL_3P3Z_1.Out.m_Int );     // Set new output
```

19.3.6 CNTRL_2p2zData

```
struct CNTRL_2p2zData
{
      CNTRL_ARG   Ref;  /* +0 This is a range of +1 */
      CNTRL_ARG   Fdbk; /* +2 This is a range of +1 */
      CNTRL_ARG   Out;  /* +4 This is a range of +1 */
      long  temp; /* +6 */
      _iq24 m_U1; /* +8 */
      _iq24 m_U2; /* +10 */
      _iq31 m_E0; /* +12 */
      _iq31 m_E1; /* +14 */
      _iq31 m_E2; /* +16 */
      _iq26 m_B2; /* +18 */
      _iq26 m_B1; /* +20 */
      _iq26 m_B0; /* +22 */
      _iq26 m_A2; /* +24 */
      _iq26 m_A1; /* +26 */
      _iq23 m_K;  /* +28 */
      _iq15 m_max;      /* +30 */
      _iq15 m_min;      /* +32 */
      int   m_PeriodCount;
      long  m_SoftRamp;
      long  m_SoftRef;
      long  m_SoftMax;
};
```

19.3.6.1 Description

This is the 2 pole 2 zero control structure.

19.3.7 CNTRL_2p2zInline

```
#if 1
#define CNTRL_2p2zInline(x) \
asm("        MOVW    DP, #_"#x"+0        ;CNTRL_2p2z"\
    "\t\n    MOVL    XAR7,#_"#x"+18      ;(COEFF) Local coefficient pointer
(XAR7)"\
\
```

```
    "\t\n    SETC    SXM,OVM"\
    "\t\n    MOV     ACC,@0                  ;(Ref)Q15"\
    "\t\n    SUB     ACC,@2                  ;(Fdbk)Q15"\
    "\t\n    LSL     ACC,#16                 ;Q31"\
\
    "\t\n    ; Diff equation"\
    "\t\n    MOVL    @8+4,ACC                ;(DBUFF+4)"\
    "\t\n    MOVL    XT,@8+8                 ;(DBUFF+8) XT=e(n-2),Q31"\
    "\t\n    QMPYL   ACC,XT,*XAR7++          ; b2*e(n-2),Q26*Q31(64-bit result)"\
\
    "\t\n    MOVDL   XT,@8+6                 ;(DBUFF+6) XT=e(n-1), e(n-2)=e(n-1)"\
    "\t\n    QMPYL   P,XT,*XAR7++            ; ACC=b3*e(n-3)+b2*e(n-2) P=b1*e(n-
1),Q26*Q31(64-bit result)"\
    "\t\n    ADDL    ACC,P                   ; 64-bit result in Q57, So ACC is in
Q25"\
\
    "\t\n    MOVDL   XT,@8+4                 ;(DBUFF+10) XT=e(n-0), e(n-1)=e(n-0)"\
    "\t\n    QMPYL   P,XT,*XAR7++            ; ACC=b2*e(n-2) P=b1*e(n-1),Q26*Q31(64-
bit result)"\
\
    "\t\n    ADDL    ACC,P                   ; ACC=b2*e(n-2)+b1*e(n-1)+b0*e(n-0),
Q25"\
    "\t\n    SFR     ACC,#1"\
    "\t\n    MOVL    @6,ACC                  ;(temp) Q24"\
\
    "\t\n    MOVL    XT,@8+2                 ;(DBUFF+2) XT=u(n-2),Q24"\
    "\t\n    QMPYL   ACC,XT,*XAR7++          ; ACC=a2*u(n-2), Q26*Q24(64-bit
result)"\
\
    "\t\n    MOVDL   XT,@8+0                 ;(DBUFF+0) XT=u(n-1), u(n-2)=u(n-
1),Q24"\
    "\t\n    QMPYL   P,XT,*XAR7++            ; P=a1*u(n-1)"\
    "\t\n                                    ; 64-bit result in Q50, So ACC is in
Q18"\
    "\t\n    ADDL    ACC,P                   ; ACC=a1*u(n-1)+a2*u(n-2),ACC in Q18"\
\
    "\t\n    LSL     ACC,#5                  ; Q23"\
    "\t\n    ADDL    ACC,ACC                 ; Q24"\
    "\t\n    ADDL    ACC,@6                  ;(temp) Q24,ACC=a1*u(n-1)+a2*u(n-
2)+b2*e(n-2)+b1*e(n-1)+b0*e(n)"\
    "\t\n    MOVL    @8+0,ACC                ; (DBUFF+0) ACC=u(n)(Q24)"\
\
    "\t\n    MOVL    XT,ACC                  ; XT = ACC iq24"\
    "\t\n    QMPYL   ACC,XT,*XAR7++          ; ACC = XT * K(23) >> 32 => iq15"\
\
    "\t\n    MINL    ACC,*XAR7++             ; Saturate the result [0,1]"\
    "\t\n    MAXL    ACC,*XAR7++"\
\
    "\t\n    MOV     @4, AL ;(Out)")

/*end of code macro*/
#endif
```

19.3.7.1 Description

Performs the 2 pole 2 zero (2p2z) control algorithm using the information stored within the 2p2z control structure that is passed as a structure to this function.

The structure should already have been declared and populated with coefficients using the function CNTRL_2p2zInit().

The feedback value from the system being controlled must be read and stored within the 2p2z structure before this function is called.

The result of the control algorithm is also stored within the structure in the Out.m_Int property.

This function is inline assembly code and it is the responsability of the user to make sure the "C" context is saved between calls.

There are CNTRL_inlineContextSave() and CNTRL_inlineContextRestore() which saves/restores the required context.

EXAMPLES
Reads the feedback value from the ADC, which will be >=0.0 and < 1.0 and calls the 2p2z control algorithm. The ePWM module 1 duty for channel A is updated using the output of the control algorithm.

```
   // Control
   CNTL_2P2Z_1.Fdbk.m_Int = ADC_getValue(ADC_MOD_1,3);  // Read feedback
   CNTRL_inlineContextSave();
   CNTRL_2p2zInline(CNTL_2P2Z_1 );                      // Run algorithm
   CNTRL_inlineContextRestore();
   PWM_setDutyA(PWM_MOD_1, CNTL_2P2Z_1.Out.m_Int );     // Set new output
```

19.3.8 CNTRL_3p3zDataFloat

```
struct CNTRL_3p3zDataFloat
{
      uint16_t    m_Ref;
      uint16_t    m_Fdbk;
      float m_A1;
      float m_A2;
      float m_A3;
      float m_B0;
      float m_B1;
      float m_B2;
      float m_B3;
      float m_E[4];
      float m_U[4];
      float m_K;
      uint16_t    m_Out;
      uint16_t    m_Min;
      uint16_t    m_Max;
};
```

19.3.8.1 Description

20 csl_cla_t0_

20.1.1.1 Description

Contains functions to execute a digital 3 pole 3 zero (3p3z) and 2 pole 2 zero (2p2z) algorithm for use in the control of switch mode power supplies (SMPS) using the CLA.

The CLA code must be declared using CLA_3p3zVMode() or CLA_2p2zVMode() and then initialized using CLA_config().

The control functions have been optimized in Assembler for maximum speed.

```
        type      instructions  us(60Mhz)
   CLA_3p3zVMode()      46        .780us
   CLA_2p2zVMode()      39        .650us
```

The controllers have sufficient information in the current time step to pre-calculate some of the result for the following time step. This pre-calculation is performed after the duty has been updated. Therefore,
due to the pre-calculation, the controller can calculate the current output and then update the duty within 290ns.

This means that, when using early ADC interrupts, the ADC can be sampled and duty updated within 940ns. However, if shadow registers are turned on then the duty will not take effect until the next PWM period.

```
              (see above)
   880ns     -------------
-------------|           |--------------
             --->290ns PWM update (1170ns from edge to update)

             (see above)
   650ns   -------------   early ADC
-----------|           |--------------
           --->290ns PWM update (940ns from edge to update)
```

The parameters for the 3p3z algorithm must be determined through control theory analysis of the system. The poles and zeros in the analogue frequency domain can be converted to the digital domain using the tool provided on the
Biricha
Digital Power website <http://www.biricha.com/resources/converter.php?type=4>

Arguments are passed to the CLA_3p3zVMode() and CLA_2p2zVMode() functions as float numbers. Macros, constants or variables cannot be used.

In the function CLA_setRef(), the argument REF is compared to the feedback value from the system under control. The CLA code reads the feedback value from the ADC and stores it within the structure during each cycle of the control loop.
The CLA code is used to update the output value based on REF
and this feedback value.

20.1.1.2 Examples

Initializes the 3p3z structure with the correct coefficients. When ADC_MOD_7 has completed a conversion the CLA code begins execution. This reads the value from the ADC result register. The duty is calculated and then PWM_MOD_3 duty is updated all within the CLA code.

```
  CLA_3p3zVMode( ClaTask, 7, 3,
            +1.46818, -0.314933, -0.153248,
             1.784224053, -1.629063952, -1.780916725, 1.632371281,
             0.48, 0.0, 240.0 );

  void main ( void )
  {
    SYS_init();
    ADC_init();

    PWM_config( PWM_MOD_3, PWM_freqToTicks(200000), PWM_COUNT_DOWN );
    PWM_pin( PWM_MOD_3, PWM_CH_A, GPIO_NON_INVERT );
    PWM_setAdcSoc( PWM_MOD_3, PWM_CH_A, PWM_INT_ZERO );

    ADC_setEarlyInterrupt( 1 );
    ADC_config( ADC_MOD_1, ADC_SH_WIDTH_7, ADC_CH_A0, ADC_TRIG_EPWM3_SOCA );
    ADC_setCallback( ADC_MOD_1, 0, ADC_INT_7 );

    CLA_setRef( CLA_getCtrlPtr(ClaTask), 2048 );
    CLA_config( CLA_MOD_7, &ClaTask, CLA_INT_ADC );
    CLA_setCallback( CLA_MOD_7, IsrFunc );

    INT_enableGlobal( 1 );

    while( 1 )
    {
    }
```

20.1.1.3 Links

file:///C:/tidcs/c28/CSL_C2803x/v100/doc/CSL_C2803x.pdf
http://focus.ti.com/lit/ug/spruge6a/spruge6a.pdf

NOTES
At power up all of the CLA to CPU message RAM is set to zero and CLA task 8 is pre-configured for use with CLA_memSet().

20.2 Api

CLA_3p3zVMode()
CLA_getCtrlPtr()
CLA_2p2zVMode()
CLA_slopeCode()
CLA_getVectorPtr()
CLA_2p2zIMode()
CLA_setCallback()
CLA_softwareStart()
CLA_isRunning()
CLA_softwareStartWait()
CLA_config()
CLA_getPieId()
CLA_ackInt()
CLA_softStartConfig()
CLA_softStartUpdate()
CLA_softStartDirection()
CLA_setRef()
CLA_memSet()
CLA_3p3zIMode()

20.2.1 CLA_3p3zVMode

void CLA_3p3zVMode(void Name,void Adc,void Pwm,void A1,void A2,void A3,void B0,void B1,void B2,void B3,void K,void MiN,void MaX)

where:
Name -
Adc - ADC module number.
Pwm - PWM module number.
A1 -
A2 -
A3 -
B0 -
B1 -
B2 -
B3 -
K -
MiN - Minimum number of ticks that the duty can be set to.
MaX - Maximum number of ticks that the duty can be set to.

20.2.1.1 Description

This macro must be called at the top of the C file, before the main function begins.

The values passed to the function call must be literals. Constants, variables or macros cannot be used.

The function creates the CLA code for a 3p3z controller.

20.2.1.2 Examples

Creates the CLA function called ClaTask. This reads the ADC value from ADC_MOD_7 and writes the duty to PWM_MOD_3.

```
CLA_3p3zVMode( ClaTask, 7, 3,
          +1.46818, -0.314933, -0.153248,
           1.784224053, -1.629063952, -1.780916725, 1.632371281,
           0.48, 0, 240 );
```

20.2.2 CLA_getCtrlPtr

void CLA_getCtrlPtr(void Mod)

where:
Mod - Selects the CLA module.

20.2.2.1 Description

```
Returns a pointer to the CLA module controller structure that holds the
reference value.
```

20.2.3 CLA_2p2zVMode

void CLA_2p2zVMode(void Name,void Adc,void Pwm,void A1,void A2,void B0,void B1,void B2,void K,void MiN,void MaX)

where:
Name -
Adc -
Pwm -
A1 -
A2 -
B0 -
B1 -
B2 -
K -
MiN - Minimum number of ticks that the duty can be set to.
MaX - Maximum number of ticks that the duty can be set to.

20.2.3.1 Description

This macro must be called at the top of the C file, before the main function begins.

The values passed to the function call must be literals. Constants, variables or macros cannot be used.

The function creates the CLA code for a 2p2z controller.

20.2.3.2 Examples

Creates the CLA function called ClaTask. This reads the ADC value from ADC_MOD_7 and writes the duty to PWM_MOD_3.

```
  CLA_2p2zVMode( ClaTask, 7, 3,
             +1.46818, -0.314933,
              1.784224053, -1.629063952, -1.780916725
              0.48, 0, 240 );
```

20.2.4 CLA_slopeCode

void CLA_slopeCode(void Name,int Comp,int Pwm,float Delta,void Steps)

where:
Name - Name of CLA code.
Comp - CMP_MOD number 1..3
Pwm - PWM_MOD number 1..6
Delta - The delta added to the CMP_MOD DAC value
Steps - The number of time Delta is added to the DAC value.

20.2.4.1 Description

This macro must be called at the top of the C file, before the main function begins.

The values passed to the function call must be literals. Constants, variables or macros cannot be used.

This function creates the CLA code to adjust the CMP_MOD DAC value.

The current DAC value is adjusted by Delta every 50ns. This means that the new DAC value must be set before the CLA slope code is executed.

The DAC is adjusted after 364ns from the PWM interrupt.
The DAC value must be valid before 280ns after the interrupt value where it is read by the CLA code.
The CLA code also clears the PWM interrupt.

Each instruction takes 16.666ns to execute assuming a 60MHz system clock.
Three instructions are used to decrement the DAC register by the value Delta.
Therefore each decrement by Delta will occur at fixed intervals of 50ns (16.666ns*3).

The number of decrements that occur during each execution of the CLA task is determined by the argument Steps. The CLA task begins executing 280ns after the interrupt for the PWM module specified occurs.

Therefore the DAC value must be valid no greater than 280ns after the interrupt as it is read in to the CLA task code. Similarly, the CLA code must finish executing before the new DAC value is set by the control function.
Otherwise the DAC value will be overwritten by the CLA slope task value.

The designer must ensure that the CLA_slopeCode function finishes before the new DAC value is written and that the Delta value is not too large such that the DAC value wraps around from zero by the end of the number of steps.

```
            16.666ns*3*Steps
364ns   _______________________
_______|                       |__________


-------
       ----
           ----  16.666ns*3
 DAC           ----
 value             ----
                       ---
                          ----------------
```

20.2.4.2 Examples

This creates the CLA function called SlopeTask. When PWM_MOD_3 generates an interrupt the CLA code is run where it decrements the CMP_MOD_1 DAC value by 1 every 50ns for 6 cycles.

```
CLA_slopeCode( SlopeTask, 1,3, -1.0, 6 );

//in the main code
//set up the comp and set the DAC value
CMP_config( CMP_MOD_1, CMP_SAMPLE_1, GPIO_NON_INVERT, CMP_DAC );
CMP_pin( CMP_MOD_1);
CMP_setDac( CMP_MOD_1, 100 );

//configure the CLA  to run after a PWM interrupt occurs
CLA_config( CLA_MOD_3, &SlopeTask, CLA_INT_PWM );

//configure the PWM
PWM_config( PWM_MOD_3, PWM_freqToTicks(200000), PWM_COUNT_DOWN );
PWM_pin( PWM_MOD_3, PWM_CH_A, GPIO_NON_INVERT );
PWM_setCallback(PWM_MOD_3, 0, PWM_INT_ZERO, PWM_INT_PRD_1 );
```

20.2.5 CLA_getVectorPtr

volatile Uint16* CLA_getVectorPtr(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.5.1 Description

Returns the CLA vector address for the CLA task. This contains the location of code to run when the CLA task is made active.

20.2.5.2 Examples

Gets the vector address for CLA_MOD_1

```
  CLA_getVectorPtr( CLA_MOD_1 );
```

20.2.6 CLA_2p2zIMode

void CLA_2p2zIMode(void Name,void Adc,void Cmp,void A1,void A2,void B0,void B1,void B2,void K,void MiN,void MaX)

where:
Name -
Adc -
Cmp -
A1 -
A2 -
B0 -
B1 -
B2 -
K -
MiN - Minimum number of ticks that the duty can be set to.
MaX - Maximum number of ticks that the duty can be set to.

20.2.6.1 Description

This macro must be called at the top of the C file, before the main function begins.

The values passed to the function call must be literals. Constants, variables or macros cannot be used.

The function creates the CLA code for a 2p2z current mode controller.

20.2.6.2 Examples

Creates the CLA function called ClaTask. This reads the ADC value from ADC_MOD_7 and writes the value to CMP_MOD_3.

```
  CLA_2p2zIMode( ClaTask, 7, 3,
            +1.46818, -0.314933,
             1.784224053, -1.629063952, -1.780916725
             0.48, 0, 240 );
```

20.2.7 CLA_setCallback

void CLA_setCallback(CLA_Module Mod,INT_IsrAddr Func)

where:
Mod - Selects the CLA module.
Func - The pointer to the interrupt function.

20.2.7.1 Description

When the CLA task has finished a CPU interrupt can be generated.

This function assigns an interrupt service routine (ISR), to the interrupt vector of the CLA.

The ISR assigned to the interrupt vector must be qualified with the interrupt keyword and must not return a value due to the nature of the interrupt function call and return sequence.

The ISR must have a function prototype that is visible to the CLA_setCallback() function as the address of the ISR is used in the function call.

The ISR could be used to examine the results of the CLA task. However, in most situations this is not necessary.

PIE controller interrupts are enabled automatically by this function for the specified ePWM module.

However, no interrupt functions will be called until the global interrupt switch is enabled. Global interrupts can be enabled by calling the INT_enableGlobal() function.

20.2.7.2 Examples

The IsrFunc() is called once the CLA task has finished.

```
interrupt void IsrFunc()
{
  CLA_ackInt(CLA_MOD_1);
}

CLA_setCallback( CLA_MOD_1, IsrFunc );
```

20.2.7.3 Notes

If a NULL pointer is passed as the function pointer, then the interrupt will be enable for the module, but not within the PIE module.

20.2.8 CLA_softwareStart

void CLA_softwareStart(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.8.1 Description

Forces the CLA task to execute.

20.2.8.2 Examples

Executes the code assigned to CLA_MOD_1 using the CLA_config() functions.

```
CLA_softwareStart(CLA_MOD_1);
```

20.2.9 CLA_isRunning

bool CLA_isRunning(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.9.1 Description

Returns true if the CLA task is running.

20.2.9.2 Examples

Blocks further execution while the CLA task is running.

```
while( CLA_isRunning( CLA_MOD_1 ) );
```

NOTES

20.2.10 CLA_softwareStartWait

void CLA_softwareStartWait(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.10.1 Description

Starts the CLA task and then waits for it to finish.

20.2.10.2 Examples

This starts CLA 1 and waits for it to complete.

```
CLA_softwareStartWait(CLA_MOD_1);
```

20.2.11 CLA_config

void CLA_config(CLA_Module Mod,Uint32* pFunc,CLA_IntMode Mode)

where:
Mod - Selects the CLA module.
pFunc - Pointer to CLA code.
Mode - CLA trigger mode.

20.2.11.1 Description

Configures the CLA module to run CLA code when the requested trigger occurs.

Valid triggers are ADC interrupts, PWM interrupts, software triggers or no triggers (CLA_INT_NONE).

The PWM and ADC modules used correspond to the CLA module number. e.g. CLA_MOD_1 can only be triggered with PWM_MOD_1 or ADC_MOD_1/

The only exception to this is CLA_MOD_8 for which there is no equivalent PWM_MOD_8 module. The CPU Timer 0 interrupt is used in place of the PWM module however the CLA_IntMode should still be specified as CLA_INT_PWM.

20.2.11.2 Examples

Assigns the code ClaTask to CLA_MOD_7. This is started when the interrupt ADC_INT_7 occurs. ADC_INT_7 must be used within the ADC configuration functions. CLA_INT_ADC can then be used as an argument for this function.

```
  CLA_3p3zCode( ClaTask, 7, 3,
            +1.46818, -0.314933, -0.153248,
             1.784224053, -1.629063952, -1.780916725, 1.632371281,
             0.48, 0.0, 240.0 );

  CLA_config( CLA_MOD_7, &ClaTask, CLA_INT_ADC );
```

20.2.11.3 Notes

If CLA_INT_PWM is used as the interrupt source then the CLA must be set up before the PWM. If this is not possible then PWM_clrInt() must be called once inside the idle loop. This is because the CLA uses the interrupt transition as a trigger rather than any particular state and the PWM is likely to have already triggered an interrupt before the CLA has finished being configured.

20.2.12 CLA_getPieId

INT_PieId CLA_getPieId(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.12.1 Description

Returns the PIE Id associated with the CLA interrupt.

This can be used when configuring the PIE controller manually using the interrupt functions from the INT CSL library.

This function does not normally need to be called as the interrupts are configured automatically using CLA_setCallback().

20.2.12.2 Examples

The function will return INT_ID_CLA2 when the ADC interrupt CLA_MOD_2 is passed as the argument.

```
  PieIdCla = CLA_getPieId( CLA_MOD_2 );
```

20.2.13 CLA_ackInt

void CLA_ackInt(CLA_Module Mod)

where:
Mod - Selects the CLA module.

20.2.13.1 Description

Used within an interrupt service routine to clear both the CLA EOC interrupt flag and the PIE group flag.

The CLA will continue to execute the set task when the next CLA trigger occurs, even if the CLA EOC interrupt flag is not cleared.

However, the CLA EOC interrupt flag must be cleared if the interrupt function assigned to this interrupt vector is to be called again when the next CLA interrupt EOC is reached. This is to say that the CLA task is independent of the CLA EOC interrupt flag whereas the EOC interrupt function is not.

20.2.13.2 Examples

Clears the CLA EOC interrupt flag and the PIE group flag for the specified interrupt.

```
interrupt void isr_cla_int6( void )
{
   CLA_ackInt( CLA_MOD_6 );
}
```

20.2.14 CLA_softStartConfig

void CLA_softStartConfig(CLA_Ctrl* Ptr,uint32_t RampMs,uint32_t UpdatePeriodNs)

where:
Ptr -
RampMs -
UpdatePeriodNs -

20.2.14.1 Description

```
Configures and enables a soft start for the CLA control code.

The 'RampMs' argument is the time in milli-seconds for the reference to reach
its steady state value. The period of execution for the update function,
CLA_SoftStartUpdate(), is specified by the argument 'UpdatePeriodNs'.

After configuring the soft start using this function the soft start is executed
by calling the update function CLA_SoftStartUpdate() at the
frequency determined by the period argument 'UpdatePeriodNs'. The update
function should preferably be called from inside an idle loop.

EXMPLES
This sets the soft start for 2 seconds with an update rate of 200kHz
(T=5000ns).

  CLA_SoftStartConfig( CLA_getCtrlPtr(Cntrl), 2000, 5000 );
```

20.2.15 CLA_softStartUpdate

void CLA_softStartUpdate(CLA_Ctrl* Ptr)

where:
Ptr -

20.2.15.1 Description

Performs an update of the CLA reference value according to the soft start parameters set using CLA_SoftStartConfig().

The reference value is updated with a value initially calculated within the function CLA_SoftStartConfig().

This function must be called at the frequency determined by the UpdatePeriodNs argument of the CLA_SoftStartConfig() function call. The update function should be called from within an idle loop.

EXMPLES
Updates the current CLA reference with the soft ramp delta value from within the main idle loop. A delay is generated which lasts for the period specified in the configuration parameters.

```
while ( 1 )
{
   CLA_softStartUpdate( CLA_getCtrlPtr(Cntrl) );
   SYS_usDelay( 5 ); // Delay for 5000ns
}
```

NOTES

20.2.16 CLA_softStartDirection

void CLA_softStartDirection(CLA_Ctrl* Ptr,int PowerUp)

where:
Ptr -
PowerUp -

20.2.16.1 Description

```
Converts the soft start to a soft stop or vice versa.

After a soft start has been configured the user may require a soft stop. This
function will reverse the ramp value allowing for the
CLA_SoftStartUpdate() function to generate a soft stop.

The ramp value may be reversed again to generate a soft start. The soft start or
soft stop is determined by the 'PowerUp' argument. If this is true the update
function will generate a soft start. If this is false the update function will
generate a soft stop. This parameter could be read from an
input pin allowing the end user to generate a soft start or soft stop.

EXMPLES
This configures the controller to perform a soft stop.
 CLA_SoftStartDirection( CLA_getCtrlPtr(Cntrl), false );
```

20.2.17 CLA_setRef

void CLA_setRef(CLA_Ctrl* Ptr,uint16_t Ref)

where:
Ptr -
Ref -

20.2.17.1 Description

```
Sets the reference for the controller.
```

20.2.18 CLA_memSet

void CLA_memSet(void* pAddr,uint16_t Data,int Count)

where:
pAddr - Start address.
Data - Data to write to memory.
Count - Number of uint16_t to write.

20.2.18.1 Description

The CLA to CPU message RAM is not writable by the main CPU.

At power up the CLA_MOD_8 task is configured with a small program that allows the CPU to request the CLA to set a single memory location.

This functions uses the default CLA_MOD_8 task to write to this location. This function then calls CLA task 8 several times with the values that are to be written to memory.

20.2.18.2 Examples

This starts CLA 1 and waits for it to complete.

```
CLA_softwareStartWait(CLA_MOD_1);
```

20.2.19 CLA_3p3zIMode

void CLA_3p3zIMode(void Name,void Adc,void Cmp,void A1,void A2,void A3,void B0,void B1,void B2,void B3,void K,void MiN,void MaX)

where:
Name -
Adc - ADC module number.
Cmp - Comp module number.
A1 -
A2 -
A3 -
B0 -
B1 -
B2 -
B3 -
K -
MiN - Minimum number of ticks that the duty can be set to.
MaX - Maximum number of ticks that the duty can be set to.

20.2.19.1 Description

```
This macro must be called at the top of the C file, before the main
function begins.

The values passed to the function call must be literals. Constants,
variables or macros cannot be used.

The function creates the CLA code for a 3p3z current mode controller.
```

20.2.19.2 Examples

```
Creates the CLA function called ClaTask. This reads the ADC value from ADC_MOD_7
and writes the duty to CMP_MOD_3.

  CLA_3p3zIMode( ClaTask, 7, 3,
            +1.46818, -0.314933, -0.153248,
             1.784224053, -1.629063952, -1.780916725, 1.632371281,
             0.48, 0, 240 );
```

20.3 Types

20.3.1 CLA_Module

```
enum CLA_Module
{
    CLA_MOD_1,
    CLA_MOD_2,
    CLA_MOD_3,
    CLA_MOD_4,
    CLA_MOD_5,
    CLA_MOD_6,
    CLA_MOD_7,
    CLA_MOD_8
};
```

20.3.1.1 Description

This uses the same style as the rest of the CSL, referring to the CLA tasks 1..8 as CLA_MOD_1..8.

20.3.2 CLA_IntMode

```
enum CLA_IntMode
{
    CLA_INT_ADC     = 0,    /* The matching ADC interrupt triggers the CLA
module */
    CLA_INT_PWM     = 2,    /* The matching PWM interrupt triggers the CLA
module */
    CLA_INT_NONE    = 1     /* There is no interrupt source for the CLA module
*/
};
```

20.3.2.1 Description

This describes the available triggers for the CLA task.

The PWM and ADC modules used correspond to the CLA module number. e.g. CLA_MOD_1 can only be triggered with PWM_MOD_1 or ADC_MOD_1/

The only exception to this is CLA_MOD_8 for which there is no equivalent PWM_MOD_8 module. The CPU Timer 0 interrupt is used in place of the PWM module however the CLA_IntMode should still be specified as CLA_INT_PWM.

20.3.3 CLA_3p3zData

```
struct CLA_3p3zData
{
      float m_PreValue; /* +0 */
      float m_U[3];     /* +2 +4 +6 */
      float m_E[3];     /* +8 +10 +12 ram */
};
```

20.3.3.1 Description

This structure is used by the 3p3z controllers for internal values.
This structure is only readable by the CPU.

20.3.4 CLA_2p2zData

```
struct CLA_2p2zData
{
      float m_PreValue; /* +0 */
      float m_U[2];     /* +2 +4 */
      float m_E[2];     /* +6 +8 ram */
};
```

20.3.4.1 Description

This structure is used by the 2p2z controllers for internal values.
This structure is only readable by the CPU.

20.3.5 CLA_Ctrl

```
struct CLA_Ctrl
{
      long  m_Ref;      /* +0 */
      long  m_Delta;
      long  m_Max;
};
```

20.3.5.1 Description

This structure is used by both controllers to set the reference and for soft start.
This structure is readable and writeable by the CPU.

21 csl_sys_c2803x_

21.1.1.1 Description

Contains functions to configure the system, including the system clock.
SYS_int() must be called before using any of CSL module functions.

By default, it is assumed that a 60MHz system clock is being used. Therefore,

```
 Low Speed Clock  (LSPCLK) = System Clock (SYSCLK) / 4 = 15MHz
```

These settings are defined in csl_sys_c2803x_Pub.h using the macros shown below.
These macros are then used through the rest of the library to calculate various frequency/ns to peripheral ticks.

```
#define USR_CLK_IN_HZ       10000000L
#define USR_PLL_MUL         SYS_PLL_MUL_6
#define USR_CLK_DIV         SYS_CLK_DIV_1
#define USR_PER_LSP_DIV     SYS_PER_CLK_DIV_4
```

If a 60MHz system clock is not being used then either change these values directly or define new values before csl_sys_c2803x_Pub.h is included.
Alternatively, define any changes within the build options as a processor define option e.g.

```
-d"USR_PLL_MUL=SYS_PLL_MUL_6"
```

The LSPCLK is used by SCI-A/B and SPI-A/B/C/D.
The SYSCLK is used by all other peripherals including the PWM.

The csl for the piccolo B assumes you are using the internal 10MHz oscillator
If you want to use the external crystal you would need to turn on the external circuits and select the external clock input after calling SYS_init(). eg

```
CLKCTL[XTALOSCOFF]     = 1
CLKCTL[OSCCLKSRC2SEL]  = 0
CLKCTL[OSCCLKSRCSEL]   = 1
```

21.1.1.2 Links

file:///C:/tidcs/c28/CSL_C280x/v100/doc/CSL_C280x.pdf
http://focus.ti.com/lit/ug/spru712f/spru712f.pdf

21.2 Api

SYS_configClk()
SYS_setPerhiperalClk()
SYS_init()

21.2.1 SYS_configClk

void SYS_configClk(SYS_PllMultiplier InMultiplier,SYS_ClockDivide InDiv,SYS_ClockOutDivide OutDiv)

where:
InMultiplier - The PLL multiplier.
InDiv - The DSP clock in divider.
OutDiv - The system clock out divider.

21.2.1.1 Description

Configures the DSP system clock.

This is called as part of SYS_init() with USR_PLL_MUL, USR_CLK_DIV and SYS_CLK_OUT_DIV_1.

If the hardware in use is not a 100MHz part or a 20MHz crystal is not being used then these macro values should be changed accordingly.

The clock can be reconfigured at runtime by calling this function with the appropriate parameters.

21.2.1.2 Examples

Overrides the default values used by SYS_init().

```
SYS_configClk( SYS_PLL_MUL_5 , SYS_CLK_DIV_2 , SYS_CLK_OUT_DIV_1 );
```

21.2.1.3 Notes

If this function is called with different parameters, the frequency and ns to ticks functions will return incorrect results.

21.2.2 SYS_setPerhiperalClk

void SYS_setPerhiperalClk(SYS_PerClockDivide LspDiv)

where:
LspDiv - Selects the low speed clock divider.

21.2.2.1 Description

Sets the values that the system clock is divided by in order to obtain the clock for the peripherals.

This is called by default as part of SYS_init() with USR_PER_LSP_DIV.

The LSPCLK is used by the SCI-A/B and SPI-A/B/C/D.
The SYSCLK is used by all the other peripherals (CPU Timers, ePWMs, eCANs, eCAPs, I2C).

21.2.2.2 Examples

Used to override the default values set by SYS_init().

```
SYS_setPerhiperalClk( SYS_PER_CLK_DIV_4 );
```

21.2.2.3 Notes

If this function is called with different parameters the frequency/ns to Ticks functions will return an incorrect result.

21.2.3 SYS_init

void SYS_init(void)

where:

21.2.3.1 Description

Initializes the Chip Support Library.

This function must be called before any of the API functions are called.

This function performs different initialization actions depending on the DSP chip being used. For all processors, this function initializes the stack, peripheral clocks and interrupt module. It copies time critical routines and the flash set up code to RAM.

The clocking options are configured to the user defined values by the function.

```
  SYS_configClk( USR_PLL_MUL, USR_CLK_DIV, SYS_CLK_OUT_DIV_1 );
  SYS_setPerhiperalClk(USR_PER_HSP_DIV, USR_PER_LSP_DIV);
```

21.2.3.2 Examples

This initializes the CSL library.

```
   SYS_init();
```

21.3 Types

21.3.1 SYS_ClockDivide

```
enum SYS_ClockDivide
{
    SYS_CLK_DIV_4    = SYS_LIT( 4,1),
    SYS_CLK_DIV_2    = SYS_LIT( 2,2),
    SYS_CLK_DIV_1    = SYS_LIT( 1,3)
};
```

21.3.1.1 Description

This is used to select the system clock divider.

21.3.2 SYS_PllMultiplier

```
enum SYS_PllMultiplier
{
    SYS_PLL_MUL_BYPASS  = SYS_LIT( 1,   0),
    SYS_PLL_MUL_1       = SYS_LIT( 1,   1),
    SYS_PLL_MUL_2       = SYS_LIT( 2,   2),
    SYS_PLL_MUL_3       = SYS_LIT( 3,   3),
    SYS_PLL_MUL_4       = SYS_LIT( 4,   4),
    SYS_PLL_MUL_5       = SYS_LIT( 5,   5),
    SYS_PLL_MUL_6       = SYS_LIT( 6,   6),
    SYS_PLL_MUL_7       = SYS_LIT( 7,   7),
    SYS_PLL_MUL_8       = SYS_LIT( 8,   8),
    SYS_PLL_MUL_9       = SYS_LIT( 9,   9),
    SYS_PLL_MUL_10      = SYS_LIT( 10, 10),
    SYS_PLL_MUL_11      = SYS_LIT( 11, 11),
    SYS_PLL_MUL_12      = SYS_LIT( 12, 12)
};
```

21.3.2.1 Description

This is used to select the system clock multiplier.

21.3.3 SYS_ClockOutDivide

```
enum SYS_ClockOutDivide
{
    SYS_CLK_OUT_DIV_4   = SYS_LIT( 4,   0),
    SYS_CLK_OUT_DIV_2   = SYS_LIT( 2,   1),
    SYS_CLK_OUT_DIV_1   = SYS_LIT( 1,   2),
    SYS_CLK_OUT_NONE    = SYS_LIT( 0,   3)
};
```

21.3.3.1 Description

This is used to select the system output clock.

21.3.4 SYS_PerClockDivide

```
enum SYS_PerClockDivide
{
    SYS_PER_CLK_DIV_1   = SYS_LIT( 1,  0),  /* 100.00MHz */
    SYS_PER_CLK_DIV_2   = SYS_LIT( 2,  1),  /* 50.00MHz */
    SYS_PER_CLK_DIV_4   = SYS_LIT( 4,  2),  /* 25.00MHz */
    SYS_PER_CLK_DIV_6   = SYS_LIT( 6,  3),  /* 16.66MHz */
    SYS_PER_CLK_DIV_8   = SYS_LIT( 8,  4),  /* 12.50MHz */
    SYS_PER_CLK_DIV_10  = SYS_LIT( 10, 5),  /* 10.00MHz */
    SYS_PER_CLK_DIV_12  = SYS_LIT( 12, 6),  /* 8.33MHz */
    SYS_PER_CLK_DIV_14  = SYS_LIT( 14, 7)   /* 7.14MHz */
};
```

21.3.4.1 Description

These are the divide options for the system clock, based on a 100MHz system clock.

21.3.5 USR_CLK_IN_HZ

```
#ifndef USR_CLK_IN_HZ
#define USR_CLK_IN_HZ 10000000L
#endif
```

21.3.5.1 Description

This is the default input frequency to the DSP chip.

NOTES
You can change this value here or in the build options. E.g.
   -d"USR_CLK_IN_HZ=10000000L"

21.3.6 USR_PLL_MUL

```
#ifndef USR_PLL_MUL
#define USR_PLL_MUL   SYS_PLL_MUL_6
#endif
```

21.3.6.1 Description

This is the default DSP multiplier to the DSP chip.

NOTES
You can change this value here or in the build options. E.g.
   -d"USR_PLL_MUL=SYS_PLL_MUL_5"

21.3.7 USR_CLK_DIV

```
#ifndef USR_CLK_DIV
#define USR_CLK_DIV   SYS_CLK_DIV_1
#endif
```

21.3.7.1 Description

This is the default DSP divider for the DSP chip.

NOTES
You can change this value here or in the build options. E.g.
    -d"USR_CLK_DIV=SYS_CLK_DIV_1"

21.3.8 USR_PER_LSP_DIV

```
#ifndef USR_PER_LSP_DIV
#define USR_PER_LSP_DIV SYS_PER_CLK_DIV_4
#endif
```

21.3.8.1 Description

This is the default divider for the DSP low speed system clock.

NOTES
You can change this value here or in the build options. E.g.
    -d"USR_PER_LSP_DIV=SYS_PER_CLK_DIV_1"

21.3.9 SYS_CLK_HZ

```
#define SYS_CLK_HZ ((1L*USR_CLK_IN_HZ * SYS_LIT_VALUE(USR_PLL_MUL)) /
(1L*SYS_LIT_VALUE(USR_CLK_DIV)))
```

21.3.9.1 Description

This is the calculated system clock in Hz based on USR_CLK_IN_HZ, USR_PLL_MUL and USR_CLK_DIV.

21.3.10 SYS_CLK_NS

```
#define SYS_CLK_NS (NS_PER_SEC/SYS_CLK_HZ)
```

21.3.10.1 Description

This is the calculated system clock in ns based on USR_CLK_IN_HZ, USR_PLL_MUL and USR_CLK_DIV.

21.3.11 SYS_CLK_LSP_HZ

```
#define SYS_CLK_LSP_HZ (SYS_CLK_HZ / SYS_LIT_VALUE(USR_PER_LSP_DIV))
```

21.3.11.1 Description

This is the calculated low speed system clock in Hz based on SYS_CLK_HZ and USR_PER_LSP_DIV.

21.3.12 SYS_CLK_PS

```
#define SYS_CLK_PS (NS_PER_SEC/(SYS_CLK_HZ/1000))
```

21.3.12.1 Description

This is the calculated system clock in ps based on USR_CLK_IN_HZ, USR_PLL_MUL and USR_CLK_DIV.

21.3.13 INT_PieId

```
enum INT_PieId
{
    INT_ID_ADCINT1H      = INT_GROUP_VAL( 1,  1  ),  /* high prioity ADC INT1 */
    INT_ID_ADCINT2H      = INT_GROUP_VAL( 1,  2  ),  /* high prioity ADC INT2 */
    INT_ID_XINT1         = INT_GROUP_VAL( 1,  4  ),  /* Group(1-12)   Index(1-8)
*/
    INT_ID_XINT2         = INT_GROUP_VAL( 1,  5  ),
    INT_ID_ADCINT9H      = INT_GROUP_VAL( 1,  6  ),
    INT_ID_TIM1          = INT_GROUP_VAL( 1,  7  ),  /* TIM2/3 are done using
int13/14 */
    INT_ID_WAKE          = INT_GROUP_VAL( 1,  8  ),
    INT_ID_TZINT1        = INT_GROUP_VAL( 2,  1  ),
    INT_ID_TZINT2        = INT_GROUP_VAL( 2,  2  ),
    INT_ID_TZINT3        = INT_GROUP_VAL( 2,  3  ),
    INT_ID_TZINT4        = INT_GROUP_VAL( 2,  4  ),
    INT_ID_TZINT5        = INT_GROUP_VAL( 2,  5  ),
    INT_ID_TZINT6        = INT_GROUP_VAL( 2,  6  ),
    INT_ID_EPWM1         = INT_GROUP_VAL( 3,  1  ),
    INT_ID_EPWM2         = INT_GROUP_VAL( 3,  2  ),
    INT_ID_EPWM3         = INT_GROUP_VAL( 3,  3  ),
    INT_ID_EPWM4         = INT_GROUP_VAL( 3,  4  ),
    INT_ID_EPWM5         = INT_GROUP_VAL( 3,  5  ),
    INT_ID_EPWM6         = INT_GROUP_VAL( 3,  6  ),
    INT_ID_EPWM7         = INT_GROUP_VAL( 3,  7  ),
    INT_ID_ECAP1         = INT_GROUP_VAL( 4,  1  ),
    INT_ID_SPIRXA        = INT_GROUP_VAL( 6,  1  ),  /* SPI-A */
    INT_ID_SPITXA        = INT_GROUP_VAL( 6,  2  ),  /* SPI-A */
    INT_ID_SPIRXB        = INT_GROUP_VAL( 6,  3  ),  /* SPI-B */
    INT_ID_SPITXB        = INT_GROUP_VAL( 6,  4  ),  /* SPI-B */
    INT_ID_SPIRXC        = INT_GROUP_VAL( 6,  5  ),  /* SPI-C */
    INT_ID_SPITXC        = INT_GROUP_VAL( 6,  6  ),  /* SPI-C */
    INT_ID_SPIRXD        = INT_GROUP_VAL( 6,  7  ),  /* SPI-D */
    INT_ID_SPITXD        = INT_GROUP_VAL( 6,  8  ),  /* SPI-D */
    INT_ID_SCIRXINTA     = INT_GROUP_VAL( 9,  1  ),  /* SCI-A-RX */
    INT_ID_SCITXINTA     = INT_GROUP_VAL( 9,  2  ),  /* SCI-A-TX */
    INT_ID_SCIRXINTB     = INT_GROUP_VAL( 9,  3  ),  /* SCI-B-RX */
    INT_ID_SCITXINTB     = INT_GROUP_VAL( 9,  4  ),  /* SCI-B-TX */
    INT_ID_ADCINT1       = INT_GROUP_VAL( 10, 1  ),
    INT_ID_ADCINT2       = INT_GROUP_VAL( 10, 2  ),
    INT_ID_ADCINT3       = INT_GROUP_VAL( 10, 3  ),
    INT_ID_ADCINT4       = INT_GROUP_VAL( 10, 4  ),
    INT_ID_ADCINT5       = INT_GROUP_VAL( 10, 5  ),
    INT_ID_ADCINT6       = INT_GROUP_VAL( 10, 6  ),
    INT_ID_ADCINT7       = INT_GROUP_VAL( 10, 7  ),
    INT_ID_ADCINT8       = INT_GROUP_VAL( 10, 8  ),
    INT_ID_CLA1          = INT_GROUP_VAL( 11, 1  ),
    INT_ID_CLA2          = INT_GROUP_VAL( 11, 2  ),
    INT_ID_CLA3          = INT_GROUP_VAL( 11, 3  ),
```

```
    INT_ID_CLA4         = INT_GROUP_VAL( 11, 4  ),
    INT_ID_CLA5         = INT_GROUP_VAL( 11, 5  ),
    INT_ID_CLA6         = INT_GROUP_VAL( 11, 6  ),
    INT_ID_CLA7         = INT_GROUP_VAL( 11, 7  ),
    INT_ID_CLA8         = INT_GROUP_VAL( 11, 8  )
};
```

21.3.13.1 Description

This is the enum for each PIE interrupt source. Each value is defined as an INT_PieGroup and INT_PieIndex.