Microchip 32-bit Chip Support Package

Help for Microchip 32-bit Chip Support Package (CSP).

© 2013-2018 Microchip Technology Inc. All rights reserved.

Peripheral Library Overview

This help documentation provides a comprehensive source of information on how to use Peripheral Libraries and Applications.

Peripheral Libraries Help

This topic provides help for the peripheral libraries that are available in the Microchip 32 bit Chip Support Package (CSP).

Description

Peripheral Libraries generates easy to understand C code which directly accesses the Peripheral registers without any abstraction layers.

The Initialization code generated in PLIB is as configured by the user in MPLAB Harmony Configurator (MHC).

SAM C20/C21 Peripheral Libraries

This Section Provides list of all the Peripheral Libraries for SAMC20/C21 Family.

Description

Below Table lists all the Peripheral Libraries for SAM C20/C21 Family.

SAM E70/S70/V70/V71 Peripheral Libraries

This Section Provides list of all the Peripheral Libraries for SAM E70/S70/V70/V71 Family.

Description

Below Table lists all the Peripheral Libraries for SAM E70/S70/V70/V71 Family.

SAM E70/S70/V70/V71 Peripheral Libraries
ACC Peripheral Library Help
AFEC Peripheral Library Help
Clock Peripheral Library Help
DACC Peripheral Library Help
EFC Peripheral Library Help
I2SC Peripheral Library Help
MPU Peripheral Library Help
NVIC Peripheral Library Help
PIO Peripheral Library Help
PWM Peripheral Library Help
QSPI Peripheral Library Help
RSTC Peripheral Library Help
RSWDT Peripheral Library Help
RTC Peripheral Library Help
RTT Peripheral Library Help
SDRAMC Peripheral Library Help
SMC Peripheral Library Help
SPI Peripheral Library Help
SSC Peripheral Library Help

SUPC Peripheral Library Help
SYSTICK Peripheral Library Help
TC Peripheral Library Help
TRNG Peripheral Library Help
TWIHS Peripheral Library Help
UART Peripheral Library Help
USART Peripheral Library Help
WDT Peripheral Library Help
XDMAC Peripheral Library Help

Peripheral Libraries

This topic provides help for the peripheral libraries that are available in the Microchip 32 bit Chip Support Package (CSP).

Description

Below are the peripheral libraries that are available in the Microchip 32 bit Chip Support Package (CSP).

ACC Peripheral Library Help

This section provides interface to use Analog Comparator Controller (ACC) peripheral.

Introduction

This section provides brief overview of ACC peripheral

Description

The Analog Comparator Controller (ACC) configures the analog comparator and generates an interrupt depending on user settings. The analog comparator embeds two 8-to-1 multiplexers that generate two internal inputs. These inputs are compared, resulting in a compare output. The hysteresis level, edge detection and polarity are configurable.

The ACC also generates a fault output which can be used by the Pulse Width Modulator (PWM) fault input.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains how to use the peripheral library.

Description

Analog comparison status can be polled by the software or can be configured to generate callback.

- With polling, the application will need to continuously poll to check if the comparison edge occurred.
- With Callback, the registered callback function will be called when comparison edge is detected. This means we do not have to poll continuously.

The table below shows the applicable function in polled and callback mode

Functions	Callback (Interrupt)	Polling
ACCx_Initialize	Yes	Yes
ACCx_StatusGet	Yes	Yes
ACCx_CallbackRegister	Yes	No

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	ACCx_Initialize	Initializes ACC x module of the device

b) Setup Functions

	Name	Description
	ACCx_CallbackRegister	Allows application to register callback with PLIB

c) Status Functions

	Name	Description
	ACCx_StatusGet	Returns comparison status of the ACC x

d) Data Types and Constants

	Name	Description
	ACC_CALLBACK	Pointer to a ACC callback function
	ACC_STATUS_SOURCE	Identifies ACC Comparison status

a) Initialization Function

ACCx_Initialize Function

Initializes ACC x module of the device

File

plib_acc.h

C

```
void ACCx_Initialize();
```

Returns

None.

Description

This function initializes ACC x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, Status API can be used to get the comparison status.

Remarks

This function must be called only once and before any other ACC function is called.

Preconditions

MHC GUI should be configured with the right values.

Example

```
ACC0_Initialize ();
```

Function

void ACCx_Initialize (void);

b) Setup Functions

ACCx_CallbackRegister Function

Allows application to register callback with PLIB

File

plib_acc.h

C

```
void ACCx_CallbackRegister(ACC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function allows application to register a callback function for the PLIB to call back when interrupt occurred.

At any point if application wants to stop the callback, it can call this function with "callback" value as NULL.

Remarks

None.

Preconditions

The ACCx_Initialize function must have been called.

Example

```
MY_APP_OBJ myAppObj;

void APP_ACC_CallbackFunction (uintptr_t context)
{
    // The context was set to an application specific object.
    // It is now retrievable easily in Callback function.
       MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;
    //Application related tasks
}

ACC0_CallbackRegister (APP_ACC_CallbackFunction, (uintptr_t)&myAppObj);
```

Parameters

Parameters	Description
callback	Pointer to the callback function implemented by the user.
context	The value of parameter will be passed back to the application unchanged, when the callback function is called. It can be used to identify any application specific value that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void ACCx_CallbackRegister (ACC_CALLBACK callback, uintptr_t context);

c) Status Functions

ACCx_StatusGet Function

Returns comparison status of the ACC x

File

plib_acc.h

C

```
bool ACCx_StatusGet(ACC_STATUS_SOURCE status);
```

Returns

Returns the comparison status based on input parameter.

Description

This function returns the current comparison status based on input parameter either comparison edge or synchronized comparator output of ACC x module.

Remarks

None.

Preconditions

None.

Example

```
bool status;
status = ACC0_StatusGet(ACC_STATUS_SOURCE_COMPARATOR_OUTPUT);
if (status)
{
    // Application related tasks
}
```

Function

bool ACCx_StatusGet (ACC_STATUS_SOURCE status);

d) Data Types and Constants

ACC_CALLBACK Type

Pointer to a ACC callback function

File

plib_acc.h

C

```
typedef void (* ACC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

ACC Callback Function pointer typedef void (*ACC_CALLBACK) (uintptr_t context);

This data type defines the required function signature for the ACC callback function. Application must register a pointer to a callback function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive callback from the PLIB.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The context parameter contains the a handle to the client context, provided at the time the callback function was registered using the ACCx_CallbackRegister function. This context handle value is passed back to the client as the "context" parameter. It can be any value (such as a pointer to the client's data) necessary to identify the client context or instance of the client that made the data transfer request.

The callback function executes in the PLIB's interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
ACC0_CallbackRegister (&APP_ACC_CallbackFunction, NULL);
void APP_ACC_CallbackFunction (uintptr_t context)
{
    //Application related tasks
}
```

ACC_STATUS_SOURCE Enumeration

Identifies ACC Comparison status

File

plib_acc.h

C

```
typedef enum {
  ACC_STATUS_SOURCE_COMPARATOR_OUTPUT,
  ACC_STATUS_SOURCE_COMPARISON_EDGE
} ACC_STATUS_SOURCE;
```

Members

Members	Description
ACC_STATUS_SOURCE_COMPARATOR_OUTPUT	ACC synchronized comparator output
ACC_STATUS_SOURCE_COMPARISON_EDGE	ACC comparison edge

Description

ACC comparison status

This enumeration identifies the ACC Comparison status either from Comparison edge or synchronized comparator output.

Remarks

None.

Files

Files

Name	Description
plib_acc.h	ACC peripheral library interface.

Description

plib_acc.h

ACC peripheral library interface.

Enumerations

	Name	Description
	ACC_STATUS_SOURCE	Identifies ACC Comparison status

Functions

	Name	Description
	ACCx_CallbackRegister	Allows application to register callback with PLIB
	ACCx_Initialize	Initializes ACC x module of the device
	ACCx_StatusGet	Returns comparison status of the ACC x

Types

	Name	Description
	ACC_CALLBACK	Pointer to a ACC callback function

Description

Analog Comparator Controller(ACC) Peripheral Library Interface Header File

This library provides documentation of all the interfaces which can be used to interact with an instance of a Analog Comparator Controller (ACC). This file must not be included in any MPLAB Project.

File Name

acc.h

Company

Microchip Technology Inc.

AFEC Peripheral Library Help

This section provides an interface to use Analog Front-End Controller (AFEC) peripheral to convert analog voltages to a corresponding digital value.

Introduction

This section provides brief overview of the AFEC peripheral

Description

The Analog Front-End Controller (AFEC) peripheral supports following features.

- 12-bit Resolution up to 16-bit Resolution by Digital Averaging

- Selectable Single-ended or Differential Input Voltage
- Selectable Single or Dual Sample-and-Hold Mode
- Programmable Gain for Maximum Full-Scale Input Range 0–VDD
- Programmable Offset Per Channel
- Integrated Multiplexers Offering Up to 12 Independent Analog Inputs
- Individual Enable and Disable of Each Channel
- Customized channel sequencer to control order of conversion
- Selectable Conversion Trigger sources
  - External trigger pin
  - Timer Counter Output
  - PWM Event output
  - Software Trigger
  - Free Run mode

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

AFEC converts all the enabled channels on software or hardware triggers. It provides both polling and callback methods to indicate the end of conversion to read the converted data from the result registers.

- With polling, the application will need to continuously poll to check if the conversion is completed.
- With callback, the registered callback function will be called once the conversion is completed. This means we do not have to poll continuously.

Callback method

This example demonstrates how to read 3 analog inputs in the callback function. Three channels are enabled and configured to start conversion on hardware trigger.

```
/* This function is called after conversion of last channel in the user sequence */
void AFEC_EventHandler(uintptr_t context)
{
/* Read the result of 3 channels*/
adc_ch5_count = AFECx_ChannelResultGet(CH5);
adc_ch6_count = AFECx_ChannelResultGet(CH6);
adc_ch0_count = AFECx_ChannelResultGet(CH0);
}
int main(void)
{
    /* Register callback function for AFEC end of conversion interrupt*/
    AFECx_CallbackRegister(AFEC_EventHandler, (uintptr_t)NULL);

    while(true)
    {

    }
}
```

Polling method

This example demonstrates how to start conversion and read the converted data in polled mode. One channel is enabled and configured to start conversion on software trigger.

```
/* Start ADC conversion */
```

```
AFECx_ConversionStart();

/* Wait till ADC conversion result is available */
while(!AFECx_ChannelResultIsReady(CH0))
{

};

/* Read the ADC result */
adc_count = AFECx_ChannelResultGet(CH0);
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	AFECx_Initialize	Initializes given instance of AFEC peripheral.

b) Setup Functions

	Name	Description
	AFECx_CallbackRegister	Registers the function to be called from interrupt
	AFECx_ChannelGainSet	Writes the gain of the channel
	AFECx_ChannelOffsetSet	Writes the channel offset

c) Transaction Functions

	Name	Description
	AFECx_ChannelResultGet	Reads the conversion result of the channel
	AFECx_ChannelsEnable	Enables the ADC channels
	AFECx_ChannelsDisable	Disables the ADC channels
	AFECx_ChannelsInterruptDisable	Disables the ADC interrupt sources
	AFECx_ChannelsInterruptEnable	Enables the ADC interrupt sources
	AFECx_ConversionSequenceSet	Sets the user sequence of the channel conversion
	AFECx_ConversionStart	Starts the ADC conversion of all the enabled channels with the software trigger

d) Status Functions

	Name	Description
	AFECx_ChannelResultIsReady	Returns the status of the channel conversion

e) Data Types and Constants

	Name	Description
	AFEC_CALLBACK	Defines the function pointer data type and function signature for the afec peripheral callback function.
	AFEC_CALLBACK_OBJECT	Callback structure
	AFEC_CHANNEL_GAIN	Identifies programmable gain setting
	AFEC_CHANNEL_MASK	Identifies AFEC channel mask
	AFEC_CHANNEL_NUM	Identifies AFEC channel number
	AFEC_INTERRUPT_MASK	Identifies channel interrupt sources mask

Description

a) Initialization Function

AFECx_Initialize Function

Initializes given instance of AFEC peripheral.

File

plib_afec.h

C

```
void AFECx_Initialize();
```

Returns

None.

Description

This function initializes given instance of AFEC peripheral of the device with the values configured in MCC GUI. Once the peripheral is initialized, peripheral can be used for conversion.

Remarks

This function must be called before any other AFEC function is called.

This function should only be called once during system initialization.

Preconditions

MCC GUI should be configured with the right values.

Example

```
AFEC0_Initialize();
```

Function

void AFECx_Initialize (void);

b) Setup Functions

AFECx_CallbackRegister Function

Registers the function to be called from interrupt

File

plib_afec.h

C

```
void AFECx_CallbackRegister(AFEC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from interrupt

Remarks

Context value can be set to NULL if not required. To disable callback function, pass NULL for the callback parameter.

Preconditions

AFECx_Initialize() must have been called first for the associated instance.

Example

```
void AFECx_Callback_Fn(uintptr_t context);

AFEC0_Initialize();
AFEC0_CallbackRegister(AFEC_Callback_Fn, NULL);
```

Parameters

Parameters	Description
callback	callback function pointer
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void AFECx_CallbackRegister (AFECx_CALLBACK callback, uintptr_t context);

AFECx_ChannelGainSet Function

Writes the gain of the channel

File

plib_afec.h

C

```
void AFECx_ChannelGainSet(AFEC_CHANNEL_NUM channel, AFEC_CHANNEL_GAIN gain);
```

Returns

None.

Description

This function writes the gain of the channel

Remarks

Input voltage range reduces as gain increases. For gain = 1, range is (0) to (Vref) For gain = 2, range is (Vref/4) to (3 * Vref/4) For gain = 4, range is (3 * Vref/8) to (5 * Vref/8)

Preconditions

AFECx_Initialize() must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelGainSet(AFEC_CH0, AFEC_CHANNEL_GAIN_2X);
```

Parameters

Parameters	Description
channel	channel number
gain	channel gain of the amplifier

Function

void AFECx_ChannelGainSet (AFEC_CHANNEL channel, AFEC_CHANNEL_GAIN gain);

AFECx_ChannelOffsetSet Function

Writes the channel offset

File

plib_afec.h

C

```
void AFECx_ChannelOffsetSet(AFEC_CHANNEL_NUM channel, uint16_t offset);
```

Returns

None.

Description

This function writes the offset value for the channel. This offset value is added to the value of sample to get the full range output (0 to Vref). Normally, this value should be set to Vref/2 in 10-bit format.

Remarks

Offset should be set at the initialization. If this function is called when conversion is on-going, offset will be applied from the next conversion. Offset is added to the sample value and thus offset limits the input voltage range. Offset less than Vref/2 will result in ADC saturation for input voltage greater than Vref/2.

Preconditions

AFECx_Initialize() must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelOffsetSet(AFEC_CH0, 512U);
```

Parameters

Parameters	Description
channel	channel number
offset	10-bit offset generated by internal DAC

Function

void AFECx_ChannelOffsetSet (AFEC_CHANNEL channel, uint16_t offset);

c) Transaction Functions

AFECx_ChannelResultGet Function

Reads the conversion result of the channel

File

plib_afec.h

C

```
uint16_t AFECx_ChannelResultGet(AFEC_CHANNEL_NUM channel);
```

Returns

uint16_t - conversion result

Description

This function reads the conversion result of the channel

Remarks

This function can be called from interrupt or by polling the status when result is available. User should decode the result based on result sign mode (signed or unsigned result) and result resolution (12, 13, 14, 15 or 16 bit result) configuration.

Preconditions

AFECx_Initialize() must have been called first for the associated instance. And conversion must have been started.

Example

```
uint16_t result;
bool status;
AFEC0_Initialize();
AFEC0_ChannelsEnable(AFEC_CH0);
AFEC0_ConversionStart();
status = AFEC0_ChannelResultIsReady(AFEC_CH0);
if (status)
{
    result = AFEC0_ChannelResultGet(AFEC_CH0);
}
```

Parameters

Parameters	Description
channel	channel number

Function

uint16_t AFECx_ChannelResultGet (AFEC_CHANNEL channel);

AFECx_ChannelsEnable Function

Enables the ADC channels

File

plib_afec.h

C

```
void AFECx_ChannelsEnable(AFEC_CHANNEL_MASK channelsMask);
```

Returns

None.

Description

This function enables channels specified in channelsMask

Remarks

This function does not disable channels which are not included in the channel mask.

Preconditions

AFECx_Initialize() function must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelsEnable(AFEC_CH0_MASK | AFEC_CH3_MASK);
```

Parameters

Parameters	Description
channelsMask	set of channel numbers

Function

void AFECx_ChannelsEnable (AFEC_CHANNEL_MASK channelsMask);

AFECx_ChannelsDisable Function

Disables the ADC channels

File

plib_afec.h

C

```
void AFECx_ChannelsDisable(AFEC_CHANNEL_MASK channelsMask);
```

Returns

None.

Description

This function disables channels specified in channelsMask

Remarks

This function does not enable channels which are not included in the channel mask.

Preconditions

AFECx_Initialize() must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelsDisable(AFEC_CH0_MASK | AFEC_CH3_MASK);
```

Parameters

Parameters	Description
channelsMask	set of channel numbers

Function

void AFECx_ChannelsDisable (AFEC_CHANNEL_MASK channelsMask);

AFECx_ChannelsInterruptDisable Function

Disables the ADC interrupt sources

File

plib_afec.h

C

```
void AFECx_ChannelsInterruptDisable(AFEC_INTERRUPT_MASK channelsInterruptMask);
```

Returns

None.

Description

This function disables interrupt sources specified in channelsInterruptMask.

Remarks

This function does not enable interrupts which are not included in the mask.

Preconditions

AFECx_Initialize() function must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelsInterruptDisable(AFEC_INTERRUPT_EOC_0_MASK);
```

Parameters

Parameters	Description
channelsInterruptMask	interrupt sources

Function

void AFECx_ChannelsInterruptDisable (AFEC_INTERRUPT_MASK channelsInterruptMask);

AFECx_ChannelsInterruptEnable Function

Enables the ADC interrupt sources

File

plib_afec.h

C

```
void AFECx_ChannelsInterruptEnable(AFEC_INTERRUPT_MASK channelsInterruptMask);
```

Returns

None.

Description

This function enables interrupt sources specified in channelsInterruptMask.

Remarks

This function does not disable interrupts which are not included in the mask.

Preconditions

AFECx_Initialize() must have been called first for the associated instance.

Example

```
AFEC0_Initialize();
AFEC0_ChannelsInterruptEnable(AFEC_INTERRUPT_EOC_0_MASK);
```

Parameters

Parameters	Description
channelsInterruptMask	interrupt sources

Function

void AFECx_ChannelsInterruptEnable (AFEC_INTERRUPT_MASK channelsInterruptMask);

AFECx_ConversionSequenceSet Function

Sets the user sequence of the channel conversion

File

plib_afec.h

C

```
void AFECx_ConversionSequenceSet(AFEC_CHANNEL_NUM * channelList, uint8_t numChannel);
```

Returns

None.

Description

This function sets the order in which channels are converted.

Remarks

Conversion order is set in this function and remains valid until user configures new conversion sequence order or reinitializes the peripheral. Array pointed to by *channelList must be valid during the call to this function. This function copies the array data into AFEC HW registers.

Preconditions

AFECx_Initialize() must have been called first for the associate instance. Conversion should not be ongoing while changing the sequence.

Example

```
AFEC_CHANNEL seq_order[4] = {AFEC_CH3, AFEC_CH5, AFEC_CH1, AFEC_CH2};
AFEC0_Initialize();
AFEC0_ConversionSequenceSet(seq_order, 0x4);
AFEC0_ChannelsEnable(AFEC_CH0_MASK | AFEC_CH1_MASK | AFEC_CH2_MASK | AFEC_CH3_MASK);
```

Parameters

Parameters	Description
*channelList	pointer to the list of the channels which describes the order of conversion
numChannel	Number of enabled channels in the list

Function

void AFECx_ConversionSequenceSet (AFEC_CHANNEL *channelList, uint8_t numChannel);

AFECx_ConversionStart Function

Starts the ADC conversion of all the enabled channels with the software trigger

File

plib_afec.h

C

```
void AFECx_ConversionStart();
```

Returns

None.

Description

This function starts ADC conversion of all the enabled channels. Trigger is common for all the enabled channels. And channels are converted in sequential order or in user decided order based on configuration.

Remarks

This function should be called only when SW trigger for conversion is selected.

Preconditions

AFECx_Initialize() function must have been called first for the associated instance and channels must have been enabled using AFECx_ChannelsEnable() function.

Example

```
AFEC0_Initialize();
AFEC0_ChannelsEnable(AFEC_CH0);
AFEC0_ConversionStart();
```

Function

void AFECx_ConversionStart();

d) Status Functions

AFECx_ChannelResultIsReady Function

Returns the status of the channel conversion

File

plib_afec.h

C

```
bool AFECx_ChannelResultIsReady(AFEC_CHANNEL_NUM channel);
```

Returns

bool - status of the channel

- false: channel is disabled or conversion is not yet finished
- true: channel is enabled and result is available

Description

This function returns the status of the channel whether conversion is complete and result is available

Remarks

None

Preconditions

AFECx_Initialize() function must have been called first for the associated instance.

Example

```
bool ch_status;
AFEC0_Initialize();
AFEC0_ChannelsEnable(AFEC_CH0);
AFEC0_ConversionStart();
ch_status = AFEC0_ChannelResultIsReady(AFEC_CH0);
```

Parameters

Parameters	Description
channel	channel number

Function

bool AFECx_ChannelResultIsReady(AFEC_CHANNEL channel);

e) Data Types and Constants

AFEC_CALLBACK Type

Defines the function pointer data type and function signature for the afec peripheral callback function.

File

plib_afec.h

C

```
typedef void (* AFEC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function pointer and function signature for the afec peripheral callback function. The afec peripheral will call back the client's function with this signature at the end of conversion.

Remarks

None.

Preconditions

AFEC_Initialize must have been called for the given afec peripheral instance and AFEC_CallbackRegister must have been called to set the function to be called.

Example

```
void AFEC_ChCallback (uintptr_t context);

AFEC_CallbackRegister(AFEC_ChCallback, NULL);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void (*AFEC_CALLBACK)(uintptr_t context)

AFEC_CALLBACK_OBJECT Structure

Callback structure

File

plib_afec.h

C

```
typedef struct {
  AFEC_CALLBACK callback_fn;
  uintptr_t context;
} AFEC_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

AFEC_CHANNEL_GAIN Enumeration

Identifies programmable gain setting

File

plib_afec.h

C

```
typedef enum {
  AFEC_CHANNEL_GAIN_X1,
  AFEC_CHANNEL_GAIN_X2,
  AFEC_CHANNEL_GAIN_X4
} AFEC_CHANNEL_GAIN;
```

Description

Programmable gain value

This enumeration identifies ADC gain of built-in programmable gain amplifier. This may be used as argument of function to set the gain.

Remarks

None.

AFEC_CHANNEL_MASK Enumeration

Identifies AFEC channel mask

File

plib_afec.h

C

```
typedef enum {
  AFEC_CH0_MASK,
  AFEC_CH1_MASK,
  AFEC_CH2_MASK,
  AFEC_CH3_MASK,
  AFEC_CH4_MASK,
  AFEC_CH5_MASK,
  AFEC_CH6_MASK,
  AFEC_CH7_MASK,
  AFEC_CH8_MASK,
  AFEC_CH9_MASK,
  AFEC_CH10_MASK,
  AFEC_CH11_MASK
} AFEC_CHANNEL_MASK;
```

Description

Analog channel mask

This enumeration identifies AFEC channel mask. This may be ORed together to enable/disable multiple channels.

Remarks

None.

AFEC_CHANNEL_NUM Enumeration

Identifies AFEC channel number

File

plib_afec.h

C

```
typedef enum {
  AFEC_CH0,
  AFEC_CH1,
  AFEC_CH2,
  AFEC_CH3,
  AFEC_CH4,
  AFEC_CH5,
  AFEC_CH6,
  AFEC_CH7,
  AFEC_CH8,
  AFEC_CH9,
  AFEC_CH10,
  AFEC_CH11
} AFEC_CHANNEL_NUM;
```

Description

Analog channel numbers

This enumeration identifies AFEC channel numbers. This may be used as argument of function to identify channel number.

Remarks

None.

AFEC_INTERRUPT_MASK Enumeration

Identifies channel interrupt sources mask

File

plib_afec.h

C

```
typedef enum {
  AFEC_INTERRUPT_EOC_0_MASK,
  AFEC_INTERRUPT_EOC_1_MASK,
  AFEC_INTERRUPT_EOC_2_MASK,
  AFEC_INTERRUPT_EOC_3_MASK,
  AFEC_INTERRUPT_EOC_4_MASK,
  AFEC_INTERRUPT_EOC_5_MASK,
  AFEC_INTERRUPT_EOC_6_MASK,
  AFEC_INTERRUPT_EOC_7_MASK,
  AFEC_INTERRUPT_EOC_8_MASK,
  AFEC_INTERRUPT_EOC_9_MASK,
  AFEC_INTERRUPT_EOC_10_MASK,
  AFEC_INTERRUPT_EOC_11_MASK
} AFEC_INTERRUPT_MASK;
```

Description

Interrupt sources mask

This enumeration identifies AFEC channel end of conversion (EOC) interrupt sources mask. This may be ORed together to enable/disable multiple interrupts.

Remarks

None.

Files

Files

Name	Description
plib_afec.h	AFEC peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_afec.h

AFEC peripheral library interface.

Enumerations

	Name	Description
	AFEC_CHANNEL_GAIN	Identifies programmable gain setting
	AFEC_CHANNEL_MASK	Identifies AFEC channel mask
	AFEC_CHANNEL_NUM	Identifies AFEC channel number
	AFEC_INTERRUPT_MASK	Identifies channel interrupt sources mask

Functions

	Name	Description
	AFECx_CallbackRegister	Registers the function to be called from interrupt
	AFECx_ChannelGainSet	Writes the gain of the channel
	AFECx_ChannelOffsetSet	Writes the channel offset
	AFECx_ChannelResultGet	Reads the conversion result of the channel
	AFECx_ChannelResultIsReady	Returns the status of the channel conversion
	AFECx_ChannelsDisable	Disables the ADC channels
	AFECx_ChannelsEnable	Enables the ADC channels
	AFECx_ChannelsInterruptDisable	Disables the ADC interrupt sources
	AFECx_ChannelsInterruptEnable	Enables the ADC interrupt sources
	AFECx_ConversionSequenceSet	Sets the user sequence of the channel conversion
	AFECx_ConversionStart	Starts the ADC conversion of all the enabled channels with the software trigger
	AFECx_Initialize	Initializes given instance of AFEC peripheral.

Structures

	Name	Description
	AFEC_CALLBACK_OBJECT	Callback structure

Types

	Name	Description
	AFEC_CALLBACK	Defines the function pointer data type and function signature for the afec peripheral callback function.

Description

AFEC Peripheral Library Interface Header File

This file defines the interface to the AFEC peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual afecheaders will be generated as required by the MCC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "AFEC" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

File Name

plib_afec.h

Company

Microchip Technology Inc.

Clock Peripheral Library Help

This section provides interface to configure and manage the Clock system of SAM E70/S70/V70/V71 Family.

Introduction

This section provides brief overview of Clock System

Description

Clock system generates and distributes clock for processor and peripherals.

Clock Generators:

- 3 to 20 MHz main crystal oscillator with bypass mode. 12 MHz or 16 MHz needed for USB operations
- High-precision Main RC oscillator with 12 MHz default frequency
- Low-power 32.768 kHz crystal oscillator with bypass mode
- Low-power 32 kHz Slow RC Oscillator (SLCK)
- 160-500 MHz programmable PLL for system clock,
- 480 MHz UTMI PLL providing clock to USB High-speed controller

Clock Distribution

- Processor Clock (HCLK) and Master Clock (MCK), programmable from a few hundred Hz to the maximum operating frequency of the device.
- Master Clock controller provides selection and division of the clock generator's output signals.
- MCK is the source clock for many peripherals. Peripheral Clock Controller enable and disable clocks to different peripheral running of from Master clock.
- Programmable Clock Outputs (PCKx) provides Clock sources independent of MCK and HCLK for USART, UART, TC, Embedded Trace Macrocell (ETM) and CAN Clocks
- Generic Clock (GCLK) with controllable division and ON/OFF control, independent of MCK and HCLK. Provided to selected peripherals.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

The MHC clock Configurator provides graphical user interface to configure clock system.

Using the Library

This section explains how to use the peripheral library

Description

The Clock peripheral library initializes clock system as configured by the user in MHC easy view.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	CLK_Initialize	Initializes hardware of the System Clock and Peripheral Clock.

Description

a) Initialization Function

CLK_Initialize Function

Initializes hardware of the System Clock and Peripheral Clock.

File

plib_clk.h

C

```
void CLK_Initialize();
```

Returns

None.

Description

This function initializes the hardware of System Clock and Peripheral Clocks.

Remarks

None.

Preconditions

None.

Example

```
//Example 1: Do not alter the configuration bit settings
CLK_Initialize ( );
```

Function

void CLK_Initialize (void)

Files

Files

Name	Description
plib_clk.h	Clock PLib interface header.

Description

This section will list only the library's interface header file(s).

plib_clk.h

Clock PLib interface header.

Functions

	Name	Description
	CLK_Initialize	Initializes hardware of the System Clock and Peripheral Clock.

Description

Clock System Service Interface Definition

This file contains the interface definition for the Clock Peripheral Library. It provides a way to interact with the Clock subsystem to manage the timing requests supported by the system.

File Name

plib_clk.h

Company

Microchip Technology Inc.

DACC Peripheral Library Help

This section provides interface to use Digital-to-Analog Converter Controller (DACC) peripheral.

Introduction

This section provides brief overview of DACC peripheral.

Description

The Digital-to-Analog Converter Controller (DACC) peripheral provides the following features:

- Up to two single-ended analog outputs or one differential analog output
- 12-bit resolution.
- Free-running mode, Max speed mode, Trigger mode or Interpolation mode.
- Provides Bypass mode which minimizes power consumption in case of a limited sampling rate conversion.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

The DACC driver provides an interface for the conversion of digital values to analog voltage.

The DAC can operate in free-running mode, where a write to the conversion-data register initiates an update on the analog output, or in triggered mode, where updates are initiated by a rising edge on an internal or external synchronization signal.

The user must ensure that new data is not written to the DAC before the last conversion is complete as illustrated below.

```
/* Write new data if DAC is ready
 if (DACCx_IsReady(DACC_CHANNEL_0))
 {
    DACCx_DataWrite (DACC_CHANNEL_1, 0xff)
 }
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Functions

	Name	Description
	DACCx_Initialize	Initializes DACC x module of the device

b) Transaction Functions

	Name	Description
	DACCx_DataWrite	Converts a Digital data to Analog value

c) Status Functions

	Name	Description
	DACCx_IsReady	Returns the status of readiness of Channel x of DACC x module for new conversion request

d) Data Types and Constants

	Name	Description
	DACC_CHANNEL_NUM	Identifies the Channel index of DACC module

a) Initialization Functions

DACCx_Initialize Function

Initializes DACC x module of the device

File

plib_dacc.h

C

```
void DACCx_Initialize();
```

Returns

None.

Description

This function initializes DACC x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, APIs can be used to convert the data.

Remarks

None.

Preconditions

MHC GUI should be configured with the right values.

Example

```
DACC0_Initialize();
```

Function

void DACCx_Initialize (void);

c) Status Functions

DACCx_IsReady Function

Returns the status of readiness of Channel x of DACC x module for new conversion request

File

plib_dacc.h

C

```
bool DACCx_IsReady(DACC_CHANNEL_NUM channel);
```

Returns

- True - When DACC is ready to accept new data for conversion.
- False - When DACC is busy and can not accept new data for conversion

Description

This function returns the status of Channel x of DACC x module, is it ready to accept new conversion requests or not.

Remarks

None.

Preconditions

DACCx_Initialize must have been called for the associated DACC instance.

Example

```
bool status = false;

if (true == DACC0_IsReady (DACC_CHANNEL_1))
{
   DACC0_DataWrite (DACC_CHANNEL_1, 0xff)
}
else
```

```
{
   //DACC is not ready to accept new conversion request
   //User Application code
}
```

Parameters

Parameters	Description
channel	Points to DACC Channel.

Function

bool DACCx_IsReady (DACC_CHANNEL_NUM channel);

b) Transaction Functions

DACCx_DataWrite Function

Converts a Digital data to Analog value

File

plib_dacc.h

C

```
void DACCx_DataWrite(DACC_CHANNEL_NUM channel, uint32_t data);
```

Returns

None

Description

This function converts a Digital data to Analog value. The behavior of this function call will vary based on the mode selected within MHC.

Remarks

None.

Preconditions

DACCx_Initialize must have been called.

Example

```
char myData[COUNT] = {"0xff","0x3E","0x7A","0x3F"};//COUNT is user
                                                   //dependent
bool status = false;

//considering count = 4
for (uint8_t i = 0; i<4; i++)
{
   DACC0_DataWrite (DACC_CHANNEL_0, myData[i]);
}
```

Parameters

Parameters	Description
channel	Points to DACC Channel
data	Digital data to be converted to Analog value.

Function

void DACCx_DataWrite (DACC_CHANNEL_NUM channel, uint32_t data)

d) Data Types and Constants

DACC_CHANNEL_NUM Enumeration

Identifies the Channel index of DACC module

File

plib_dacc.h

C

```
typedef enum {
  DACC_CHANNEL_0,
  DACC_CHANNEL_1
} DACC_CHANNEL_NUM;
```

Members

Members	Description
DACC_CHANNEL_0	DACC Channel 0
DACC_CHANNEL_1	DACC Channel 1

Description

DACC Channel Index

This enumeration identifies the Channel index of DACC module.

Remarks

None.

Files

Files

Name	Description
plib_dacc.h	DACC peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_dacc.h

DACC peripheral library interface.

Enumerations

	Name	Description
	DACC_CHANNEL_NUM	Identifies the Channel index of DACC module

Functions

	Name	Description
	DACCx_DataWrite	Converts a Digital data to Analog value
	DACCx_Initialize	Initializes DACC x module of the device

	DACCx_IsReady	Returns the status of readiness of Channel x of DACC x module for new conversion request

Description

Digital-to-Analog Converter Controller (DACC) Peripheral Library(PLIB) Interface Header File

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Digital-to-Analog Converter Controller (DACC). This file must not be included in any MPLAB Project.

File Name

plib_dacc.h

Company

Microchip Technology Inc.

EFC Peripheral Library Help

This section provides interface to use Embedded Flash Controller (EFC) peripheral.

Introduction

This section provides brief overview of EFC peripheral.

Description

The Enhanced Embedded Flash Controller (EEFC) manages the programming, erasing, locking and unlocking sequences of the Flash using a full set of commands.

Using the Library

This section explains how to use the peripheral library

Description

This library can be used in two different modes:

- **Interrupt Mode:** In this mode of operation EFC Interrupts are enabled and application will get callback after transfer completion
- **Non-Interrupt Mode:** In this mode of operation EFC Interrupts are disabled and application needs to poll for transaction completion.

Following table describes which function is supported in which mode of EFC Peripheral Library:

Functions	Interrupt Mode	Non-Interrupt Mode
EFCx_CallbackRegister	Yes	No
EFCx_Read	Yes	Yes
EFCx_SectorErase	Yes	Yes
EFCx_PageWrite	Yes	Yes
EFCx_QuadWordWrite	Yes	Yes
EFCx_RegionUnlock	Yes	Yes
EFCx_RegionLock	Yes	Yes
EFCx_IsBusy	Yes	Yes
EFCx_ErrorGet	Yes	Yes

Interrupt mode:

When "Interrupt Mode" option is selected in MHC GUI, the generated code for that particular EFC PLIB instance will be Non-blocking in nature. In this particular mode, application will call the EFC library transaction functions and will be get the control

back with request accepted or rejected.

Once the transaction is completed Application will be notified through the call back registered using EFCx_CallbackRegister().

To know the status of transaction if it was success or failure Application can call EFCx_ErrorGet().

Non-Interrupt mode:

When "Interrupt Mode" option is De-Selected in MHC GUI, the generated code for that particular EFC PLIB instance will still be Non-blocking in nature. In this particular mode, application will call the EFC library transaction functions and will be get the control back with request accepted or rejected.

Application needs to poll for transaction completion using EFCx_IsBusy() API.

To know the status of transaction if it was success or failure Application can call EFCx_ErrorGet().

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

d) Data Types and Constants

	Name	Description
	EFC_CALLBACK	Defines the data type and function signature for the EFC peripheral callback function.
	EFC_ERROR	Defines the data type for the EFC Error.

a) Setup Functions

Functions

	Name	Description
	EFCx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the operation is complete.

Description

EFCx_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the operation is complete.

File

plib_efc.h

C

```
void EFCx_CallbackRegister(EFC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the EFCx is ready to receive new command.

It also passes a context value that is passed into the function when it is called.

This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. Note that the value of NULL will still be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the EFC_CALLBACK type definition for additional information.

Preconditions

None.

Example

```
EFCx_CallbackRegister(MyCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the EFC_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void EFCx_CallbackRegister(EFC_CALLBACK callback, uintptr_t context)

b) Transaction Functions

Functions

	Name	Description
	EFCx_RegionLock	Locks a Flash region.
	EFCx_RegionUnlock	Unlocks a Flash region.
	EFCx_SectorErase	Erases a Sector in the FLASH.
	EFCx_PageWrite	Writes data of size equivalent to page size to a given FLASH address.
	EFCx_QuadWordWrite	Writes a 128-bit data to a given address in FLASH memory.
	EFCx_Read	Reads length number of bytes from a given address in FLASH memory.

Description

EFCx_RegionLock Function

Locks a Flash region.

File

plib_efc.h

C

```
void EFCx_RegionLock(uint32_t address);
```

Returns

None.

Description

This function is used to lock a certain region of on-chip flash. It takes in address as a parameter and locks the region associated with it.

Preconditions

Validate if EFCx controller is ready to accept new request by calling EFCx_IsBusy()

Example

```
while(EFC0_IsBusy());
EFCx0_RegionLock(0x00500000);
```

Function

void EFCx_RegionLock(uint32_t address);

EFCx_RegionUnlock Function

Unlocks a Flash region.

File

plib_efc.h

C

```
void EFCx_RegionUnlock(uint32_t address);
```

Returns

None.

Description

This function is used to Unlock a certain region of on-chip flash. It takes in address as a parameter and unlocks the region associated with it.

Preconditions

Validate if EFCx controller is ready to accept new request by calling EFCx_IsBusy()

Example

```
while(EFC0_IsBusy());
EFCx0_RegionUnlock(0x00500000);
```

Parameters

Parameters	Description
address	Address of the page to be unlocked

Function

void EFCx_RegionUnlock(uint32_t address);

EFCx_SectorErase Function

Erases a Sector in the FLASH.

File

plib_efc.h

C

```
bool EFCx_SectorErase(uint32_t address);
```

Returns

- true: On successfull erase request.
- false: On failure to send erase request.

Description

This function is used to erase a sector (collection of pages).

Remarks

Application needs to poll for busy bit or wait for callback to be called before sending next request.

Preconditions

Validate if EFCx controller is ready to accept new request by calling EFCx_IsBusy()

Example

```
EFC0_SectorErase( 0x500000 );
while(EFC0_IsBusy());
```

Parameters

Parameters	Description
address	FLASH address to be Erased

Function

bool EFCx_SectorErase(uint32_t address)

EFCx_PageWrite Function

Writes data of size equivalent to page size to a given FLASH address.

File

plib_efc.h

C

```
bool EFCx_PageWrite(uint32_t* data, uint32_t address);
```

Returns

- true: On successfull write request.
- false: On failure to send write request.

Description

This function takes a 32-bit address, a pointer to data of size equivalent to page size and writes it to the given location in FLASH memory.

Remarks

Application needs to poll for busy bit or wait for callback to be called before sending next request.

Preconditions

Validate if EFCx controller is ready to accept new request by calling EFCx_IsBusy()

The Page to be written should be in Erased State.

Example

```
EFC0_PageWrite( 0x500000, &buffer);
while(EFC0_IsBusy());
```

Parameters

Parameters	Description
address	FLASH address to be modified
data	pointer to data buffer

Function

bool EFCx_PageWrite(uint32_t* data, uint32_t address)

EFCx_QuadWordWrite Function

Writes a 128-bit data to a given address in FLASH memory.

File

plib_efc.h

C

```
bool EFCx_QuadWordWrite(uint32_t* data, uint32_t address);
```

Returns

- true: On successfull write request.
- false: On failure to send write request.

Description

This function takes a 32-bit address, a pointer to the 128-bit data and writes it to the given location in FLASH memory.

Remarks

Application needs to poll for busy bit or wait for callback to be called before sending next request.

Preconditions

Validate if EFCx controller is ready to accept new request by calling EFCx_IsBusy()

The Memory to be written should be in Erased State.

Example

```
EFC0_QuadWordWrite( 0x500000, &buffer);
while(EFC0_IsBusy());
```

Parameters

Parameters	Description
address	FLASH address to be modified
data	pointer to 128-bit data buffer

Function

bool EFCx_QuadWordWrite(uint32_t* data, uint32_t address)

EFCx_Read Function

Reads length number of bytes from a given address in FLASH memory.

File

plib_efc.h

C

```
bool EFCx_Read(uint32_t * data, uint32_t length, uint32_t address);
```

Returns

- true: On successfull read
- false: On failure to read.

Description

Reads length number of bytes from a given address in FLASH memory into the user buffer.

Preconditions

None

Example

```
uint8_t buffer[256];
EFC0_Read( &buffer, 256, 0x500000);
```

Parameters

Parameters	Description
data	pointer to user data buffer
length	Number of bytes to read
address	FLASH address to be read from

Function

bool EFCx_Read(uint32_t *data, uint32_t length, uint32_t address)

c) Status Functions

Functions

	Name	Description
	EFCx_IsBusy	Returns the current status of EFCx controller.
	EFCx_ErrorGet	Returns the error encountered by EFCx controller.

Description

EFCx_IsBusy Function

Returns the current status of EFCx controller.

File

plib_efc.h

C

```
bool EFCx_IsBusy();
```

Returns

- true: If EFCx peripheral is busy doing an operation.
- false: If EFCx peripheral is ready to accept new request..

Description

This function is used to check is EFCx is busy or not

Preconditions

None.

Example

```
while(EFC0_IsBusy());
EFC0_SectorErase( 0x500000 );
```

Function

bool EFCx_IsBusy(void)

EFCx_ErrorGet Function

Returns the error encountered by EFCx controller.

File

plib_efc.h

C

```
uint32_t EFCx_ErrorGet();
```

Returns

uint32_t - EFCx controller error type

Description

This function is used to check error encountered by EFCx controller.

Return value can be mapped to EFC_ERROR to identify particular error.

Preconditions

None.

Example

```
uint32_t error = EFCx_ErrorGet();
```

Function

EFC_ERROR EFCx_ErrorGet(void)

d) Data Types and Constants

EFC_CALLBACK Type

Defines the data type and function signature for the EFC peripheral callback function.

File

plib_efc.h

C

```
typedef void (* EFC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

This data type defines the function signature for the EFC peripheral callback function. The EFC peripheral will callback the client's function with this signature when EFC is ready to accept new operation.

Preconditions

EFCx_CallbackRegister must have been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

EFC_ERROR Enumeration

Defines the data type for the EFC Error.

File

plib_efc.h

C

```
typedef enum {
  EFC_ERROR_NONE = 0x1,
  EFC_CMD_ERROR = 0x2,
  EFC_LOCK_ERROR = 0x4,
  EFC_FLERR_ERROR = 0x8,
  EFC_ECC_ERROR = 0xF0000
} EFC_ERROR;
```

Members

Members	Description
EFC_ERROR_NONE = 0x1	No Error Occured
EFC_CMD_ERROR = 0x2	In-valid command
EFC_LOCK_ERROR = 0x4	Flash region is locked
EFC_FLERR_ERROR = 0x8	Flash Error
EFC_ECC_ERROR = 0xF0000	Flash Encountered an ECC error

Description

This enum is used to define error encountered during last Flash operation.

Remarks

None.

Files

Files

Name	Description
plib_efc.h	Interface definition of EFC Plib.

Description

This section will list only the library's interface header file(s).

plib_efc.h

Interface definition of EFC Plib.

Enumerations

	Name	Description
	EFC_ERROR	Defines the data type for the EFC Error.

Functions

	Name	Description
	EFCx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the operation is complete.
	EFCx_ErrorGet	Returns the error encountered by EFCx controller.
	EFCx_IsBusy	Returns the current status of EFCx controller.
	EFCx_PageWrite	Writes data of size equivalent to page size to a given FLASH address.
	EFCx_QuadWordWrite	Writes a 128-bit data to a given address in FLASH memory.
	EFCx_Read	Reads length number of bytes from a given address in FLASH memory.
	EFCx_RegionLock	Locks a Flash region.
	EFCx_RegionUnlock	Unlocks a Flash region.
	EFCx_SectorErase	Erases a Sector in the FLASH.

Types

	Name	Description
	EFC_CALLBACK	Defines the data type and function signature for the EFC peripheral callback function.

Description

Interface definition of EFC PLIB.

This file defines the interface for the EFC Plib. It allows user to Program, Erase and lock the on-chip FLASH memory.

File Name

plib_efc.h

Company

Microchip Technology Inc.

I2SC Peripheral Library Help

This section provides interface to use I2SC Peripheral.

Introduction

This library provides an interface to use the I2SC (Inter-IC Sound Controller) module.

Description

The I2SC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2SC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- DI: Serial Data Input for receiving sample digital audio data (ADCDAT)
- DO: Serial Data Output for transmitting digital audio data (DACDAT)
- CK: Serial Clock, also known as bit clock (BCLK)
- WS: Word Select, also known as Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

In Master mode, the module generates both the BCLK on the CK pin and the LRCK on the WS pin. In certain devices, while in

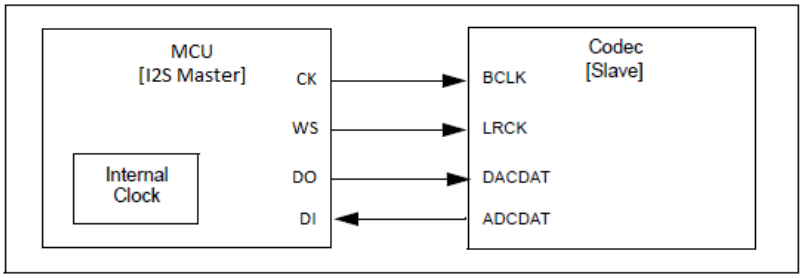
Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of CK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out DO, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

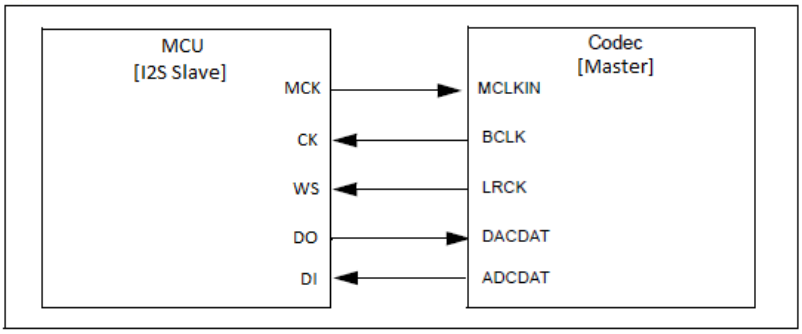
Master Mode

Master Generating its Own Clock – Output BCLK and LRCK



Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out



Audio Formats

The I2SC Module supports just one audio format, I2S. Left Justified format is not available.

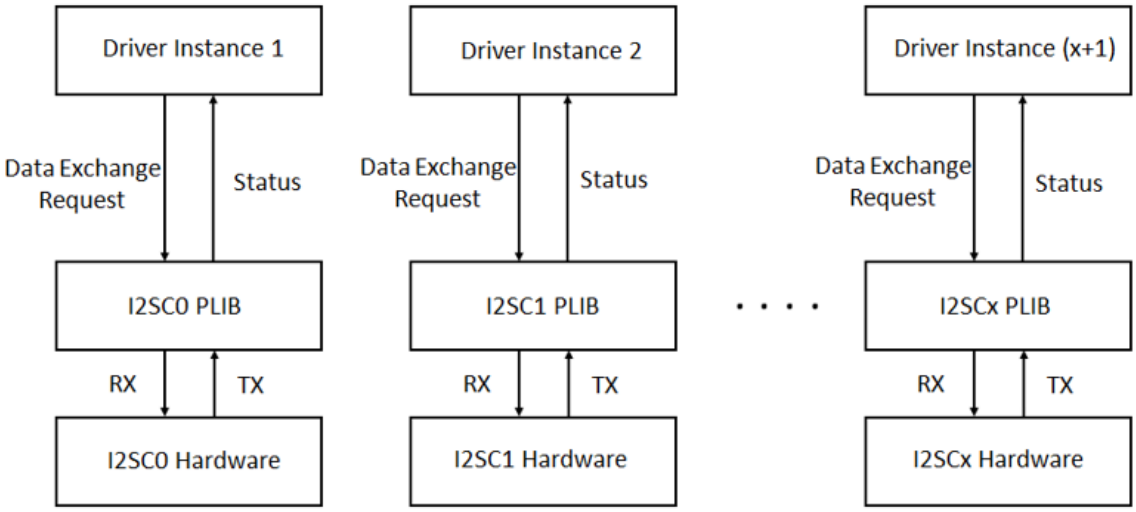
Using the Library

This topic describes the basic architecture of the I2SC Peripheral Library and provides information and examples on how to use it.

Description

Abstraction Model

The I2SC module sits between the I2S Driver, and the actual hardware.



Interface Header File: `plib_i2sc.h`

The interface to the I2SC Peripheral Library is defined in the `plib_i2sc.h` header file. Any C language source (`.c`) file that uses the I2SC Peripheral Library should include plib_i2sc.h.

Library Source Files:

The I2SC Peripheral Library library source files are provided in the `csp/peripheral/i2sc_xxxx/src` or `csp/peripheral/i2sc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

Usage Model

The only usage model for the I2SC Peripheral Library is to use a Interrupt/DMA model. Therefore the remaining functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right clock (LRCLK) for the I2S stream.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

Configuring the Library

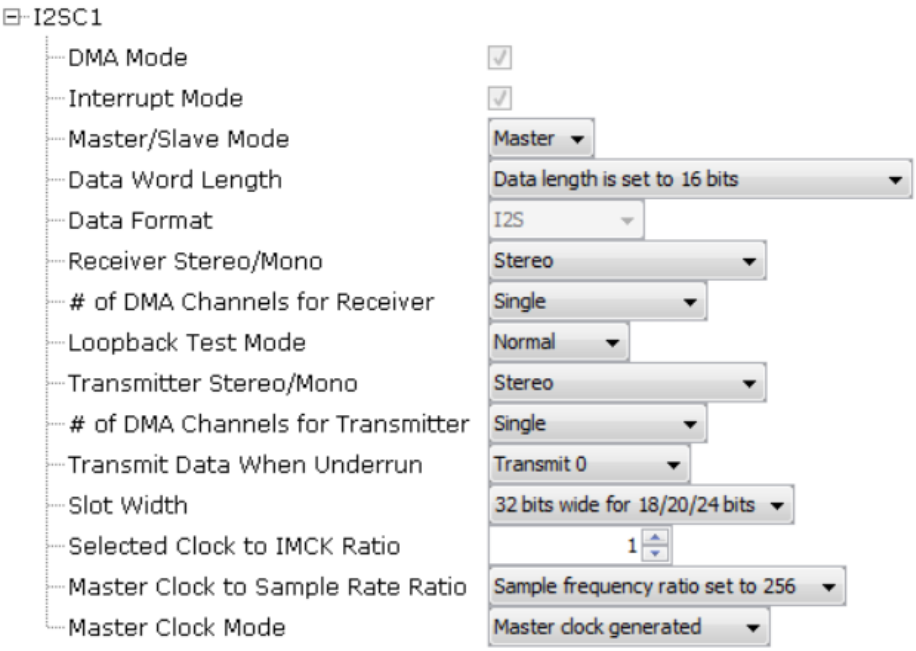
This section describes how to configure the peripheral library using MHC

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the I2SC peripheral by clicking on the appropriate instance under *Peripherals->I2SC* in the Available Components section of MHC. (The I2SC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2SCx peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options (example shown for instance I2SC1):



Default values are shown. As mentioned earlier, **DMA** and **Interrupt Mode** are always enabled.

Master/Slave Mode can be either Master -- the I2SC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks

Data Word Length can be selected from 8 to 32-bits

Data Format is currently always I2S

Receiver Stereo/Mono can be either Stereo or Mono (left channel dup'ed to right)

# of DMA Channels for Receiver can either Single or Multiple (1/channel)

Loopback Test Mode can be Normal or Loop mode

Transmitter Stereo/Mono can be either Stereo or Mono (left channel dup'ed to right)

Transmit Data When Underrun can be Transmit 0 or Transmit previous

Slot Width can be 24 or 32-bits wide

The following options are only shown if the Master/Slave mode is Master:

Selected Clock for IMCK Ratio is selectable as any integer

Master Clock to Sample Rate Ratio can be set from 32 to 2048

Master Clock Mode can be either Master Clock Generated or not

Library Interface

a) Initialization Function

	Name	Description
	I2SCx_Initialize	Initializes I2SCx module of the device

b) Transaction Functions

	Name	Description
	I2SCx_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal

Description

This section describes the Application Programming Interface (API) functions of the I2SC Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

I2SCx_Initialize Function

Initializes I2SCx module of the device

File

plib_i2sc.h

C

```
void I2SCx_Initialize();
```

Returns

None.

Description

This function initializes I2SCx module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Remarks

This function must be called only once and before any other I2SC function is called.

Preconditions

MHC GUI should be configured with the right values.

Example

```
I2SC0_Initialize();
```

Function

void I2SCx_Initialize (void);

b) Transaction Functions

I2SCx_LRCLK_Get Function

Get the level of the I2S LRCLK (left/right clock) signal

File

plib_i2sc.h

C

```
uint32_t I2SCx_LRCLK_Get();
```

Returns

State of the LRCLK pin for the I2SCx module -- 1 if high, 0 if low

Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize itself to the LRCLK signal.

Remarks

None.

Preconditions

None.

Example

```
I2SC1_LRCLK_Get();
```

Function

uint32_t I2SCx_LRCLK_Get(void);

Files

Files

Name	Description
plib_i2sc.h	I2SC PLIB Header File for documentation

Description

This section will list only the library's interface header file(s).

plib_i2sc.h

I2SC PLIB Header File for documentation

Functions

	Name	Description
	I2SCx_Initialize	Initializes I2SCx module of the device
	I2SCx_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal

Description

I2SC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of an Inter-IC Sound Controller (I2SC). This file must not be included in any MPLAB Project.

File Name

plib_i2sc.h

Company

Microchip Technology Inc.

MPU Peripheral Library Help

This section provides interface to use Memory Protection Unit (MPU)

Introduction

This section provides brief overview of MPU

Description

The Memory Protection Unit (MPU) divides the memory map into 8 or 16 regions, and defines the start address, size, memory type, data access permission and instruction access permission.

Using the Library

This section explains how to use the peripheral library

Description

Memory Protection Unit will be initialized as part of system Initialization with configurations provided to the memory regions through MHC.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

The Memory Protection Unit (MPU) divides the memory map into 8 or 16 regions, and defines the start address, size, memory attributes, data access permission and instruction access permission.

Memory Attributes:

Memory attributes can used to define memory type and cache behaviour.

- **Strongly Ordered:** The processor preserves transaction order and the write buffers are not permitted.
- **Device Memory:** The processor preserves transaction order and the write buffers are permitted for device access. Device memory is used for memory mapped peripheral registers.
- **Normal Memory**: The processor can re-order transactions for efficiency, or perform speculative reads.
  - Write through cache: On hits it writes to the cache and the main memory, On misses it updates the block in the main memory not bringing that block to the cache.
  - Write-back cache: On hits it writes to the cache setting dirty bit for the block, the main memory is not updated. On misses it updates the block in the main memory not bringing that block to the cache.
  - Write-back with write allocate: On hits it writes to the cache setting dirty bit for the block, the main memory is not updated. On misses it updates the block in the main memory and brings the block to the cache
  - Non-cacheable memory

Data Access Permission

Data access permission can be used to define flash as read-only to protect from accidental writes and define non-accessible region to detect stack overflow conditions.

- Data access permission(AP[2:0])

AP[2:0]	Access permission
000	User: No Access, Privileged: No Access
001	User: No Access, Privileged: Read/Write
010	User: Read only, Privileged: Read/Write
011	User: Read/Write, Privileged: Read/Write
100	Reserved
101	User: No Access, Privileged: Read only
110	User: Read only, Privileged: Read only
111	User: Read only, Privileged: Read only

Instruction Access Permission:

Instruction access permission can be used to define non-executable memory region.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

Name	Description
MPU_Initialize	Initializes Memory Protection for configured regions.

a) Initialization Function

MPU_Initialize

Initializes Memory Protection for configured regions.

C

```
void MPU_Initialize( void );
```

Returns

None.

Description

This function initializes the Memory Protection Unit for configured regions.

Attributes for the Region to be configured

- Start address of the Region to be protected
- Region Size
- Type of the memory with cache policy to be used
- Data access rights

Preconditions

None.

Example

```
MPU_Initialize();
```

NVIC Peripheral Library Help

This section provides an interface to use NVIC Peripheral Library

Introduction

This section provides brief overview of the NVIC peripheral

Description

The Nested Vectored Interrupt Controller (NVIC) supports:

- Tightly coupled interrupt controller provides low interrupt latency

- An implementation-defined number of interrupts, in the range 1-240 interrupts.
- A programmable priority level for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Supports Vectored interrupt where each interrupt has its own vector stored in a vector table.
- Relocatable Vector table defines the entry address of the processor exceptions and the peripheral interrupts to automatically service the required exception or interrupt
- Non-Maskable Interrupt
- Software interrupt generation
- The processor automatically stacks its state on exception entry and unstacks this state on exception exit, with no instruction overhead. This provides low latency exception handling

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This Section provides information on how to use the NVIC Peripheral Library.

Description

The NVIC peripheral library initializes interrupt controller as configured by the user in MHC. User can enable interrupt, configures priority and specify interrupt vector in the MHC easy view.

Library Interface

This section describes the Application Programming Interface (API) functions of the NVIC Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

Name	Description
NVIC_Initialize	Initializes the Interrupt controller (NVIC)

a) Initialization Function

NVIC_Initialize

Initializes the Interrupt controller (NVIC)

Returns

None.

Description

This function initializes the interrupt controller (NVIC) as configured by the user in MHC.

Preconditions

None.

Example

```
NVIC_Initialize();
```

Function

void NVIC_Initialize(void)

PIO Peripheral Library Help

This section provides interface to use Parallel Input Output (PIO) controller.

Introduction

This section provides brief overview of Parallel Input Output (PIO) controller.

Description

The Parallel Input/Output Controller (PIO) manages up to 32 fully programmable input/output lines. Each I/O line may be dedicated as a general-purpose I/O or be assigned to a function of an embedded peripheral. This ensures effective optimization of the pins of the product.

Each I/O line of the PIO Controller features the following:

- An input change interrupt enabling level change detection on any I/O line
- Additional Interrupt modes enabling rising edge, falling edge, low-level or high-level detection on any I/O line
- A glitch filter providing rejection of glitches lower than one-half of peripheral clock cycle
- A debouncing filter providing rejection of unwanted pulses from key or push button operations
- Multi-drive capability similar to an open drain I/O line
- Control of the I/O line pullup and pulldown

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library

Using the Library

This section explains how to use the Peripheral Library.

Description

PIO PLIB provides two kinds of functions which can be used to control PIO Pins:

1. Pin Functions
2. Port Functions

Pin functions take a particular pin as input and operates only on that pin without affecting any other pins. Whereas port functions can operate on multiple pins of the same port together.

PIO PLIB also provides feature of callback on individual pins. Callback can be registered using PIO_PinInterruptCallbackRegister function for the pins for which interrupt has been enabled in MHC Pin Manager.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialize Function

	Name	Description
	PIO_Initialize	Initialize the PIO library.

b) Pin Functions

	Name	Description
	PIO_PinClear	Clears the selected pin.
	PIO_PinInputEnable	Enables selected IO pin as input.
	PIO_PinInterruptDisable	Disables IO interrupt on selected IO pin.
	PIO_PinInterruptEnable	Enables IO interrupt on selected IO pin.
	PIO_PinLatchRead	Read the value driven on the selected pin.
	PIO_PinOutputEnable	Enables selected IO pin as output.
	PIO_PinRead	Read the selected pin value.
	PIO_PinSet	Sets the selected pin.
	PIO_PinToggle	Toggles the selected pin.
	PIO_PinWrite	Writes the selected pin.
	PIO_PinInterruptCallbackRegister	Allows application to register callback for every pin.

c) Port Functions

	Name	Description
	PIO_PortClear	Clear the selected IO pins of a port.
	PIO_PortInputEnable	Enables selected IO pins of a port as input.
	PIO_PortInterruptDisable	Disables IO interrupt on selected IO pins of a port.
	PIO_PortInterruptEnable	Enables IO interrupt on selected IO pins of a port.
	PIO_PortLatchRead	Read the latched value on all the I/O lines of the selected port.
	PIO_PortOutputEnable	Enables selected IO pins of a port as output(s).
	PIO_PortRead	Read all the I/O lines of the selected port port.
	PIO_PortSet	Set the selected IO pins of a port.
	PIO_PortToggle	Toggles the selected IO pins of a port.
	PIO_PortWrite	Write the value on the masked I/O lines of the selected port.

d) Data Types and Constants

	Name	Description
	PIO_PIN_CALLBACK	Pointer to a PIO Pin-Event handler function.
	PIO_PIN	Identifies the available PIO port pins.
	PIO_PORT	Identifies the available PIO Ports.

a) Initialize Function

PIO_Initialize Function

Initialize the PIO library.

File

plib_pio.h

C

```
void PIO_Initialize();
```

Returns

None.

Description

This function initializes the PIO library and all ports and pins configured in the MHC Pin Manager.

Remarks

None.

Preconditions

None.

Example

```
PIO_Initialize();
```

Function

void PIO_Initialize(void)

b) Pin Functions

PIO_PinClear Function

Clears the selected pin.

File

plib_pio.h

C

```
void PIO_PinClear(PIO_PIN pin);
```

Returns

None.

Description

This function drives '0' on the selected I/O line/pin.

Remarks

None.

Preconditions

None.

Example

```
PIO_PinClear(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinClear(PIO_PIN pin)

PIO_PinInputEnable Function

Enables selected IO pin as input.

File

plib_pio.h

C

```
void PIO_PinInputEnable(PIO_PIN pin);
```

Returns

None.

Description

This function enables selected IO pin as input.

Remarks

None.

Preconditions

None.

Example

```
PIO_PinInputEnable(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinInputEnable(PIO_PIN pin)

PIO_PinInterruptDisable Function

Disables IO interrupt on selected IO pin.

File

plib_pio.h

C

```
void PIO_PinInterruptDisable(PIO_PIN pin);
```

Returns

None.

Description

This function disables IO interrupt on selected IO pin.

Remarks

None.

Preconditions

Corresponding pin must be configured in interrupt mode in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
PIO_PinInterruptDisable(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinInterruptDisable(PIO_PIN pin)

PIO_PinInterruptEnable Function

Enables IO interrupt on selected IO pin.

File

plib_pio.h

C

```
void PIO_PinInterruptEnable(PIO_PIN pin);
```

Returns

None.

Description

This function enables interrupt on selected IO pin.

Remarks

None.

Preconditions

Corresponding pin must be configured in interrupt mode in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
PIO_PinInterruptEnable(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinInterruptEnable(PIO_PIN pin)

PIO_PinLatchRead Function

Read the value driven on the selected pin.

File

plib_pio.h

C

```
bool PIO_PinLatchRead(PIO_PIN pin);
```

Returns

Returns the value driven on the selected I/O pin.

Description

This function reads the data driven on the selected I/O line/pin. Whatever data is written/driven on I/O line by using any of the PIO PLIB APIs, will be read by this API.

Remarks

To read actual pin value, PIO_PinRead API should be used.

Preconditions

None.

Example

```
bool value;
value = PIO_PinLatchRead(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

bool PIO_PinLatchRead (PIO_PIN pin)

PIO_PinOutputEnable Function

Enables selected IO pin as output.

File

plib_pio.h

C

```
void PIO_PinOutputEnable(PIO_PIN pin);
```

Returns

None.

Description

This function enables selected IO pin as output.

Remarks

None.

Preconditions

None.

Example

```
PIO_PinOutputEnable(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinOutputEnable(PIO_PIN pin)

PIO_PinRead Function

Read the selected pin value.

File

plib_pio.h

C

```
bool PIO_PinRead(PIO_PIN pin);
```

Returns

Returns the read value of the selected I/O pin.

Description

This function reads the selected pin value. it reads the value regardless of pin configuration, whether uniquely as an input, or driven by the PIO Controller, or driven by peripheral.

Remarks

To read the latched value on this pin, PIO_PinLatchRead API should be used.

Preconditions

Reading the I/O line levels requires the clock of the PIO Controller to be enabled, otherwise this API reads the levels present on the I/O line at the time the clock was disabled.

Example

```
bool value;
value = PIO_PinRead(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

bool PIO_PinRead(PIO_PIN pin)

PIO_PinSet Function

Sets the selected pin.

File

plib_pio.h

C

```
void PIO_PinSet(PIO_PIN pin);
```

Returns

None.

Description

This function drives '1' on the selected I/O line/pin.

Remarks

None.

Preconditions

None.

Example

```
PIO_PinSet(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinSet(PIO_PIN pin)

PIO_PinToggle Function

Toggles the selected pin.

File

plib_pio.h

C

```
void PIO_PinToggle(PIO_PIN pin);
```

Returns

None.

Description

This function toggles/inverts the value on the selected I/O line/pin.

Remarks

None.

Preconditions

PIO_Initialize() must have been called.

Example

```
PIO_PinToggle(PIO_PIN_PB3);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN

Function

void PIO_PinToggle(PIO_PIN pin)

PIO_PinWrite Function

Writes the selected pin.

File

plib_pio.h

C

```
void PIO_PinWrite(PIO_PIN pin, bool value);
```

Returns

None.

Description

This function writes/drives the "value" on the selected I/O line/pin.

Remarks

None.

Preconditions

The desired pin must be configured as an output pin in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
PIO_PinWrite(PIO_PIN_PB3, true);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN
value	value to be written on the selected pin: • true - set pin to high (1). • false - clear pin to low (0).

Function

void PIO_PinWrite(PIO_PIN pin, bool value)

PIO_PinInterruptCallbackRegister Function

Allows application to register callback for every pin.

File

plib_pio.h

C

```
bool PIO_PinInterruptCallbackRegister(PIO_PIN pin, const PIO_PIN_CALLBACK callBack, uintptr_t
context);
```

Returns

Callback registration status:

- true: Callback was successfully registered
- false: Callback was not registered

Description

This function allows application to register an event handling function for the PLIB to call back when I/O interrupt occurs on the selected pin.

At any point if application wants to stop the callback, it can call this function with "eventHandler" value as NULL.

If a pin is not configured for interrupt in Pin Manager and yet its callback registration is attempted using this API, then registration doesn't happen and API returns false indicating the same.

Remarks

None.

Preconditions

Corresponding pin must be configured in interrupt mode in MHC Pin Manager.

The PIO_Initialize function must have been called.

Example

```
if(PIO_PinInterruptCallbackRegister(PIO_PIN_PB3, &APP_PinHandler, NULL))
{
    // callback got registered
}
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the application unchanged, when the eventHandler function is called. It can be used to identify any application specific value.

Function

```
bool PIO_PinInterruptCallbackRegister(
    PIO_PIN pin,
const     PIO_PIN_CALLBACK callBack,
uintptr_t context
);
```

c) Port Functions

PIO_PortClear Function

Clear the selected IO pins of a port.

File

plib_pio.h

C

```
void PIO_PortClear(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function clears (to '0') the selected IO pins of a port.

Remarks

If the port has less than 32-bits, unimplemented pins will be ignored.

Implemented pins are Right aligned in the 32-bit value.

Preconditions

None.

Example

```
// Clear PC5 and PC7 pins to 0
PIO_PortClear(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be cleared. • 1's - Will clear corresponding IO pins to low (to 0). • 0's - Will remain unchanged.

Function

void PIO_PortClear(PIO_PORT port, uint32_t mask)

PIO_PortInputEnable Function

Enables selected IO pins of a port as input.

File

plib_pio.h

C

```
void PIO_PortInputEnable(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function enables selected IO pins of a port as input.

Remarks

None.

Preconditions

None.

Example

```
// Make PC5 and PC7 pins as input
PIO_PortInputEnable(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One or more of the of the IO ports from the enum PIO_PORT.
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be setup as inputs. • 1's - Will set corresponding IO pins as input(s). • 0's - Will cause the direction of the corresponding IO pins to remain unchanged.

Function

void PIO_PortInputEnable(PIO_PORT port, uint32_t mask)

PIO_PortInterruptDisable Function

Disables IO interrupt on selected IO pins of a port.

File

plib_pio.h

C

```
void PIO_PortInterruptDisable(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function disables IO interrupt on selected IO pins of selected port.

Remarks

None.

Preconditions

Corresponding pins must be configured in interrupt mode in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
// Disable IO interrupt for PB9 and PB1 pins
PIO_PortInterruptDisable(PIO_PORT_C, 0x0202);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	Is a 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will have interrupt disabled. • The bit positions of mask value which are set as 1, IO interrupt of corresponding IO pin of the selected port will be disabled. • The bit positions of mask value which are cleared to 0, IO interrupt of corresponding IO pin of the selected port will remain unchanged.

Function

void PIO_PortInterruptDisable(PIO_PORT port, uint32_t mask)

PIO_PortInterruptEnable Function

Enables IO interrupt on selected IO pins of a port.

File

plib_pio.h

C

```
void PIO_PortInterruptEnable(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function enables interrupt on selected IO pins of selected port.

Remarks

None.

Preconditions

Corresponding pins must be configured in interrupt mode in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
// Enable IO interrupt for PC5 and PC7 pins
PIO_PortInterruptEnable(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	Is a 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will have interrupt enabled. • The bit positions of mask value which are set as 1, IO interrupt of corresponding IO pin of the selected port will be enabled. • The bit positions of mask value which are cleared to 0, IO interrupt of corresponding IO pin of the selected port will remain unchanged.

Function

void PIO_PortInterruptEnable(PIO_PORT port, uint32_t mask)

PIO_PortLatchRead Function

Read the latched value on all the I/O lines of the selected port.

File

plib_pio.h

C

```
uint32_t PIO_PortLatchRead(PIO_PORT port);
```

Returns

Returns the data driven on all the I/O lines of the selected port.

Description

This function reads the latched data values on all the I/O lines of the selected port. Bit values returned in each position indicate corresponding pin levels.

- 1 - Pin is high.
- 0 - Pin is low.

Remarks

If the port has less than 32-bits, unimplemented pins will read as low (0).

Implemented pins are Right aligned in the 32-bit return value.

Preconditions

None.

Example

```
uint32_t value;
value = PIO_PortLatchRead(PIO_PORT_C);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT

Function

uint32_t PIO_PortLatchRead (PIO_PORT port)

PIO_PortOutputEnable Function

Enables selected IO pins of a port as output(s).

File

plib_pio.h

C

```
void PIO_PortOutputEnable(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function enables selected IO pins of the given port as output(s).

Remarks

None.

Preconditions

None.

Example

```
// Make PC5 and PC7 pins as output
PIO_PortOutputEnable(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be setup as outputs. • 1's - Will set corresponding IO pins as output(s). • 0's - Will cause the direction of the corresponding IO pins to remain unchanged.

Function

void PIO_PortOutputEnable(PIO_PORT port, uint32_t mask)

PIO_PortRead Function

Read all the I/O lines of the selected port port.

File

plib_pio.h

C

```
uint32_t PIO_PortRead(PIO_PORT port);
```

Returns

Returns the read value of all the I/O lines of the selected port port.

Description

This function reads the live data values on all the I/O lines of the selected port. Bit values returned in each position indicate corresponding pin levels.

- 1 - Pin is high.
- 0 - Pin is low.

This function reads the value regardless of pin configuration, whether it is set as as an input, driven by the PIO Controller, or driven by a peripheral.

Remarks

If the port has less than 32-bits, unimplemented pins will read as low (0).

Implemented pins are Right aligned in the 32-bit return value.

Preconditions

Reading the I/O line levels requires the clock of the PIO Controller to be enabled, otherwise this API reads the levels present on the I/O line at the time the clock was disabled.

Example

```
uint32_t value;
value = PIO_PortRead(PIO_PORT_C);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT

Function

uint32_t PIO_PortRead(PIO_PORT port)

PIO_PortSet Function

Set the selected IO pins of a port.

File

plib_pio.h

C

```
void PIO_PortSet(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function sets (to '1') the selected IO pins of a port.

Remarks

If the port has less than 32-bits, unimplemented pins will be ignored.

Implemented pins are Right aligned in the 32-bit value.

Preconditions

None.

Example

```
// Set PC5 and PC7 pins to 1
PIO_PortSet(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be set. • 1's - Will set corresponding IO pins to high (to 1). • 0's - Will remain unchanged.

Function

void PIO_PortSet(PIO_PORT port, uint32_t mask)

PIO_PortToggle Function

Toggles the selected IO pins of a port.

File

plib_pio.h

C

```
void PIO_PortToggle(PIO_PORT port, uint32_t mask);
```

Returns

None.

Description

This function toggles (or invert) the selected IO pins of a port.

Remarks

If the port has less than 32-bits, unimplemented pins will be ignored.

Implemented pins are Right aligned in the 32-bit value.

Preconditions

PIO_Initialize() must have been called.

Example

```
// Toggles PC5 and PC7 pins
PIO_PortToggle(PIO_PORT_C, 0x00A0);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be toggled. • 1's - Will toggle (invert) corresponding IO pins. • 0's - Will remain unchanged.

Function

void PIO_PortToggle(PIO_PORT port, uint32_t mask)

PIO_PortWrite Function

Write the value on the masked I/O lines of the selected port.

File

plib_pio.h

C

```
void PIO_PortWrite(PIO_PORT port, uint32_t mask, uint32_t value);
```

Returns

None.

Description

This function writes the data values driven on selected output lines of the selected port. Bit values in each position indicate corresponding pin levels.

- 1 - Pin is driven high.
- 0 - Pin is driven low.

Remarks

If the port has less than 32-bits, unimplemented pins will be ignored.

Implemented pins are Right aligned in the 32-bit value.

Preconditions

The desired pins lines of the selected port must be setup as output(s) in MHC Pin Manager.

PIO_Initialize() must have been called.

Example

```
// Write binary value 0011 to the pins PC3, PC2, PC1 and PC0 respectively.
PIO_PortWrite(PIO_PORT_C, 0x0F, 0xF563D453);
```

Parameters

Parameters	Description
port	One of the IO ports from the enum PIO_PORT
mask	A 32 bit value in which positions of 0s and 1s decide which IO pins of the selected port will be written. • 1's - Will write to corresponding IO pins. • 0's - Will remain unchanged.
value	Value which has to be written/driven on the I/O lines of the selected port for which mask bits are '1'. Values for the corresponding mask bit '0' will be ignored.

Function

void PIO_PortWrite(PIO_PORT port, uint32_t mask, uint32_t value);

d) Data Types and Constants

PIO_PIN_CALLBACK Type

Pointer to a PIO Pin-Event handler function.

File

plib_pio.h

C

```
typedef void (* PIO_PIN_CALLBACK)(PIO_PIN pin, uintptr_t context);
```

Returns

None.

Description

PIO Pin-Event Handler Function Pointer

This data type defines the required function signature for the PIO pin-event handling callback function. The client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive calls back from the PLIB when a configured pin event occurs.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the PIO_PinInterruptCallbackRegister function. This context handle value is passed back to the client as the "context" parameter. It can be any value (such as a pointer to the client's data) necessary to identify the client context.

The event handler function executes in the PLIB's interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

A function matching this signature:

```
void APP_PinEventHandler(PIO_PIN pin, uintptr_t context)
{
    // Do Something
}
```

Is registered as follows:

```
PIO_PinInterruptCallbackRegister(PIO_PIN_PA5, &APP_PinEventHandler, NULL);
```

Parameters

Parameters	Description
pin	One of the IO pins from the enum PIO_PIN
context	Value identifying the context of the client that registered the event handling function

PIO_PIN Enumeration

Identifies the available PIO port pins.

File

plib_pio.h

C

```
typedef enum {
  PIO_PIN_PA0,
  PIO_PIN_PA1,
  PIO_PIN_PA2,
  PIO_PIN_PA3,
  PIO_PIN_PA4,
  PIO_PIN_PA5,
  PIO_PIN_PA6,
  PIO_PIN_PA7,
  PIO_PIN_PA8,
  PIO_PIN_PA9,
  PIO_PIN_PA10,
  PIO_PIN_PA11,
  PIO_PIN_PA12,
  PIO_PIN_PA13,
  PIO_PIN_PA14,
  PIO_PIN_PA15,
  PIO_PIN_PA16,
  PIO_PIN_PA17,
```

```
PIO_PIN_PA18,
PIO_PIN_PA19,
PIO_PIN_PA20,
PIO_PIN_PA21,
PIO_PIN_PA22,
PIO_PIN_PA23,
PIO_PIN_PA24,
PIO_PIN_PA25,
PIO_PIN_PA26,
PIO_PIN_PA27,
PIO_PIN_PA28,
PIO_PIN_PA29,
PIO_PIN_PA30,
PIO_PIN_PA31,
PIO_PIN_PB0,
PIO_PIN_PB1,
PIO_PIN_PB2,
PIO_PIN_PB3,
PIO_PIN_PB4,
PIO_PIN_PB5,
PIO_PIN_PB6,
PIO_PIN_PB7,
PIO_PIN_PB8,
PIO_PIN_PB9,
PIO_PIN_PB12,
PIO_PIN_PB13,
PIO_PIN_PC0,
PIO_PIN_PC1,
PIO_PIN_PC2,
PIO_PIN_PC3,
PIO_PIN_PC4,
PIO_PIN_PC5,
PIO_PIN_PC6,
PIO_PIN_PC7,
PIO_PIN_PC8,
PIO_PIN_PC9,
PIO_PIN_PC10,
PIO_PIN_PC11,
PIO_PIN_PC12,
PIO_PIN_PC13,
PIO_PIN_PC14,
PIO_PIN_PC15,
PIO_PIN_PC16,
PIO_PIN_PC17,
PIO_PIN_PC18,
PIO_PIN_PC19,
PIO_PIN_PC20,
PIO_PIN_PC21,
PIO_PIN_PC22,
PIO_PIN_PC23,
PIO_PIN_PC24,
PIO_PIN_PC25,
PIO_PIN_PC26,
PIO_PIN_PC27,
PIO_PIN_PC28,
PIO_PIN_PC29,
PIO_PIN_PC30,
PIO_PIN_PC31,
PIO_PIN_PD0,
PIO_PIN_PD1,
PIO_PIN_PD2,
PIO_PIN_PD3,
PIO_PIN_PD4,
PIO_PIN_PD5,
PIO_PIN_PD6,
PIO_PIN_PD7,
PIO_PIN_PD8,
```

```
  PIO_PIN_PD9,
  PIO_PIN_PD10,
  PIO_PIN_PD11,
  PIO_PIN_PD12,
  PIO_PIN_PD13,
  PIO_PIN_PD14,
  PIO_PIN_PD15,
  PIO_PIN_PD16,
  PIO_PIN_PD17,
  PIO_PIN_PD18,
  PIO_PIN_PD19,
  PIO_PIN_PD20,
  PIO_PIN_PD21,
  PIO_PIN_PD22,
  PIO_PIN_PD23,
  PIO_PIN_PD24,
  PIO_PIN_PD25,
  PIO_PIN_PD26,
  PIO_PIN_PD27,
  PIO_PIN_PD28,
  PIO_PIN_PD29,
  PIO_PIN_PD30,
  PIO_PIN_PD31,
  PIO_PIN_PE0,
  PIO_PIN_PE1,
  PIO_PIN_PE2,
  PIO_PIN_PE3,
  PIO_PIN_PE4,
  PIO_PIN_PE5,
  PIO_PIN_NONE
} PIO_PIN;
```

Members

Members	Description
PIO_PIN_NONE	This element should not be used in any of the PIO APIs. It will be used by other modules or application to denote that none of the PIO Pin is used

Description

PIO Port Pins

This enumeration identifies the available PIO port pins.

Remarks

The caller should not rely on the specific numbers assigned to any of these values as they may change from one processor to the next.

Not all pins are available on all devices. Refer to the specific device data sheet to determine which pins are supported.

PIO_PORT Enumeration

Identifies the available PIO Ports.

File

plib_pio.h

C

```
typedef enum {
  PIO_PORT_A,
  PIO_PORT_B,
  PIO_PORT_C,
  PIO_PORT_D,
  PIO_PORT_E
} PIO_PORT;
```

Description

PIO Port

This enumeration identifies the available PIO Ports.

Remarks

The caller should not rely on the specific numbers assigned to any of these values as they may change from one processor to the next.

Not all ports are available on all devices. Refer to the specific device data sheet to determine which ports are supported.

Files

Files

Name	Description
plib_pio.h	PIO PLIB Documentation File

Description

This section will list only the library's interface header file(s).

plib_pio.h

PIO PLIB Documentation File

Enumerations

	Name	Description
	PIO_PIN	Identifies the available PIO port pins.
	PIO_PORT	Identifies the available PIO Ports.

Functions

	Name	Description
	PIO_Initialize	Initialize the PIO library.
	PIO_PinClear	Clears the selected pin.
	PIO_PinInputEnable	Enables selected IO pin as input.
	PIO_PinInterruptCallbackRegister	Allows application to register callback for every pin.
	PIO_PinInterruptDisable	Disables IO interrupt on selected IO pin.
	PIO_PinInterruptEnable	Enables IO interrupt on selected IO pin.
	PIO_PinLatchRead	Read the value driven on the selected pin.
	PIO_PinOutputEnable	Enables selected IO pin as output.
	PIO_PinRead	Read the selected pin value.
	PIO_PinSet	Sets the selected pin.
	PIO_PinToggle	Toggles the selected pin.
	PIO_PinWrite	Writes the selected pin.
	PIO_PortClear	Clear the selected IO pins of a port.
	PIO_PortInputEnable	Enables selected IO pins of a port as input.
	PIO_PortInterruptDisable	Disables IO interrupt on selected IO pins of a port.
	PIO_PortInterruptEnable	Enables IO interrupt on selected IO pins of a port.
	PIO_PortLatchRead	Read the latched value on all the I/O lines of the selected port.
	PIO_PortOutputEnable	Enables selected IO pins of a port as output(s).
	PIO_PortRead	Read all the I/O lines of the selected port port.
	PIO_PortSet	Set the selected IO pins of a port.

	PIO_PortToggle	Toggles the selected IO pins of a port.
	PIO_PortWrite	Write the value on the masked I/O lines of the selected port.

Types

	Name	Description
	PIO_PIN_CALLBACK	Pointer to a PIO Pin-Event handler function.

Description

PIO PLIB

This library provides documentation of all the interface to control and interact with Parallel Input/Output controller (PIO) module.

File Name

plib_pio.h

Company

Microchip Technology Inc.

PWM Peripheral Library Help

This section provides an interface to use Pulse Width Modulation Controller (PWM) peripheral

Introduction

This section provides brief overview of the PWM peripheral

Description

The Pulse Width Modulation Controller (PWM) generates output pulses on 4 channels independently according to parameters defined per channel. Each channel controls two complementary square output waveforms. Characteristics of the output waveforms such as period, duty-cycle, polarity and dead-times are configured through the user interface. Each channel selects and uses one of the clocks provided by the clock generator. The clock generator provides several clocks resulting from the division of the PWM peripheral clock. External triggers can be managed to allow output pulses to be modified in real time.

Channels can be linked together as synchronous channels to be able to update their duty-cycle or dead-time at the same time.

The PWM provides 8 independent comparison units capable of comparing a programmed value to the counter of the synchronous channels (counter of channel 0). These comparisons are intended to

generate software interrupts, to trigger pulses on the 2 independent event lines (in order to synchronize ADC conversions with a lot of flexibility independently of the PWM outputs) and to trigger DMA Controller transfer requests.

The PWM provides a fault protection mechanism with 8 fault inputs, capable to detect a fault condition and to override the PWM outputs asynchronously (outputs forced to ‘0’, ‘1’ or Hi-Z).

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

This example shows how to configure PWM to generate 3 phase PWM signals with dead time for motor control application.

The duty cycle of the PWM is updated in the interrupt handler.

```
/* Duty cycle increment value */
#define DUTY_INCREMENT (10U)

/* Save PWM period */
uint16_t period;

/* This function is called after PWM0 counter event */
void PWM_CounterEventHandler(uintptr_t context)
{
    static uint16_t duty0 = 0U;
    static uint16_t duty1 = 2500U;
    static uint16_t duty2 = 5000U;

    PWMx_ChannelDutySet(PWM_CHANNEL_0, duty0);
    PWMx_ChannelDutySet(PWM_CHANNEL_1, duty1);
    PWMx_ChannelDutySet(PWM_CHANNEL_2, duty2);

    /* Increment duty cycle values */
    duty0 += DUTY_INCREMENT;
    duty1 += DUTY_INCREMENT;
    duty2 += DUTY_INCREMENT;

    if (duty0 > period)
        duty0 = 0U;
    if (duty1 > period)
        duty1 = 0U;
    if (duty2 > period)
        duty2 = 0U;
}

int main ( void )
{
    /* Register callback function for Channel 0 counter event */
    PWMx_CallbackRegister(PWM_CounterEventHandler, (uintptr_t)NULL);

    /* Read the period */
    period = PWM0_ChannelPeriodGet(PWM_CHANNEL_0);

    /* Start all synchronous channels by starting channel 0*/
    PWMx_ChannelsStart(PWM_CHANNEL_0_MASK);
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Functions

	Name	Description
	PWMx_Initialize	Initializes given instance of PWM peripheral

b) Setup Functions

	Name	Description
	PWMx_CallbackRegister	Registers the function to be called from interrupt

c) Transaction Functions

	Name	Description
	PWMx_ChannelCounterEventDisable	Disables counter event of given channels
	PWMx_ChannelCounterEventEnable	Enables counter event of given channels
	PWMx_ChannelDeadTimeSet	Writes dead time values of given PWM channel.

	PWMx_ChannelDutySet	Writes the duty cycle value of given PWM channel
	PWMx_ChannelPeriodGet	Reads the period value of given PWM channel
	PWMx_ChannelPeriodSet	Sets the period value of given PWM channel
	PWMx_ChannelsStart	Starts the given PWM channels
	PWMx_ChannelsStop	Stops the given PWM channels
	PWMx_CompareValueSet	Writes the compare value for given PWM peripheral and given comparison unit
	PWMx_SyncUpdateEnable	This sets the synchronous update unlock bit
	PWMx_FaultStatusClear	This function clears the status of the given fault id

d) Status Functions

	Name	Description
	PWMx_ChannelCounterEventStatusGet	Disables counter event of given channels

e) Data Types and Constants

	Name	Description
	PWM_CALLBACK	Defines the data type and function signature for the PWM peripheral callback function.
	PWM_CALLBACK_OBJECT	Callback structure
	PWM_CHANNEL_MASK	Identifies PWM channel mask
	PWM_CHANNEL_NUM	Identifies PWM channel number in a PWM peripheral
	PWM_COMPARE	Identifies PWM compare unit
	PWM_FAULT_ID	Identifies PWM fault input ids

a) Initialization Functions

PWMx_Initialize Function

Initializes given instance of PWM peripheral

File

plib_pwm.h

C

```
void PWMx_Initialize();
```

Returns

None.

Description

This function initializes given instance of PWM periphral with the values configured in MCC GUI. This initializes all the selected channels of given peripheal in MCC GUI.

Remarks

This function must be called before any other PWM function is called.

This function should only be called once during system initialization.

Preconditions

MCC GUI should be configured with the right values.

Example

```
PWM0_Initialize();
```

Function

void PWMx_Initialize (void);

b) Setup Functions

PWMx_CallbackRegister Function

Registers the function to be called from interrupt

File

plib_pwm.h

C

```
void PWMx_CallbackRegister(PWM_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from interrupt

Remarks

Context value can be set to NULL if not required. To disable callback function, pass NULL for the callback parameter.

Preconditions

PWMx_Initialize() must have been called first for the associated instance.

Example

```
void PWM_Callback_Fn(uintptr_t context);

PWM0_Initialize();
PWMx_CallbackRegister(PWM_Callback_Fn, NULL);
```

Parameters

Parameters	Description
callback	callback function pointer
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void PWMx_CallbackRegister (PWM_CALLBACK callback, uintptr_t context);

c) Transaction Functions

PWMx_ChannelCounterEventDisable Function

Disables counter event of given channels

File

plib_pwm.h

C

```
void PWMx_ChannelCounterEventDisable(PWM_CHANNEL_MASK channelMask);
```

Returns

None

Description

This function disables the counter event of channels specified in channelMask.

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
PWM0_Initialize();
PWM0_ChannelCounterEventDisable(PWM_CHANNEL_0_MASK);
```

Parameters

Parameters	Description
channelMask	set of channel numbers

Function

void PWMx_ChannelCounterEventDisable (PWM_CHANNEL_MASK channelMask);

PWMx_ChannelCounterEventEnable Function

Enables counter event of given channels

File

plib_pwm.h

C

```
void PWMx_ChannelCounterEventEnable(PWM_CHANNEL_MASK channelMask);
```

Returns

None

Description

This function enables the counter event of channels specified in channelMask. For left-aligned mode, counter event occurs when counter value = period value For center-aligned mode, counter event is based on configuration counter value = period value or counter value = period value and counter value = 0 while decrementing

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
PWM0_Initialize();
PWM0_ChannelCounterEventEnable(PWM_CHANNEL_0_MASK);
```

Parameters

Parameters	Description
channelMask	set of channel numbers

Function

void PWMx_ChannelCounterEventEnable (PWM_CHANNEL_MASK channelMask);

PWMx_ChannelDeadTimeSet Function

Writes dead time values of given PWM channel.

File

plib_pwm.h

C

```
void PWMx_ChannelDeadTimeSet(PWM_CHANNEL_NUM channel, uint16_t deadtime_high, uint16_t
deadtime_low);
```

Returns

None

Description

This function writes the dead time value in double buffer register. Update to the actual dead time register happens at the next period border. For left-aligned mode, update occurs when counter value = period value For center-aligned mode, update occurs when counter value = 0 while decrementing

Remarks

Dead time is inserted between the edges of the two complementary outputs DTOHx and DTOLx to safely drive the external power control switches.

In sync mode, unlock bit (PWM_SCUC.UPDULOCK) should be set explicitly to trigger the update of dead time value at the next PWM period border. Use API PWMx_SyncUpdateEnable to set the PWM_SCUC.UPDULOCK bit.

Preconditions

PWMx_Initialize() function must have been called first for associated channel. And associated PWM channel must be running.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
PWM0_ChannelDeadTimeSet(PWM_CHANNEL_0, 0x250, 0x50);
```

Parameters

Parameters	Description
channel	PWM channel number
deadtime_high	dead time inserted on high side output waveform
deadtime_low	dead time inserted on low side output waveform

Function

void PWMx_ChannelDeadTimeSet (PWM_CHANNEL_NUM channel, uint16_t deadtime_high, uint16_t deadtime_low);

PWMx_ChannelDutySet Function

Writes the duty cycle value of given PWM channel

File

plib_pwm.h

C

```
void PWMx_ChannelDutySet(PWM_CHANNEL_NUM channel, uint16_t duty);
```

Returns

channel - PWM channel number duty - duty cycle value

Description

This function writes the duty cycle value in double buffer register. Update to the actual duty cycle register happens at the next period border. For left-aligned mode, update occurs when counter value = period value For center-aligned mode, update occurs when counter value = 0 while decrementing OR counter value = period value while incrementing

Remarks

In sync mode update method 0 (manual update of duty cycle), unlock bit (PWM_SCUC.UPDULOCK) should be set explicitly to trigger the update of duty cycle value at the next PWM period border. Use API PWMx_SyncUpdateEnable to set the PWM_SCUC.UPDULOCK bit.

Preconditions

PWMx_Initialize() function must have been called first for the associated instance. And associated channel must be running.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
PWM0_ChannelDutySet(PWM_CHANNEL_0, 0x200);
```

Parameters

Parameters	Description
channel	PWM channel number
duty	duty cycle value which determines ON time of waveform

Function

void PWMx_ChannelDutySet(PWM_CHANNEL_NUM channel);

PWMx_ChannelPeriodGet Function

Reads the period value of given PWM channel

File

plib_pwm.h

C

```
uint16_t PWMx_ChannelPeriodGet(PWM_CHANNEL_NUM channel);
```

Returns

The PWM channel's period value

Description

This function reads the period value of given PWM channel.

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
uint16_t period;
PWM0_Initialize();
period = PWM0_ChannelPeriodGet(PWM_CHANNEL_0);
```

Parameters

Parameters	Description
channel	PWM channel

Function

uint16_t PWMx_ChannelPeriodGet (PWM_CHANNEL_NUM channel);

PWMx_ChannelPeriodSet Function

Sets the period value of given PWM channel

File

plib_pwm.h

C

```
void PWMx_ChannelPeriodSet(PWM_CHANNEL_NUM channel, uint16_t period);
```

Returns

None.

Description

This function writes the period value in double buffer register. Update to the actual period register happens at the next period border. For left-aligned mode, update occurs when counter value = period value For center-aligned mode, update occurs when counter value = 0 while decrementing

Remarks

In sync mode, unlock bit (PWM_SCUC.UPDULOCK) should be set explicitly to trigger the update of period value at the next PWM period border. Use API PWMx_SyncUpdateEnable to set the PWM_SCUC.UPDULOCK bit.

Preconditions

PWMx_Initialize() function must have been called first for the associated instance. And associated channel must be running.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
PWM0_ChannelPeriodSet(PWM_CHANNEL_0, 0x500);
```

Parameters

Parameters	Description
channel	PWM channel number
period	period value corresponding to PWM frequency

Function

void PWMx_ChannelPeriodSet (PWM_CHANNEL_NUM channel, uint32_t period);

PWMx_ChannelsStart Function

Starts the given PWM channels

File

plib_pwm.h

C

```
void PWMx_ChannelsStart(PWM_CHANNEL_MASK channelMask);
```

Returns

None.

Description

This function enables the PWM channels specified in channelMask. If channels are configured in sync mode, only channel 0 should be enabled which will ensure all sync channels are started together.

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
```

Parameters

Parameters	Description
channelMask	set of channel numbers

Function

void PWMx_ChannelsStart (PWM_CHANNEL_MASK channelMask);

PWMx_ChannelsStop Function

Stops the given PWM channels

File

plib_pwm.h

C

```
void PWMx_ChannelsStop(PWM_CHANNEL_MASK channelMask);
```

Returns

None.

Description

This function disables the PWM channels specified in channelMask. If channels are configured in sync mode, only channel 0 should be disabled which will ensure all sync channels are stopped together.

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
```

```
PWM0_ChannelsStop(PWM_CHANNEL_0_MASK);
```

Parameters

Parameters	Description
channelMask	set of channel numbers

Function

void PWMx_ChannelsStop (PWM_CHANNEL_MASK channelMask);

PWMx_CompareValueSet Function

Writes the compare value for given PWM peripheral and given comparison unit

File

plib_pwm.h

C

```
void PWMx_CompareValueSet(PWM_COMPARE cmp_unit, uint16_t cmp_value);
```

Returns

None

Description

This function writes compare value in a double buffer register. Update to the actual compare value register happens at the PWM period border when compare period counter expires.

Remarks

When compare value matches the counter of PWM channel 0, interrupt is generated and also pulse is generated on PWM event line which can be used to synchronize ADC conversion.

Preconditions

PWMx_Initialize() function must have been called first for the associated instance. And associated channel must be running.

Example

```
PWM0_Initialize();
PWM0_ChannelsStart(PWM_CHANNEL_0_MASK);
PWM0_CompareValueSet(PWM_COMPARE_0, 0x25);
```

Parameters

Parameters	Description
cmp_unit	compare unit number
cmp_value	compare value which is compared with channel 0 counter

Function

void PWMx_CompareValueSet (PWM_COMPARE cmp_unit, uint16_t cmp_value);

PWMx_SyncUpdateEnable Function

This sets the synchronous update unlock bit

File

plib_pwm.h

C

```
void PWMx_SyncUpdateEnable();
```

Returns

None

Description

This function sets the synchronous update unlock bit which enables update of period, duty and dead-time values from double buffer register to actual register at the next PWM period border.

Remarks

In sync mode update method 1 (manual update), this function has to be called to update period, duty and dead-time values to actual register. In sync mode update method 2 (automatic update), this function has to be called to update only period and dead-time. Duty value update happens automatically.

Preconditions

PWMx_Initialize() function must have been called first for the associated instance. And new period or duty or dead-time value must have been written in a double buffer register.

Example

```
PWM0_Initialize();
PWM0_SyncUpdateEnable();
```

Function

void PWMx_SyncUpdateEnable (void);

PWMx_FaultStatusClear Function

This function clears the status of the given fault id

File

plib_pwm.h

C

```
void PWMx_FaultStatusClear(PWM_FAULT_ID fault_id);
```

Returns

None

Description

This function clears the fault status of the given id. When fault mode is set as "fault is active until cleared at peripheral level and also at the register level", this function clears the fault status at the register level.

Remarks

Fault becomes inactive after this function call, only if fault input level is cleared.

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
PWM0_Initialize();
PWM0_FaultStatusClear(PWM_FAULT_ID_0);
```

Function

void PWMx_FaultStatusClear(PWM_FAULT_ID fault_id);

d) Status Functions

PWMx_ChannelCounterEventStatusGet Function

Disables counter event of given channels

File

plib_pwm.h

C

```
bool PWMx_ChannelCounterEventStatusGet(PWM_CHANNEL_NUM channel);
```

Returns

status of an interrupt

- True - counter event of a channel occurred
- False - counter event of a channel is not yet occurred

Description

This function disables the counter event of channels specified in channelMask.

Remarks

None

Preconditions

PWMx_Initialize() function must have been called first for the associated instance.

Example

```
bool status;
PWM0_Initialize();
status = PWM0_ChannelCounterEventStatusGet(PWM_CHANNEL_0);
```

Parameters

Parameters	Description
channel	channel number

Function

bool PWMx_ChannelCounterEventStatusGet (PWM_CHANNEL_MASK channelMask);

e) Data Types and Constants

PWM_CALLBACK Type

Defines the data type and function signature for the PWM peripheral callback function.

File

plib_pwm.h

C

```
typedef void (* PWM_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the PWM peripheal callback function. The library will call back the client's function with this signature from the interrupt context.

Remarks

None.

Preconditions

PWMx_Initialize must have been called for the given PWM peripheral instance and PWMx_CallbackRegister must have been called to register the function to be called.

Example

```
// callback function
void PWM_Counter_Callback (uintptr_t context);

// register callback function
PWMx_CallbackRegister(PWM_Counter_Callback, NULL);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void (*PWMx_CALLBACK)(uintptr_t context)

PWM_CALLBACK_OBJECT Structure

Callback structure

File

plib_pwm.h

C

```
typedef struct {
  PWM_CALLBACK callback_fn;
  uintptr_t context;
} PWM_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

PWM_CHANNEL_MASK Enumeration

Identifies PWM channel mask

File

plib_pwm.h

C

```
typedef enum {
  PWM_CHANNEL_0_MASK,
  PWM_CHANNEL_1_MASK,
  PWM_CHANNEL_2_MASK,
  PWM_CHANNEL_3_MASK
} PWM_CHANNEL_MASK;
```

Description

PWM channel number mask

This enumeration identifies PWM channel mask. This can be ORed together to start/stop channels together.

Remarks

None.

PWM_CHANNEL_NUM Enumeration

Identifies PWM channel number in a PWM peripheral

File

plib_pwm.h

C

```
typedef enum {
  PWM_CHANNEL_0,
  PWM_CHANNEL_1,
  PWM_CHANNEL_2,
  PWM_CHANNEL_3
} PWM_CHANNEL_NUM;
```

Description

PWM channel number

This enumeration identifies PWM channel number in a PWM peripheral. This can be used as an argument in the APIs

Remarks

None.

PWM_COMPARE Enumeration

Identifies PWM compare unit

File

plib_pwm.h

C

```
typedef enum {
  PWM_COMPARE_0,
  PWM_COMPARE_1,
  PWM_COMPARE_2,
  PWM_COMPARE_3,
  PWM_COMPARE_4,
  PWM_COMPARE_5,
  PWM_COMPARE_6,
  PWM_COMPARE_7
} PWM_COMPARE;
```

Description

PWM compare unit

This enumeration identifies PWM compare unit.

Remarks

None.

PWM_FAULT_ID Enumeration

Identifies PWM fault input ids

File

plib_pwm.h

C

```
typedef enum {
  PWM_FAULT_ID_0,
  PWM_FAULT_ID_1,
  PWM_FAULT_ID_2,
  PWM_FAULT_ID_3,
  PWM_FAULT_ID_4,
  PWM_FAULT_ID_5,
  PWM_FAULT_ID_6,
  PWM_FAULT_ID_7
} PWM_FAULT_ID;
```

Description

PWM fault ID

This enumeration defines the PWM fault IDs

Remarks

None.

Files

Files

Name	Description
plib_pwm.h	PWM peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_pwm.h

PWM peripheral library interface.

Enumerations

	Name	Description
	PWM_CHANNEL_MASK	Identifies PWM channel mask
	PWM_CHANNEL_NUM	Identifies PWM channel number in a PWM peripheral
	PWM_COMPARE	Identifies PWM compare unit
	PWM_FAULT_ID	Identifies PWM fault input ids

Functions

	Name	Description
	PWMx_CallbackRegister	Registers the function to be called from interrupt
	PWMx_ChannelCounterEventDisable	Disables counter event of given channels
	PWMx_ChannelCounterEventEnable	Enables counter event of given channels
	PWMx_ChannelCounterEventStatusGet	Disables counter event of given channels
	PWMx_ChannelDeadTimeSet	Writes dead time values of given PWM channel.
	PWMx_ChannelDutySet	Writes the duty cycle value of given PWM channel
	PWMx_ChannelPeriodGet	Reads the period value of given PWM channel
	PWMx_ChannelPeriodSet	Sets the period value of given PWM channel
	PWMx_ChannelsStart	Starts the given PWM channels
	PWMx_ChannelsStop	Stops the given PWM channels
	PWMx_CompareValueSet	Writes the compare value for given PWM peripheral and given comparison unit
	PWMx_FaultStatusClear	This function clears the status of the given fault id
	PWMx_Initialize	Initializes given instance of PWM peripheral
	PWMx_SyncUpdateEnable	This sets the synchronous update unlock bit

Structures

	Name	Description
	PWM_CALLBACK_OBJECT	Callback structure

Types

	Name	Description
	PWM_CALLBACK	Defines the data type and function signature for the PWM peripheral callback function.

Description

PWM Peripheral Library Interface Header File

This file defines the interface to the PWM peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual PWMheaders will be generated as required by the MCC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "PWM" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

File Name

plib_pwm.h

Company

Microchip Technology Inc.

QSPI Peripheral Library Help

This section provides interface to use Quad Serial Peripheral Interface (QSPI) peripheral.

Introduction

This section provides brief overview of QSPI peripheral.

Description

The Quad SPI Interface (QSPI) is a synchronous serial data link that provides communication with external devices in Master mode. It is similar to SPI protocol except that it has additional data lines.

QSPI can be used in normal SPI mode or in Serial Memory mode to connect to external Flash memories. The normal SPI has four communication lines: Chip Select, Clock, MOSI, and MISO. For QSPI, additional data lines are available. So the command/data/address are sent through single, quad or dual IO based on the mode selected. As data is sent over multiple lines, it helps in increasing bandwidth compared to standard SPI transfer. With the support of the Quad SPI protocol, the QSPI allows the system to use high-performance serial Flash memories which are small and inexpensive, in place of larger and more expensive parallel Flash memories.

The QSPI allows the system to execute code directly from a serial Flash memory (XIP) without code shadowing to RAM. The serial Flash memory mapping is seen in the system as other memories such as SRAM, Flash memory.

Using the Library

This section explains how to use the peripheral library

Description

QSPI Peripheral Library operates only in Serial Memory Mode to interface with QSPI based Serial Flash Memories operating in Single-bit SPI, Dual SPI and Quad SPI.

QSPI Peripheral library provides non-Blocking API's and they can be used to perform below functionalities on the QSPI Slave device.

- Send a command
  - qspi_command_xfer_t data type needs to filled to send a command to the slave device
- Register read/write
  - qspi_register_xfer_t data type needs to be filled to read data from a register or to write data to register.
- Memory read/write
  - qspi_memory_xfer_t data type needs to be filled to read data from flash memory or to write data to flash memory.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	QSPIx_Initialize	Initializes given instance of the QSPI peripheral.

b) Transaction Functions

	Name	Description
	QSPIx_CommandWrite	Writes command to QSPI slave device.
	QSPIx_RegisterRead	Reads particular register of QSPI slave device.
	QSPIx_RegisterWrite	Writes to particular register of QSPI slave device.
	QSPIx_MemoryRead	Reads from the specified address of the serial flash device.

	QSPIx_MemoryWrite	Writes to the specified address of the serial flash device.

c) Data Types and Constants

	Name	Description
	QSPI_ADDRESS_LENGTH	Defines the data type to specify address length
	QSPI_LANE_WIDTH	Defines the data type to specify lane width
	QSPI_OPTION_LENGTH	Defines the data type to specify option length
	qspi_command_xfer_t	Defines the data type for the QSPI commands transfer.
	qspi_register_xfer_t	Defines the data type for the QSPI register data transfer.
	qspi_memory_xfer_t	Defines the data type for the QSPI memory transfer.

a) Initialization Function

QSPIx_Initialize Function

Initializes given instance of the QSPI peripheral.

File

plib_qspi.h

C

```
void QSPIx_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the QSPI peripheral as configured by the user.

Preconditions

Memory Protection Unit(MPU) has to be enabled and configured for QSPI memory.

Example

```
QSPI0_Initialize();
```

Function

void QSPIx_Initialize(void)

b) Transaction Functions

QSPIx_CommandWrite Function

Writes command to QSPI slave device.

File

plib_qspi.h

C

```
bool QSPIx_CommandWrite(qspi_command_xfer_t * qspi_command_xfer, uint32_t address);
```

Returns

- True on transfer request success.
- False on transfer request failure.

Description

This function can be used to send commands to QSPI slave device which does not include any exchange of data.

Preconditions

QSPIx_Initialize must have been called for the associated QSPI instance.

Example

```
#define WRITE_ENABLE_CODE 0x06

static qspi_command_xfer_t qspi_command_xfer;

// Use QAUD SPI Lane
qspi_command_xfer.width = QUAD_CMD;

// Send Write enable command
qspi_command_xfer->qspi_code.instruction = WRITE_ENABLE_CODE;

if (TRUE != QSPI0_CommandWrite(&qspi_command_xfer, 0))
{
    // Handle Error
}
```

Parameters

Parameters	Description
*qspi_command_xfer	pointer to QSPI command transfer structure holding the instruction code register and instruction frame register information.
address	Instruction address to be sent with the instruction frame. Can be passed as 0 if no address associated with instruction.

Function

bool QSPIx_CommandWrite(qspi_command_xfer_t *qspi_command_xfer, uint32_t address)

QSPIx_RegisterRead Function

Reads particular register of QSPI slave device.

File

plib_qspi.h

C

```
bool QSPIx_RegisterRead(qspi_register_xfer_t * qspi_register_xfer, uint32_t * rx_data, uint8_t
rx_data_length);
```

Returns

- True on transfer request success.
- False on transfer request failure.

Description

This function can be used to read a particular register of QSPI slave device which has a command code associated to it.

Preconditions

QSPIx_Initialize must have been called for the associated QSPI instance.

Example

```
#define READ_STATUS_REG_CODE 0x05

static qspi_register_xfer_t qspi_register_xfer;
static uint8_t reg_data;

// Use QAUD SPI Lane
qspi_register_xfer.width = QUAD_CMD;
qspi_register_xfer.dummy_cycles = 2;

// Read status register of 1 Byte width
qspi_register_xfer.instruction = READ_STATUS_REG_CODE;

if (TRUE != QSPI0_RegisterRead(&qspi_register_xfer, (uint32_t *)&reg_data, 1))
{
    // Handle Error
}
```

Parameters

Parameters	Description
*qspi_register_xfer	pointer to QSPI register transfer structure holding the instruction code register and instruction frame register information.
*rx_data	Pointer to receive buffer where the register data will be stored.
rx_data_length	Register width in bytes.

Function

bool QSPIx_RegisterRead(qspi_register_xfer_t *qspi_register_xfer, uint32_t *rx_data, uint8_t rx_data_length)

QSPIx_RegisterWrite Function

Writes to particular register of QSPI slave device.

File

plib_qspi.h

C

```
bool QSPIx_RegisterWrite(qspi_register_xfer_t * qspi_register_xfer, uint32_t * tx_data, uint8_t
tx_data_length);
```

Returns

- True on transfer request success.
- False on transfer request failure.

Description

This function can be used to write to a particular register of QSPI slave device which has a command code associated to it.

Preconditions

QSPIx_Initialize must have been called for the associated QSPI instance.

Write enable command has to be sent before write register request and

Example

```
#define WRITE_STATUS_REG_CODE 0x01
#define QUAD_ENABLE_BIT       (1 << 1)

static qspi_register_xfer_t qspi_register_xfer;
static uint8_t reg_data = QUAD_ENABLE_BIT;

qspi_register_xfer.width = QUAD_CMD;
```

```
// Write status register of 1 Byte width
qspi_register_xfer.instruction = WRITE_STATUS_REG_CODE;

if (TRUE != QSPI0_RegisterWrite(&qspi_register_xfer, (uint32_t *)&reg_data, 1))
{
    // Handle Error
}
```

Parameters

Parameters	Description
*qspi_register_xfer	pointer to QSPI register transfer structure holding the instruction code register and instruction frame register information.
*tx_data	Pointer to transmit buffer holding the data to write into register.
tx_data_length	Register width in bytes.

Function

bool QSPIx_RegisterWrite(qspi_register_xfer_t *qspi_register_xfer, uint32_t *tx_data, uint8_t tx_data_length)

QSPIx_MemoryRead Function

Reads from the specified address of the serial flash device.

File

plib_qspi.h

C

```
bool QSPIx_MemoryRead(qspi_memory_xfer_t * qspi_memory_xfer, uint32_t * rx_data, uint32_t
rx_data_length, uint32_t address);
```

Returns

- True on transfer request success.
- False on transfer request failure.

Description

This function can be used to read n bytes of data from the specified address of the serial flash device connected as a QSPI slave.

Preconditions

QSPIx_Initialize must have been called for the associated QSPI instance.

Example

```
#define READ_MEMORY_CODE  0x0B

static qspi_memory_xfer_t qspi_memory_xfer;
static uint32_t buffer[4];
uint32_t address = 0x0

// Read memory location starting from address 0x0
qspi_memory_xfer.instruction = READ_MEMORY_CODE;

// Use QAUD SPI Lane
qspi_memory_xfer.width = QUAD_CMD;
qspi_memory_xfer.dummy_cycles = 6;

if (TRUE != QSPI0_MemoryRead(&qspi_memory_xfer, buffer, sizeof(buffer), address))
{
    // Handle Error
}
```

Parameters

Parameters	Description
*qspi_memory_xfer	pointer to QSPI memory transfer structure holding the instruction code register and instruction frame register information.
*rx_data	Pointer to receive buffer where the data read from memory will be stored.
rx_data_length	number of bytes to read.
address	Memory location to read from.

Function

bool QSPIx_MemoryRead(qspi_memory_xfer_t *qspi_memory_xfer, uint32_t *rx_data, uint32_t rx_data_length, uint32_t address)

QSPIx_MemoryWrite Function

Writes to the specified address of the serial flash device.

File

plib_qspi.h

C

```
bool QSPIx_MemoryWrite(qspi_memory_xfer_t * qspi_memory_xfer, uint32_t * tx_data, uint32_t
tx_data_length, uint32_t address);
```

Returns

- True on transfer request success.
- False on transfer request failure.

Description

This function can be used to write maximum of one page of data to the specified address of the serial flash device connected as a QSPI slave.

If the number of bytes to be written is greater than the page size then QSPIx_MemoryWrite API needs to be called multiple times.

Remarks

Before sending next write request QSPIx_RegisterRead() can be called to read the status register of the slave device to check if the previous write operation is completed.

Preconditions

QSPIx_Initialize must have been called for the associated QSPI instance.

Write enable command has to be sent before write memory request.

Example

```
#define WRITE_PAGE_CODE       0x02
#define WRITE_IN_PROGRESS      (1 << 0)

static qspi_memory_xfer_t qspi_memory_xfer;
static uint8_t buffer[256];
uint32_t address = 0x0;

for (int i = 0; i < 256; i++)
    buffer[i] = i;

// Write to memory location starting from address 0x0
qspi_memory_xfer.instruction = WRITE_PAGE_CODE;

// Use QAUD SPI Lane
qspi_memory_xfer.width = QUAD_CMD;
```

```
if (TRUE != QSPI0_MemoryWrite(&qspi_memory_xfer, buffer, sizeof(buffer), address))
{
    // Handle Error
}

// Query the status register and wait until WIP bit is cleared.
while(status_reg & WRITE_IN_PROGRESS);
```

Parameters

Parameters	Description
*qspi_memory_xfer	pointer to QSPI memory transfer structure holding the instruction code register and instruction frame register information.
*tx_data	Pointer to transmit buffer holding the data to write into memory.
tx_data_length	number of bytes to write.
address	Memory location to write to.

Function

bool QSPIx_MemoryWrite(qspi_memory_xfer_t *qspi_memory_xfer, uint32_t *tx_data, uint32_t tx_data_length, uint32_t address)

c) Data Types and Constants

QSPI_ADDRESS_LENGTH Enumeration

Defines the data type to specify address length

File

plib_qspi.h

C

```
typedef enum {
  ADDRL_24_BIT = QSPI_IFR_ADDRL_24_BIT,
  ADDRL_32_BIT = QSPI_IFR_ADDRL_32_BIT
} QSPI_ADDRESS_LENGTH;
```

Description

QSPI Address Length

This data type can be used to specify the address length for QSPI transaction

Remarks

None.

QSPI_LANE_WIDTH Enumeration

Defines the data type to specify lane width

File

plib_qspi.h

C

```
typedef enum {
  SINGLE_BIT_SPI = QSPI_IFR_WIDTH_SINGLE_BIT_SPI,
  DUAL_OUTPUT = QSPI_IFR_WIDTH_DUAL_OUTPUT,
  QUAD_OUTPUT = QSPI_IFR_WIDTH_QUAD_OUTPUT,
```

```
  DUAL_IO = QSPI_IFR_WIDTH_DUAL_IO,
  QUAD_IO = QSPI_IFR_WIDTH_QUAD_IO,
  DUAL_CMD = QSPI_IFR_WIDTH_DUAL_CMD,
  QUAD_CMD = QSPI_IFR_WIDTH_QUAD_CMD
} QSPI_LANE_WIDTH;
```

Description

QSPI Lane Width

This data type can be used to specify the lane length to be used for QSPI transaction

Remarks

None.

QSPI_OPTION_LENGTH Enumeration

Defines the data type to specify option length

File

plib_qspi.h

C

```
typedef enum {
  OPTL_1_BIT = QSPI_IFR_OPTL_OPTION_1BIT,
  OPTL_2_BIT = QSPI_IFR_OPTL_OPTION_2BIT,
  OPTL_4_BIT = QSPI_IFR_OPTL_OPTION_4BIT,
  OPTL_8_BIT = QSPI_IFR_OPTL_OPTION_8BIT
} QSPI_OPTION_LENGTH;
```

Description

QSPI Option Length

This data type can be used to specify the option length for QSPI transaction

Remarks

None.

qspi_command_xfer_t Structure

Defines the data type for the QSPI commands transfer.

File

plib_qspi.h

C

```
typedef struct {
  uint8_t instruction;
  QSPI_LANE_WIDTH width;
  bool addr_en;
  QSPI_ADDRESS_LENGTH addr_len;
} qspi_command_xfer_t;
```

Description

QSPI Command Transfer structure

This data type should be used to send the QSPI instruction code register and instruction frame register information for sending a command to QSPI slave.

Remarks

None.

qspi_register_xfer_t Structure

Defines the data type for the QSPI register data transfer.

File

plib_qspi.h

C

```
typedef struct {
  uint8_t instruction;
  QSPI_LANE_WIDTH width;
  uint8_t dummy_cycles;
} qspi_register_xfer_t;
```

Description

QSPI Register Transfer structure

This data type should be used to send the QSPI instruction code register and instruction frame register information for reading/writing QSPI slave registers.

Remarks

None.

qspi_memory_xfer_t Structure

Defines the data type for the QSPI memory transfer.

File

plib_qspi.h

C

```
typedef struct {
  uint8_t instruction;
  uint8_t option;
  QSPI_LANE_WIDTH width;
  QSPI_ADDRESS_LENGTH addr_len;
  bool option_en;
  QSPI_OPTION_LENGTH option_len;
  bool continuous_read_en;
  uint8_t dummy_cycles;
} qspi_memory_xfer_t;
```

Description

QSPI Memory Transfer structure

This data type should be used to send the QSPI instruction code register and instruction frame register information for reading/writing QSPI slave memory.

Remarks

None.

Files

Files

Name	Description
plib_qspi.h	Interface definition of the Quad Serial Peripheral Interface Plib (QSPI).

Description

This section will list only the library's interface header file(s).

plib_qspi.h

Interface definition of the Quad Serial Peripheral Interface Plib (QSPI).

Enumerations

	Name	Description
	QSPI_ADDRESS_LENGTH	Defines the data type to specify address length
	QSPI_LANE_WIDTH	Defines the data type to specify lane width
	QSPI_OPTION_LENGTH	Defines the data type to specify option length

Functions

	Name	Description
	QSPIx_CommandWrite	Writes command to QSPI slave device.
	QSPIx_Initialize	Initializes given instance of the QSPI peripheral.
	QSPIx_MemoryRead	Reads from the specified address of the serial flash device.
	QSPIx_MemoryWrite	Writes to the specified address of the serial flash device.
	QSPIx_RegisterRead	Reads particular register of QSPI slave device.
	QSPIx_RegisterWrite	Writes to particular register of QSPI slave device.

Structures

	Name	Description
	qspi_command_xfer_t	Defines the data type for the QSPI commands transfer.
	qspi_memory_xfer_t	Defines the data type for the QSPI memory transfer.
	qspi_register_xfer_t	Defines the data type for the QSPI register data transfer.

Description

Interface definition of QSPI PLIB.

This file defines the interface for the QSPI Plib. It allows user to setup QSPI and transfer data to and from slave devices attached.

File Name

plib_qspi.h

Company

Microchip Technology Inc.

RSTC Peripheral Library Help

This section provides interface to use Reset Controller (RSTC) peripheral.

Introduction

This section provides brief overview of Reset Controller (RSTC)

Description

The Reset Controller (RSTC) handles all the resets of the system and generates following reset signals.

- **Processor Reset:** Resets processor and also resets the Watchdog Timer
- **Peripheral Reset:** Resets the peripheral

- **External Reset:** Drives the NRST pin

The following are the sources of reset and it resorts which one caused the reset.

- **General Reset:** A general reset occurs when a VDDIO power-on-reset is detected, a brownout or a voltage regulation loss is detected by the Supply Controller.
- **Backup Reset**: A backup reset occurs when the chip exits from Backup mode.
- **Watchdog Reset:** The watchdog reset occurs when it times out
- **Software Reset:** The RSTC offers commands to assert processor reset and external reset
- **User Reset:** A user reset is generated when a low level is detected on the NRST pin

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains how to use the peripheral library

Description

RSTC Peripheral library provides API's for the following:

- To generate software reset to reset the processor and all the embedded peripherals including the memory.
- To find the cause of last device reset
- Poll NRST pin status or register interrupt callback when it is used as GPIO

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	RSTCx_Initialize	Initializes RSTC module with the user configuration

b) Setup Functions

	Name	Description
	RSTCx_CallbackRegister	Allows application to register callback with PLIB

c) Transaction Functions

	Name	Description
	RSTCx_Reset	Resets the processor and all the embedded peripherals including the memory system and, in particular, the Remap Command or this function asserts NRST Pin

d) Status Functions

	Name	Description
	RSTCx_NRSTPinRead	Reads the NRST pin level, sampled on each "Master Clock(MCK)" rising edge
	RSTCx_ResetCauseGet	Reports the cause of the last reset

e) Data Types and Constants

	Name	Description
	RSTC_CALLBACK	Pointer to a RSTC callback function
	RSTC_RESET_CAUSE	Identifies the type of reset
	RSTC_RESET_TYPE	Identifies the type of reset, either Processor reset or External Reset

a) Initialization Function

RSTCx_Initialize Function

Initializes RSTC module with the user configuration

File

plib_rstc.h

C

```
void RSTCx_Initialize();
```

Returns

None.

Description

This function initializes RSTC x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, RSTC APIs can be used to generate reset or to get the cause of reset.

Remarks

None.

Preconditions

MHC GUI should be configured with the right values.

Example

```
RSTC0_Initialize();
```

Function

void RSTCx_Initialize (void);

b) Setup Functions

RSTCx_CallbackRegister Function

Allows application to register callback with PLIB

File

plib_rstc.h

C

```
void RSTCx_CallbackRegister(RSTC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function allows application to register a callback function for the PLIB to call back when interrupt occurred.

At any point if application wants to stop the callback, it can call this function with "callback" value as NULL.

Remarks

None.

Preconditions

The RSTCx_Initialize function must have been called.

Example

```
MY_APP_OBJ myAppObj;
void APP_RSTC_CallbackFunction(uintptr_t context)
{
    // The context was set to an application specific object.
    // It is now retrievable easily in the event handler.
       MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;
    //Application related tasks
}

RSTC0_CallbackRegister(APP_RSTC_CallbackFunction, (uintptr_t)&myAppObj);
```

Parameters

Parameters	Description
callback	Pointer to the callback function implemented by the user
context	The value of parameter will be passed back to the application unchanged, when the callback function is called. It can be used to identify any application specific value that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void RSTCx_CallbackRegister (RSTC_CALLBACK callback, uintptr_t context);

c) Transaction Functions

RSTCx_Reset Function

Resets the processor and all the embedded peripherals including the memory system and, in particular, the Remap Command or this function asserts NRST Pin

File

plib_rstc.h

C

```
void RSTCx_Reset(RSTC_RESET_TYPE type);
```

Returns

None.

Description

This function resets the processor and all the embedded peripherals, if RSTC_RESET_TYPE type is "RSTC_RESET_PROC". This function asserts the NRST pin, if RSTC_RESET_TYPE type is "RSTC_RESET_EXT"

Remarks

None.

Preconditions

None.

Example

```
RSTC0_Reset (RSTC_RESET_PROC);
```

Parameters

Parameters	Description
RSTC_RESET_TYPE	Reset type, either Processor reset, External Reset or both.

Function

void RSTCx_Reset (RSTC_RESET_TYPE type);

d) Status Functions

RSTCx_NRSTPinRead Function

Reads the NRST pin level, sampled on each "Master Clock(MCK)" rising edge

File

plib_rstc.h

C

```
bool RSTCx_NRSTPinRead();
```

Returns

bool - Returns True when NRST pin is high Returns False when NRST pin is Low

Description

This function helps to read the NRST pin status in firmware, When NSRT pin is configured to neither generate reset nor generate interrupt, the NRST pin could be used as digital input pin.

Remarks

None.

Preconditions

None.

Example

```
if (RSTC0_NRSTPinRead ())
{
    //Application related tasks
}
```

Function

bool RSTCx_NRSTPinRead (void);

RSTCx_ResetCauseGet Function

Reports the cause of the last reset

File

plib_rstc.h

C

```
RSTC_RESET_CAUSE RSTCx_ResetCauseGet();
```

Returns

RSTC_RESET_CAUSE - Identifies cause of reset, either General, Backup, Watchdog, Software or User Reset

Description

This function is used to know the cause of the last reset.

Remarks

None.

Preconditions

None.

Example

```
if (RSTC_WATCHDOG_RESET == RSTC0_ResetCauseGet ())
{
    //Application related tasks
}
```

Function

RSTC_RESET_CAUSE RSTCx_ResetCauseGet (void);

e) Data Types and Constants

RSTC_CALLBACK Type

Pointer to a RSTC callback function

File

plib_rstc.h

C

```
typedef void (* RSTC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

RSTC Callback Function pointer typedef void (*RSTC_CALLBACK) (uintptr_t context);

This data type defines the required function signature for the RSTC callback function. Application must register a pointer to a callback function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive callback from the PLIB.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The context parameter contains a handle to the client context, provided at the time the callback function was registered using the RSTCx_CallbackRegister function. This context handle value is passed back to the client as the "context" parameter. It can be any value (such as a pointer to the client's data) necessary to identify the client context or instance of the client that made the request.

The callback function executes in the PLIB's interrupt context. It is recommended of the application to not perform process

intensive or blocking operations with in this function.

Example

```
void APP_RSTC_CallbackFunction(uintptr_t context)
{
    //Application related tasks
}

RSTC0_CallbackRegister (APP_RSTC_CallbackFunction, NULL);
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the callback function.

RSTC_RESET_CAUSE Enumeration

Identifies the type of reset

File

plib_rstc.h

C

```
typedef enum {
  RSTC_GENERAL_RESET,
  RSTC_BACKUP_RESET,
  RSTC_WATCHDOG_RESET,
  RSTC_SOFTWARE_RESET,
  RSTC_USER_RESET
} RSTC_RESET_CAUSE;
```

Members

Members	Description
RSTC_GENERAL_RESET	First power reset
RSTC_BACKUP_RESET	Reset after return from Backup mode
RSTC_WATCHDOG_RESET	Reset when Watchdog fault occurred
RSTC_SOFTWARE_RESET	Reset caused by dedicated software instruction
RSTC_USER_RESET	Reset occurs when NRST pin is detected low

Description

User Reset type

This enumeration identifies either General, Backup, Watchdog, Software or User Reset

Remarks

None.

RSTC_RESET_TYPE Enumeration

Identifies the type of reset, either Processor reset or External Reset

File

plib_rstc.h

C

```
typedef enum {
  RSTC_RESET_PROC,
  RSTC_RESET_EXT,
```

```
    RSTC_RESET_EXT_PROC
} RSTC_RESET_TYPE;
```

Members

Members	Description
RSTC_RESET_PROC	Processor reset
RSTC_RESET_EXT	External reset
RSTC_RESET_EXT_PROC	Processor and External reset

Description

Reset type

This enumeration identifies either Processor reset or External Reset.

Remarks

None.

Files

Files

Name	Description
plib_rstc.h	RSTC peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_rstc.h

RSTC peripheral library interface.

Enumerations

	Name	Description
	RSTC_RESET_CAUSE	Identifies the type of reset
	RSTC_RESET_TYPE	Identifies the type of reset, either Processor reset or External Reset

Functions

	Name	Description
	RSTCx_CallbackRegister	Allows application to register callback with PLIB
	RSTCx_Initialize	Initializes RSTC module with the user configuration
	RSTCx_NRSTPinRead	Reads the NRST pin level, sampled on each "Master Clock(MCK)" rising edge
	RSTCx_Reset	Resets the processor and all the embedded peripherals including the memory system and, in particular, the Remap Command or this function asserts NRST Pin
	RSTCx_ResetCauseGet	Reports the cause of the last reset

Types

	Name	Description
	RSTC_CALLBACK	Pointer to a RSTC callback function

Description

Reset Controller (RSTC) Peripheral Library(PLIB) Interface Header File

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Reset Controller (RSTC). This file must not be included in any MPLAB Project.

File Name

plib_rstc.h

Company

Microchip Technology Inc.

RSWDT Peripheral Library Help

This section provides interface to use Reinforced Safety Watchdog Timer (RSWDT) peripheral.

Introduction

This section provides brief overview of Reinforced Safety Watchdog Timer (RSWDT) peripheral.

Description

The Reinforced Safety Watchdog Timer (RSWDT) works in parallel with the Watchdog Timer (WDT) to reinforce safe watchdog operations.

The RSWDT is a 12-bit down counter that allows watchdog period of up to 16 seconds with 32kHz slow clock. It should be fed periodically before it counts down to zero. If the timer counts down to zero, it will either reset the device.

The RSWDT uses the Main RC oscillator divided clock (frequency is equivalent to Slow RC) if the Main RC oscillator is enabled by the application or if the WDT is driven by the 32kHz Slow clock.

If the WDT clock source (for example, the 32 kHz crystal oscillator) fails, the system lock-up is no longer monitored by the WDT and the RSWDT performs the monitoring with Main RC oscillator and resets the device.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This Section provides information on how to use the Peripheral Library.

Description

The RSWDT module runs independent of the rest of the system, causing system resets whenever it times out. However, the application software should ensure that the timeout never occurs by resetting the RSWDT periodically as long as the software is in a known healthy state. If the system hangs or program execution is corrupted, the RSWDT will not receive its periodic reset, and will eventually time out and cause a system reset.

In many systems, there is an obvious upper bound to the amount of time each iteration of the main application loop can be expected to run, before a malfunction can be assumed (either due to a deadlock waiting on hardware or software, or due to other means). When the RSWDT is configured with a timeout period equal to this upper bound, a malfunction in the system will force a full system reset to allow for a graceful recovery. The upper bound is specified by 12-bit watch dog period value.

Window Mode

Just as there is a reasonable upper bound to the time the main program loop should take for each iteration, there is also in many applications a lower bound, i.e. a minimum time for which each loop iteration should run for under normal circumstances. To guard against a system failure resetting the Watchdog in a tight loop (or a failure in the system application causing the main loop to run faster than expected) a watch dog delta value can be specified to disallow resetting of the Watchdog counter before a certain period of time. Any attempt to reset the watchdog when the watchdog counter has not counted down to delta value results in

device reset.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	RSWDT_Initialize	Initializes given instance of the RSWDT peripheral.

b) Setup Functions

	Name	Description
	RSWDT_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

c) Transaction Functions

	Name	Description
	RSWDT_Enable	Enables the RSWDT peripheral with/without Windowmode.
	RSWDT_Disable	Disables the RSWDT peripheral.
	RSWDT_Clear	Restart the RSWDT counter.

d) Data Types and Constants

	Name	Description
	RSWDT_CALLBACK	Defines the data type and function signature for the RSWDT peripheral callback function.
	RSWDT_MODE	Defines the data type for the RSWDT Modes.

a) Initialization Function

RSWDT_Initialize Function

Initializes given instance of the RSWDT peripheral.

File

plib_rswdt.h

C

```
void RSWDT_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the RSWDT peripheral as configured by the user.

Preconditions

None.

Example

```
RSWDT_Initialize();
```

Function

void RSWDT_Initialize(void)

b) Setup Functions

RSWDT_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

File

plib_rswdt.h

C

```
void RSWDT_CallbackRegister(RSWDT_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the Timeout event occurs. It also passes a context value that is passed into the function when it is called. This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. Note that the value of NULL will still be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the RSWDT_CALLBACK type definition for additional information. This feature may or may not be available based on device.

Preconditions

RSWDT_Initialize must have been called for the associated RSWDT instance. Reset mode must be disabled in MCC configuration

Example

```
RSWDT_CallbackRegister(MyCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the RSWDT_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void RSWDT_CallbackRegister(RSWDT_CALLBACK callback, uintptr_t context)

c) Transaction Functions

RSWDT_Enable Function

Enables the RSWDT peripheral with/without Windowmode.

File

plib_rswdt.h

C

```
void RSWDT_Enable(RSWDT_MODE mode);
```

Returns

None.

Description

This function Enables the RSWDT peripheral.

Remarks

Certain devices does not allow disabling Watchdog timers.

Preconditions

None.

Example

```
RSWDT_Enable(RSWDT_MODE_WINDOW);
```

Parameters

Parameters	Description
mode	Selects the watch dog timer mode.

Function

void RSWDT_Enable(RSWDT_MODE mode)

RSWDT_Disable Function

Disables the RSWDT peripheral.

File

plib_rswdt.h

C

```
void RSWDT_Disable();
```

Returns

None.

Description

This function is used to stop the RSWDT counter and disable RSWDT peripheral.

Remarks

Certain devices does not allow disabling Watchdog timers once enabled.

Preconditions

None

Example

```
RSWDT_Disable();
```

Function

void RSWDT_Disable(void)

RSWDT_Clear Function

Restart the RSWDT counter.

File

plib_rswdt.h

C

```
void RSWDT_Clear();
```

Returns

None.

Description

This function is used to restart the RSWDT counter to avoid timeout reset/interrupt.

Preconditions

None

Example

```
//Application loop
while(1)
{
    RSWDT_Clear();
    //user code
}
```

Function

void RSWDT_Clear(void)

d) Data Types and Constants

RSWDT_CALLBACK Type

Defines the data type and function signature for the RSWDT peripheral callback function.

File

plib_rswdt.h

C

```
typedef void (* RSWDT_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the RSWDT peripheral callback function. The RSWDT peripheral will call back the client's function with this signature when the Timeout event has occurred. This feature may or may not be available based on device.

Preconditions

RSWDT_Initialize must have been called for the given RSWDT peripheral instance and RSWDT_CallbackRegister must have

been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

RSWDT_MODE Enumeration

Defines the data type for the RSWDT Modes.

File

plib_rswdt.h

C

```
typedef enum {
  RSWDT_MODE_NORMAL,
  RSWDT_MODE_WINDOW
} RSWDT_MODE;
```

Members

Members	Description
RSWDT_MODE_NORMAL	Normal Mode of Operation
RSWDT_MODE_WINDOW	Window mode of operation

Description

RSWDT Mode

This may be used to initialize the RSWDT in window or normal mode.

Remarks

None.

Files

Files

Name	Description
plib_rswdt.h	Interface definition of the Watch Dog Timer Plib (RSWDT).

Description

This section will list only the library's interface header file(s).

plib_rswdt.h

Interface definition of the Watch Dog Timer Plib (RSWDT).

Enumerations

	Name	Description
	RSWDT_MODE	Defines the data type for the RSWDT Modes.

Functions

	Name	Description
	RSWDT_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.
	RSWDT_Clear	Restart the RSWDT counter.
	RSWDT_Disable	Disables the RSWDT peripheral.
	RSWDT_Enable	Enables the RSWDT peripheral with/without Windowmode.
	RSWDT_Initialize	Initializes given instance of the RSWDT peripheral.

Types

	Name	Description
	RSWDT_CALLBACK	Defines the data type and function signature for the RSWDT peripheral callback function.

Description

Interface definition of RSWDT PLIB.

This file defines the interface for the RSWDT Plib. It allows user to setup timeout duration and restart watch dog timer.

File Name

plib_rswdt.h

Company

Microchip Technology Inc.

RTC Peripheral Library Help

This section provides interface to use Real-Time Clock (RTC) peripheral

Introduction

This section provides brief overview of Real-Time Clock (RTC) peripheral.

Description

The RTC provides a full binary-coded decimal (BCD) clock that includes century (19/20), year (with leap years), month, day, hours, minutes, and seconds. The RTC can operate in 24-hour mode or in 12-hour mode with an AM/PM indicator. The reference clock is the Slow Clock (SLCK) which can be driven internally or by an external 32.768 kHz crystal.

The RTC continues to run in the device's low-power sleep modes, to track the current time and date information. The RTC can be used as a source to wake up the system at a scheduled time or periodically using the alarm functions.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library.

Description

The RTC keeps track of current time and generate alarm at desired time. The RTC Alarm has five programmable fields: month, day, hours, minutes, and seconds.

Each of these fields can be enabled or disabled individually to match the alarm condition:

- If all the fields are enabled, an alarm flag is generated (the corresponding flag is asserted and an interrupt generated if enabled) at a given month, date, hour/minute/second.
- If only the “seconds” field is enabled, then an alarm is generated every minute.

Depending on the combination of fields enabled, a large number of possibilities are available to the user application ranging from minutes to 365/366 days.

To avoid unwanted side effects, verification is performed on century, year, month, day, hours, minutes and seconds when user setup time and alarm. If any field fails its check the new data is not loaded into the destination register/counter and returns "false" status. When the user application programs acceptable time/date fields, time set and alarm set API returns "true".

This example demonstrates how to set the RTC time and alarm time to generate alarm interrupt at desired time of the day.

```
void RTC0_Callback( uintptr_t context, uint32_t int_cause )
{
    if (int_cause & RTC_INT_ALARM)
    {
        /* Alarm has Occurred */
    }
}

int main ( void )
{
    /* Initialize System Time and Alarm Time */
    struct tm sys_time;
    struct tm alarm_time;

    /* Register Callback */
    RTC0_CallbackRegister(RTC0_Callback, (uintptr_t) NULL);


    /* Set Time and date
      15-01-2018 12:00:00 Monday */
    sys_time.tm_hour = 12;
    sys_time.tm_sec = 00;
    sys_time.tm_min = 00;
    sys_time.tm_mon = 0;
    sys_time.tm_year = 118;
    sys_time.tm_mday = 15;
    sys_time.tm_wday = 1;

    if (RTC0_TimeSet(&sys_time) == false)
    {
        // Invalid time and/or data input
    }

    /* Set Alarm Time and date. Generate alarm every day when Hour, Minute and Seconds match.
       15-01-2018 12:00:20 Monday */
    alarm_time.tm_hour = 12;
    alarm_time.tm_sec = 20;
    alarm_time.tm_min = 00;
    alarm_time.tm_mon = 0;
    alarm_time.tm_year = 118;
    alarm_time.tm_mday = 15;
    alarm_time.tm_wday = 1;

    if (RTC0_AlarmSet(&alarm_time, RTC_ALARM_MASK_HHMISS) == false)
    {
        // Invalid Alarm Time and/or data input
    }
```

```
    /* Get System Time */
    while ( true )
    {
        RTC0_TimeGet(&sys_time);
        printf("System time is:   %02d:%02d:%02d\r",sys_time.tm_hour, sys_time.tm_min,
sys_time.tm_sec);
    }

}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Functions

	Name	Description
	RTCx_Initialize	Initializes given instance of the RTC peripheral.

b) Setup Functions

	Name	Description
	RTCx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

c) Transaction Functions

	Name	Description
	RTCx_TimeSet	Sets the Time for the RTC peripheral.
	RTCx_AlarmSet	Sets up the Alarm
	RTCx_InterruptDisable	Disables the specified RTC interrupt
	RTCx_InterruptEnable	Enables the specified RTC interrupt
	RTCx_TimeGet	Reads the current time

d) Data Types and Constants

	Name	Description
	RTC_ALARM_MASK	Defines the data type for the RTC Alarm Mask.
	RTC_CALLBACK	Defines the data type and function signature for the RTC peripheral callback function.
	RTC_INT_MASK	Defines the Interrupt mask for RTC events.

a) Initialization Functions

RTCx_Initialize Function

Initializes given instance of the RTC peripheral.

File

plib_rtc.h

C

```
void RTCx_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the RTC peripheral as configured by the user in MHC.

Preconditions

None.

Example

```
RTC0_Initialize();
```

Function

void RTCx_Initialize(void)

b) Setup Functions

RTCx_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

File

plib_rtc.h

C

```
void RTCx_CallbackRegister(RTC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the Alarm occurs, It also passes a context value that is passed into the function when it is called. This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. In this case the value NULL will be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the RTC_CALLBACK type definition for additional information.

Preconditions

None.

Example

```
RTC0_CallbackRegister(MyCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the RTC_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void RTCx_CallbackRegister(RTC_CALLBACK callback, uintptr_t context)

c) Transaction Functions

RTCx_TimeSet Function

Sets the Time for the RTC peripheral.

File

plib_rtc.h

C

```
bool RTCx_TimeSet(struct tm * time);
```

Returns

- True - Valid Time and/or Date is configured.
- False - Invalid Time and/or Date input provided.

Description

This function updates the time for RTC peripheral as configured by the user.

Remarks

The structure can be deleted once the API returns to free up the memory

Preconditions

None.

Example

```
struct tm dateTime;
//Hour
dateTime.tm_hour = 12;
//second
dateTime.tm_sec = 0;
//Minute
dateTime.tm_min = 34;
//Month
dateTime.tm_mon = 11;
//Year
dateTime.tm_year = 2017;
//day of the month
dateTime.tm_mday = 14;
//day of the week
dateTime.tm_wday = 1;
RTC0_TimeSet( &dateTime );
```

Parameters

Parameters	Description
time	Pointer to struct tm.

Function

bool RTCx_TimeSet(struct tm *time)

RTCx_AlarmSet Function

Sets up the Alarm

File

plib_rtc.h

C

```
bool RTCx_AlarmSet(struct tm * alarmTime, RTC_ALARM_MASK mask);
```

Returns

- True - Valid Alarm Time and/or Date is configured.
- False - Invalid Alarm Time and/or Date input.

Description

This function is used to set up the alarm time and Alarm mask based on user input.

Preconditions

None

Example

```
struct tm alarmTime;
//Hour
alarmTime.tm_hour = 12;
//second
alarmTime.tm_sec = 0;
//Minute
alarmTime.tm_min = 34;
//Month
alarmTime.tm_mon = 11;
//Year
alarmTime.tm_year = 2017;
//day of the month
alarmTime.tm_mday = 14;
//set up alarm for every month
RTC0_AlarmSet( &alarmTime, RTC_ALARM_MASK_DDHHMISS );
```

Remark: The structure can be deleted once the API returns to free up the memory

Parameters

Parameters	Description
alarmTime	Alarm Time
mask	Alarm mask to be used for notification

Function

bool RTCx_AlarmSet(struct tm *alarmTime, RTC_ALARM_MASK mask);

RTCx_InterruptDisable Function

Disables the specified RTC interrupt

File

plib_rtc.h

C

```
void RTCx_InterruptDisable(RTC_INT_MASK interrupt);
```

Returns

None.

Description

This function is used to disable a specific RTC interrupt source

Preconditions

None.

Example

```
RTC0_InterruptDisable(RTC_INT_ALARM);
```

Parameters

Parameters	Description
interrupt	Interrupt source to be disabled.

Function

void RTCx_InterruptDisable(RTC_INT_MASK interrupt)

RTCx_InterruptEnable Function

Enables the specified RTC interrupt

File

plib_rtc.h

C

```
void RTCx_InterruptEnable(RTC_INT_MASK interrupt);
```

Returns

None.

Description

This function is used to enable a specific RTC interrupt source

Preconditions

None.

Example

```
RTC0_InterruptEnable(RTC_INT_ALARM);
```

Parameters

Parameters	Description
interrupt	Interrupt source to be enabled.

Function

void RTCx_InterruptEnable(RTC_INT_MASK interrupt)

RTCx_TimeGet Function

Reads the current time

File

plib_rtc.h

C

```
void RTCx_TimeGet(struct tm * time);
```

Returns

None

Description

This function is used to read the current date and time from the RTC.

Preconditions

None.

Example

```
struct tm dateTime;
//get the current date and time
RTC0_TimeGet( &dateTime );
```

Parameters

Parameters	Description
time	This is an output parameter that will be filled with the current date and time.

Function

void RTCx_TimeGet(struct tm *time);

d) Data Types and Constants

RTC_ALARM_MASK Enumeration

Defines the data type for the RTC Alarm Mask.

File

plib_rtc.h

C

```
typedef enum {
  RTC_ALARM_MASK_OFF = 0x00,
  RTC_ALARM_MASK_SS = 0x01,
  RTC_ALARM_MASK_MI = 0x02,
  RTC_ALARM_MASK_HH = 0x04,
  RTC_ALARM_MASK_DD = 0x08,
  RTC_ALARM_MASK_MO = 0x10,
  RTC_ALARM_MASK_MISS = 0x03,
  RTC_ALARM_MASK_HHMISS = 0x07,
  RTC_ALARM_MASK_DDHHMISS = 0x0f,
  RTC_ALARM_MASK_MODDHHMISS = 0x1f
} RTC_ALARM_MASK;
```

Members

Members	Description
RTC_ALARM_MASK_OFF = 0x00	NO Alarm
RTC_ALARM_MASK_SS = 0x01	Every minute, seconds alarm enable
RTC_ALARM_MASK_MI = 0x02	Minute alarm enable
RTC_ALARM_MASK_HH = 0x04	Hour alarm enable
RTC_ALARM_MASK_DD = 0x08	Date alarm enable
RTC_ALARM_MASK_MO = 0x10	Month alarm enable
RTC_ALARM_MASK_MISS = 0x03	Every hour
RTC_ALARM_MASK_HHMISS = 0x07	Every day
RTC_ALARM_MASK_DDHHMISS = 0x0f	Every month

RTC_ALARM_MASK_MODDHHMISS = 0x1f	Every year

Description

RTC Alarm Mask

This is used to set up the alarm for RTC.

Remarks

None.

RTC_CALLBACK Type

Defines the data type and function signature for the RTC peripheral callback function.

File

plib_rtc.h

C

```
typedef void (* RTC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the RTC peripheral callback function. The RTC peripheral will call back the client's function with this signature once the alarm match occurs.

Preconditions

RTC_CallbackRegister must have been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

RTC_INT_MASK Enumeration

Defines the Interrupt mask for RTC events.

File

plib_rtc.h

C

```
typedef enum {
  RTC_INT_ALARM = 0x02,
  RTC_INT_TIME = 0x08,
  RTC_INT_CALENDAR = 0x10
} RTC_INT_MASK;
```

Members

Members	Description
RTC_INT_ALARM = 0x02	Alarm Event
RTC_INT_TIME = 0x08	Time Event
RTC_INT_CALENDAR = 0x10	Calendar enable

Description

RTC Interrupt Mask

This may be used to check the interrupt source for RTC.

Remarks

None.

Files

Files

Name	Description
plib_rtc.h	Interface definition of the Real Time Counter and Calendar (RTC) Plib.

Description

This section will list only the library's interface header file(s).

plib_rtc.h

Interface definition of the Real Time Counter and Calendar (RTC) Plib.

Enumerations

	Name	Description
	RTC_ALARM_MASK	Defines the data type for the RTC Alarm Mask.
	RTC_INT_MASK	Defines the Interrupt mask for RTC events.

Functions

	Name	Description
	RTCx_AlarmSet	Sets up the Alarm
	RTCx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.
	RTCx_Initialize	Initializes given instance of the RTC peripheral.
	RTCx_InterruptDisable	Disables the specified RTC interrupt
	RTCx_InterruptEnable	Enables the specified RTC interrupt
	RTCx_TimeGet	Reads the current time
	RTCx_TimeSet	Sets the Time for the RTC peripheral.

Types

	Name	Description
	RTC_CALLBACK	Defines the data type and function signature for the RTC peripheral callback function.

Description

Interface definition of RTC PLIB.

This file defines the interface for the RTC Plib. It allows user to setup alarm and retrieve current date and time.

File Name

plib_rtc.h

Company

Microchip Technology Inc.

RTT Peripheral Library Help

This section provides interface to use Real-Time Timer (RTT) peripheral.

Introduction

This section provides brief overview of Real-Time Timer (RTT) peripheral.

Description

The Real-time Timer (RTT) is built around a 32-bit counter used to count roll-over events of the programmable 16-bit prescaler driven from the 32-kHz slow clock source.

The RTT can also be configured to be driven by the 1Hz RTC signal, thus taking advantage of a calibrated 1Hz clock.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This Section explains how to use the Peripheral Library.

Description

RTT module generates a periodic interrupt and/or triggers an alarm on a programmed value.

- Periodic interrupt is generated when prescaler roll-over generates clock to increment real-time timer counter
- Alarm interrupt is generated when the time value matches the programmed alarm value.

The RTT continues to run in the device's low-power sleep modes, and it can be used as a source to wake up the system using alarm function.

Periodic Interrupt

This example below demonstrates how to use RTT for periodic interrupt generation.

```
void Timeout_Handler (RTT_INTERRUPT_TYPE type, uintptr_t context)
{
    if(type == RTT_PERIODIC)
    {
        LED_Toggle();
    }
}

int main ( void )
{

    /* Register Callback */
    RTT0_CallbackRegister(Timeout_Handler, (uintptr_t) NULL);

    /* Enable RTT */
    RTT0_Enable();


    while ( true )
    {
```

```
    }

}
```

The below example demonstrates how to set alarm and watch it being triggered when the timer reaches the corresponding value.

```
void Timeout_Handler (RTT_INTERRUPT_TYPE type, uintptr_t context)
{
    if(type == RTT_ALARM)
    {
        // Alarm has occurred
    }
}

int main ( void )
{

    /* Register Callback */
    RTT0_CallbackRegister(Timeout_Handler, (uintptr_t) NULL);

    /* Set Alarm Value */
    RTT_AlarmValueSet(10);

    /* Enable RTT */
    RTT0_Enable();

    while ( true )
    {

    }

}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	RTTx_Initialize	Initialize the RTT peripheral

b) Setup Functions

	Name	Description
	RTTx_CallbackRegister	Sets the pointer to the function (and it's context) to be called.

c) Transaction Functions

	Name	Description
	RTTx_AlarmValueSet	Sets the Alarm value for the RTT peripheral.
	RTTx_Disable	Disables the real Time Timer
	RTTx_DisableInterrupt	Disable RTT interrupts
	RTTx_Enable	Enables the Real Time Timer
	RTTx_EnableInterrupt	Enable RTT interrupts
	RTTx_FrequencyGet	Returns the current RTT frequency
	RTTx_PrescalarUpdate	Updates the prescalar for the RTT peripheral.
	RTTx_TimerValueGet	Returns the current timer value

d) Data Types and Constants

	Name	Description
	RTT_CALLBACK	Defines the data type and function signature for the RTT peripheral callback function.
	RTT_INTERRUPT_TYPE	Defines the data type for the RTT INTERRUPT TYPE.
	RTT_OBJECT	Defines the data type for the RTT OBJECT.

a) Initialization Function

RTTx_Initialize Function

Initialize the RTT peripheral

File

plib_rtt.h

C

```
void RTTx_Initialize();
```

Returns

None.

Description

This function rests the RTT and initialize the RTT as per MHC selection

Preconditions

None.

Example

```
RTT0_Initialize();
```

Function

void RTTx_Initialize(void);

b) Setup Functions

RTTx_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called.

File

plib_rtt.h

C

```
void RTTx_CallbackRegister(RTT_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the interrupt occurs. It also passes a context value that is passed into the function when it is called. This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. In this case the value NULL will be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the RTT_CALLBACK type definition for additional information.

Preconditions

None.

Example

```
void My_callback(uintptr_t context)
{
    //Interrupt recieved stop RTT0
    RTT0_Disable();
}
int main(void)
{
    RTT0_Initialize();

    //Enable callback for the periodic interrupt
    RTT0_CallbackRegister(My_callback, NULL);

    //start RTT0
    RTT0_Enable();
}
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the RTT_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void RTTx_CallbackRegister(RTT_CALLBACK callback, uintptr_t context);

c) Transaction Functions

RTTx_AlarmValueSet Function

Sets the Alarm value for the RTT peripheral.

File

plib_rtt.h

C

```
void RTTx_AlarmValueSet(uint32_t alarm);
```

Returns

None.

Description

This function updates the Alarm compare value for RTT peripheral.

Preconditions

None.

Example

```
RTT0_AlarmValueSet(100);
```

Parameters

Parameters	Description
alarm	The alarm value to be updated

Function

void RTTx_AlarmValueSet(uint32_t alarm);

RTTx_Disable Function

Disables the real Time Timer

File

plib_rtt.h

C

```
void RTTx_Disable();
```

Returns

None.

Description

This API is used to stop the real time timer.

Preconditions

RTT should be running.

Example

```
void My_callback(uintptr_t context)
{
    //Interrupt recieved stop RTT0
    RTT0_Disable();
}
int main(void)
{
    RTT0_Initialize();

    //Enable callback for the periodic interrupt
    RTT0_CallbackRegister(My_callback, NULL);

    //start RTT0
    RTT0_Enable();
}
```

Function

void RTTx_Disable(struct tm *time)

RTTx_DisableInterrupt Function

Disable RTT interrupts

File

plib_rtt.h

C

```
void RTTx_DisableInterrupt(RTT_INTERRUPT_TYPE type);
```

Returns

None.

Description

This function is used to disable periodic and alarm interrupts

Preconditions

None.

Example

```
//disable Alarm interrupt for RTT0
RTT0_DisableInterrupt(RTT_ALARM);
```

Parameters

Parameters	Description
type	Interrupt to be disabled

Function

void RTTx_DisableInterrupt(RTT_INTERRUPT_TYPE type);

RTTx_Enable Function

Enables the Real Time Timer

File

plib_rtt.h

C

```
void RTTx_Enable();
```

Returns

None.

Description

This API is used to start the real time timer.

Preconditions

RTTx_Initialize must be called in order to configure prescalar and source clock

Example

```
RTT0_Initialize();
RTT0_Enable();
```

Function

void RTTx_Enable(void);

RTTx_EnableInterrupt Function

Enable RTT interrupts

File

plib_rtt.h

C

```
void RTTx_EnableInterrupt(RTT_INTERRUPT_TYPE type);
```

Returns

None.

Description

This function is used to enable periodic and alarm interrupts

Preconditions

None.

Example

```
//Enable Alarm interrupt for RTT0
RTT0_EnableInterrupt(RTT_ALARM);
```

Parameters

Parameters	Description
type	Interrupt to be enabled

Function

void RTTx_EnableInterrupt (RTT_INTERRUPT_TYPE type);

RTTx_FrequencyGet Function

Returns the current RTT frequency

File

plib_rtt.h

C

```
uint32_t RTTx_FrequencyGet();
```

Returns

Returns the operating frequency of RTT

Description

This function is used to get the frequncy of the RTT timer

Preconditions

None.

Example

```
uint32_t frequency = 0;
frequency = RTT0_FrequencyGet();
```

Function

uint32_t RTTx_FrequencyGet(void)

RTTx_PrescalarUpdate Function

Updates the prescalar for the RTT peripheral.

File

plib_rtt.h

C

```
void RTTx_PrescalarUpdate(uint16_t prescale);
```

Returns

None.

Description

This function is used to update the prescalar for the RTT peripheral

Preconditions

None.

Example

```
RTT0_PrescalarUpdate(10);
```

Parameters

Parameters	Description
prescale	The prescale value to be loaded.

Function

void RTTx_PrescalarUpdate(uint16_t prescale);

RTTx_TimerValueGet Function

Returns the current timer value

File

plib_rtt.h

C

```
uint32_t RTTx_TimerValueGet();
```

Returns

The current timer count

Description

This function is used the counter value of RTT. This value can be use to calculate the time elapsed.

Preconditions

None.

Example

```
uint32_t frequency = 0;
uint32_t count = 0;
uint64_t time_elapsed;

frequency = RTT0_FrequencyGet();
count = RTT0_TimerValueGet();

time_elapsed = count/frequency;
```

Function

uint32_t RTTx_TimerValueGet(void);

d) Data Types and Constants

RTT_CALLBACK Type

Defines the data type and function signature for the RTT peripheral callback function.

File

plib_rtt.h

C

```
typedef void (* RTT_CALLBACK)(RTT_INTERRUPT_TYPE type, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the RTT peripheral callback function. The RTT peripheral will call back the client's function with this signature once the alarm match or periodic interrupt occurs.

Preconditions

RTT_CallbackRegister must have been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).
type	Reason for the interrupt

RTT_INTERRUPT_TYPE Enumeration

Defines the data type for the RTT INTERRUPT TYPE.

File

plib_rtt.h

C

```
typedef enum {
  RTT_PERIODIC,
  RTT_ALARM
} RTT_INTERRUPT_TYPE;
```

Members

Members	Description
RTT_PERIODIC	Periodic interrupt
RTT_ALARM	Alarm Interrupt

Description

RTT_INTERRUPT_TYPE

This may be used for the interrupt type.

Remarks

None.

RTT_OBJECT Structure

Defines the data type for the RTT OBJECT.

File

plib_rtt.h

C

```
typedef struct {
  RTT_CALLBACK callback_periodic;
  uintptr_t context_periodic;
} RTT_OBJECT;
```

Members

Members	Description
RTT_CALLBACK callback_periodic;	A pointer to a function with a calling signature defined by the RTT_CALLBACK data type.
uintptr_t context_periodic;	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Description

RTT_OBJECT

This structure defines RTT object parameters.

Remarks

None.

Files

Files

Name	Description
plib_rtt.h	Interface definition of rtt Plib.

Description

This section will list only the library's interface header file(s).

plib_rtt.h

Interface definition of rtt Plib.

Enumerations

	Name	Description
	RTT_INTERRUPT_TYPE	Defines the data type for the RTT INTERRUPT TYPE.

Functions

	Name	Description
	RTTx_AlarmValueSet	Sets the Alarm value for the RTT peripheral.
	RTTx_CallbackRegister	Sets the pointer to the function (and it's context) to be called.
	RTTx_Disable	Disables the real Time Timer
	RTTx_DisableInterrupt	Disable RTT interrupts
	RTTx_Enable	Enables the Real Time Timer
	RTTx_EnableInterrupt	Enable RTT interrupts
	RTTx_FrequencyGet	Returns the current RTT frequency
	RTTx_Initialize	Initialize the RTT peripheral
	RTTx_PrescalarUpdate	Updates the prescalar for the RTT peripheral.
	RTTx_TimerValueGet	Returns the current timer value

Structures

	Name	Description
	RTT_OBJECT	Defines the data type for the RTT OBJECT.

Types

	Name	Description
	RTT_CALLBACK	Defines the data type and function signature for the RTT peripheral callback function.

Description

Interface definition of RTT PLIB.

This file defines the interface for the rtt Plib. It allows user to start, stop and configure the on-chip real time timer.

File Name

plib_rtt.h

Company

Microchip Technology Inc.

SDRAMC Peripheral Library Help

This section provides interface to use SDRAMC peripheral.

Introduction

This section provides brief overview of SDRAMC peripheral.

Description

The SDRAM Controller (SDRAMC) extends the memory capabilities of a chip by providing the interface to external 16-bitDRAM devices. The page size supports ranges from 2048 to 8192 and the number of columns from 256 to 2048. It supports byte (8-bit),

half-word (16-bit) and word (32-bit) accesses.

The SDRAMC supports a read or write burst length of one location. It keeps track of the active row in each bank, thus maximizing SDRAM performance, e.g., the application may be placed in one bank and data in the other banks. For optimized performance, it is advisable to avoid accessing different rows in the same bank.

The SDRAMC supports a CAS latency of 1, 2 or 3 and optimizes the read access depending on the frequency.

Supports Self-refresh, Power-down and Deep Power-down modes to minimize power consumption on the SDRAM device.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains how to use the peripheral library

Description

SDRAMC controller is initialized as configured in the MHC as part of System Initialization.

Library Interface

a) Initialization Function

	Name	Description
	SDRAMC_Initialize	Initializes and Enables the SDRAM Controller.

Description

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

SDRAMC_Initialize Function

Initializes and Enables the SDRAM Controller.

File

plib_sdramc.h

C

```
void SDRAMC_Initialize();
```

Returns

None.

Description

This function Enables the external memory controller module(s).

Remarks

This routine must be called before any attempt to access external memory.

Not all features are available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
SDRAMC_Initialize();
```

Function

void SDRAMC_Initialize (void)

Files

Files

Name	Description
plib_sdramc.h	SDRAMC Controller (SDRAMC).

Description

This section will list only the library's interface header file(s).

plib_sdramc.h

SDRAMC Controller (SDRAMC).

Functions

	Name	Description
	SDRAMC_Initialize	Initializes and Enables the SDRAM Controller.

Description

The SDRAMC System Memory interface provides a simple interface to manage the externally connected SDRAM device. This file contains the source code to initialize the SDRAMC controller

File Name

plib_sdramc.h

Company

Microchip Technology Inc.

SMC Peripheral Library Help

This section provides interface to use Static Memory Controller (SMC) peripheral.

Introduction

This section provides brief overview of Static Memory Controller (SMC).

Description

The Static Memory Controller (SMC) is part of the External Bus Interface (EBI). The EBI is a highly flexible module for interfacing external memories and memory addressable peripherals such as LCD controllers and advanced communication controllers. The

EBI module has four separate Chip Select blocks with individual address ranges and wait state control. Additional Chip Select lines can be decoded externally.

The SMC handles several types of external memory and peripheral devices, such as SRAM, PSRAM, PROM, EPROM, EEPROM, LCD Module, NOR Flash and NAND Flash.

The SMC generates the signals that control the access to the external memory devices or peripheral devices. It has 4 chip selects, a 24-bit address bus, and a configurable 8 or 16-bit data bus. Separate read and write control signals allow for direct memory and peripheral interfacing. Read and write signal waveforms are fully adjustable

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains how to use the peripheral library

Description

Static Memory Controller (SMC) is initialized as configured in the MHC as part of System Initialization.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	SMC_Initialize	Initializes Static Memory Controller (SMC) interface.

a) Initialization Function

SMC_Initialize Function

Initializes Static Memory Controller (SMC) interface.

File

plib_smc.h

C

```
void SMC_Initialize();
```

Returns

None.

Description

This function initializes the Static Memory Controller (SMC) peripheral interface.

Remarks

None.

Preconditions

None.

Example

```
//Example 1: Do not alter the configuration bit settings
SMC_Initialize();
```

Function

void SMC_Initialize (void)

Files

Files

Name	Description
plib_smc.h	Static Memory Controller (SMC) peripheral library interface. This file contains the source code to initialize the SMC controller

Description

This section will list only the library's interface header file(s).

plib_smc.h

Static Memory Controller (SMC) peripheral library interface. This file contains the source code to initialize the SMC controller

Functions

	Name	Description
	SMC_Initialize	Initializes Static Memory Controller (SMC) interface.

Description

This file defines the interface to the SMC peripheral library. This library provides access to and control of the associated peripheral instance.

File Name

plib_smc.h

Company

Microchip Technology Inc.

SPI Peripheral Library Help

This section provides an interface to use Serial Peripheral Interface (SPI) peripheral.

Introduction

This library provides an interface to use Serial Peripheral Interface (SPI) module.

Description

The Serial Peripheral Interface (SPI) is a high-speed synchronous data transfer interface using three or four pins. It allows fast

communication between a master device and one or more peripheral devices. The master initiates and controls all data transactions. The SPI master initiates a communication cycle by pulling low the Slave Select (SS) pin of the desired slave. Master and slave prepare data to be sent in their respective shift registers, and the master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from master to slave on the Master Out - Slave In (MOSI) line, and from slave to master on the Master In - Slave Out (MISO) line.

Configuring the Library

This section describes how to configure the peripheral library using MHC

Description

The SPI library can transfer a block of data in a blocking and non-blocking manner.

- To transfer block of data in non-blocking manner, the peripheral interrupt is used to manage the transfer and hence the "Interrupt Mode" option should be **enabled** in MHC
- To transfer block of data in blocking manner, the CPU polls the peripheral register and hence the "Interrupt Mode" option should be **disabled** in MHC

Using the Library

This section explains how to use the peripheral library

Description

The SPI peripheral library supports following SPI transfers:

- Master Write: The master writes a block of data to the slave
- Master Read: The master reads a block of data from the slave
- Master Write/Read: The master writes and then reads back a block of data from slave.

The buffer model can process a block of data in a blocking and non-blocking manner.

- With blocking, the read/write APIs will block until all of the data is transferred. The CPU polls the peripheral register continuously to manage the transfer.
- With non-blocking, the read/write APIs will return immediately. The peripheral interrupt is used to manage the transfer. The user application can either use polling or callback method to indicate the transfer status.
  - With polling, the application will need to continuously poll to check if the transfer is completed.
  - With callback, the registered callback function will be called once the transfer is completed. This means we do not have to poll continuously.

Blocking Transfer

The below example shows the blocking transfer.

```
#define TX_NUM_BYTES      4
#define RX_NUM_BYTES      10

uint8_t txBuffer[TX_NUM_BYTES];
uint8_t rxBuffer[RX_NUM_BYTES];


int main(void)
{
    SPIx_WriteRead(&txBuffer, TX_NUM_BYTES, &rxBuffer, RX_NUM_BYTES);
}
```

Non-blocking Transfer with polling

The example code demonstrates polling method.

```
#define TX_NUM_BYTES      4
#define RX_NUM_BYTES      10

uint8_t txBuffer[TX_NUM_BYTES];
uint8_t rxBuffer[RX_NUM_BYTES];

int main(void)
{
```

```
    /* Submit Write-Read Request */
    SPIx_WriteRead(&txBuffer, TX_NUM_BYTES, &rxBuffer, RX_NUM_BYTES);

    /* Poll and wait for the transfer to complete */
    while(SPIx_IsBusy());

}
```

Non-blocking Transfer with callback

The example code demonstrates callback method.

```
#define TX_NUM_BYTES     4
#define RX_NUM_BYTES     10

uint8_t txBuffer[TX_NUM_BYTES];
uint8_t rxBuffer[RX_NUM_BYTES];

void SPIx_Callback(uintptr_t context)
{
    // Transfer is completed successfully
}

int main(void)
{
    /* Register Callback function */
    SPIx_CallbackRegister(SPIx_Callback, (uintptr_t)NULL );

    /* Submit Write Request */
    SPIx_WriteRead(&txBuffer, TX_NUM_BYTES, &rxBuffer, RX_NUM_BYTES);
}
```

Library Interface

a) Initialization Function

	Name	Description
	SPIx_Initialize	Initializes SPI x module of the device

b) Setup Functions

	Name	Description
	SPIx_TransferSetup	Setup SPI operational parameters as desired by the client.
	SPIx_CallbackRegister	Allows application to register callback with PLIB.

c) Transaction Functions

	Name	Description
	SPIx_Read	Read data on SPI x peripheral
	SPIx_Write	Write data on SPI x peripheral
	SPIx_WriteRead	Write and Read data on SPI x peripheral

d) Status Functions

	Name	Description
	SPIx_IsBusy	Returns transfer status of SPI x

e) Data Types and Constants

	Name	Description
	SPI_CALLBACK	Pointer to a SPI Callback function.
	SPI_CLOCK_PHASE	Identifies SPI Clock Phase Options
	SPI_CLOCK_POLARITY	Identifies SPI Clock Polarity Options
	SPI_DATA_BITS	Identifies SPI bits per transfer

	SPI_TRANSFER_SETUP	Identifies the setup parameters which can be changed dynamically.

Description

This section describes the Application Programming Interface (API) functions of the SPI Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

SPIx_Initialize Function

Initializes SPI x module of the device

File

plib_spi.h

C

```
void SPIx_Initialize();
```

Returns

None.

Description

This function initializes SPI x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Remarks

This function must be called only once and before any other SPI function is called.

Preconditions

MHC GUI should be configured with the right values.

Example

```
SPI1_Initialize();
```

Function

void SPIx_Initialize (void);

b) Setup Functions

SPIx_TransferSetup Function

Setup SPI operational parameters as desired by the client.

File

plib_spi.h

C

```
bool SPIx_TransferSetup(SPI_TRANSFER_SETUP * setup, uint32_t spiSourceClock);
```

Returns

- true: if setup was successful, API returns true.

- false: if spiSourceClock and SPI clock frequencies are such that resultant SCBR value is out of the possible range, then API returns false.

Description

This function setup SPI x with the values needed by the client dynamically. Values passed through setup structure will be the new setup for SPI transmission. Even if user is not intending to modify all the setup parameters, he must update his setup structure with all its parameters. If user do not update all the setup structure elements, then unexpected SPI behavior may occur.

User need not call this function if he has configured the setup in GUI and there is no dynamic change needed in any of the parameters.

Remarks

User need not call this function if he has configured the setup in GUI and there is no dynamic change needed in any of the parameters.

Preconditions

SPI x must first be initialized using SPIx_Initialize().

Example

```
SPI_TRANSFER_SETUP setup;
setup.clockFrequency = 1000000;
setup.clockPhase = SPI_CLOCK_PHASE_TRAILING_EDGE;
setup.clockPolarity = SPI_CLOCK_POLARITY_IDLE_LOW;
setup.dataBits = SPI_DATA_BITS_8;

// Assuming 150 MHz as peripheral Master clock frequency
if (SPI1_TransferSetup (&setup, 150000000) == false)
{
    // this means setup could not be done, debug the reason.
}
```

Parameters

Parameters	Description
spiSourceClock	Source Clock frequency in Hz on which SPI module is running.
*setup	pointer to the data structure of type SPI_TRANSFER_SETUP which has the list of elements to be setup for a client.

Function

bool SPIx_TransferSetup(SPI_TRANSFER_SETUP *setup, uint32_t spiSourceClock);

SPIx_CallbackRegister Function

Allows application to register callback with PLIB.

File

plib_spi.h

C

```
void SPIx_CallbackRegister(const SPI_CALLBACK* callback, uintptr_t context);
```

Returns

None.

Description

This function allows application to register a callback function for the PLIB to call back when requested data transfer operation has completed. The callback should be registered before the client performs transfer operation.

At any point if application wants to stop the callback, it can call this function with "callback" value as NULL.

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The SPIx_Initialize function must have been called.

Example

```
SPI1_CallbackRegister(&APP_SPITransferHandler, (uintptr_t)NULL);

void APP_SPITransferHandler(uintptr_t context)
{
    //Transfer was completed without error, do something else now.
}
```

Parameters

Parameters	Description
callback	Pointer to the callback function implemented by the user
context	The value of parameter will be passed back to the application unchanged, when the callback function is called. It can be used to identify any application specific value that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void SPIx_CallbackRegister

(

const SPI_CALLBACK* callback,

uintptr_t context

);

c) Transaction Functions

SPIx_Read Function

Read data on SPI x peripheral

File

plib_spi.h

C

```
bool SPIx_Read(void* pReceiveData, size_t rxSize);
```

Returns

- In Blocking mode, API returns True once the transfer is complete. It returns False if rxSize is 0 and receive data pointer is NULL.
- In interrupt mode, if previous buffer request is not completed and a new transfer request comes, then this API will reject the new request and will return "False". Also, Same as blocking mode, It returns False if rxSize is 0 and receive data pointer is NULL.

Description

This function should be used to read "rxSize" number of bytes on SPI x module. Received data is saved in the location pointed by pReceiveData.

The function will work differently as per the configuration done in MHC as described below:

Remarks

Non-blocking interrupt mode configuration implementation of this function will be used by Harmony driver implementation APIs.

Preconditions

The SPIx_Initialize function must have been called.

Callback has to be registered using SPIx_CallbackRegister API if the peripheral instance has been configured in Interrupt mode and transfer completion status needs to be communicated back to application via callback.

Example

```
uint8_t     rxBuffer[10];
size_t      rxSize = 10;
bool        reqAccepted;

reqAccepted = SPI1_Read(&rxBuffer, rxSize);
```

Parameters

Parameters	Description
*pReceiveData	Pointer to the location where received data has to be stored. It is user's responsibility to ensure pointed location has sufficient memory to store the read data. if it is NULL, that means only data transmission is expected. For 9 to 15bit mode, received data will be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 5 16-bit data are to be received, the receive size should be 10 bytes.

Function

bool SPIx_Read(void* pReceiveData, size_t rxSize);

SPIx_Write Function

Write data on SPI x peripheral

File

plib_spi.h

C

```
bool SPIx_Write(void* pTransmitData, size_t txSize);
```

Returns

- In Blocking mode, API returns True once the transfer is complete. It returns False if txSize parameter is 0 and transmit data pointer is NULL.

- In interrupt mode, if previous buffer request is not completed and a new transfer request comes, then this API will reject the new request and will return "False". Also, Same as blocking mode, It returns False if txSize parameter is 0 and transmit data pointer is NULL.

Description

This function should be used to write "txSize" number of bytes on SPI x module. Data pointed by pTransmitData is transmitted.

Remarks

Non-blocking interrupt mode configuration implementation of this function will be used by Harmony driver implementation APIs.

Preconditions

The SPIx_Initialize function must have been called.

Callback has to be registered using SPIx_CallbackRegister API if the peripheral instance has been configured in Interrupt mode and transfer completion status needs to be communicated back to application via callback.

Example

```
uint8_t     txBuffer[4];
size_t      txSize = 4;
bool        reqAccepted;

reqAccepted = SPI1_Write(&txBuffer, txSize);
```

Parameters

Parameters	Description
*pTransmitData	Pointer to the data which has to be transmitted. if it is NULL, that means only data receiving is expected. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 5 16-bit data are to be transmitted, the transmit size should be 10 bytes.

Function

bool SPIx_Write(void* pTransmitData, size_t txSize);

SPIx_WriteRead Function

Write and Read data on SPI x peripheral

File

plib_spi.h

C

```
bool SPIx_WriteRead(void* pTransmitData, size_t txSize, void* pReceiveData, size_t rxSize);
```

Returns

- In Blocking mode, API returns True once the transfer is complete. It returns False if either of the size parameters are 0 and corresponding data pointer is NULL.

- In interrupt mode, if previous buffer request is not completed and a new transfer request comes, then this API will reject the new request and will return "False". Also, Same as blocking mode, It returns False if either of the size parameters are 0 and corresponding data pointer is NULL.

Description

This function should be used to write "txSize" number of bytes and read "rxSize" number of bytes on SPI x module. Data pointed by pTransmitData is transmitted and received data is saved in the location pointed by pReceiveData.

Remarks

Non-blocking interrupt mode configuration implementation of this function. will be used by Harmony driver implementation APIs.

Preconditions

The SPIx_Initialize function must have been called.

Callback has to be registered using SPIx_CallbackRegister API if the peripheral instance has been configured in Interrupt mode and transfer completion status needs to be communicated back to application via callback.

Example

```
uint8_t     txBuffer[4];
uint8_t     rxBuffer[10];
size_t      txSize = 4;
size_t      rxSize = 10;
bool        reqAccepted;

reqAccepted = SPI1_WriteRead(&txBuffer, txSize, &rxBuffer, rxSize);
```

Parameters

Parameters	Description
*pTransmitData	Pointer to the data which has to be transmitted. if it is NULL, that means only data receiving is expected. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 5 16-bit data are to be transmitted, the transmit size should be 10 bytes.
*pReceiveData	Pointer to the location where received data has to be stored. It is user's responsibility to ensure pointed location has sufficient memory to store the read data. if it is NULL, that means only data transmission is expected. For 9 to 15bit mode, received data will be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 5 16-bit data are to be received, the receive size should be 10 bytes. if "n" number of bytes has to be received AFTER transmitting "m" number of bytes, then "rxSize" should be set as "m+n".

Function

bool SPIx_WriteRead
(
void* pTransmitData,
size_t txSize,
void* pReceiveData,
size_t rxSize
);

d) Status Functions

SPIx_IsBusy Function

Returns transfer status of SPI x

File

plib_spi.h

C

```
bool SPIx_IsBusy();
```

Returns

Returns the current status of transfer happening on SPI x.

- true: Transfer is still in progress
- false: Transfer is completed

Description

This function returns transfer status of last successful Write, Read or WriteRead request on SPI x module in interrupt mode.

Remarks

This API is available only for interrupt mode as blocking mode transfer APIs will always return only after completing the transfer.

Preconditions

None.

Example

```
if (SPI1_IsBusy() == false)
{
    //Data Transfer is complete, do something else.
}
```

Function

bool SPIx_IsBusy (void):

e) Data Types and Constants

SPI_CALLBACK Type

Pointer to a SPI Callback function.

File

plib_spi.h

C

```
typedef void (* SPI_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

SPI Callback Function Pointer

This data type defines the required function signature for the SPI callback function. Application must register a pointer to a callback function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive callback from the PLIB.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The context parameter contains the a handle to the client context, provided at the time the callback function was registered using the SPIx_CallbackRegister function. This context handle value is passed back to the client as the "context" parameter. It can be any value (such as a pointer to the client's data) necessary to identify the client context or instance of the client that made the data transfer request.

The callback function executes in the PLIB's interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
SPI1_CallbackRegister(&APP_SPITransferHandler, (uintptr_t)NULL);
void APP_SPITransferHandler(uintptr_t context)
{
    //Transfer was completed without error, do something else now.
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the callback function

SPI_CLOCK_PHASE Enumeration

Identifies SPI Clock Phase Options

File

plib_spi.h

C

```
typedef enum {
  SPI_CLOCK_PHASE_TRAILING_EDGE,
  SPI_CLOCK_PHASE_LEADING_EDGE
} SPI_CLOCK_PHASE;
```

Members

Members	Description
SPI_CLOCK_PHASE_TRAILING_EDGE	Input data is valid on clock trailing edge and output data is ready on leading edge
SPI_CLOCK_PHASE_LEADING_EDGE	Input data is valid on clock leading edge and output data is ready on trailing edge

Description

SPI Clock Phase

This enumeration identifies possible SPI Clock Phase Options.

Remarks

None.

SPI_CLOCK_POLARITY Enumeration

Identifies SPI Clock Polarity Options

File

plib_spi.h

C

```
typedef enum {
  SPI_CLOCK_POLARITY_IDLE_LOW,
  SPI_CLOCK_POLARITY_IDLE_HIGH
} SPI_CLOCK_POLARITY;
```

Members

Members	Description
SPI_CLOCK_POLARITY_IDLE_LOW	The inactive state value of clock is logic level zero
SPI_CLOCK_POLARITY_IDLE_HIGH	The inactive state value of clock is logic level one

Description

SPI Clock Polarity

This enumeration identifies possible SPI Clock Polarity Options.

Remarks

None.

SPI_DATA_BITS Enumeration

Identifies SPI bits per transfer

File

plib_spi.h

C

```
typedef enum {
  SPI_DATA_BITS_8,
  SPI_DATA_BITS_9,
  SPI_DATA_BITS_10,
  SPI_DATA_BITS_11,
  SPI_DATA_BITS_12,
  SPI_DATA_BITS_13,
  SPI_DATA_BITS_14,
  SPI_DATA_BITS_15,
  SPI_DATA_BITS_16
} SPI_DATA_BITS;
```

Members

Members	Description
SPI_DATA_BITS_8	8 bits per transfer
SPI_DATA_BITS_9	9 bits per transfer
SPI_DATA_BITS_10	10 bits per transfer
SPI_DATA_BITS_11	11 bits per transfer
SPI_DATA_BITS_12	12 bits per transfer
SPI_DATA_BITS_13	13 bits per transfer
SPI_DATA_BITS_14	14 bits per transfer
SPI_DATA_BITS_15	15 bits per transfer
SPI_DATA_BITS_16	16 bits per transfer

Description

SPI Data Bits

This enumeration identifies number of bits per SPI transfer.

Remarks

For 9 to 15bit modes, data should be right aligned in the 16 bit memory location.

SPI_TRANSFER_SETUP Structure

Identifies the setup parameters which can be changed dynamically.

File

plib_spi.h

C

```
typedef struct {
  uint32_t clockFrequency;
  SPI_CLOCK_PHASE clockPhase;
  SPI_CLOCK_POLARITY clockPolarity;
  SPI_DATA_BITS dataBits;
} SPI_TRANSFER_SETUP;
```

Members

Members	Description
uint32_t clockFrequency;	Baud Rate or clock frequency
SPI_CLOCK_PHASE clockPhase;	Clock Phase
SPI_CLOCK_POLARITY clockPolarity;	Clock Polarity
SPI_DATA_BITS dataBits;	Number of bits per transfer

Description

SPI Setup Parameters

This structure identifies the possible setup parameters for SPI which can be changed dynamically if needed.

Remarks

None.

Files

Files

Name	Description
plib_spi.h	SPI PLIB Header File for documentation

Description

This section will list only the library's interface header file(s).

plib_spi.h

SPI PLIB Header File for documentation

Enumerations

	Name	Description
	SPI_CLOCK_PHASE	Identifies SPI Clock Phase Options
	SPI_CLOCK_POLARITY	Identifies SPI Clock Polarity Options
	SPI_DATA_BITS	Identifies SPI bits per transfer

Functions

	Name	Description
	SPIx_CallbackRegister	Allows application to register callback with PLIB.
	SPIx_Initialize	Initializes SPI x module of the device
	SPIx_IsBusy	Returns transfer status of SPI x
	SPIx_Read	Read data on SPI x peripheral
	SPIx_TransferSetup	Setup SPI operational parameters as desired by the client.
	SPIx_Write	Write data on SPI x peripheral
	SPIx_WriteRead	Write and Read data on SPI x peripheral

Structures

	Name	Description
	SPI_TRANSFER_SETUP	Identifies the setup parameters which can be changed dynamically.

Types

	Name	Description
	SPI_CALLBACK	Pointer to a SPI Callback function.

Description

SPI PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Peripheral Interface (SPI) controller. This file must not be included in any MPLAB Project.

File Name

plib_spi.h

Company

Microchip Technology Inc.

SSC Peripheral Library Help

This section provides interface to use Serial Synchronous Controller(SSC) peripheral.

Description

Introduction

This library provides an interface to use the SSC (Serial Synchronous Controller) module.

Description

The SSC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module. The SSC hardware also provides for other protocols, which can be configured manually.

The SSC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- RD: Serial Data Input for receiving sample digital audio data (ADCDAT)
- TD: Serial Data Output for transmitting digital audio data (DACDAT)
- TK/RK: Transmit/Receive Serial Clock, also known as bit clock (BCLK)
- TF/RF: Transmit/Receive Frame Clock, also known as Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

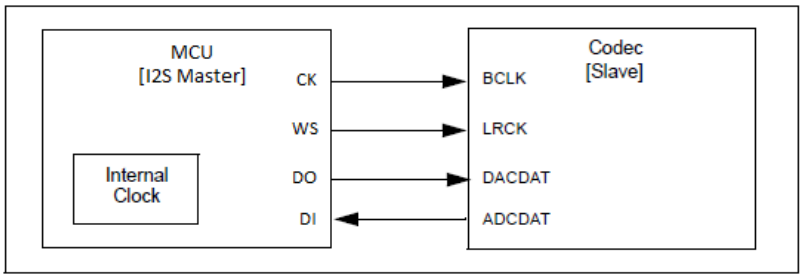
In Master mode, the module generates both the BCLK on the TK pin and the LRCK on the TF pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of BCLK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out TD, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

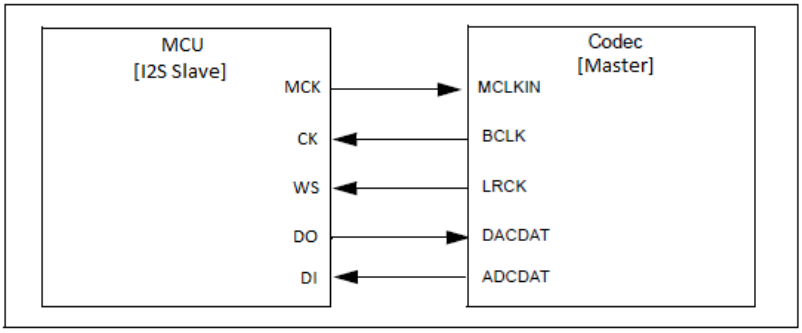
Master Mode

Master Generating its Own Clock – Output BCLK and LRCK



Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out



Audio Formats

The SSC Module supports two audio formats, I2S and Left Justified, which are selectable via MHC.

Configuring the Library

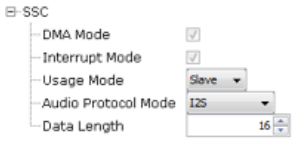
This section describes how to configure the peripheral library using MHC

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the SSC peripheral by clicking on the appropriate instance under *Peripherals->SSC* in the Available Components section of MHC. (The SSC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the SSC peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options):



Default values are shown. As mentioned earlier, **DMA** and **Interrupt Mode** are always enabled.

Usage Mode can be either Master -- the SSC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

Audio Protocol Mode can be either I2S, Left Justified, or Custom. In the latter case, a number of other options are then available, which allow the SSC peripheral to be used for a number of protocols (not currently supported by existing drivers).

Data Length is set to the number of data bits per channel.

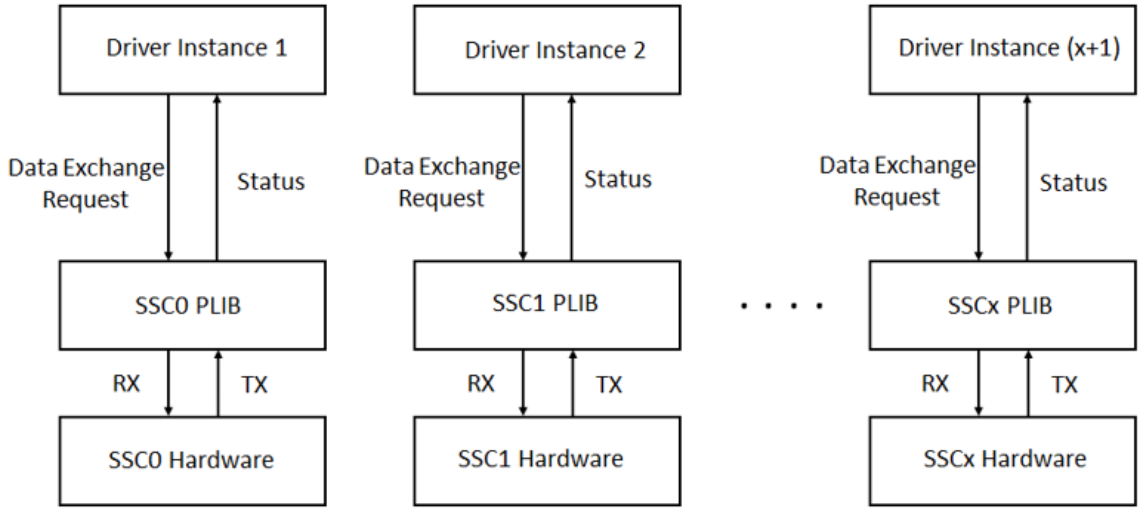
Using the Library

This topic describes the basic architecture of the SSC Peripheral Library and provides information and examples on how to use it.

Description

Abstraction Model

The SSC module sits between the I2S Driver, and the actual hardware.



Note: Some devices may have only one instance of an SSC interface (SSC0 only).

Interface Header File: plib_ssc.h

The interface to the SSC Peripheral Library is defined in the plib_ssc.h header file. Any C language source (.c) file that uses the SSC Peripheral Library should include plib_ssc.h.

Library Source Files:

The SSC Peripheral Library library source files are provided in the `csp/peripheral/ssc_xxxx/src` or `csp/peripheral/ssc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

Usage Model

The only usage model for the SSC Peripheral Library is to use a Interrupt/DMA model. Therefore many functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`

Library Interface

a) Initialization Function

	Name	Description
	SSC_Initialize	Initializes SSC module of the device

b) Transaction Functions

	Name	Description
	SSC_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal
	SSCx_BaudSet	Changes the baud rate (samples/second) of the interface.

Description

a) Initialization Function

SSC_Initialize Function

Initializes SSC module of the device

File

plib_ssc.h

C

```
void SSC_Initialize();
```

Returns

None.

Description

This function initializes SSC module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Remarks

This function must be called only once and before any other SSC function is called.

Preconditions

MHC GUI should be configured with the right values.

Example

```
SSC_Initialize();
```

Function

void SSC_Initialize (void);

b) Transaction Functions

SSC_LRCLK_Get Function

Get the level of the I2S LRCLK (left/right clock) signal

File

plib_ssc.h

C

```
uint32_t SSC_LRCLK_Get();
```

Returns

State of the LRCLK pin for the SSC module -- 1 if high, 0 if low if the audio format is I2S, and 0 if high, 1 if low if the format is Left Jusutifed

Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or

other external source, this allows the caller to synchronize itself to the LRCLK signal.

Remarks

None.

Preconditions

None.

Example

```
SSC_LRCLK_Get();
```

Function

uint32_t SSC_LRCLK_Get(void);

SSCx_BaudSet Function

Changes the baud rate (samples/second) of the interface.

File

plib_ssc.h

C

```
void SSCx_BaudSet(const uint32_t baud);
```

Returns

None.

Description

This function Cchanges the baud rate, or samples/second of the interface.

Remarks

None.

Preconditions

None.

Example

```
SSC_BaudSet(44100);
```

Function

void SSC_BaudSet (const uint32_t baud);

c) Data Types and Constants

Files

Files

Name	Description
plib_ssc.h	SSC PLIB Header File for documentation

Description

This section will list only the library's interface header file(s).

plib_ssc.h

SSC PLIB Header File for documentation

Functions

	Name	Description
	SSC_Initialize	Initializes SSC module of the device
	SSC_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal
	SSCx_BaudSet	Changes the baud rate (samples/second) of the interface.

Description

SSC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Synchronous Controller (SSC). This file must not be included in any MPLAB Project.

File Name

plib_ssc.h

Company

Microchip Technology Inc.

SUPC Peripheral Library Help

This section provides an interface to use Supply Controller (SUPC) peripheral

Introduction

This section provides brief overview of the SUPC peripheral.

Description

The Supply Controller controls the Embedded Voltage Regulators used by the Core Power Supply and Backup Low Power Mode.

In Backup Low Power Mode the current consumption is reduced to a few microamps for Backup power retention. Exit from this mode can be achieved via multiple wake-up sources.

The Supply Controller also generates the Slow Clock (SCLK) by selecting either the 22-42kHz Low Power RC oscillator or the 32kHz Low Power Crystal oscillator.

The purpose of Backup Low Power Mode is to achieve the lowest power consumption possible in a system that wakes up periodically in order to perform tasks that do not require a fast startup time.

The supply controller, zero-power power-on reset, RTT, RTC, backup registers, and 32kHz oscillator (RC or crystal oscillator selected by software) are all running whilst the regulator and core supply are off.

Following wakeup sources can wakeup the device from backup mode.

- 14 Wakeup Inputs with Programmable De-bouncing
- Real-Time Clock Alarm
- Real-Time Timer Alarm
- Supply Monitor Detection on VDDIO, with Programmable Scan Period and Voltage Threshold

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

SUPC Peripheral library provides API's and they can be used to perform below functionalities.

- To put the device either in Sleep, Backup or Wait mode.
- To read and write the General Purpose Backup Register (GPBR).
- Generate supply monitor callback to save critical data to prepare the system for a power down. Supply monitor can generate interrupt or Reset the device when supply voltage crosses below the specified threshold level.

The examples shows the code to enter sleep, backup and wait mode.

```
while(1)
  {
      switch(cmd)
      {
          case SLEEP_MODE:
          {
              SUPC_SleepModeEnter();
              break;
          }
          case WAIT_MODE:
          {
              SUPC_WaitModeEnter(PMC_FSMR_FLPM_FLASH_DEEP_POWERDOWN, WAITMODE_WKUP_WKUP7);
              break;
          }
          case BACKUP_MODE:
          {
              SUPC_BackupModeEnter();
              break;
          }
      }
  }
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	SUPCx_Initialize	Initializes SUPC x module of the device

b) Setup Functions

	Name	Description
	SUPCx_CallbackRegister	Allows application to register callback with PLIB.

c) Transaction Functions

	Name	Description
	SUPCx_BackupModeEnter	Puts the device into Backup mode
	SUPCx_GPBRRead	Reads the data from General Purpose Backup Register(GPBR)
	SUPCx_GPBRWrite	Writes data to the General Purpose Backup Register(GPBR)
	SUPCx_SleepModeEnter	Puts the device into Sleep mode
	SUPCx_WaitModeEnter	Puts the device into Wait mode

d) Data Types and Constants

	Name	Description
	SUPC_CALLBACK	Pointer to a SUPC callback function
	WAITMODE_FLASH_STATE	Identifies Flash Low-power mode
	WAITMODE_WKUP_SOURCE	Identifies the wakeup input source

a) Initialization Function

SUPCx_Initialize Function

Initializes SUPC x module of the device

File

plib_supc.h

C

```
void SUPCx_Initialize();
```

Returns

None.

Description

This function initializes SUPC x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, APIs can be used to put the device into Low-power mode.

Remarks

This function must be called only once and before any other SUPC function is called.

Preconditions

MHC GUI should be configured with the right values.

Example

```
SUPC0_Initialize ();
```

Function

void SUPCx_Initialize (void);

b) Setup Functions

SUPCx_CallbackRegister Function

Allows application to register callback with PLIB.

File

plib_supc.h

C

```
void SUPCx_CallbackRegister(SUPC_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function allows application to register a callback function for the PLIB to callback when a interrupt occurred due to Supply monitor detection.

The callback should be registered before the client performs transfer operation.

At any point if application wants to stop the callback, it can call this function with "callback" value as NULL.

Remarks

None.

Preconditions

The SUPCx_Initialize function must have been called.

Example

```
MY_APP_OBJ myAppObj;
void APP_SUPC_CallbackFunction (uintptr_t context)
{
    // The context was set to an application specific object.
    // It is now retrievable easily in the event handler.
       MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;
    //Application related tasks
}

SUPC0_CallbackRegister(APP_SUPC_CallbackFunction, (uintptr_t)&myAppObj);
```

Parameters

Parameters	Description
callback	Pointer to the callback function implemented by the user
context	The value of parameter will be passed back to the application unchanged, when the callback function is called. It can be used to identify any application specific value that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void SUPCx_CallbackRegister (SUPC_CALLBACK callback, uintptr_t context);

c) Transaction Functions

SUPCx_BackupModeEnter Function

Puts the device into Backup mode

File

plib_supc.h

C

```
void SUPCx_BackupModeEnter();
```

Returns

None.

Description

This functions is used to put the device into Backup mode to achieve the lowest power consumption possible in a system which is performing periodic wakeup to perform tasks but not requiring fast startup time.

The Supply Controller, zero-power power-on reset, RTT, RTC, backup SRAM, backup registers and 32 kHz oscillator (RC or crystal oscillator selected by software in the Supply Controller) are running. The regulator and the core supply are off.

Exit from Backup mode occurs as a result of one of the following enabled wakeup events and it causes device reset:

- WKUP0 to WKUP13 pins (level transition, configurable debouncing)
- Supply Monitor alarm
- RTC alarm
- RTT alarm

Remarks

None.

Preconditions

The peripherals (Supply Monitor, RTT, RTC etc) must be configured to generate wakeup event.

Example

```
SUPC0_BackupModeEnter ();
```

Function

void SUPCx_BackupModeEnter (void);

SUPCx_GPBRRead Function

Reads the data from General Purpose Backup Register(GPBR)

File

plib_supc.h

C

```
uint32_t SUPCx_GPBRRead(GPBR_REGS_INDEX reg);
```

Returns

32 bit data available in specified GPBR register.

Description

This functions is used to read the data available in GPBR register when the device wakeup from Low power modes.

Remarks

None.

Preconditions

None

Example

```
uint32_t data;
```

```
data = SUPC0_GPBRRead (GPBR_REGS_2);
```

Parameters

Parameters	Description
reg	Identifies the GPBR register

Function

uint32_t SUPCx_GPBRRead (GPBR_REGS_INDEX reg);

SUPCx_GPBRWrite Function

Writes data to the General Purpose Backup Register(GPBR)

File

plib_supc.h

C

```
void SUPCx_GPBRWrite(GPBR_REGS_INDEX reg, uint32_t data);
```

Returns

None.

Description

This functions is used to write the data to GPBR register.

Remarks

None.

Preconditions

None

Example

```
SUPC0_GPBRWrite (GPBR_REGS_2, 0x12345678);
```

Parameters

Parameters	Description
reg	Identifies the GPBR register
data	32bit data to be stored into GPBR register

Function

void SUPCx_GPBRWrite (GPBR_REGS_INDEX reg, uint32_t data);

SUPCx_SleepModeEnter Function

Puts the device into Sleep mode

File

plib_supc.h

C

```
void SUPCx_SleepModeEnter();
```

Returns

None.

Description

This functions is used to put the device into sleep mode to optimize power consumption.

In this mode, only the core clock is stopped and peripheral clock remains enabled.

Processor wakeup is triggered by an interrupt.

Remarks

None.

Preconditions

None

Example

```
SUPC0_SleepModeEnter ();
```

Function

void SUPCx_SleepModeEnter (void);

SUPCx_WaitModeEnter Function

Puts the device into Wait mode

File

plib_supc.h

C

```
void SUPCx_WaitModeEnter(WAITMODE_FLASH_STATE flash_lpm, WAITMODE_WKUP_SOURCE source);
```

Returns

None.

Description

This functions is used to put the device into Wait mode to achieve very low power consumption while maintaining the whole device in a powered state for a startup time of less than 10Î¼s.

In Wait mode, the clocks of the core, peripherals and memories are stopped. However, the core, peripherals and memories power supplies are still powered.

When entering Wait mode, the embedded Flash can be placed in one of the low_power modes (Deep_power_down or Standby mode).

Exit from Wait mode occurs as a result of one of the following enabled wakeup events:

- WKUP0 to WKUP13 pins
- RTC alarm
- RTT alarm
- USB alarm
- GMAC wake on LAN event

A fast startup is enabled upon the detection of a programmed level on one of the 14 wake-up inputs (WKUP) or upon an active alarm from the RTC, RTT and USB Controller. The polarity of the 14 wakeup inputs is programmable by writing the PMC Fast Startup Polarity register (PMC_FSPR).

Remarks

None.

Preconditions

The peripherals (RTT, RTC etc) must be configured to generate wakeup event.

Example

```
SUPC0_WaitModeEnter (PMC_FSMR_FLPM_FLASH_DEEP_POWERDOWN,
```

```
                                WAITMODE_WKUP_RTC);
```

Parameters

Parameters	Description
WAITMODE_FLASH_STATE	Identifies Flash Low_power mode
WAITMODE_WKUP_SOURCE	Identifies possible wakeup input source

Function

```
void SUPCx_WaitModeEnter (   WAITMODE_FLASH_STATE flash_lpm,
                             WAITMODE_WKUP_SOURCE source);
```

d) Data Types and Constants

SUPC_CALLBACK Type

Pointer to a SUPC callback function

File

plib_supc.h

C

```
typedef void (* SUPC_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

SUPC Callback Function pointer typedef void (*SUPC_CALLBACK) (uintptr_t context);

This data type defines the required function signature for the SUPC callback function. Application must register a pointer to a callback function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive callback from the PLIB.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The context parameter contains a handle to the client context, provided at the time the callback function was registered using the SUPCx_CallbackRegister function. This context handle value is passed back to the client as the "context" parameter. It can be any value (such as a pointer to the client's data) necessary to identify the client context or instance of the client that made the request.

The callback function executes in the PLIB's interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_SUPC_CallbackFunction (uintptr_t context)
{
    //Application related tasks
}

SUPC0_CallbackRegister (APP_SUPC_CallbackFunction, NULL);
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the callback function.

WAITMODE_FLASH_STATE Enumeration

Identifies Flash Low-power mode

File

plib_supc.h

C

```
typedef enum {
  WAITMODE_FLASH_STANDBY,
  WAITMODE_FLASH_DEEPSLEEP
} WAITMODE_FLASH_STATE;
```

Members

Members	Description
WAITMODE_FLASH_STANDBY	Flash is in Standby mode
WAITMODE_FLASH_DEEPSLEEP	Flash is in Deep-powerdown mode

Description

Flash Low-power mode states

This enumeration identifies Flash Low-power mode state when system enters Wait mode.

Remarks

None.

WAITMODE_WKUP_SOURCE Enumeration

Identifies the wakeup input source

File

plib_supc.h

C

```
typedef enum {
  WAITMODE_WKUP_WKUP0,
  WAITMODE_WKUP_WKUP1,
  WAITMODE_WKUP_WKUP2,
  WAITMODE_WKUP_WKUP3,
  WAITMODE_WKUP_WKUP4,
  WAITMODE_WKUP_WKUP5,
  WAITMODE_WKUP_WKUP6,
  WAITMODE_WKUP_WKUP7,
  WAITMODE_WKUP_WKUP8,
  WAITMODE_WKUP_WKUP9,
  WAITMODE_WKUP_WKUP10,
  WAITMODE_WKUP_WKUP11,
  WAITMODE_WKUP_WKUP12,
  WAITMODE_WKUP_WKUP13,
  WAITMODE_WKUP_GMAC,
  WAITMODE_WKUP_DEBUG,
  WAITMODE_WKUP_RTT,
  WAITMODE_WKUP_RTC,
  WAITMODE_WKUP_USB
} WAITMODE_WKUP_SOURCE;
```

Members

Members	Description
WAITMODE_WKUP_WKUP0	Wakeup 0 pin configured for Fast Startup Input 0
WAITMODE_WKUP_WKUP1	Wakeup 1 pin configured for Fast Startup Input 1
WAITMODE_WKUP_WKUP2	Wakeup 2 pin configured for Fast Startup Input 2
WAITMODE_WKUP_WKUP3	Wakeup 3 pin configured for Fast Startup Input 3
WAITMODE_WKUP_WKUP4	Wakeup 4 pin configured for Fast Startup Input 4
WAITMODE_WKUP_WKUP5	Wakeup 5 pin configured for Fast Startup Input 5
WAITMODE_WKUP_WKUP6	Wakeup 6 pin configured for Fast Startup Input 6
WAITMODE_WKUP_WKUP7	Wakeup 7 pin configured for Fast Startup Input 7
WAITMODE_WKUP_WKUP8	Wakeup 8 pin configured for Fast Startup Input 8
WAITMODE_WKUP_WKUP9	Wakeup 9 pin configured for Fast Startup Input 9
WAITMODE_WKUP_WKUP10	Wakeup 10 pin configured for Fast Startup Input 10
WAITMODE_WKUP_WKUP11	Wakeup 11 pin configured for Fast Startup Input 11
WAITMODE_WKUP_WKUP12	Wakeup 12 pin configured for Fast Startup Input 12
WAITMODE_WKUP_WKUP13	Wakeup 13 pin configured for Fast Startup Input 13
WAITMODE_WKUP_GMAC	Wakeup source configured for GMAC
WAITMODE_WKUP_DEBUG	Wakeup source configured for debug request
WAITMODE_WKUP_RTT	Wakeup source configured for RTT alarm
WAITMODE_WKUP_RTC	Wakeup source configured for RTC alarm
WAITMODE_WKUP_USB	Wakeup source configured for USB alarm

Description

Wakeup Input Source

This enumeration identifies possible wakeup input source for exit from Wait mode.

Remarks

None.

Files

Files

Name	Description
plib_supc.h	SUPC PLIB Header File for documentation

Description

This section will list only the library's interface header file(s).

plib_supc.h

SUPC PLIB Header File for documentation

Enumerations

	Name	Description
	WAITMODE_FLASH_STATE	Identifies Flash Low-power mode
	WAITMODE_WKUP_SOURCE	Identifies the wakeup input source

Functions

	Name	Description
	SUPCx_BackupModeEnter	Puts the device into Backup mode
	SUPCx_CallbackRegister	Allows application to register callback with PLIB.
	SUPCx_GPBRRead	Reads the data from General Purpose Backup Register(GPBR)
	SUPCx_GPBRWrite	Writes data to the General Purpose Backup Register(GPBR)
	SUPCx_Initialize	Initializes SUPC x module of the device
	SUPCx_SleepModeEnter	Puts the device into Sleep mode
	SUPCx_WaitModeEnter	Puts the device into Wait mode

Types

	Name	Description
	SUPC_CALLBACK	Pointer to a SUPC callback function

Description

Supply Controller (SUPC) Peripheral Library (PLIB) Interface Header File

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Supply Controller (SUPC) controller. This file must not be included in any MPLAB Project.

File Name

plib_supc.h

Company

Microchip Technology Inc.

SYSTICK Peripheral Library Help

This section provides interface to use System Timer (SysTick) peripheral.

Introduction

This section provides brief overview of System Timer (SysTick)

Description

ARM® Cortex® CPU includes an optional system timer (SysTick) that provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism.

When enabled, the 24-bit timer counter counts down from the value in VAL. When the counter reaches zero, it reloads the value in LOAD on the next clock edge. It then decrements on subsequent clocks. This reloading when the counter reaches zero is called wrapping. The timer interrupt is activated on the transition from 1 to 0, therefore it activates every n+1 clock ticks. . If a period of 100 is required 99 should be written to the SysTick Reload Value register.

The SysTick can be polled by software or can be configured to generate an interrupt.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

When the timer is enabled, it decrements by one on every rising edge of the input clock, and generates an interrupt when the timer value transitions to zero (Timer expiry). This is used to generate periodic interrupt generation.

The SysTick can be polled by software or can be configured to generate an interrupt.

- With polling, the application will need to continuously poll to check if the timer has expired.
- With interrupt, the registered callback function will be called when the timer expires. This means we do not have to poll continuously.

Callback method

This example demonstrates how to use SysTick to generate periodic callback.

```
/* This function is called after Timer expires */
void SYSTICK_EventHandler(uintptr_t context)
{
    /* Toggle LED */
    LED_Toggle();
}

int main(void)
{
    /* Initialize all modules */
    SYS_Initialize ( NULL );
    /* Register Callback */
    SYSTICK_TimerCallbackSet(&SYSTICK_EventHandler, (uintptr_t) NULL);

    /* Start the Timer */
    SYSTICK_TimerStart();
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	SYSTICK_TimerInitialize	Initializes SysTick.

c) Transaction Functions

	Name	Description
	SYSTICK_TimerCounterGet	Get the SysTick current Value
	SYSTICK_TimerFrequencyGet	Get the SysTick frequency Value
	SYSTICK_TimerPeriodGet	Get the SysTick Load Value
	SYSTICK_TimerStart	Starts System Timer.
	SYSTICK_TimerStop	Stops System Timer.
	SYSTICK_TimerPeriodSet	Set the SysTick Load Value
	SYSTICK_TimerRestart	Restarts SysTick.
	SYSTICK_DelayMs	Blocking function to generate delay in milliseconds

d) Status Functions

	Name	Description
	SYSTICK_TimerPeriodHasExpired	Returns the current status of the systick

e) Data Types and Constants

	Name	Description
	SYSTICK_CALLBACK	Defines the data type and function signature for the SysTick callback function.
	SYSTICK_OBJECT	Defines the data type for the SYSTICK OBJECT.

a) Initialization Function

SYSTICK_TimerInitialize Function

Initializes SysTick.

File

plib_systick.h

C

```
void SYSTICK_TimerInitialize();
```

Returns

None.

Description

This function initializes the System Timer as configured by the user from within the MHC.

Preconditions

None.

Example

```
SYSTICK_TimerInitialize();
```

Function

void SYSTICK_TimerInitialize(void)

b) Setup Functions

Functions

	Name	Description
	SYSTICK_TimerCallbackSet	Sets the pointer to the function (and it's context) to be called when the given systick reaches 0.

Description

SYSTICK_TimerCallbackSet Function

Sets the pointer to the function (and it's context) to be called when the given systick reaches 0.

File

plib_systick.h

C

```
void SYSTICK_TimerCallbackSet(SYSTICK_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when systick reaches 0. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

This function is available only in interrupt or non-blocking mode of operation.

Remarks

The context parameter is ignored if the pointer passed is NULL.

To disable the callback function, pass a NULL for the callback parameter.

Preconditions

SYSTICK_Initialize must have been called for the associated USART instance.

Example

```
SYSTICK_TimerCallbackSet(MyCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the SYSTICK_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void SYSTICK_TimerCallbackSet (SYSTICK_CALLBACK callback, uintptr_t context)

c) Transaction Functions

SYSTICK_TimerCounterGet Function

Get the SysTick current Value

File

plib_systick.h

C

```
uint32_t SYSTICK_TimerCounterGet();
```

Returns

None.

Description

This function returns the SysTick current value

Preconditions

32-bit current value

Example

```
uint32_t value;

value = SYSTICK_TimerCounterGet();
```

Function

uint32_t SYSTICK_TimerCounterGet(void)

SYSTICK_TimerFrequencyGet Function

Get the SysTick frequency Value

File

plib_systick.h

C

```
uint32_t SYSTICK_TimerFrequencyGet();
```

Returns

Systick Frequency value

Description

This function returns the SysTick frequency value

Preconditions

None.

Example

```
uint32_t frequency;

frequency = SYSTICK_TimerFrequencyGet();
```

Function

uint32_t SYSTICK_TimerFrequencyGet(void)

SYSTICK_TimerPeriodGet Function

Get the SysTick Load Value

File

plib_systick.h

C

```
uint32_t SYSTICK_TimerPeriodGet();
```

Returns

32-bit period value.

Description

This function returns the SysTick Load value

Preconditions

None

Example

```
uint32_t period;
```

```
period = SYSTICK_TimerPeriodGet();
```

Function

uint32_t SYSTICK_TimerPeriodGet(void)

SYSTICK_TimerStart Function

Starts System Timer.

File

plib_systick.h

C

```
void SYSTICK_TimerStart();
```

Returns

None.

Description

This function resets and starts the System Timer

Preconditions

SYSTICK_Initialize should have been called to set up SysTick.

Example

```
SYSTICK_Initialize();
SYSTICK_TimerStart();
```

Function

void SYSTICK_TimerStart(void)

SYSTICK_TimerStop Function

Stops System Timer.

File

plib_systick.h

C

```
void SYSTICK_TimerStop();
```

Returns

None.

Description

This function stops the System Timer

Preconditions

None.

Example

```
SYSTICK_TimerStops();
```

Function

void SYSTICK_TimerStop(void)

SYSTICK_TimerPeriodSet Function

Set the SysTick Load Value

File

plib_systick.h

C

```
void SYSTICK_TimerPeriodSet(uint32_t period);
```

Returns

None.

Description

This function is used to update SysTick Load value

Preconditions

Systick should be stopped prior to setting up the new Load value by calling SYSTICK_TimerStop. This will make sure that the new value is used for the next Tick

Example

```
SYSTICK_TimerStop();
SYSTICK_TimerPeriodSet(0x00004567);
SYSTICK_TimerStart();
```

Parameters

Parameters	Description
Period	Timer period to set

Function

void SYSTICK_TimerPeriodSet(uint32_t period)

SYSTICK_TimerRestart Function

Restarts SysTick.

File

plib_systick.h

C

```
void SYSTICK_TimerRestart();
```

Returns

None.

Description

This function Restarts systick timer with preloaded configuration.

Preconditions

SYSTICK_Initialize should have been called to set up SysTick..

Example

```
SYSTICK_Initialize();
SYSTICK_TimerRestart();
```

Function

void SYSTICK_TimerRestart(void)

SYSTICK_DelayMs Function

Blocking function to generate delay in milliseconds

File

plib_systick.h

C

```
void SYSTICK_DelayMs(uint32_t ms);
```

Returns

None.

Description

This function generates accurate delay in units of milliseconds using SysTick timer in interrupt mode

Preconditions

SYSTICK_Initialize should have been called to set up SysTick in interrupt mode

Example

```
SYSTICK_Initialize();
SYSTICK_TimerStart();

// Generates 100ms blocking delay
SYSTICK_DelayMs(100)
```

Parameters

Parameters	Description
ms	The delay in units of milliseconds

Function

void SYSTICK_DelayMs(uint32_t ms)

d) Status Functions

SYSTICK_TimerPeriodHasExpired Function

Returns the current status of the systick

File

plib_systick.h

C

```
bool SYSTICK_TimerPeriodHasExpired();
```

Returns

- True - timer period has expired
- False - timer period is not expired

Description

This function is used to identify if the Systick underflow has happened. This API can be used in polling mode

Preconditions

SYSTICK_Initialize should have been called to set up SysTick.

Example

```
SYSTICK_Initialize();
SYSTICK_TimerStart();

if(SYSTICK_TimerPeriodHasExpired())
{
    //application code
}
```

Function

bool SYSTICK_TimerPeriodHasExpired(void)

e) Data Types and Constants

SYSTICK_CALLBACK Type

Defines the data type and function signature for the SysTick callback function.

File

plib_systick.h

C

```
typedef void (* SYSTICK_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the SysTick callback function. SysTick will call back the client's function with this signature.

Remarks

None.

Preconditions

SYSTICK_Initialize and SYSTICK_TimerCallbackSet must have been called to set the function to be called.

Example

```
ticks = 0;

void MyCallback ( uintptr_t context )
{
    ticks++;
}

SYSTICK_TimerCallbackSet(MyCallback, (uintptr_t) NULL);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

SYSTICK_OBJECT Structure

Defines the data type for the SYSTICK OBJECT.

File

plib_systick.h

C

```
typedef struct {
  SYSTICK_CALLBACK callback;
  uintptr_t context;
} SYSTICK_OBJECT;
```

Members

Members	Description
SYSTICK_CALLBACK callback;	A pointer to a function with a calling signature defined by the SYSTICK_CALLBACK data type.
uintptr_t context;	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Description

SYSTICK_OBJECT

This structure defines SYSTICK object parameters.

Remarks

None.

Files

Files

Name	Description
plib_systick.h	SysTick peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_systick.h

SysTick peripheral library interface.

Functions

	Name	Description
	SYSTICK_DelayMs	Blocking function to generate delay in milliseconds
	SYSTICK_TimerCallbackSet	Sets the pointer to the function (and it's context) to be called when the given systick reaches 0.
	SYSTICK_TimerCounterGet	Get the SysTick current Value

	SYSTICK_TimerFrequencyGet	Get the SysTick frequency Value
	SYSTICK_TimerInitialize	Initializes SysTick.
	SYSTICK_TimerPeriodGet	Get the SysTick Load Value
	SYSTICK_TimerPeriodHasExpired	Returns the current status of the systick
	SYSTICK_TimerPeriodSet	Set the SysTick Load Value
	SYSTICK_TimerRestart	Restarts SysTick.
	SYSTICK_TimerStart	Starts System Timer.
	SYSTICK_TimerStop	Stops System Timer.

Structures

	Name	Description
	SYSTICK_OBJECT	Defines the data type for the SYSTICK OBJECT.

Types

	Name	Description
	SYSTICK_CALLBACK	Defines the data type and function signature for the SysTick callback function.

Description

SysTick Peripheral Library Interface Header File

This file defines the interface to the SysTick peripheral library. This library provides access to and control of the associated peripheral instance.

File Name

plib_systick.h

Company

Microchip Technology Inc.

TC Peripheral Library Help

This section provides an interface to use Timer Counter (TC) peripheral in Timer, Compare, Capture and Quadrature decoder mode.

Introduction

This section provides brief overview of the TC peripheral

Description

A Timer Counter (TC) module includes three identical TC channels. All channels of the Timer Counter are independent and identical in operation except when the QDEC is enabled.

- Timer mode is used for periodic interrupt generation.
- The TC channel in compare mode generates one or two PWM signals with the same frequency and independently programmable duty cycles, or generates different types of one-shot or repetitive pulses on TIOAx and TIOBx output
- The Capture mode allows the TC channel to perform measurements such as pulse timing, frequency, period, duty cycle and phase on TIOAx and TIOBx input.
- The quadrature decoder (QDEC) mode is used to measure the position and speed of the motor

Timer

This section provides interface to use the TC peripheral in Timer mode

Description

Introduction

This section provides brief overview of the TC peripheral in Timer mode

Description

The TC channel in timer mode is used for periodic interrupt generation.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

When the timer is enabled, it increments by one on every rising edge of the input clock, and generates an interrupt on a period match (Timer expiry period). Timer mode is used for periodic interrupt generation.

It provides both polling and callback methods to indicate period match has occurred.

- With polling, the application will need to continuously poll to check if the timer has expired.
- With callback, the registered callback function will be called when the timer expires. This means we do not have to poll continuously.

Callback method

This example demonstrates how to use TC channel in timer mode to generate periodic callback.

```
/* This function is called after period expires */
void TC0_CH0_TimerInterruptHandler(uintptr_t context)
{
    /* Toggle LED */
    LED_Toggle();
}

int main(void)
{

    /* Register callback function for CH0 period interrupt */
    TC0_CH0_TimerCallbackRegister(TC0_CH0_TimerInterruptHandler, (uintptr_t)NULL);

    /* Start the timer channel 0*/
    TC0_CH0_TimerStart();
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the TC Peripheral Library in Timer mode

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	TCx_CHy_TimerInitialize	Initializes given instance of TC channel

b) Setup Functions

	Name	Description
	TCx_CHy_TimerCallbackRegister	Registers the function to be called from interrupt

c) Transaction Functions

	Name	Description
	TCx_CHy_TimerCounterGet	Reads the timer channel counter value
	TCx_CHy_TimerFrequencyGet	Provides the given timer's counter-increment frequency.
	TCx_CHy_TimerPeriodGet	Reads the period value of given timer channel
	TCx_CHy_TimerPeriodSet	Sets the period value of a given timer channel.
	TCx_CHy_TimerStart	Starts the given TC channel counter
	TCx_CHy_TimerStop	Stops the given TC channel counter
	TCx_CHy_TimerCompareSet	Sets the compare value of a given timer channel.

d) Status Functions

	Name	Description
	TCx_CHy_TimerPeriodHasExpired	Checks whether timer period is elapsed

e) Data Types and Constants

	Name	Description
	TC_TIMER_STATUS	Identifies channel interrupt source mask
	TC_TIMER_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_TIMER_CALLBACK_OBJECT	Callback structure

a) Initialization Function

TCx_CHy_TimerInitialize Function

Initializes given instance of TC channel

File

plib_tc.h

C

```
void TCx_CHy_TimerInitialize();
```

Returns

None.

Description

This function initializes given instance of TC channel in timer mode with the options configured in MCC GUI.

Remarks

This function must be called before any other TC function is called. This function is normally only be called once during system initialization.

Preconditions

MCC GUI should be configured with the desired values.

Example

```
TC1_CH0_TimerInitialize();
```

Function

void TCx_CHy_TimerInitialize (void);

b) Setup Functions

TCx_CHy_TimerCallbackRegister Function

Registers the function to be called from interrupt

File

plib_tc.h

C

```
void TCx_CHy_TimerCallbackRegister(TC_TIMER_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from interrupt.

Remarks

Context value can be set to NULL if not required. To disable callback function, pass NULL for the callback parameter.

Preconditions

TCx_CHy_TimerInitialize() must have been called first.

Example

```
void TC_CallbackFn (TC_TIMER_STATUS status, uintptr_t context );

TC0_CH1_TimerInitialize();
TC0_CH1_TimerCallbackRegistert(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
callback	callback function pointer
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void TCx_CHy_TimerCallbackRegister (TC_TIMER_CALLBACK callback, uintptr_t context);

c) Transaction Functions

TCx_CHy_TimerCounterGet Function

Reads the timer channel counter value

File

plib_tc.h

C

```
uint16_t TCx_CHy_TimerCounterGet();
```

Returns

The timer's counter value

Description

This function reads the timer channel counter value.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel.

Example

```
uint16_t counter;

TC0_CH2_TimerInitialize();
TC0_CH2_TimerStart();
counter = TC0_CH2_TimerCounterGet();
```

Function

uint16_t TCx_CHy_TimerCounterGet (void);

TCx_CHy_TimerFrequencyGet Function

Provides the given timer's counter-increment frequency.

File

plib_tc.h

C

```
uint32_t TCx_CHy_TimerFrequencyGet();
```

Returns

The frequency (in Hz) at which the timer's counter increments.

Description

This function provides the frequency at which the given counter increments. It can be used to convert differences between counter values to real time or real-time intervals to timer period values.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel.

Example

```
uint32_t frequency;

TC0_CH1_TimerInitialize();
frequency = TC0_CH1_TimerFrequencyGet();
```

Function

uint32_t TCx_CHy_TimerFrequencyGet (void);

TCx_CHy_TimerPeriodGet Function

Reads the period value of given timer channel

File

plib_tc.h

C

```
uint16_t TCx_CHy_TimerPeriodGet();
```

Returns

The timer's period value.

Description

This function reads the value of period of given timer channel.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel.

Example

```
uint16_t period;

TC0_CH1_TimerInitialize();
period = TC0_CH1_TimerPeriodGet();
```

Function

uint16_t TCx_CHy_TimerPeriodGet (void);

TCx_CHy_TimerPeriodSet Function

Sets the period value of a given timer channel.

File

plib_tc.h

C

```
void TCx_CHy_TimerPeriodSet(uint16_t period);
```

Returns

None.

Description

This function writes the period value. When timer counter matches period value, counter is reset and interrupt can be generated.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel.

Example

```
TC0_CH1_TimerInitialize();
TC0_CH1_TimerPeriodSet(0x500ul);
```

Parameters

Parameters	Description
Period	Period value of timer which decides the timer delay

Function

void TCx_CHy_TimerPeriodSet (uint16_t period);

TCx_CHy_TimerStart Function

Starts the given TC channel counter

File

plib_tc.h

C

```
void TCx_CHy_TimerStart();
```

Returns

None.

Description

This function enables the clock and starts the counter of associated channel.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel.

Example

```
TC0_CH1_TimerInitialize();
TC0_CH1_TimerStart();
```

Function

void TCx_CHy_TimerStart (void);

TCx_CHy_TimerStop Function

Stops the given TC channel counter

File

plib_tc.h

C

```
void TCx_CHy_TimerStop();
```

Returns

None.

Description

This function stops the clock and thus counter of associated channel.

Remarks

None

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel.

Example

```
TC1_CH2_TimerInitialize();
TC1_CH2_TimerStart();
TC1_CH2_TimerStop();
```

Function

void TCx_CHy_TimerStop (void);

TCx_CHy_TimerCompareSet Function

Sets the compare value of a given timer channel.

File

plib_tc.h

C

```
void TCx_CHy_TimerCompareSet(uint16_t compare);
```

Returns

None.

Description

This function writes the compare value. When timer counter matches compare value, interrupt can be generated.

Remarks

This function is available only when TC timer mode is used by SYS_TIME module. SYS_TIME uses compare match interrupt to generate dynamic delay.

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel.

Example

```
TC0_CH1_TimerInitialize();
TC0_CH1_TimerCompareSet(0x500ul);
```

Parameters

Parameters	Description
compare	compare value of the timer

Function

void TCx_CHy_TimerCompareSet (uint16_t compare);

d) Status Functions

TCx_CHy_TimerPeriodHasExpired Function

Checks whether timer period is elapsed

File

plib_tc.h

C

```
bool TCx_CHy_TimerPeriodHasExpired();
```

Returns

- True - timer period has expired
- False - timer period is not expired

Description

This function checks the status of the timer period interrupt.

Remarks

This function is available only in polling mode

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel.

Example

```
bool status;

TC0_CH1_TimerInitialize();
status = TC0_CH1_TimerPeriodHasExpired();
```

Function

bool TCx_CHy_TimerPeriodHasExpired (void);

e) Data Types and Constants

TC_TIMER_STATUS Enumeration

Identifies channel interrupt source mask

File

plib_tc.h

C

```
typedef enum {
  TC_TIMER_NONE,
  TC_TIMER_COMPARE_MATCH,
  TC_TIMER_PERIOD_MATCH,
  TC_TIMER_STATUS_MSK
} TC_TIMER_STATUS;
```

Description

Interrupt source mask for Timer mode

This enumeration identifies TC timer mode interrupt source mask.

Remarks

None.

TC_TIMER_CALLBACK Type

Defines the function pointer data type and function signature for the tc channel callback function.

File

plib_tc.h

C

```
typedef void (* TC_TIMER_CALLBACK)(TC_TIMER_STATUS status, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer for Timer mode

This data type defines the function pointer and function signature for the TC channel callback function. The library will call back the client's function with this signature from the interrupt routine.

Remarks

None.

Preconditions

TCx_CHy_Initialize must have been called for the given TC channel instance and TCx_CHy_TimerCallbackRegister must have been called to register the function to be called.

Example

```
void TC_CallbackFn (TC_TIMER_STATUS status, uintptr_t context );

TC0_CH1_TimerCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
status	Event status in the timer mode
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

typedef void (*TC_TIMER_CALLBACK) (TC_TIMER_STATUS status, uintptr_t context);

TC_TIMER_CALLBACK_OBJECT Structure

Callback structure

File

plib_tc.h

C

```
typedef struct {
```

```
    TC_TIMER_CALLBACK callback_fn;
    uintptr_t context;
  } TC_TIMER_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

Compare

This section provides interface to use the TC peripheral in Compare mode

Introduction

This section provides brief overview of the TC peripheral in Compare mode

Description

The TC channel in compare mode generates one or two PWM signals with the same frequency and independently programmable duty cycles, or generates different types of one-shot or repetitive pulses.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

The TC channel in compare mode generates one or two PWM signals with the same frequency and independently programmable duty cycles, or generates different types of one-shot or repetitive pulses. In this mode, TIOAx is configured as an output and TIOBx is defined as an output if it is not used as an external event.

Library Interface

This section describes the Application Programming Interface (API) functions of the TC Peripheral Library in Compare mode

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	TCx_CHy_CompareInitialize	Initializes given TC channel in compare mode.

b) Setup Functions

	Name	Description
	TCx_CHy_CompareCallbackRegister	Registers the function to be called from interrupt.

c) Transaction Functions

	Name	Description
	TCx_CHy_CompareASet	Sets the Compare-A value of the given timer channel in compare mode.
	TCx_CHy_CompareBSet	Sets the Compare-B value of the given timer channel in compare mode.
	TCx_CHy_CompareFrequencyGet	Provides the given timer's counter-increment frequency in compare mode.
	TCx_CHy_ComparePeriodGet	Gets the period value of given timer channel in compare mode.
	TCx_CHy_ComparePeriodSet	Sets the period value of a given TC channel in compare mode.
	TCx_CHy_CompareStart	Starts the given TC channel compare counter.
	TCx_CHy_CompareStop	Stops the given TC channel counter in compare mode.

d) Status Functions

	Name	Description
	TCx_CHy_CompareStatusGet	Identifies status of the compare events

e) Data Types and Constants

	Name	Description
	TC_COMPARE_STATUS	Identifies channel interrupt source mask
	TC_COMPARE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_COMPARE_CALLBACK_OBJECT	Callback structure

a) Initialization Function

TCx_CHy_CompareInitialize Function

Initializes given TC channel in compare mode.

File

plib_tc.h

C

```
void TCx_CHy_CompareInitialize();
```

Returns

None.

Description

This function initializes given instance of a TC channel in compare mode with the options selected in MCC GUI.

Remarks

This function must be called before any other TC function is called. This function should only be called once during system initialization.

Preconditions

MCC GUI should be configured with the desired values.

Example

```
TCx_CHy_CompareInitialize();
```

Function

void TCx_CHy_CompareInitialize (void)

b) Setup Functions

TCx_CHy_CompareCallbackRegister Function

Registers the function to be called from interrupt.

File

plib_tc.h

C

```
void TCx_CHy_CompareCallbackRegister(TC_COMPARE_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from interrupt

Remarks

Context value can be set to NULL if not required.

To disable callback function, pass NULL for the callback parameter.

Preconditions

TCx_CHy_CompareInitialize must have been called first.

Example

```
void TC_CallbackFn(TC_COMPARE_STATUS status, uintptr_t context);

TC0_CH1_CompareInitialize();
TC0_CH1_CompareCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
callback	Callback function pointer.
context	Value provided back to the caller by the callback (usually a pointer to the caller's context for multi-instance clients).

Function

void TCx_CHy_CompareCallbackRegister (TC_COMPARE_CALLBACK callback, uintptr_t context)

c) Transaction Functions

TCx_CHy_CompareASet Function

Sets the Compare-A value of the given timer channel in compare mode.

File

plib_tc.h

C

```
void TCx_CHy_CompareASet(uint16_t value);
```

Returns

None.

Description

This function sets the Compare-A value of the given timer channel, which decides the ON time the waveform generated in compare mode.

Remarks

This function is applicable only for compare mode.

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel to configure channel in compare mode.

Example

```
TC0_CH1_CompareInitialize();
TC0_CH1_CompareASet(0x100);
```

Parameters

Parameters	Description
value	Compare-A value.

Function

void TCx_CHy_CompareASet (uint16_t value)

TCx_CHy_CompareBSet Function

Sets the Compare-B value of the given timer channel in compare mode.

File

plib_tc.h

C

```
void TCx_CHy_CompareBSet(uint16_t value);
```

Returns

None.

Description

This function sets the Compare-B value of the given timer channel, which decides the ON time the waveform generated in compare mode.

Remarks

This function is applicable only for compare mode.

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel to configure channel in compare mode.

Example

```
TC0_CH1_CompareInitialize();
TC0_CH1_CompareBSet(0x200);
```

Parameters

Parameters	Description
value	Compare-B value.

Function

void TCx_CHy_CompareBSet (uint16_t value)

TCx_CHy_CompareFrequencyGet Function

Provides the given timer's counter-increment frequency in compare mode.

File

plib_tc.h

C

```
uint32_t TCx_CHy_CompareFrequencyGet();
```

Returns

The frequency (in Hz) at which the timer's counter increments.

Description

This function provides the frequency at which the given counter increments in compare mode. It can be used to convert differences between counter values to real time or real-time intervals to timer period values.

Remarks

None

Preconditions

TCx_CHy_CompareInitialize function must have been called first for the given channel.

Example

```
uint32_t frequency;
TC0_CH1_CompareInitialize();
frequency = TC0_CH1_CompareFrequencyGet();
```

Function

uint32_t TCx_CHy_CompareFrequencyGet (void)

TCx_CHy_ComparePeriodGet Function

Gets the period value of given timer channel in compare mode.

File

plib_tc.h

C

```
uint32_t TCx_CHy_ComparePeriodGet();
```

Returns

The timer's period value.

Description

This function gets the current value of period of given timer channel in compare mode.

Remarks

The caller must know the number of significant bits of timer. Period value is right-aligned.

Preconditions

TCx_CHy_CompareInitialize() function must have been called first for the given channel.

Example

```
uint32_t period;
TC0_CH1_CompareInitialize();
period = TC0_CH1_ComparePeriodGet();
```

Function

uint32_t TCx_CHy_ComparePeriodGet (void)

TCx_CHy_ComparePeriodSet Function

Sets the period value of a given TC channel in compare mode.

File

plib_tc.h

C

```
void TCx_CHy_ComparePeriodSet(uint32_t period);
```

Returns

None.

Description

This function writes the TC compare timer's period value. When counter matches the period value, it is reset, the waveform phase is updated and an interrupt can be generated.

Remarks

The caller must know the number of significant bits of timer. The Period value is right-aligned.

Preconditions

TCx_CHy_CompareInitialize() function must have been called first for the given channel.

Example

```
TC0_CH1_CompareInitialize();
TC0_CH1_ComparePeriodSet(0x500U);
```

Parameters

Parameters	Description
period	The timer period value necessary to achieve the desired waveform frequency.

Function

void TCx_CHy_ComparePeriodSet (uint32_t period);

TCx_CHy_CompareStart Function

Starts the given TC channel compare counter.

File

plib_tc.h

C

```
void TCx_CHy_CompareStart();
```

Returns

None.

Description

This function enables the clock and starts the counter of associated channel in compare mode so that the TC module may start generating the selected waveform.

Remarks

None

Preconditions

TCx_CHy_CompareInitialize() function must have been called first for the given channel.

Example

```
TC0_CH1_CompareInitialize();
TC0_CH1_CompareStart();
```

Function

void TCx_CHy_CompareStart (void)

TCx_CHy_CompareStop Function

Stops the given TC channel counter in compare mode.

File

plib_tc.h

C

```
void TCx_CHy_CompareStop();
```

Returns

None.

Description

This function stops the clock and thus counter of associated channel, freezing the associated waveform generation.

Remarks

None

Preconditions

TCx_CHy_CompareInitialize function must have been called first for the given channel.

Example

```
TC0_CH1_CompareInitialize();
TC0_CH1_CompareStart();
// Generate waveform a long as desired, then...
TC0_CH1_CompareStop();
```

Function

void TCx_CHy_CompareStop (void)

d) Status Functions

TCx_CHy_CompareStatusGet Function

Identifies status of the compare events

File

plib_tc.h

C

```
TC_COMPARE_STATUS TCx_CHy_CompareStatusGet();
```

Returns

TC_COMPARE_STATUS - status of the capture events

Description

This function returns the status of the events that occur in compare mode.

Remarks

This function is applicable only for compare mode. Event status bits are cleared after reading the status register.

Preconditions

TCx_CHy_CompareInitialize() function must have been called first for the given channel to configure channel in capture mode.

Example

```
if ((TC0_CH1_CompareStatusGet() & TC_COMPARE_C) == true)
{
    TC0_CH1_CompareASet(0x1000U);
    TC0_CH1_CompareBSet(0x500U);
}
```

Function

TC_COMPARE_STATUS TCx_CHy_CompareStatusGet (void)

e) Data Types and Constants

TC_COMPARE_STATUS Enumeration

Identifies channel interrupt source mask

File

plib_tc.h

C

```
typedef enum {
  TC_COMPARE_NONE,
  TC_COMPARE_A,
  TC_COMPARE_B,
  TC_COMPARE_C,
  TC_COMPARE_STATUS_MSK
} TC_COMPARE_STATUS;
```

Description

Interrupt source mask for compare mode

This enumeration identifies TC compare mode interrupt source mask.

Remarks

None.

TC_COMPARE_CALLBACK Type

Defines the function pointer data type and function signature for the tc channel callback function.

File

plib_tc.h

C

```
typedef void (* TC_COMPARE_CALLBACK)(TC_COMPARE_STATUS status, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer for Compare mode

This data type defines the function pointer and function signature for the tc channel callback function. The library will call back the client's function with this signature from the interrupt routine.

Remarks

None.

Preconditions

TCx_CHy_Initialize must have been called for the given TC channel instance and TCx_CHy_CompareCallbackRegister must have been called to register the function to be called.

Example

```
void TC_CallbackFn (TC_COMPARE_STATUS status, uintptr_t context );

TC0_CH1_CompareCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
status	Event status in compare mode
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

typedef void (*TC_COMPARE_CALLBACK) (TC_COMPARE_STATUS status, uintptr_t context);

TC_COMPARE_CALLBACK_OBJECT Structure

Callback structure

File

plib_tc.h

C

```
typedef struct {
  TC_COMPARE_CALLBACK callback_fn;
  uintptr_t context;
} TC_COMPARE_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

Capture

This section provides interface to use the TC peripheral in Capture mode

Introduction

This section provides brief overview of the TC peripheral in Capture mode

Description

The TC channel in capture mode is used to measure pulse time, frequency, period and duty cycle.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

The Capture mode allows the TC channel to perform measurements such as pulse timing, frequency, period, duty cycle and phase on TIOAx and TIOBx signals.

In Capture mode, TIOAx and TIOBx are configured as inputs. The counter value is captured in the buffer on the selected input edge and it is used as a timestamp for the event. User can select up to two input events and the respective timestamp is captured in Capture A and Capture B buffers, it can be used to measure pulse, period, frequency and duty cycle of the external input.

It provides both polling and callback methods to indicate capture event has occurred.

- With polling, the application will need to continuously poll to check if the capture event has occurred.
- With callback, the registered callback function will be called when the capture event occurs. This means we do not have to poll continuously.

Callback method

This example demonstrates how to use TC channel in capture mode to measure off time and period. The rising edge is captured in Capture A and Falling edge is captured in Capture B register, and then the timer is reset.

```
/* This function is called after Capture B */
void TC0_CH0_CaptureInterruptHandler(uintptr_t context)
{
    /* Read Captured values */
    off_time = TC0_CH0_CaptureAGet();
    period = TC0_CH0_CaptureBGet();
}

int main(void)
```

```
{

    /* Register callback function for CH0 period interrupt */
    TC0_CH0_CaptureCallbackRegister(TC0_CH0_CaptureInterruptHandler, (uintptr_t)NULL);

    /* Start the Capture channel 0*/
    TC0_CH0_CaptureStart();
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the TC Peripheral Library in Capture mode Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	TCx_CHy_CaptureInitialize	Initializes given instance of TC channel to capture mode.

b) Setup Functions

	Name	Description
	TCx_CHy_CaptureCallbackRegister	Registers the function to be called from the capture-event.

c) Transaction Functions

	Name	Description
	TCx_CHy_CaptureAGet	Returns the Capture-A value.
	TCx_CHy_CaptureBGet	Returns the Capture-B value.
	TCx_CHy_CaptureStart	Starts the given TC channel capture counter.
	TCx_CHy_CaptureStop	Stops the given TC channel counter.

d) Status Functions

	Name	Description
	TCx_CHy_CaptureStatusGet	Identifies status of the capture events

e) Data Types and Constants

	Name	Description
	TC_CAPTURE_STATUS	Identifies channel interrupt source mask
	TC_CAPTURE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_CAPTURE_CALLBACK_OBJECT	Callback structure

a) Initialization Function

TCx_CHy_CaptureInitialize Function

Initializes given instance of TC channel to capture mode.

File

plib_tc.h

C

```
void TCx_CHy_CaptureInitialize();
```

Returns

None.

Description

This function initializes given instance of TC channel to capture mode with the options configured in MCC GUI, allowing it to be ready to start capturing selected signal event counter values.

Remarks

This function must be called before any other TC function is called. This function is normally only called once during system initialization.

Preconditions

MCC GUI should be configured with the desired values.

Example

```
TC0_CH1_CaptureInitialize();
```

Function

void TCx_CHy_CaptureInitialize (void)

b) Setup Functions

TCx_CHy_CaptureCallbackRegister Function

Registers the function to be called from the capture-event.

File

plib_tc.h

C

```
void TCx_CHy_CaptureCallbackRegister(TC_CAPTURE_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from the interrupt when the selected capture event occurs.

Remarks

Context value can be set to NULL if not required.

To disable callback function, pass NULL for the callback parameter.

Preconditions

TCx_CHy_CaptureInitialize() must have been called first.

Example

```
void TC_CallbackFn(TC_CAPTURE_STATUS status, uintptr_t context);

TC0_CH1_CaptureInitialize();
TC0_CH1_CaptureCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
callback	Callback function pointer.

context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

void TCx_CHy_CaptureCallbackRegister (TC_CAPTURE_CALLBACK callback, uintptr_t context)

c) Transaction Functions

TCx_CHy_CaptureAGet Function

Returns the Capture-A value.

File

plib_tc.h

C

```
uint16_t TCx_CHy_CaptureAGet();
```

Returns

Most recent Capture-A value.

Description

This function provides the Capture-A value that was stored when the selected event occurred on the input signal. The caller should call the TCx_CHy_CaptureAEventOccured function to identify if the Capture-A value has been updated by the selected signal event.

Remarks

This function is applicable only for capture mode.

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel to configure channel in capture mode.

Example

```
uint16_t capture;


if (TC0_CH1_CaptureAEventOccured() == true)
{
    capture = TC0_CH1_CaptureAGet();
}
```

Function

uint16_t TCx_CHy_CaptureAGet (void)

TCx_CHy_CaptureBGet Function

Returns the Capture-B value.

File

plib_tc.h

C

```
uint16_t TCx_CHy_CaptureBGet();
```

Returns

Most recent Capture-B value.

Description

This function provides the Capture-B value that was stored when the selected event occurred on the input signal. The caller should call the TCx_CHy_CaptureBEventOccured function to identify if the Capture-B value has been updated by the selected signal event.

Remarks

This function is applicable only for capture mode.

Preconditions

TCx_CHy_TimerInitialize function must have been called first for the given channel to configure channel in capture mode.

Example

```
uint16_t capture;


if (TC0_CH1_CaptureBEventOccured() == true)
{
    capture = TC0_CH1_CaptureBGet();
}
```

Function

uint16_t TCx_CHy_CaptureBGet (void)

TCx_CHy_CaptureStart Function

Starts the given TC channel capture counter.

File

plib_tc.h

C

```
void TCx_CHy_CaptureStart();
```

Returns

None.

Description

This function enables the clock and starts the counter for the associated channel so that it can start capturing signal-event counter values.

Remarks

None

Preconditions

TCx_CHy_CaptureInitialize() function must have been called first for the given channel.

Example

```
TC0_CH1_CaptureInitialize();
TC0_CH1_CaptureStart();
```

Function

void TCx_CHy_CaptureStart (void);

TCx_CHy_CaptureStop Function

Stops the given TC channel counter.

File

plib_tc.h

C

```
void TCx_CHy_CaptureStop();
```

Returns

None.

Description

This function stops the clock and thus counter of associated channel, preventing it from capturing signal-event counter values.

Remarks

None

Preconditions

TCx_CHy_CaptureInitialize() function must have been called first for the given channel.

Example

```
TC0_CH1_CaptureInitialize();
TC0_CH1_CaptureStart();
// Capture signal event counter values.
TC0_CH1_CaptureStop();
```

Function

void TCx_CHy_CaptureStop (void);

d) Status Functions

TCx_CHy_CaptureStatusGet Function

Identifies status of the capture events

File

plib_tc.h

C

```
TC_CAPTURE_STATUS TCx_CHy_CaptureStatusGet();
```

Returns

TC_CAPTURE_STATUS - status of the capture events

Description

This function returns the status of the events that occur in capture mode.

Remarks

This function is applicable only for capture mode. Event status bits are cleared after reading the status register.

Preconditions

TCx_CHy_TimerInitialize() function must have been called first for the given channel to configure channel in capture mode.

Example

```
uint16_t capture;

if ((TC0_CH1_CaptureStatusGet() & TC_CAPTURE_A_LOAD) == true)
{
    capture = TC0_CH1_CaptureAGet();
}
```

Function

TC_CAPTURE_STATUS TCx_CHy_CaptureStatusGet (void)

e) Data Types and Constants

TC_CAPTURE_STATUS Enumeration

Identifies channel interrupt source mask

File

plib_tc.h

C

```
typedef enum {
  TC_CAPTURE_NONE,
  TC_CAPTURE_COUNTER_OVERFLOW,
  TC_CAPTURE_LOAD_OVERRUN,
  TC_CAPTURE_A_LOAD,
  TC_CAPTURE_B_LOAD,
  TC_CAPTURE_STATUS_MSK
} TC_CAPTURE_STATUS;
```

Description

Interrupt source mask for capture mode

This enumeration identifies TC capture mode interrupt source mask.

Remarks

None.

TC_CAPTURE_CALLBACK Type

Defines the function pointer data type and function signature for the tc channel callback function.

File

plib_tc.h

C

```
typedef void (* TC_CAPTURE_CALLBACK)(TC_CAPTURE_STATUS status, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer for Capture mode

This data type defines the function pointer and function signature for the tc channel callback function. The library will call back the client's function with this signature from the interrupt routine.

Remarks

None.

Preconditions

TCx_CHy_Initialize must have been called for the given TC channel instance and TCx_CHy_CaptureCallbackRegister must have been called to register the function to be called.

Example

```
void TC_CallbackFn (TC_CAPTURE_STATUS status, uintptr_t context );

TC0_CH1_CaptureCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
status	Event status in capture mode
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

typedef void (*TC_CAPTURE_CALLBACK) (TC_CAPTURE_STATUS status, uintptr_t context);

TC_CAPTURE_CALLBACK_OBJECT Structure

Callback structure

File

plib_tc.h

C

```
typedef struct {
  TC_CAPTURE_CALLBACK callback_fn;
  uintptr_t context;
} TC_CAPTURE_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

Quadrature Decoder

This section provides interface to use the TC peripheral as Quadrature decoder

Introduction

This section provides brief overview of the TC peripheral in Quadrature Decoder mode

Description

The quadrature decoder (QDEC) mode is used to measure the position and speed of the motor.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

The TC embeds a quadrature decoder (QDEC) connected in front of the timers and driven by TIOA0, TIOB0 and TIOB1 inputs. When enabled, the QDEC performs the input lines filtering, decoding of quadrature signals and connects to the timers/counters in order to read the position and speed of the motor through the user interface.

Library Interface

This section describes the Application Programming Interface (API) functions of the TC Peripheral Library in Quadrature Decoder mode

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	TCx_QuadratureInitialize	Initializes given instance of TC channel in quadrature mode.

b) Setup Functions

	Name	Description
	TCx_QuadratureCallbackRegister	Registers the function to be called from interrupt.

c) Transaction Functions

	Name	Description
	TCx_QuadratureRevolutionsGet	Reads the number of revolutions from the quadrature encoder.
	TCx_QuadraturePositionGet	Reads the angular position from the quadrature encoder.
	TCx_QuadratureSpeedGet	Reads the quadrature index change speed.
	TCx_QuadratureStart	Starts the given TC channel counter in quadrature mode.
	TCx_QuadratureStop	Stops the given TC channel counter in quadrature mode.

d) Status Functions

	Name	Description
	TCx_QuadratureStatusGet	Identifies status of the quadrature events

e) Data Types and Constants

	Name	Description
	TC_QUADRATURE_STATUS	Identifies channel interrupt source mask
	TC_QUADRATURE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_QUADRATURE_CALLBACK_OBJECT	Callback structure

Description

a) Initialization Function

TCx_QuadratureInitialize Function

Initializes given instance of TC channel in quadrature mode.

File

plib_tc.h

C

```
void TCx_QuadratureInitialize();
```

Returns

None.

Description

This function initializes given instance of TC channel in quadrature mode with options configured in MCC GUI.

Remarks

This function must be called before any other TC function is called.

This function is normally called only be called once during system initialization.

Preconditions

MCC GUI should be configured with the desired options.

Example

```
TC0_QuadratureInitialize();
```

Function

void TCx_QuadratureInitialize (void);

b) Setup Functions

TCx_QuadratureCallbackRegister Function

Registers the function to be called from interrupt.

File

plib_tc.h

C

```
void TCx_QuadratureCallbackRegister(TC_QUADRATURE_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function registers the callback function to be called from interrupt

Remarks

Context value can be set to NULL if not required.

To disable callback function, pass NULL for the callback parameter.

Preconditions

TCx_QuadratureInitialize must have been called first.

Example

```
void TC_CallbackFn(TC_QUADRATURE_STATUS status, uintptr_t context);

TC0_QuadratureInitialize();
TC0_QuadratureCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
callback	Callback function pointer.
context	Value provided back to the caller by the callback (usually a pointer to the caller's context for multi-instance clients).

Function

void TCx_QuadratureCallbackRegister (TC_QUADRATURE_CALLBACK callback, uintptr_t context)

c) Transaction Functions

TCx_QuadratureRevolutionsGet Function

Reads the number of revolutions from the quadrature encoder.

File

plib_tc.h

C

```
int16_t TCx_QuadratureRevolutionsGet();
```

Returns

Number of revolutions of the encoder.

Description

This function reads the number of revolutions from the quadrature encoder by reading as tracked by the channel counter.

Remarks

In counter-clockwise direction, revolution counter works in down counting mode. This function is available only if quadrature encoder provides index pulse.

Preconditions

TCx_QuadratureInitialize function must have been called first for the given channel.

Example

```
int16_t revolutions;

TC0_QuadratureInitialize();
```

```
TC0_QuadratureStart();
revolutions = TC0_QuadratureRevolutionsGet();
```

Function

int16_t TCx_QuadratureRevolutionsGet (void)

TCx_QuadraturePositionGet Function

Reads the angular position from the quadrature encoder.

File

plib_tc.h

C

```
int16_t TCx_QuadraturePositionGet();
```

Returns

Position of the encoder.

Description

This function reads the position from the quadrature encoder by reading as tracked by the channel counter.

Remarks

In counter-clockwise direction, position counter works in down counting mode.

Preconditions

TCx_QuadratureInitialize function must have been called first for the given channel.

Example

```
int16_t position;

TC0_QuadratureInitialize();
TC0_QuadratureStart();
position = TC0_QuadraturePositionGet();
```

Function

int16_t TCx_QuadraturePositionGet (void)

TCx_QuadratureSpeedGet Function

Reads the quadrature index change speed.

File

plib_tc.h

C

```
uint16_t TCx_QuadratureSpeedGet();
```

Returns

Number of quadrature pulses counted in given time base which is used to determine speed of motion tracked by the encoder.

Description

This function reads the number of quadrature pulses captured in timer channel for given time base.

Remarks

Caller must calculate the actual speed based on returned speed count value, time base and number of quadrature encoder pulses.

Preconditions

TCx_QuadratureInitialize function must have been called first for the given channel.

Example

```
uint16_t speed;
TC0_QuadratureInitialize();
TC0_QuadratureStart();
speed = TC0_QuadratureSpeedGet();
```

Function

uint16_t TCx_QuadratureSpeedGet (void)

TCx_QuadratureStart Function

Starts the given TC channel counter in quadrature mode.

File

plib_tc.h

C

```
void TCx_QuadratureStart();
```

Returns

None.

Description

This function enables the clock and starts the counter of associated channel in quadrature mode, allowing the TC to begin tracking the quadrature signals.

Remarks

None.

Preconditions

TCx_QuadratureInitialize function must have been called first for the given channel.

Example

```
TC0_QuadratureInitialize();
TC0_QuadratureStart();
```

Function

void TCx_QuadratureStart (void)

TCx_QuadratureStop Function

Stops the given TC channel counter in quadrature mode.

File

plib_tc.h

C

```
void TCx_QuadratureStop();
```

Returns

None.

Description

This function stops the clock and thus counter of associated channel in quadrature mode, stopping tracking of the quadrature signals.

Remarks

None.

Preconditions

TCx_QuadratureInitialize function must have been called first for the given channel.

Example

```
TC0_QuadratureInitialize();
TC0_QuadratureStart();
TC0_QuadratureStop();
```

Function

void TCx_QuadratureStop (void)

d) Status Functions

TCx_QuadratureStatusGet Function

Identifies status of the quadrature events

File

plib_tc.h

C

```
TC_QUADRATURE_STATUS TCx_QuadratureStatusGet();
```

Returns

TC_QUADRATURE_STATUS - status of the capture events

Description

This function returns the status of the events that occur in quadrature mode.

Remarks

This function is applicable only for quadrature mode. Event status bits are cleared after reading the status register.

Preconditions

TCx_CHy_QuadratureInitialize() function must have been called first for the given channel to configure channel in quadrature mode.

Example

```
uint32_t position;

if ((TC0_QuadratureStatusGet() & TC_QUADRATURE_INDEX) == true)
{
    position = TC0_QuadraturePositionGet()
}
```

Function

TC_QUADRATURE_STATUS TCx_CHy_QuadratureStatusGet (void)

e) Data Types and Constants

TC_QUADRATURE_STATUS Enumeration

Identifies channel interrupt source mask

File

plib_tc.h

C

```
typedef enum {
  TC_QUADRATURE_NONE,
  TC_QUADRATURE_INDEX,
  TC_QUADRATURE_DIR_CHANGE,
  TC_QUADRATURE_ERROR,
  TC_QUADRATURE_STATUS_MSK
} TC_QUADRATURE_STATUS;
```

Description

Interrupt source mask for quadrature mode

This enumeration identifies TC quadrature mode interrupt source mask.

Remarks

None.

TC_QUADRATURE_CALLBACK Type

Defines the function pointer data type and function signature for the tc channel callback function.

File

plib_tc.h

C

```
typedef void (* TC_QUADRATURE_CALLBACK)(TC_QUADRATURE_STATUS status, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer for Quadrature mode

This data type defines the function pointer and function signature for the tc channel callback function. The library will call back the client's function with this signature from the interrupt routine.

Remarks

None.

Preconditions

TCx_Initialize must have been called for the given TC channel instance and TCx_QuadratureCallbackRegister must have been called to register the function to be called.

Example

```
void TC_CallbackFn (TC_QUADRATURE_STATUS status, uintptr_t context );
```

```
TC0_QuadratureCallbackRegister(TC_CallbackFn, NULL);
```

Parameters

Parameters	Description
status	Event status in quadrature mode
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Function

typedef void (*TC_QUADRATURE_CALLBACK) (TC_QUADRATURE_STATUS status, uintptr_t context);

TC_QUADRATURE_CALLBACK_OBJECT Structure

Callback structure

File

plib_tc.h

C

```
typedef struct {
  TC_QUADRATURE_CALLBACK callback_fn;
  uintptr_t context;
} TC_QUADRATURE_CALLBACK_OBJECT;
```

Description

Callback structure

This stores the callback function pointer and context

Remarks

None.

Files

Files

Name	Description
plib_tc.h	TC peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_tc.h

TC peripheral library interface.

Enumerations

	Name	Description
	TC_CAPTURE_STATUS	Identifies channel interrupt source mask
	TC_COMPARE_STATUS	Identifies channel interrupt source mask
	TC_QUADRATURE_STATUS	Identifies channel interrupt source mask
	TC_TIMER_STATUS	Identifies channel interrupt source mask

Functions

	Name	Description
	TCx_CHy_CaptureAGet	Returns the Capture-A value.
	TCx_CHy_CaptureBGet	Returns the Capture-B value.
	TCx_CHy_CaptureCallbackRegister	Registers the function to be called from the capture-event.
	TCx_CHy_CaptureInitialize	Initializes given instance of TC channel to capture mode.
	TCx_CHy_CaptureStart	Starts the given TC channel capture counter.
	TCx_CHy_CaptureStatusGet	Identifies status of the capture events
	TCx_CHy_CaptureStop	Stops the given TC channel counter.
	TCx_CHy_CompareASet	Sets the Compare-A value of the given timer channel in compare mode.
	TCx_CHy_CompareBSet	Sets the Compare-B value of the given timer channel in compare mode.
	TCx_CHy_CompareCallbackRegister	Registers the function to be called from interrupt.
	TCx_CHy_CompareFrequencyGet	Provides the given timer's counter-increment frequency in compare mode.
	TCx_CHy_CompareInitialize	Initializes given TC channel in compare mode.
	TCx_CHy_ComparePeriodGet	Gets the period value of given timer channel in compare mode.
	TCx_CHy_ComparePeriodSet	Sets the period value of a given TC channel in compare mode.
	TCx_CHy_CompareStart	Starts the given TC channel compare counter.
	TCx_CHy_CompareStatusGet	Identifies status of the compare events
	TCx_CHy_CompareStop	Stops the given TC channel counter in compare mode.
	TCx_CHy_TimerCallbackRegister	Registers the function to be called from interrupt
	TCx_CHy_TimerCompareSet	Sets the compare value of a given timer channel.
	TCx_CHy_TimerCounterGet	Reads the timer channel counter value
	TCx_CHy_TimerFrequencyGet	Provides the given timer's counter-increment frequency.
	TCx_CHy_TimerInitialize	Initializes given instance of TC channel
	TCx_CHy_TimerPeriodGet	Reads the period value of given timer channel
	TCx_CHy_TimerPeriodHasExpired	Checks whether timer period is elapsed
	TCx_CHy_TimerPeriodSet	Sets the period value of a given timer channel.
	TCx_CHy_TimerStart	Starts the given TC channel counter
	TCx_CHy_TimerStop	Stops the given TC channel counter
	TCx_QuadratureCallbackRegister	Registers the function to be called from interrupt.
	TCx_QuadratureInitialize	Initializes given instance of TC channel in quadrature mode.
	TCx_QuadraturePositionGet	Reads the angular position from the quadrature encoder.
	TCx_QuadratureRevolutionsGet	Reads the number of revolutions from the quadrature encoder.
	TCx_QuadratureSpeedGet	Reads the quadrature index change speed.
	TCx_QuadratureStart	Starts the given TC channel counter in quadrature mode.
	TCx_QuadratureStatusGet	Identifies status of the quadrature events
	TCx_QuadratureStop	Stops the given TC channel counter in quadrature mode.

Structures

	Name	Description
	TC_CAPTURE_CALLBACK_OBJECT	Callback structure
	TC_COMPARE_CALLBACK_OBJECT	Callback structure
	TC_QUADRATURE_CALLBACK_OBJECT	Callback structure
	TC_TIMER_CALLBACK_OBJECT	Callback structure

Types

	Name	Description
	TC_CAPTURE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.

	TC_COMPARE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_QUADRATURE_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.
	TC_TIMER_CALLBACK	Defines the function pointer data type and function signature for the tc channel callback function.

Description

TC Peripheral Library Interface Header File

This file defines the interface to the TC peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual tcheaders will be generated as required by the MCC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "TC" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

This interface supports the following different modes of the TC peripheral.

- Timer Mode
- Capture Mode
- Compare Mode
- Quadrature Mode

Only one of these modes (and only the associated interface functions) will be active (and thus will be generated at a time.)

File Name

plib_tc.h

Company

Microchip Technology Inc.

TRNG Peripheral Library Help

This section provides interface to use True Random Number Generator (TRNG) peripheral.

Introduction

This section provides brief overview of True Random Number Generator (TRNG) peripheral.

Description

As soon as the TRNG is enabled, the module provides a new 32-bit random data, for every 84 peripheral clock cycles.

The TRNG can optionally generate interrupt when a new random value is available.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This Section provides information on how to use the Peripheral Library.

Description

The library provides both blocking and callback methods to generate and read the random number.

- With blocking API, it enable TRNG module, waits for new random number to be available and then returns the newly generated random number.
- With callback, it enables TRNG module and the registered callback function will be called once the new random value is available.

Blocking API

The example code to generate 32-bit random number using blocking API is given below.

```
uint32_t randomNumber
randomNumber = TRNG0_ReadData();
```

Callback method

This example demonstrates how to use callback to generate 32-bit random number.

```
/* Random number is provided as part of callback */
void TRNG_EventHandler(uint32_t randomNumber, uintptr_t context)
{

}

int main(void)
{
    /* Register Callback */
    TRNG_CallbackRegister(&TRNG_EventHandler, (uintptr_t) NULL);

    /* Enable TRNG */
    TRNG_RandomNumberGenerate();
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Setup Functions

	Name	Description
	TRNGx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the random number is ready to be read.

b) Data Types and Constants

	Name	Description
	TRNG_CALLBACK	Defines the data type and function signature for the TRNG peripheral callback function.

a) Setup Functions

TRNGx_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the random number is ready to be read.

File

plib_trng.h

C

```
void TRNGx_CallbackRegister(TRNG_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the random number is ready. It also accepts a context value that is passed into the function when it is called. This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. In this case the value NULL will be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the TRNG_CALLBACK type definition for additional information.

Preconditions

None.

Example

```
uint32_t randomValue;
void testCallback( uintptr_t context, uint32_t random)
{
    randomValue= random;
}
TRNG0_CallbackRegister(testCallback, NULL);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the TRNG_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void TRNGx_CallbackRegister(TRNG_CALLBACK callback, uintptr_t context)

b) Transaction Functions

Functions

	Name	Description
	TRNGx_ReadData	Generates and return a 32-bit random number
	TRNGx_RandomNumberGenerate	Initiates generation of random number using the TRNG peripheral.

Description

TRNGx_ReadData Function

Generates and return a 32-bit random number

File

plib_trng.h

C

```
uint32_t TRNGx_ReadData();
```

Returns

uint32_t:- 32-bit true random number

Description

This function returns a 32-bit random integer.

Remarks

This API is required when the library is configured as blocking. In non-blocking mode the random number will be passed via callback.

Preconditions

None

Example

```
uint32_t data = TRNG0_ReadData();
```

Function

uint32_t TRNGx_ReadData(void)

TRNGx_RandomNumberGenerate Function

Initiates generation of random number using the TRNG peripheral.

File

plib_trng.h

C

```
void TRNGx_RandomNumberGenerate();
```

Returns

None.

Description

This function triggers the TRNG peripheral to generate a 32-bit random number. The API will return in case of interrupt mode and wait for random number generation to complete in case of polled mode.

Remarks

This API should be used when the library is configured in interrupt mode.

Preconditions

None.

Example

```
TRNG0_RandomNumberGenerate();
```

Function

void TRNGx_RandomNumberGenerate(void)

b) Data Types and Constants

TRNG_CALLBACK Type

Defines the data type and function signature for the TRNG peripheral callback function.

File

plib_trng.h

C

```
typedef void (* TRNG_CALLBACK)(uint32_t random, uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the TRNG peripheral callback function. The TRNG peripheral will call back the client's function with this signature when the random data is ready to be read.

Preconditions

TRNGx_CallbackRegister must have been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).
random	The generated 32-bit random number that will be passed via callback.

Files

Files

Name	Description
plib_trng.h	Interface definition of the True Random Number Generator Plib (TRNG).

Description

This section will list only the library's interface header file(s).

plib_trng.h

Interface definition of the True Random Number Generator Plib (TRNG).

Functions

	Name	Description
	TRNGx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the random number is ready to be read.
	TRNGx_RandomNumberGenerate	Initiates generation of random number using the TRNG peripheral.
	TRNGx_ReadData	Generates and return a 32-bit random number

Types

	Name	Description
	TRNG_CALLBACK	Defines the data type and function signature for the TRNG peripheral callback function.

Description

Interface definition of TRNG PLIB.

This file defines the interface for the True Random Number Generator (TRNG) Plib. It allows user to generate random numbers.

File Name

plib_trng.h

Company

Microchip Technology Inc.

TWIHS Peripheral Library Help

This section provides interface to use Two-wire Interface (TWIHS) peripheral.

Introduction

This section provides brief overview of Two-wire Interface (TWIHS) peripheral.

Description

The Two-wire Interface (TWIHS) interconnects components on a unique two-wire bus, made up of one clock line and one data line with speeds of up to 400 kbit/s in Fast mode and up to 3.4 Mbit/s in Highspeed slave mode only, based on a byte-oriented transfer format. It can be used with any Two-wire Interface bus Serial EEPROM and I²C-compatible devices, such as a Real-Time Clock (RTC), Dot Matrix/ Graphic LCD Controller and temperature sensor. The TWIHS is programmable as a master or a slave with sequential or single-byte access. Multiple master capability is supported.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains how to use the Peripheral Library

Description

The TWIHS peripheral library supports following I2C transfers:

- Master Write: The master writes a block of data to the slave
- Master Read: The master reads a block of data from the slave
- Master Write/Read: The master writes and then reads back a block of data from slave.

The block of data is transferred in non-blocking manner using peripheral interrupt. It provides both polling and callback methods to indicate the end of transfer.

- With polling, the application will need to continuously poll to check if the transfer is completed.
- With callback, the registered callback function will be called when the transfer is completed. This means we do not have to poll continuously.

The example code demonstrates write operation using polling method.

```
#define APP_SLAVE_ADDR 0x0057
#define NUM_BYTES      10

uint8_t myWriteData [NUM_BYTES] = {'1', '0', ' ', 'B', 'Y', 'T', 'E', 'S', '!', '!',};
```

```
int main(void)
{
    /* Write data to the TWIHS0 Slave */
    TWIHSx_Write(APP_SLAVE_ADDR, &myWriteData[0], NUM_BYTES);

    /* Poll and wait for the transfer to complete */
    while(TWIHSx_IsBusy());

    /* Check if any error occurred */
    if(TWIHSx_ErrorGet() == TWIHS_ERROR_NONE)
    {
        //Transfer is completed successfully
    }
    else
    {
        //Error occurred during transfer.
    }
}
```

The example code demonstrates write operation using callback method.

```
#define APP_SLAVE_ADDR 0x0057
#define NUM_BYTES      10

uint8_t myWriteData [NUM_BYTES] = {'1', '0', ' ', 'B', 'Y', 'T', 'E', 'S', '!', '!',};

void TWIHSx_Callback(uintptr_t context)
{
    if(TWIHSx_ErrorGet() == TWIHS_ERROR_NONE)
    {
        //Transfer is completed successfully
    }
    else
    {
        //Error occurred during transfer.
    }
}

int main(void)
{
    /* Register Callback function */
    TWIHSx_CallbackRegister(TWIHSx_Callback, (uintptr_t)NULL);

    /* Submit Write Request */
    TWIHSx_Write(APP_SLAVE_ADDR, &myWriteData[0], NUM_BYTES);
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	TWIHSx_Initialize	Initializes given instance of the TWIHS peripheral.

b) Setup Function

	Name	Description
	TWIHSx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the given TWIHS's transfer events occur.

c) Transaction Functions

	Name	Description
	TWIHSx_Read	Reads data from the slave.
	TWIHSx_Write	Writes data onto the slave.
	TWIHSx_WriteRead	Write and Read data from Slave.

d) Status Functions

	Name	Description
	TWIHSx_IsBusy	Returns the TWIHS Peripheral busy status.
	TWIHSx_ErrorGet	Returns the error during transfer.

e) Data Types and Constants

	Name	Description
	TWIHS_CALLBACK	Defines the data type and function signature for the TWIHS peripheral callback function.
	TWIHS_ERROR	TWIHS Error Status data type.

a) Initialization Function

TWIHSx_Initialize Function

Initializes given instance of the TWIHS peripheral.

File

plib_twihs.h

C

```
void TWIHSx_Initialize();
```

Returns

None

Description

This function initializes the given instance of the TWIHS peripheral as configured by the user from within the MHC.

Remarks

Stops the TWIHS if it was already running and reinitializes it.

Preconditions

None.

Example

```
TWIHS1_Initialize();
```

Function

void TWIHSx_Initialize(void)

b) Setup Function

TWIHSx_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the given TWIHS's transfer events occur.

File

plib_twihs.h

C

```
void TWIHSx_CallbackRegister(TWIHS_CALLBACK callback, uintptr_t contextHandle);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the given TWIHS's transfer events occur. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

Remarks

The context parameter is ignored if the pointer passed is NULL. To disable the callback function, pass a NULL for the callback parameter. See the TWIHS_CALLBACK type definition for additional information.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
TWIHS1_CallbackRegister(MyTWIHSCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the TWIHS_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void TWIHSx_CallbackRegister(TWIHS_CALLBACK callback, uintptr_t contextHandle)

c) Transaction Functions

TWIHSx_Read Function

Reads data from the slave.

File

plib_twihs.h

C

```
bool TWIHSx_Read(uint16_t address, uint8_t * pdata, size_t length);
```

Returns

- True - Read request was submitted successfully.
- False - Failed while submitting the read request.

Description

This function reads the data of size in bytes equal to length into pdata buffer from the slave having given address. Master will generate Stop condition after completion of the read.

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
uint8_t myData [NUM_BYTES];

if(!TWIHS1_Read( SLAVE_ADDR, &myData[0], NUM_BYTES ))
{
    // error handling
}
```

Parameters

Parameters	Description
address	7-bit / 10-bit slave address.
pdata	pointer to destination data buffer
length	length of data buffer in number of bytes.

Function

bool TWIHSx_Read(uint16_t address, uint8_t *pdata, size_t length)

TWIHSx_Write Function

Writes data onto the slave.

File

plib_twihs.h

C

```
bool TWIHSx_Write(uint16_t address, uint8_t * pdata, size_t length);
```

Returns

- True - Write request was submitted successfully.
- False - Failed while submitting the write request.

Description

This function writes the data from pdata buffer of size in bytes equal to length onto the slave with given address. Master will generate Stop condition after completion of the write.

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
uint8_t myData [NUM_BYTES] = {'1', '0', ' ', 'B', 'Y', 'T', 'E', 'S', '!', '!',};

if(!TWIHS1_Write( SLAVE_ADDR, &myData[0], NUM_BYTES ))
{
    // error handling
```

```
}
```

Parameters

Parameters	Description
address	7-bit / 10-bit slave address.
pdata	pointer to source data buffer
length	length of data buffer in number of bytes.

Function

bool TWIHSx_Write(uint16_t address, uint8_t *pdata, size_t length)

TWIHSx_WriteRead Function

Write and Read data from Slave.

File

plib_twihs.h

C

```
bool TWIHSx_WriteRead(uint16_t address, uint8_t * wdata, size_t wlength, uint8_t * rdata,
size_t rlength);
```

Returns

- True - Write read request was submitted successfully.
- False - Failed while submitting the write read request.

Description

This function writes data from wdata buffer to the slave of given address and of size(in bytes) equal to wlength. Master then generates a repeated start and do a read operation. Read operation reads data into rdata buffer of size(in bytes) equal to rlength from the slave of given address. Master generates a stop condition after reading the data.

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
uint8_t myTxData [NUM_BYTES] = {'1', '0', ' ', 'B', 'Y', 'T', 'E', 'S', '!', '!'};
uint8_t myRxData [NUM_BYTES] = {0};

if(!TWIHSx_WriteRead( SLAVE_ADDR, &myTxData[0], NUM_BYTES, myRxData, NUM_BYTES ))
{
    // error handling
}
```

Parameters

Parameters	Description
address	7-bit / 10-bit slave address.
wdata	pointer to write data buffer
wlength	write data length in bytes.
rdata	pointer to read data buffer.
rlength	read data length in bytes.

Function

bool TWIHSx_WriteRead(uint16_t address, uint8_t *wdata, size_t wlength, uint8_t *rdata, size_t rlength)

d) Status Functions

TWIHSx_IsBusy Function

Returns the TWIHS Peripheral busy status.

File

plib_twihs.h

C

```
bool TWIHSx_IsBusy();
```

Returns

- True - Busy.
- False - Not busy.

Description

This function returns the TWIHS Peripheral busy status

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
if(!TWIHS1_IsBusy())
{
    // add another transfer request
}
```

Function

bool TWIHSx_IsBusy(void)

TWIHSx_ErrorGet Function

Returns the error during transfer.

File

plib_twihs.h

C

```
TWIHS_ERROR TWIHSx_ErrorGet();
```

Returns

- TWIHS_ERROR_NONE - Transfer was successful.
- TWIHS_ERROR_NACK - Slave returned NACK during the transfer.

Description

This function returns the error during transfer.

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the associated TWIHS instance.

Example

```
if(TWIHS_ERROR_NONE == TWIHS1_ErrorGet())
{
    //TWIHS transfer is completed, go to next state.
}
```

Function

TWIHS_ERROR TWIHSx_ErrorGet(void)

e) Data Types and Constants

TWIHS_CALLBACK Type

Defines the data type and function signature for the TWIHS peripheral callback function.

File

plib_twihs.h

C

```
typedef void (* TWIHS_CALLBACK)(uintptr_t contextHandle);
```

Returns

None.

Description

TWIHS Transfer Event Callback Function Pointer.

This data type defines the function signature for the TWIHS peripheral callback function. The TWIHS peripheral will call back the client's function with this signature when the TWIHS Transfer event has occurred.

Remarks

None.

Preconditions

TWIHSx_Initialize must have been called for the given TWIHS peripheral instance and TWIHSx_CallbackRegister must have been called to set the function to be called.

Example

```
// Data tracking what each of my instances is doing.
MY_DATA myData[2];

void MyTWIHSCallback ( uintptr_t context )
{
    MY_DATA *mePtr = (MY_DATA *)context;

    if(TWIHS_ERROR_NONE == TWIHS1_ErrorGet())
    {
        //Handle complete status here based on context
    }
    else
    {
        //Handle error here
    }
}
```

```
TWIHS1_CallbackSet(MyTWIHSCallback, &myData[0]);
TWIHS2_CallbackSet(MyTWIHSCallback, &myData[1]);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

TWIHS_ERROR Enumeration

TWIHS Error Status data type.

File

plib_twihs.h

C

```
typedef enum {
  TWIHS_ERROR_NONE,
  TWIHS_ERROR_NACK
} TWIHS_ERROR;
```

Members

Members	Description
TWIHS_ERROR_NONE	No Error
TWIHS_ERROR_NACK	Slave returned Nack

Description

TWIHS Error Status

This enum defines the list of TWIHS Error Status. Any one error from the list will be returned if the error status is queried.

Remarks

None.

Files

Files

Name	Description
plib_twihs.h	TWIHS peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_twihs.h

TWIHS peripheral library interface.

Enumerations

	Name	Description
	TWIHS_ERROR	TWIHS Error Status data type.

Functions

	Name	Description
	TWIHSx_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the given TWIHS's transfer events occur.
	TWIHSx_ErrorGet	Returns the error during transfer.
	TWIHSx_Initialize	Initializes given instance of the TWIHS peripheral.
	TWIHSx_IsBusy	Returns the TWIHS Peripheral busy status.
	TWIHSx_Read	Reads data from the slave.
	TWIHSx_Write	Writes data onto the slave.
	TWIHSx_WriteRead	Write and Read data from Slave.

Types

	Name	Description
	TWIHS_CALLBACK	Defines the data type and function signature for the TWIHS peripheral callback function.

Description

TWIHS Peripheral Library Interface Header File

This file defines the interface to the TWIHS peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual plib_twihsheaders will be generated as required by the MCC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "TWIHS" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

File Name

plib_twihs.h

Company

Microchip Technology Inc.

UART Peripheral Library Help

This section provides interface to use UART peripheral.

Introduction

This library provides brief overview of UART peripheral.

Description

The Universal Asynchronous Receiver Transceiver (UART) provides one full duplex universal asynchronous serial link.

- Data Bits: 8 data bits
- Stop bits: 1 stop bit
- Parity: Even, Odd, Mark or Space Parity Generation

Configuring the Library

This section describes how to configure the peripheral library using MHC

Description

The UART library supports byte and buffer model to transfer data. The buffer model can transfer a block of data in a blocking and non-blocking manner.

- To transfer block of data in non-blocking manner, the peripheral interrupt is used to manage the transfer and hence the "Interrupt Mode" option should be **enabled** in MHC
- To use byte model or transfer block of data in blocking manner, the CPU polls the peripheral register and hence the "Interrupt Mode" option should be **disabled** in MHC

Using the Library

This section describes how to use the Peripheral Library

Description

The UART library supports byte and buffer model to transfer data. The buffer model can process a block of data in a blocking or non-blocking manner.

Byte-by-Byte Transfer Model:

The Byte-by-Byte Model allows the application to transfer data through the UART one byte at a time. With this model, the driver reads one byte from the receive buffer or writes one byte to the transmit buffer. The application must check if data has been received before reading the data. Similarly, it must check if the transmit buffer is empty before writing to the transmit buffer. The byte-by-byte data transfer model places the responsibility of maintaining the UART peripheral on the Application.

The below example shows the byte-by-byte model usage.

```
 char rxData;
 char txData = 'A';

int main(void)
{
    if(UARTx_ReceiverIsReady() == true)
    {
        rxData = UARTx_ReadByte();
    }

    if(UARTx_TransmitterIsReady() == true)
    {
        UARTx_WriteByte(txData);
    }

}
```

Buffer Transfer Model:

The buffer model can process a block of data in a blocking and non-blocking manner.

- With blocking, the read/write APIs will block until all of the data is transferred. The CPU polls the peripheral register continuously to manage the transfer.
- With non-blocking, the read/write APIs will return immediately. The peripheral interrupt is used to manage the transfer. The user application can either use polling or callback method to indicate the transfer status.
  - With polling, the application will need to continuously poll to check if the transfer is completed.
  - With callback, the registered callback function will be called once the transfer is completed. This means we do not have to poll continuously.

Blocking Transfer

The below example shows the blocking transfer.

```
char myData[6] = {"hello"};

int main(void)
{
   /* Write 6 bytes of data */
   UARTx_Write(&myData, 6)
```

```
}
```

Non-blocking Transfer with callback

The below examples shows non-blocking transfer with callback.

```
char readBuffer[10];

void UARTx_Callback(uintptr_t context)
{
    if(UARTx_ErrorGet() != UART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

int main (void)
{
    /* Register Callback function for Read operation*/
    UARTx_ReadCallbackRegister(UARTx_Callback, (uintptr_t)NULL);

    /* Submit Read request to read 10 bytes of data */
    UARTx_Read(&readBuffer, 10);

}
```

Non-blocking Transfer with polling

The below examples shows non-blocking transfer with polling.

```
char readBuffer[10];

int main(void)
{

    UARTx_Read(&readBuffer, 10);

    /* Poll and wait for the transfer to complete */
    while(UARTx_ReadIsBusy());

    if(UARTx_ErrorGet() != UART_ERROR_NONE)
    {
        // Error occurred while receiving the buffer
    }
    else
    {
        //  Received the buffer without any error
    }
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	UARTx_Initialize	Initializes given instance of the UART peripheral.

b) Setup Functions

	Name	Description
	UARTx_SerialSetup	Sets the UART serial communication settings dynamically.

	UARTx_ReadCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given UART's read events occur.
	UARTx_WriteCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given UART's write events occur.

c) Transaction Functions

	Name	Description
	UARTx_Read	Submits request to read n-Bytes of data to the given UART peripheral.
	UARTx_Write	Submits a write buffer to the given UART peripheral to transfer.
	UARTx_ReadByte	Submits request to read a byte of data to the given UART peripheral.
	UARTx_WriteByte	Submits a byte of data to the given UART peripheral to transfer.

d) Status Functions

	Name	Description
	UARTx_ErrorGet	Gets the error of the given UART peripheral instance.
	UARTx_ReadCountGet	Returns the count of number of bytes processed for a given UART read operation.
	UARTx_WriteCountGet	Returns the count of number of bytes processed for a given UART write operation.
	UARTx_ReadIsBusy	Returns the read request status associated with the given UART peripheral instance.
	UARTx_WriteIsBusy	Returns the write request status associated with the given UART peripheral instance.
	UARTx_ReceiverIsReady	Returns the hardware status of the UART Receiver.
	UARTx_TransmitterIsReady	Returns the hardware status of the UART Transmitter.

e) Data Types and Constants

	Name	Description
	UART_CALLBACK	Defines the data type and function signature for the UART peripheral callback function.
	UART_ERROR	Defines the data type for the UART peripheral errors.
	UART_PARITY	Defines the type of parity for UART peripheral.
	UART_SERIAL_SETUP	Defines the data type for the UART serial setup.

a) Initialization Function

UARTx_Initialize Function

Initializes given instance of the UART peripheral.

File

plib_uart.h

C

```
void UARTx_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the UART peripheral as configured by the user in MHC.

Remarks

Stops the UART if it was already running and reinitializes it.

Preconditions

None.

Example

```
UART1_Initialize();
```

Function

void UARTx_Initialize(void)

b) Setup Functions

UARTx_SerialSetup Function

Sets the UART serial communication settings dynamically.

File

plib_uart.h

C

```
bool UARTx_SerialSetup(UART_SERIAL_SETUP * setup, uint32_t srcClkFreq);
```

Returns

- true - Serial setup was updated Successfully.
- false - Failure while updating serial setup.

Description

This function sets the UART serial communication settings dynamically.

Remarks

srcClkFreq overrides any change in the peripheral clock frequency. If configured to zero, PLIB takes the peripheral clock frequency from MHC.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

The UART transmit or receive transfer status should not be busy.

Example

```
UART_SERIAL_SETUP setup = {
        115200,
        UART_PARITY_ODD
    };

UART1_SerialSetup(&setup, 0);
```

Parameters

Parameters	Description
setup	Pointer to the structure containing the serial setup.
srcClkFreq	UART Peripheral Clock Source Frequency.

Function

void UARTx_SerialSetup(UART_SERIAL_SETUP * setup, uint32_t srcClkFreq)

UARTx_ReadCallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the given UART's read events occur.

File

plib_uart.h

C

```
bool UARTx_ReadCallbackRegister(UART_CALLBACK callback, uintptr_t context);
```

Returns

- True - Callback registration was successful
- False - Callback registration failed

Description

This function sets the pointer to a client function to be called "back" when the given UART's read events occur. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

This function is available only in interrupt mode of operation.

Remarks

The context parameter is ignored if the pointer passed is NULL. To disable the callback function, pass a NULL for the callback parameter. See the UART_CALLBACK type definition for additional information.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
void MyUartCallback(uintptr_t context)
{
    if(UART1_ErrorGet() != UART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

UART1_ReadCallbackRegister(MyUartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the UART_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

bool UARTx_ReadCallbackRegister(UART_CALLBACK callback, uintptr_t context)

UARTx_WriteCallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the given UART's write events occur.

File

plib_uart.h

C

```
bool UARTx_WriteCallbackRegister(UART_CALLBACK callback, uintptr_t context);
```

Returns

- True - Callback registration was successful
- False - Callback registration failed

Description

This function sets the pointer to a client function to be called "back" when the given UART's write events occur. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

This function is available only in interrupt mode of operation.

Remarks

The context parameter is ignored if the pointer passed is NULL. To disable the callback function, pass a NULL for the callback parameter. See the UART_CALLBACK type definition for additional information.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
void MyUartCallback(uintptr_t context)
{
    if(UART1_ErrorGet() != UART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

UART1_WriteCallbackRegister(MyUartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the UART_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

bool UARTx_WriteCallbackRegister(UART_CALLBACK callback, uintptr_t context)

c) Transaction Functions

UARTx_Read Function

Submits request to read n-Bytes of data to the given UART peripheral.

File

plib_uart.h

C

```
bool UARTx_Read(void * buffer, const size_t size);
```

Returns

- True - Read request was successful.
- False - Read request has failed.

Description

This function submits a request to read n-Bytes of data to the UART peripheral. The behavior of this function call will vary based on the mode selected within MHC.

If the peripheral is configured for the interrupt mode, this function call is always non-blocking. Call to this function submits the buffer and the size to the peripheral library and returns immediately. User can check the transfer status either through callback mechanism or by calling UARTx_ReadIsBusy.

If the peripheral is configured for the non-interrupt mode, this function call returns only after requested size is processed.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
//Example to use in non-interrupt

#define COUNT  6
char rxData[COUNT] = {}; //COUNT is expected size

if(UART1_Read(&rxData, COUNT) == true)
{
    //The transfer is completed
}
```

Parameters

Parameters	Description
buffer	Pointer to the user buffer.
size	Number of bytes to be read.

Function

bool UARTx_Read(void *buffer, const size_t size)

UARTx_Write Function

Submits a write buffer to the given UART peripheral to transfer.

File

plib_uart.h

C

```
bool UARTx_Write(void * buffer, const size_t size);
```

Returns

- True - Write request was successful.
- False - Write request has failed.

Description

This function submits a write buffer to the UART peripheral to transfer. The behavior of this function call will vary based on the

mode selected within MHC.

If the peripheral is configured for the interrupt mode, this function call is always non-blocking. Call to this function submits the buffer and the size to the peripheral library and returns immediately. User can check the transfer status either through callback mechanism or by calling UARTx_WriteIsBusy.

If the peripheral is configured for the non-interrupt mode, this function call returns only after requested size is transferred.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
//Example to use in non-interrupt mode
char myData[6] = {"hello"};

if(UART1_Write(&myData, 6) == true)
{
    //The transfer is completed
}
```

Parameters

Parameters	Description
buffer	Pointer to the user buffer.
size	Number of bytes to be transferred.

Function

bool UARTx_Write(void *buffer, const size_t size)

UARTx_ReadByte Function

Submits request to read a byte of data to the given UART peripheral.

File

plib_uart.h

C

```
int UARTx_ReadByte();
```

Returns

Read byte.

Description

This function submits request to read a byte of data to the given UART peripheral. This Function is available only in non-interrupt mode.

Remarks

None.

Preconditions

- UARTx_Initialize must have been called for the associated UART instance.
- Receiver readiness must be confirmed using UARTx_ReceiverIsReady.

Example

```
char rxData;

if(UART1_ReceiverIsReady() == true)
{
```

```
    rxData = UART1_ReadByte();
}
```

Function

int UARTx_ReadByte(void)

UARTx_WriteByte Function

Submits a byte of data to the given UART peripheral to transfer.

File

plib_uart.h

C

```
void UARTx_WriteByte(int data);
```

Returns

None

Description

This function submits a byte of data to the UART peripheral to transfer. This Function is available only in non-interrupt mode.

Remarks

None.

Preconditions

- UARTx_Initialize must have been called for the associated UART instance.
- Transmitter readiness must be confirmed using UARTx_TransmitterIsReady.

Example

```
UART1_WriteByte(0xAA);
```

Parameters

Parameters	Description
data	Data byte to be transferred.

Function

void UARTx_WriteByte(int data)

d) Status Functions

UARTx_ErrorGet Function

Gets the error of the given UART peripheral instance.

File

plib_uart.h

C

```
UART_ERROR UARTx_ErrorGet();
```

Returns

Errors occurred as listed by UART_ERROR.

Description

This function returns the errors associated with the given UART peripheral instance. It reports all the UART errors ORed together. The call to this function also clears all the associated errors.

Remarks

UART errors are normally associated with the receiver. Hence, it is recommended to use this function in receive context only.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
if ((UART_ERROR_OVERRUN & UART1_ErrorGet()) == UART_ERROR_OVERRUN)
{
    //Errors are cleared by the API call, take respective action
    //for the overrun error case.
}
```

Function

UART_ERROR UARTx_ErrorGet(void)

UARTx_ReadCountGet Function

Returns the count of number of bytes processed for a given UART read operation.

File

plib_uart.h

C

```
size_t UARTx_ReadCountGet();
```

Returns

Count of bytes processed for the last read request.

In case of error, the API returns the count of successful read bytes.

Description

This function returns the count of number of bytes processed for the last UART read operation.

This function is available only in interrupt mode of operation.

Remarks

This function resets the read count every time a new transfer is submitted.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
size_t count; //COUNT_EXPECTED is the expected Receive count value

count = UART1_ReadCountGet();

if(COUNT_EXPECTED > count)
{
    //More bytes are expected, wait
}
```

Function

size_t UARTx_ReadCountGet(void)

UARTx_WriteCountGet Function

Returns the count of number of bytes processed for a given UART write operation.

File

plib_uart.h

C

```
size_t UARTx_WriteCountGet();
```

Returns

Count of bytes completed/processed for the last write request.

Description

This function returns the count of number of bytes processed for the last UART write request.

This function is available only in interrupt mode of operation.

Remarks

This function resets the write count every time a new transfer is submitted.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
size_t count; //COUNT_EXPECTED is the expected Transmit count value

count = UART1_WriteCountGet();

if(COUNT_EXPECTED > count)
{
    //More bytes are expected to transmit, wait
}
```

Function

size_t UARTx_WriteCountGet(void)

UARTx_ReadIsBusy Function

Returns the read request status associated with the given UART peripheral instance.

File

plib_uart.h

C

```
bool UARTx_ReadIsBusy();
```

Returns

- true - UART is busy in processing the read request.
- false - UART is free and ready to accept the new read request.

Description

This function returns the status of the read request which was made using UART1_Read function. It can be used to poll the read status if callback is not intended.

This function is available only in interrupt mode of operation.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
if(true == UART1_ReadIsBusy())
{
    //UART is currently processing the previous read request.
}
else
{
    // read is completed
}
```

Function

bool UARTx_ReadIsBusy(void)

UARTx_WriteIsBusy Function

Returns the write request status associated with the given UART peripheral instance.

File

plib_uart.h

C

```
bool UARTx_WriteIsBusy();
```

Returns

- true - UART is busy in processing write request.
- false - UART is free and ready to accept the new write request.

Description

This function returns the status of the write request which was made using UART1_Write function. It can be used to poll the write status if callback is not intended.

This function is available only in interrupt mode of operation.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
if(true == UART1_WriteIsBusy())
{
    //UART is currently processing the write request.
}
else
{
    // write is completed.
}
```

Function

bool UARTx_WriteIsBusy(void)

UARTx_ReceiverIsReady Function

Returns the hardware status of the UART Receiver.

File

plib_uart.h

C

```
bool UARTx_ReceiverIsReady();
```

Returns

- true - Receive hardware FIFO has at least one data available to read.
- false - Receive hardware FIFO is empty.

Description

This function returns the hardware status associated with the Receive hardware FIFO of the given UART peripheral instance.

When Receiver is ready, user can read the available data.

This function is available only in non-interrupt mode of operation and it should be used to check receiver readiness before calling UART1_ReadByte function.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
char rxData;
if(true == UART1_ReceiverIsReady())
{
    // At least one data is available in Receive FIFO, hence can read a byte
    rxData = UART1_ReadByte();
}
else
{
    //Receiver is empty
}
```

Function

bool UARTx_ReceiverIsReady(void)

UARTx_TransmitterIsReady Function

Returns the hardware status of the UART Transmitter.

File

plib_uart.h

C

```
bool UARTx_TransmitterIsReady();
```

Returns

- true - Transmit hardware FIFO has empty space to accept the data.
- false - Transmit hardware FIFO is full.

Description

This function returns the hardware status associated with the Transmit hardware FIFO of the given UART peripheral instance.When Transmitter is ready, user can submit data to transmit.

This function is available only in non-interrupt mode of operation and it should be used to check transmitter readiness before calling UART1_WriteByte function.

Remarks

None.

Preconditions

UARTx_Initialize must have been called for the associated UART instance.

Example

```
if(true == UART1_TransmitterIsReady())
{
    // Empty space is available in Transmitter FIFO, hence can write a byte
    UART1_WriteByte('A');
}
else
{
    //Transmitter is busy
}
```

Function

bool UARTx_TransmitterIsReady(void)

e) Data Types and Constants

UART_CALLBACK Type

Defines the data type and function signature for the UART peripheral callback function.

File

plib_uart.h

C

```
typedef void (* UART_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the UART peripheral callback function. The UART peripheral will call back the client's function with this signature when the UART buffer event has occurred.

Remarks

None.

Preconditions

- UARTx_Initialize must have been called for the given UART peripheral instance.
- Callback must have been registered using UARTx_WriteCallbackRegister or UARTx_ReadCallbackRegister.

Example

```
void MyUartCallback(uintptr_t context)
```

```
{
    if(UART1_ErrorGet() != UART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

UART1_WriteCallbackRegister(MyUartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multiple clients).

UART_ERROR Enumeration

Defines the data type for the UART peripheral errors.

File

plib_uart.h

C

```
typedef enum {
  UART_ERROR_NONE,
  UART_ERROR_OVERRUN,
  UART_ERROR_PARITY,
  UART_ERROR_FRAMING
} UART_ERROR;
```

Description

UART Errors

This may be used to check the type of error occurred with the UART peripheral during error status.

Remarks

None.

UART_PARITY Enumeration

Defines the type of parity for UART peripheral.

File

plib_uart.h

C

```
typedef enum {
  UART_PARITY_NONE = 0,
  UART_PARITY_ODD = 1,
  UART_PARITY_EVEN = 2,
  UART_PARITY_MARK = 3,
  UART_PARITY_SPACE = 4
} UART_PARITY;
```

Description

UART Parity

This defines the type of parity for UART peripheral.

Remarks

None.

UART_SERIAL_SETUP Structure

Defines the data type for the UART serial setup.

File

plib_uart.h

C

```
typedef struct {
  uint32_t baudRate;
  UART_PARITY parity;
} UART_SERIAL_SETUP;
```

Description

UART Serial Setup

This can be used to define a serial setup which may then be used to change the serial setup of the UART dynamically.

Remarks

None.

Files

Files

Name	Description
plib_uart.h	UART peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_uart.h

UART peripheral library interface.

Enumerations

	Name	Description
	UART_ERROR	Defines the data type for the UART peripheral errors.
	UART_PARITY	Defines the type of parity for UART peripheral.

Functions

	Name	Description
	UARTx_ErrorGet	Gets the error of the given UART peripheral instance.
	UARTx_Initialize	Initializes given instance of the UART peripheral.
	UARTx_Read	Submits request to read n-Bytes of data to the given UART peripheral.
	UARTx_ReadByte	Submits request to read a byte of data to the given UART peripheral.
	UARTx_ReadCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given UART's read events occur.
	UARTx_ReadCountGet	Returns the count of number of bytes processed for a given UART read operation.

	UARTx_ReadIsBusy	Returns the read request status associated with the given UART peripheral instance.
	UARTx_ReceiverIsReady	Returns the hardware status of the UART Receiver.
	UARTx_SerialSetup	Sets the UART serial communication settings dynamically.
	UARTx_TransmitterIsReady	Returns the hardware status of the UART Transmitter.
	UARTx_Write	Submits a write buffer to the given UART peripheral to transfer.
	UARTx_WriteByte	Submits a byte of data to the given UART peripheral to transfer.
	UARTx_WriteCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given UART's write events occur.
	UARTx_WriteCountGet	Returns the count of number of bytes processed for a given UART write operation.
	UARTx_WriteIsBusy	Returns the write request status associated with the given UART peripheral instance.

Structures

	Name	Description
	UART_SERIAL_SETUP	Defines the data type for the UART serial setup.

Types

	Name	Description
	UART_CALLBACK	Defines the data type and function signature for the UART peripheral callback function.

Description

UART Peripheral Library Interface Header File

This file defines the interface to the UART peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual plib_uartheaders will be generated as required by the MHC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "UART" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

File Name

plib_uart.h

Company

Microchip Technology Inc.

USART Peripheral Library Help

This section provides interface to use USART peripheral.

Introduction

This library provides brief overview of USART peripheral.

Description

The Universal Synchronous Asynchronous Receiver Transceiver (USART) provides one full duplex universal synchronous asynchronous serial link. Data frame format is widely programmable (data length, parity,number of stop bits) to support a maximum of standards.

- Data Bits: 5, 6, 7, 8 or 9 data bits
- Stop bits: 1, 1.5 or 2 stop bits

- Parity: Even, Odd, Mark or Space Parity Generation

Configuring the Library

This section describes how to configure the peripheral library using MHC

Description

The USART library supports byte and buffer model to transfer data. The buffer model can transfer a block of data in a blocking and non-blocking manner.

- To transfer block of data in non-blocking manner, the peripheral interrupt is used to manage the transfer and hence the "Interrupt Mode" option should be **enabled** in MHC
- To use byte model or transfer block of data in blocking manner, the CPU polls the peripheral register and hence the "Interrupt Mode" option should be **disabled** in MHC

Using the Library

This section describes how to use the Peripheral Library

Description

The USART library supports byte and buffer model to transfer data. The buffer model can process a block of data in a blocking or non-blocking manner.

Byte-by-Byte Transfer Model:

The Byte-by-Byte Model allows the application to transfer data through the USART one byte at a time. With this model, the driver reads one byte from the receive buffer or writes one byte to the transmit buffer. The application must check if data has been received before reading the data. Similarly, it must check if the transmit buffer is empty before writing to the transmit buffer. The byte-by-byte data transfer model places the responsibility of maintaining the USART peripheral on the Application.

The below example shows the byte-by-byte model usage.

```
 char rxData;
 char txData = 'A';

int main(void)
{
    if(USARTx_ReceiverIsReady() == true)
    {
        rxData = USARTx_ReadByte();
    }

    if(USARTx_TransmitterIsReady() == true)
    {
        USARTx_WriteByte(txData);
    }

}
```

Buffer Transfer Model:

The buffer model can process a block of data in a blocking and non-blocking manner.

- With blocking, the read/write APIs will block until all of the data is transferred. The CPU polls the peripheral register continuously to manage the transfer.
- With non-blocking, the read/write APIs will return immediately. The peripheral interrupt is used to manage the transfer. The user application can either use polling or callback method to indicate the transfer status.
  - With polling, the application will need to continuously poll to check if the transfer is completed.
  - With callback, the registered callback function will be called once the transfer is completed. This means we do not have to poll continuously.

Blocking Transfer

The below example shows the blocking transfer.

```
char myData[6] = {"hello"};

int main(void)
{
   /* Write 6 bytes of data */
   USARTx_Write(&myData, 6)

}
```

Non-blocking Transfer with callback

The below examples shows non-blocking transfer with callback.

```
char readBuffer[10];

void USARTx_Callback(uintptr_t context)
{
    if(USARTx_ErrorGet() != USART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

int main (void)
{
    /* Register Callback function for Read operation*/
    USARTx_ReadCallbackRegister(USARTx_Callback, (uintptr_t)NULL);

    /* Submit Read request to read 10 bytes of data */
    USARTx_Read(&readBuffer, 10);

}
```

Non-blocking Transfer with polling

The below examples shows non-blocking transfer with polling.

```
char readBuffer[10];

int main(void)
{

    USARTx_Read(&readBuffer, 10);

    /* Poll and wait for the transfer to complete */
    while(USARTx_ReadIsBusy());

    if(USARTx_ErrorGet() != USART_ERROR_NONE)
    {
        // Error occurred while receiving the buffer
    }
    else
    {
        //  Received the buffer without any error
    }
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a)Initialization Function

	Name	Description
	USARTx_Initialize	Initializes given instance of the USART peripheral.

b) Setup Functions

	Name	Description
	USARTx_ReadCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given USART's read events occur.
	USARTx_WriteCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given USART's write events occur.
	USARTx_SerialSetup	Sets the USART serial communication settings dynamically.

c) Transaction Functions

	Name	Description
	USARTx_Read	Submits a read buffer to the given USART peripheral to process.
	USARTx_Write	Submits a write buffer to the given USART peripheral to transfer.
	USARTx_ReadByte	Submits request to read a byte of data to the given USART peripheral.
	USARTx_WriteByte	Submits a byte of data to the given USART peripheral to transfer.

d) Status Functions

	Name	Description
	USARTx_ReadIsBusy	Returns the read request status associated with the given USART peripheral instance.
	USARTx_WriteIsBusy	Returns the write request status associated with the given USART peripheral instance.
	USARTx_TransmitterIsReady	Returns the hardware status of the USART Transmitter.
	USARTx_ReceiverIsReady	Returns the hardware status of the USART Receiver.
	USARTx_ErrorGet	Gets the error of the given USART peripheral instance.
	USARTx_ReadCountGet	Returns the count of number of bytes processed for a given USART read operation.
	USARTx_WriteCountGet	Returns the count of number of bytes processed for a given USART write operation.

e) Data Types and Constants

	Name	Description
	USART_CALLBACK	Defines the data type and function signature for the USART peripheral callback function.
	USART_SERIAL_SETUP	Defines the data type for the USART serial setup.
	USART_DATA_BIT	Defines the data type for the USART data bit length.
	USART_ERROR	Defines the data type for the USART peripheral errors.
	USART_PARITY	Defines the type of parity for USART peripheral.
	USART_STOP_BIT	Defines the length of stop bits.

a)Initialization Function

USARTx_Initialize Function

Initializes given instance of the USART peripheral.

File

plib_usart.h

C

```
void USARTx_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the USART peripheral as configured by the user in MHC.

Remarks

Stops the USART if it was already running and reinitializes it.

Preconditions

None.

Example

```
USART1_Initialize();
```

Function

void USARTx_Initialize(void)

b) Setup Functions

USARTx_ReadCallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the given USART's read events occur.

File

plib_usart.h

C

```
bool USARTx_ReadCallbackRegister(USART_CALLBACK callback, uintptr_t context);
```

Returns

- True - Callback registration was successful
- False - Callback registration failed

Description

This function sets the pointer to a client function to be called "back" when the given USART's read events occur. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

This function is available only in interrupt mode of operation.

Remarks

The context parameter is ignored if the pointer passed is NULL. To disable the callback function, pass a NULL for the callback parameter. See the USART_CALLBACK type definition for additional information.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
void MyUsartCallback(uintptr_t context)
{
    if(USART1_ErrorGet() != USART_ERROR_NONE)
    {
```

```
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

USART1_ReadCallbackRegister(MyUsartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the USART_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

bool USARTx_ReadCallbackRegister(USART_CALLBACK callback, uintptr_t context)

USARTx_WriteCallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the given USART's write events occur.

File

plib_usart.h

C

```
bool USARTx_WriteCallbackRegister(USART_CALLBACK callback, uintptr_t context);
```

Returns

- True - Callback registration was successful
- False - Callback registration failed

Description

This function sets the pointer to a client function to be called "back" when the given USART's write events occur. It also passes a context value (usually a pointer to a context structure) that is passed into the function when it is called.

This function is available only in interrupt mode of operation.

Remarks

The context parameter is ignored if the pointer passed is NULL. To disable the callback function, pass a NULL for the callback parameter. See the USART_CALLBACK type definition for additional information.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
void MyUsartCallback(uintptr_t context)
{
    if(USART1_ErrorGet() != USART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}
```

```
USART1_WriteCallbackRegister(MyUsartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the USART_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

bool USARTx_WriteCallbackRegister(USART_CALLBACK callback, uintptr_t context)

USARTx_SerialSetup Function

Sets the USART serial communication settings dynamically.

File

plib_usart.h

C

```
bool USARTx_SerialSetup(USART_SERIAL_SETUP * setup, uint32_t srcClkFreq);
```

Returns

- true - Serial setup was updated Successfully.
- false - Failure while updating serial setup.

Description

This function sets the USART serial communication settings dynamically.

Remarks

srcClkFreq overrides any change in the peripheral clock frequency. If configured to zero, PLIB takes the peripheral clock frequency from MHC.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

The USART transmit or receive transfer status should not be busy.

Example

```
USART_SERIAL_SETUP setup = {
        115200,
        USART_DATA_8_BIT,
        USART_PARITY_ODD,
        USART_STOP_1_BIT
    };

USART1_SerialSetup(&setup, 0);
```

Parameters

Parameters	Description
setup	Pointer to the structure containing the serial setup.
srcClkFreq	USART Peripheral Clock Source Frequency.

Function

void USARTx_SerialSetup(USART_SERIAL_SETUP * setup, uint32_t srcClkFreq)

c) Transaction Functions

USARTx_Read Function

Submits a read buffer to the given USART peripheral to process.

File

plib_usart.h

C

```
bool USARTx_Read(void * buffer, const size_t size);
```

Returns

- True - Read request was successful.
- False - Read request has failed.

Description

This function submits a request to read n-Bytes of data to the UART peripheral. The behavior of this function call will vary based on the mode selected within MHC.

If the peripheral is configured for the interrupt mode, this function call is always non-blocking. Call to this function submits the buffer and the size to the peripheral library and returns immediately. User can check the transfer status either through callback mechanism or by calling USARTx_ReadIsBusy.

If the peripheral is configured for the non-interrupt mode, this function call returns only after requested size is processed.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
//Example to use in non-interrupt

#define COUNT  6
char rxData[COUNT] = {}; //COUNT is expected size

if(USART1_Read(&rxData, COUNT) == true)
{
    //The transfer is completed
}
```

Parameters

Parameters	Description
buffer	Pointer to the user buffer.
size	Number of bytes to be transferred.

Function

bool USARTx_Read(void *buffer, const size_t size)

USARTx_Write Function

Submits a write buffer to the given USART peripheral to transfer.

File

plib_usart.h

C

```
bool USARTx_Write(void * buffer, const size_t size);
```

Returns

- True - Write request was successful.
- False - Write request has failed.

Description

This function submits a write buffer to the USART peripheral to transfer. The behavior of this function call will vary based on the mode selected within MHC.

If the peripheral is configured for the interrupt mode, this function call is always non-blocking. Call to this function submits the buffer and the size to the peripheral library and returns immediately. User can check the transfer status either through callback mechanism or by calling USARTx_WriteIsBusy.

If the peripheral is configured for the non-interrupt mode, this function call returns only after requested size is transferred.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
//Example to use in non-interrupt mode
char myData[6] = {"hello"};

if(USART1_Write(&myData, 6) == true)
{
    //The transfer is completed
}
```

Parameters

Parameters	Description
buffer	Pointer to the user buffer.
size	Number of bytes to be transferred.

Function

bool USARTx_Write(void *buffer, const size_t size)

USARTx_ReadByte Function

Submits request to read a byte of data to the given USART peripheral.

File

plib_usart.h

C

```
int USARTx_ReadByte();
```

Returns

Read byte.

Description

This function submits request to read a byte of data to the given USART peripheral. This Function is available only in non-interrupt

mode.

Remarks

None.

Preconditions

- USARTx_Initialize must have been called for the associated USART instance.
- Receiver readiness must be confirmed using USARTx_ReceiverIsReady.

Example

```
char rxData;

if(USART1_ReceiverIsReady() == true)
{
    rxData = USART1_ReadByte();
}
```

Function

int USARTx_ReadByte(void)

USARTx_WriteByte Function

Submits a byte of data to the given USART peripheral to transfer.

File

plib_usart.h

C

```
void USARTx_WriteByte(int data);
```

Returns

None

Description

This function submits a byte of data to the USART peripheral to transfer. This Function is available only in non-interrupt mode.

Remarks

None.

Preconditions

- USARTx_Initialize must have been called for the associated USART instance.
- Transmitter readiness must be confirmed using USARTx_TransmitterIsReady.

Example

```
USART1_WriteByte(0xAA);
```

Parameters

Parameters	Description
data	Data byte to be transferred.

Function

void USARTx_WriteByte(int data)

d) Status Functions

USARTx_ReadIsBusy Function

Returns the read request status associated with the given USART peripheral instance.

File

plib_usart.h

C

```
bool USARTx_ReadIsBusy();
```

Returns

- true - USART is busy in processing the read request.
- false - USART is free and ready to accept the new read request.

Description

This function returns the status of the read request which was made using USART1_Read function. It can be used to poll the read status if callback is not intended.

This function is available only in interrupt mode of operation.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
if(true == USART1_ReadIsBusy())
{
    //USART is currently processing the previous read request.
}
else
{
    // read is completed
}
```

Function

bool USARTx_ReadIsBusy(void)

USARTx_WriteIsBusy Function

Returns the write request status associated with the given USART peripheral instance.

File

plib_usart.h

C

```
bool USARTx_WriteIsBusy();
```

Returns

- true - USART is busy in processing write request.
- false - USART is free and ready to accept the new write request.

Description

This function returns the status of the write request which was made using USART1_Write function. It can be used to poll the write status if callback is not intended.

This function is available only in interrupt mode of operation.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
if(true == USART1_WriteIsBusy())
{
    //USART is currently processing the write request.
}
else
{
    // write is completed.
}
```

Function

bool USARTx_WriteIsBusy(void)

USARTx_TransmitterIsReady Function

Returns the hardware status of the USART Transmitter.

File

plib_usart.h

C

```
bool USARTx_TransmitterIsReady();
```

Returns

- true - Transmit hardware FIFO has empty space to accept the data.
- false - Transmit hardware FIFO is full.

Description

This function returns the hardware status associated with the Transmit hardware FIFO of the given USART peripheral instance.When Transmitter is ready, user can submit data to transmit.

This function is available only in non-interrupt mode of operation and it should be used to check transmitter readiness before calling USART1_WriteByte function.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
if(true == USART1_TransmitterIsReady())
{
    // Empty space is available in Transmitter FIFO, hence can write a byte
    USART1_WriteByte('A');
}
else
{
    //Transmitter is busy
}
```

Function

bool USARTx_TransmitterIsReady(void)

USARTx_ReceiverIsReady Function

Returns the hardware status of the USART Receiver.

File

plib_usart.h

C

```
bool USARTx_ReceiverIsReady();
```

Returns

- true - Receive hardware FIFO has at least one data available to read.
- false - Receive hardware FIFO is empty.

Description

This function returns the hardware status associated with the Receive hardware FIFO of the given USART peripheral instance.

When Receiver is ready, user can read the available data.

This function is available only in non-interrupt mode of operation and it should be used to check receiver readiness before calling USART1_ReadByte function.

Remarks

None.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
char rxData;
if(true == USART1_ReceiverIsReady())
{
    // At least one data is available in Receive FIFO, hence can read a byte
    rxData = USART1_ReadByte();
}
else
{
    //Receiver is empty
}
```

Function

bool USARTx_ReceiverIsReady(void)

USARTx_ErrorGet Function

Gets the error of the given USART peripheral instance.

File

plib_usart.h

C

```
USART_ERROR USARTx_ErrorGet();
```

Returns

Errors occurred as listed by USART_ERROR.

Description

This function returns the errors associated with the given USART peripheral instance. It reports all the USART errors ORed

together. The call to this function also clears all the associated errors.

Remarks

USART errors are normally associated with the receiver. Hence, it is recommended to use this function in receive context only.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
if ((USART_ERROR_OVERRUN & USART1_ErrorGet()) == USART_ERROR_OVERRUN)
{
    //Errors are cleared by the API call, take respective action
    //for the overrun error case.
}
```

Function

USART_ERROR USARTx_ErrorGet(void)

USARTx_ReadCountGet Function

Returns the count of number of bytes processed for a given USART read operation.

File

plib_usart.h

C

```
size_t USARTx_ReadCountGet();
```

Returns

Count of bytes processed for the last read request.

In case of error, the API returns the count of successful read bytes.

Description

This function returns the count of number of bytes processed for the last USART read operation.

This function is available only in interrupt mode of operation.

Remarks

This function resets the read count every time a new transfer is submitted.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
size_t count; //COUNT_EXPECTED is the expected Receive count value

count = USART1_ReadCountGet();

if(COUNT_EXPECTED > count)
{
    //More bytes are expected, wait
}
```

Function

size_t USARTx_ReadCountGet(void)

USARTx_WriteCountGet Function

Returns the count of number of bytes processed for a given USART write operation.

File

plib_usart.h

C

```
size_t USARTx_WriteCountGet();
```

Returns

Count of bytes completed/processed for the last write request.

Description

This function returns the count of number of bytes processed for the last USART write request.

This function is available only in interrupt mode of operation.

Remarks

This function resets the write count every time a new transfer is submitted.

Preconditions

USARTx_Initialize must have been called for the associated USART instance.

Example

```
size_t count; //COUNT_EXPECTED is the expected Transmit count value

count = USART1_WriteCountGet();

if(COUNT_EXPECTED > count)
{
    //More bytes are expected to transmit, wait
}
```

Function

size_t USARTx_WriteCountGet(void)

e) Data Types and Constants

USART_CALLBACK Type

Defines the data type and function signature for the USART peripheral callback function.

File

plib_usart.h

C

```
typedef void (* USART_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the USART peripheral callback function. The USART peripheral will call back the client's function with this signature when the USART buffer event has occurred.

Remarks

None.

Preconditions

- USARTx_Initialize must have been called for the given USART peripheral instance.
- Callback must have been registered using USARTx_WriteCallbackRegister or USARTx_ReadCallbackRegister.

Example

```
void MyUsartCallback(uintptr_t context)
{
    if(USART1_ErrorGet() != USART_ERROR_NONE)
    {
        //Handle error case
    }
    else
    {
        //Transfer completed successfully
    }
}

USART1_WriteCallbackRegister(MyUsartCallback, (uintptr_t)NULL);
```

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multiple clients).

USART_SERIAL_SETUP Structure

Defines the data type for the USART serial setup.

File

plib_usart.h

C

```
typedef struct {
  uint32_t baudRate;
  USART_DATA_BIT dataWidth;
  USART_PARITY parity;
  USART_STOP_BIT stopBits;
} USART_SERIAL_SETUP;
```

Description

USART Serial Setup

This can be used to define a serial setup which may then be used to change the serial setup of the USART dynamically.

Remarks

None.

USART_DATA_BIT Enumeration

Defines the data type for the USART data bit length.

File

plib_usart.h

C

```
typedef enum {
  USART_DATA_BIT_5,
  USART_DATA_BIT_6,
  USART_DATA_BIT_7,
  USART_DATA_BIT_8,
  USART_DATA_BIT_9
} USART_DATA_BIT;
```

Description

USART Data Bits

This is used to define length of data bits.

Remarks

None.

USART_ERROR Enumeration

Defines the data type for the USART peripheral errors.

File

plib_usart.h

C

```
typedef enum {
  USART_ERROR_NONE,
  USART_ERROR_OVERRUN,
  USART_ERROR_PARITY,
  USART_ERROR_FRAMING
} USART_ERROR;
```

Description

USART Errors

This may be used to check the type of error occurred with the USART peripheral during error status.

Remarks

None.

USART_PARITY Enumeration

Defines the type of parity for USART peripheral.

File

plib_usart.h

C

```
typedef enum {
  USART_PARITY_NONE,
  USART_PARITY_ODD,
  USART_PARITY_EVEN,
  USART_PARITY_MARK,
  USART_PARITY_SPACE,
  USART_PARITY_MULTIDROP
} USART_PARITY;
```

Description

USART Parity

This defines the type of parity for USART peripheral.

Remarks

None.

USART_STOP_BIT Enumeration

Defines the length of stop bits.

File

plib_usart.h

C

```
typedef enum {
  USART_STOP_BIT_1,
  USART_STOP_BIT_1_5,
  USART_STOP_BIT_2
} USART_STOP_BIT;
```

Description

USART Stop Bits

This data type defines the length of stop bits for USART peripheral.

Remarks

None.

Files

Files

Name	Description
plib_usart.h	USART peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_usart.h

USART peripheral library interface.

Enumerations

	Name	Description
	USART_DATA_BIT	Defines the data type for the USART data bit length.
	USART_ERROR	Defines the data type for the USART peripheral errors.
	USART_PARITY	Defines the type of parity for USART peripheral.
	USART_STOP_BIT	Defines the length of stop bits.

Functions

	Name	Description
	USARTx_ErrorGet	Gets the error of the given USART peripheral instance.

	USARTx_Initialize	Initializes given instance of the USART peripheral.
	USARTx_Read	Submits a read buffer to the given USART peripheral to process.
	USARTx_ReadByte	Submits request to read a byte of data to the given USART peripheral.
	USARTx_ReadCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given USART's read events occur.
	USARTx_ReadCountGet	Returns the count of number of bytes processed for a given USART read operation.
	USARTx_ReadIsBusy	Returns the read request status associated with the given USART peripheral instance.
	USARTx_ReceiverIsReady	Returns the hardware status of the USART Receiver.
	USARTx_SerialSetup	Sets the USART serial communication settings dynamically.
	USARTx_TransmitterIsReady	Returns the hardware status of the USART Transmitter.
	USARTx_Write	Submits a write buffer to the given USART peripheral to transfer.
	USARTx_WriteByte	Submits a byte of data to the given USART peripheral to transfer.
	USARTx_WriteCallbackRegister	Sets the pointer to the function (and it's context) to be called when the given USART's write events occur.
	USARTx_WriteCountGet	Returns the count of number of bytes processed for a given USART write operation.
	USARTx_WriteIsBusy	Returns the write request status associated with the given USART peripheral instance.

Structures

	Name	Description
	USART_SERIAL_SETUP	Defines the data type for the USART serial setup.

Types

	Name	Description
	USART_CALLBACK	Defines the data type and function signature for the USART peripheral callback function.

Description

USART Peripheral Library Interface Header File

This file defines the interface to the USART peripheral library. This library provides access to and control of the associated peripheral instance.

Remarks

This header is for documentation only. The actual plib_usartheaders will be generated as required by the MHC (where is the peripheral instance number).

Every interface symbol has a lower-case 'x' in it following the "USART" abbreviation. This 'x' will be replaced by the peripheral instance number in the generated headers. These are the actual functions that should be used.

File Name

plib_usart.h

Company

Microchip Technology Inc.

WDT Peripheral Library Help

This section provides interface to use Watchdog Timer (WDT) peripheral.

Introduction

This section provides brief overview of Watchdog Timer (WDT)

Description

The Watchdog timer (WDT) is designed to give an added level of safety in critical systems, to ensure a system reset is triggered in the case of a deadlock or other software malfunction that prevents normal device operation.

The Watchdog timer is a 12-bit down counter that allows watchdog period of up to 16 seconds with 32kHz slow clock. It should be fed periodically before it counts down to zero. If the timer counts down to zero, it can either reset the device or generate an interrupt.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This Section explains how to use the Peripheral Library.

Description

The Watchdog Timer (WDT) runs independent of the rest of the system, causing system resets whenever it times out. However, the application software should ensure that the timeout never occurs by resetting the WDT periodically as long as the software is in a known healthy state. If the system hangs or program execution is corrupted, the WDT will not receive its periodic reset, and will eventually time out and cause a system reset.

In many systems, there is an obvious upper bound to the amount of time each iteration of the main application loop can be expected to run, before a malfunction can be assumed (either due to a deadlock waiting on hardware or software, or due to other means). When the Watchdog is configured with a timeout period equal to this upper bound, a malfunction in the system will force a full system reset to allow for a graceful recovery. The upper bound is specified by 12-bit Watch dog period value.

Window Mode

Just as there is a reasonable upper bound to the time the main program loop should take for each iteration, there is also in many applications a lower bound, i.e. a minimum time for which each loop iteration should run for under normal circumstances. To guard against a system failure resetting the Watchdog in a tight loop (or a failure in the system application causing the main loop to run faster than expected) a watch dog delta value can be specified to disallow resetting of the Watchdog counter before a certain period of time. Any attempt to reset the watchdog when the watchdog counter has not counted down to delta value results in device reset.

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	WDT_Initialize	Initializes given instance of the WDT peripheral.

b) Setup Functions

	Name	Description
	WDT_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

c) Transaction Functions

	Name	Description
	WDT_Clear	Restart the WDT counter.

d) Data Types and Constants

	Name	Description
	WDT_CALLBACK	Defines the data type and function signature for the WDT peripheral callback function.

a) Initialization Function

WDT_Initialize Function

Initializes given instance of the WDT peripheral.

File

plib_wdt.h

C

```
void WDT_Initialize();
```

Returns

None.

Description

This function initializes the given instance of the WDT peripheral as configured by the user.

Preconditions

None.

Example

```
WDT_Initialize();
```

Function

void WDT_Initialize(void)

b) Setup Functions

WDT_CallbackRegister Function

Sets the pointer to the function (and it's context) to be called when the Timeout events occur.

File

plib_wdt.h

C

```
void WDT_CallbackRegister(WDT_CALLBACK callback, uintptr_t context);
```

Returns

None.

Description

This function sets the pointer to a client function to be called "back" when the Timeout event occurs. It also passes a context value that is passed into the function when it is called. This function is available only in interrupt mode of operation.

Remarks

The context value may be set to NULL if it is not required. Note that the value of NULL will still be passed to the callback function. To disable the callback function, pass a NULL for the callback parameter. See the WDT_CALLBACK type definition for additional information. This feature may or may not be available based on device.

Preconditions

WDT_Initialize must have been called for the associated WDT instance. Reset mode must be disabled in MCC configuration

Example

```
WDT_CallbackRegister(MyCallback, &myData);
```

Parameters

Parameters	Description
callback	A pointer to a function with a calling signature defined by the WDT_CALLBACK data type.
context	A value (usually a pointer) passed (unused) into the function identified by the callback parameter.

Function

void WDT_CallbackRegister(WDT_CALLBACK callback, uintptr_t context)

c) Transaction Functions

WDT_Clear Function

Restart the WDT counter.

File

plib_wdt.h

C

```
void WDT_Clear();
```

Returns

None.

Description

This function is used to restart the WDT counter to avoid timeout reset/interrupt.

Preconditions

None

Example

```
//Application loop
while(1)
{
    WDT_Clear();
    //user code
}
```

Function

void WDT_Clear(void)

d) Data Types and Constants

WDT_CALLBACK Type

Defines the data type and function signature for the WDT peripheral callback function.

File

plib_wdt.h

C

```
typedef void (* WDT_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

Callback Function Pointer

This data type defines the function signature for the WDT peripheral callback function. The WDT peripheral will call back the client's function with this signature when the Timeout event has occurred. This feature may or may not be available based on device.

Preconditions

WDT_Initialize must have been called for the given WDT peripheral instance and WDT_CallbackRegister must have been called to set the function to be called.

Parameters

Parameters	Description
context	Allows the caller to provide a context value (usually a pointer to the callers context for multi-instance clients).

Files

Files

Name	Description
plib_wdt.h	Interface definition of the Watch Dog Timer Plib (WDT).

Description

This section will list only the library's interface header file(s).

plib_wdt.h

Interface definition of the Watch Dog Timer Plib (WDT).

Functions

	Name	Description
	WDT_CallbackRegister	Sets the pointer to the function (and it's context) to be called when the Timeout events occur.
	WDT_Clear	Restart the WDT counter.
	WDT_Initialize	Initializes given instance of the WDT peripheral.

Types

	Name	Description
	WDT_CALLBACK	Defines the data type and function signature for the WDT peripheral callback function.

Description

Interface definition of WDT PLIB.

This file defines the interface for the WDT Plib. It allows user to setup timeout duration and restart watch dog timer.

File Name

plib_wdt.h

Company

Microchip Technology Inc.

XDMAC Peripheral Library Help

This section provides interface to use DMA Controller (XDMAC) peripheral.

Introduction

This section provides brief overview of DMA controller (DMAC) peripheral.

Description

The DMA Controller (XDMAC) can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. It enables high data transfer rates with minimum CPU intervention, and frees up CPU time. With access to all peripherals, the DMAC can handle automatic transfer of data between communication modules. XDMAC has several DMA channels and each channel is fully programmable and provides both peripheral or memory-to-memory transfers.

Configuring the Library

This section explains how to configure the peripheral library using MHC

Description

Use MHC to configure the library.

Using the Library

This section explains the operation and typical steps to use the peripheral library

Description

DMA controller requires source address, destination address and transfer count to initiate transfer. It transfers data on each DMA trigger and decrements the transfer count. The DMA generates completion interrupt to notify CPU at the end of the DMA transfer. .

User can use either callback mechanism or polling mechanism to determine DMA completion.

- With polling, the application will need to continuously poll to check if the DMA has completed the transfer.
- With callback, the registered callback function will be called when the transfer is completed. This means we do not have to poll continuously.

Callback method

This example demonstrates DMA transfer with callback

```
#define USART1_TRANSMIT_ADDRESS      (&USART1_REGS->US_THR)

uint8_t txBuffer[] = “Hello World!”;

void XDMAC_Callback(XDMAC_TRANSFER_EVENT status, uintptr_t context)
{
      if(status == XDMAC_TRANSFER_COMPLETE)
      {
           // Transfer is completed.
      }
}

 int main ( void )
{
      SYS_Initialize ( NULL );

      XDMAC_ChannelCallbackRegister(XDMAC_CHANNEL_0, XDMAC_Callback, 0);

      XDMAC_ChannelTransfer(XDMAC_CHANNEL_0, txBuffer, (const void *)USART1_TRANSMIT_ADDRESS,
sizeof(txBuffer));

}
```

Polling method

This example demonstrates DMA transfer with polling

```
#define USART1_TRANSMIT_ADDRESS      (&USART1_REGS->US_THR)

uint8_t txBuffer[] = “Hello World!”;

int main ( void )
{
      SYS_Initialize ( NULL );

      XDMAC_ChannelTransfer(XDMAC_CHANNEL_0, txBuffer, (const void *)USART1_TRANSMIT_ADDRESS,
sizeof(txBuffer));

      while(XDMAC_ChannelIsBusy(XDMAC_CHANNEL_0));
}
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	XDMAC_Initialize	Initializes the XDMAC controller of the device.

b) Setup Functions

	Name	Description
	XDMAC_ChannelCallbackRegister	This function allows a XDMAC PLIB client to set an event handler.
	XDMAC_ChannelSettingsGet	Returns the channel settings of a specific XDMAC Channel.
	XDMAC_ChannelSettingsSet	Sets the channel settings of a specific XDMAC Channel.
	XDMAC_ChannelBlockLengthSet	Sets the channel Block Length of a specific XDMAC Channel.

c) Transaction Functions

	Name	Description
	XDMAC_ChannelTransfer	Adds a data transfer to a XDMAC channel and enables the channel to start data transfer.
	XDMAC_ChannelLinkedListTransfer	Sets up a multi-block data transfer on a specified XDMAC channel using linked list feature.
	XDMAC_ChannelDisable	Disables the specified channel.

d) Status Functions

	Name	Description
	XDMAC_ChannelIsBusy	Returns the busy status of a specific XDMAC Channel.

e) Data Types and Constants

	Name	Description
	XDMAC_CHANNEL	This lists the set of channels available for data transfer using XDMAC.
	XDMAC_CHANNEL_CALLBACK	Pointer to a XDMAC Transfer Event handler function.
	XDMAC_CHANNEL_CONFIG	Defines the channel settings.
	XDMAC_DESCRIPTOR_CONTROL	Defines the descriptor control for linked list operation.
	XDMAC_DESCRIPTOR_VIEW_0	View 0
	XDMAC_DESCRIPTOR_VIEW_1	View 1
	XDMAC_DESCRIPTOR_VIEW_2	View 2
	XDMAC_DESCRIPTOR_VIEW_3	View 3
	XDMAC_MICRO_BLOCK_CONTROL	Defines the control parameters for linked list operation.
	XDMAC_TRANSFER_EVENT	Enumeration of possible XDMAC transfer events.

a) Initialization Function

XDMAC_Initialize Function

Initializes the XDMAC controller of the device.

File

plib_xdmac.h

C

```
void XDMAC_Initialize();
```

Returns

None.

Description

This function initializes the XDMAC controller of the device as configured by the user from within the XDMAC manager of MCC.

Remarks

Stops the XDMAC controller if it was already running and reinitializes it.

Preconditions

None.

Example

```
XDMAC_Initialize();
```

Function

void XDMAC_Initialize(void)

b) Setup Functions

XDMAC_ChannelCallbackRegister Function

This function allows a XDMAC PLIB client to set an event handler.

File

plib_xdmac.h

C

```
void XDMAC_ChannelCallbackRegister(XDMAC_CHANNEL channel, const XDMAC_CHANNEL_CALLBACK
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to set an event handler. The client may want to receive transfer related events in cases when it submits a XDMAC PLIB transfer request. The event handler should be set before the client intends to perform operations that could generate events.

This function accepts a contextHandle parameter. This parameter could be set by the client to contain (or point to) any client specific data object that should be associated with this XDMAC channel.

Remarks

None.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
MY_APP_OBJ myAppObj;

void APP_XDMACTransferEventHandler(XDMAC_TRANSFER_EVENT event,
        uintptr_t contextHandle)
{
    switch(event)
    {
        case XDMAC_TRANSFER_COMPLETE:
            // This means the data was transferred.
            break;

        case XDMAC_TRANSFER_ERROR:
            // Error handling here.
            break;
```

```
        default:
            break;
    }
}

// User registers an event handler with XDMAC channel. This is done once.
XDMAC_ChannelCallbackRegister(channel, APP_XDMACTransferEventHandler,
        (uintptr_t)&myAppObj);
```

Parameters

Parameters	Description
channel	A specific XDMAC channel from which the events are expected.
eventHandler	Pointer to the event handler function.
contextHandle	Value identifying the context of the application/driver/middleware that registered the event handling function.

Function

void XDMAC_ChannelCallbackRegister
(
    XDMAC_CHANNEL channel,
const XDMAC_CHANNEL_CALLBACK eventHandler,
const uintptr_t contextHandle
)

XDMAC_ChannelSettingsGet Function

Returns the channel settings of a specific XDMAC Channel.

File

plib_xdmac.h

C

```
XDMAC_CHANNEL_CONFIG XDMAC_ChannelSettingsGet(XDMAC_CHANNEL channel);
```

Returns

Settings of a specific channel.

Description

This function returns the channel settings of the XDMAC channel.

This function can be used along with the XDMAC_ChannelSettingsSet to update any specific setting of the XDMAC channel.

Remarks

Use macro definitions from the packs header to compare with the settings.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
//Update the transfer direction dynamically which was set to
//PER2MEM from within DMA manager to MEM2PER.
XDMAC_CHANNEL_CONFIG settings = 0;
settings = XDMAC_ChannelSettingsGet(XDMAC_CHANNEL_0);
settings = (settings & ~XDMAC_CC_DSYNC_Msk) | XDMAC_CC_DSYNC_MEM2PER;
XDMAC_ChannelSettingsSet(XDMAC_CHANNEL_0, settings);
```

Parameters

Parameters	Description
channel	A specific XDMAC channel

Function

XDMAC_CHANNEL_SETTINGS XDMAC_ChannelSettingsGet (XDMAC_CHANNEL channel)

XDMAC_ChannelSettingsSet Function

Sets the channel settings of a specific XDMAC Channel.

File

plib_xdmac.h

C

```
bool XDMAC_ChannelSettingsSet(XDMAC_CHANNEL channel, XDMAC_CHANNEL_CONFIG setting);
```

Returns

Settings update status.

- True - Channel settings are updated successfully.
- False - Channel settings update failed.

Description

This function sets the channel settings of the XDMAC channel.

This function can be used along with the XDMAC_ChannelSettingsGet to update any specific setting of the XDMAC channel. Any ongoing transaction of the specified XDMAC channel will be aborted when this function is called.

Remarks

Use macro definitions from the packs header to construct a new setting.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
//Update the transfer direction dynamically which was set to
//PER2MEM from within DMA manager to MEM2PER.
XDMAC_CHANNEL_CONFIG settings = 0;
settings = XDMAC_ChannelSettingsGet(XDMAC_CHANNEL_0);
settings = (settings & ~XDMAC_CC_DSYNC_Msk) | XDMAC_CC_DSYNC_MEM2PER;
XDMAC_ChannelSettingsSet(XDMAC_CHANNEL_0, settings);
```

Parameters

Parameters	Description
channel	A specific XDMAC channel
setting	Settings for a channel

Function

bool XDMAC_ChannelSettingsSet (XDMAC_CHANNEL channel,

XDMAC_CHANNEL_SETTINGS setting)

XDMAC_ChannelBlockLengthSet Function

Sets the channel Block Length of a specific XDMAC Channel.

File

plib_xdmac.h

C

```
void XDMAC_ChannelBlockLengthSet(XDMAC_CHANNEL channel, uint16_t length);
```

Returns

None

Description

This function sets the channel Block Length of the XDMAC channel. The length of the block is (BLEN+1) microblocks. Any ongoing transaction of the specified XDMAC channel will be aborted when this function is called.

Remarks

None

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
XDMAC_ChannelBlockLengthSet(XDMAC_CHANNEL_0, 16);
```

Parameters

Parameters	Description
channel	A specific XDMAC channel
length	The number of microblocks of data

Function

void XDMAC_ChannelBlockLengthSet (XDMAC_CHANNEL channel, uint16_t length);

c) Transaction Functions

XDMAC_ChannelTransfer Function

Adds a data transfer to a XDMAC channel and enables the channel to start data transfer.

File

plib_xdmac.h

C

```
bool XDMAC_ChannelTransfer(XDMAC_CHANNEL channel, const void * srcAddr, const void * destAddr,
size_t blockSize);
```

Returns

- True - If transfer request is accepted.
- False - If previous transfer is in progress and the request is rejected.

Description

This function adds a single block data transfer characteristics for a specific XDMAC channel id it is not busy already. It also enables the channel to start data transfer.

If the requesting client registered an event callback with the PLIB, the PLIB will issue a XDMAC_TRANSFER_COMPLETE event if the transfer was processed successfully and XDMAC_TRANSFER_ERROR event if the transfer was not processed successfully.

Remarks

When DMA transfer buffers are placed in cacheable memory, cache maintenance operation must be performed by cleaning and invalidating cache for DMA buffers located in cacheable SRAM region using CMSIS APIs. The buffer start address must be aligned to cache line and buffer size must be multiple of cache line. Refer to device documentation to find the cache line size.

Invalidate cache lines having received buffer before using it to load the latest data in the actual memory to the cache SCB_InvalidateDCache_by_Addr((uint32_t *)&readBuffer, sizeof(readBuffer));

Clean cache lines having source buffer before submitting a transfer request to XDMAC to load the latest data in the cache to the actual memory SCB_CleanDCache_by_Addr((uint32_t *)&writeBuffer, sizeof(writeBuffer));

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
// Transfer 10 bytes of data to UART TX using XDMAC channel 1
MY_APP_OBJ myAppObj;
uint8_t buf[10] = {0,1,2,3,4,5,6,7,8,9};
void *srcAddr = (uint8_t *) buf;
void *destAddr = (uin8_t*) &U1TXREG;
size_t size = 10;

// User registers an event handler with PLIB. This is done once.
XDMAC_ChannelCallbackRegister(APP_XDMACTransferEventHandler,
    (uintptr_t)&myAppObj);

if (XDMAC_ChannelTransfer(XDMAC_CHANNEL_1, srcAddr, destAddr, size) == true)
{
    // do something else
}
else
{
    // try again?
}
```

Parameters

Parameters	Description
channel	A specific XDMAC channel
srcAddr	Source of the XDMAC transfer
destAddr	Destination of the XDMAC transfer
blockSize	Size of the transfer block

Function

bool XDMAC_ChannelTransfer

(

XDMAC_CHANNEL channel,

const void *srcAddr,

const void *destAddr,

size_t blockSize

)

XDMAC_ChannelLinkedListTransfer Function

Sets up a multi-block data transfer on a specified XDMAC channel using linked list feature.

File

plib_xdmac.h

C

```
bool XDMAC_ChannelLinkedListTransfer(XDMAC_CHANNEL channel, uint32_t firstDescriptorAddress,
XDMAC_DESCRIPTOR_CONTROL* firstDescriptorControl);
```

Returns

- True - If transfer request is accepted.
- False - If previous transfer is in progress and the request is rejected.

Description

This function sets up multi-block data transfer for a XDMAC channel by linked list operation mode if channel is not busy already. That is each block of data to be transferred is configured as descriptors of a linked list. Each descriptor can be defined in any available descriptor view formats and the views are available as XDMAC_DESCRIPTOR_VIEW_x.

Specified XDMAC channel will be enabled by default upon linked list addition. Upon each block transfer completion XDMAC PLIB will call event handler if the event handler is registered through XDMAC_ChannelCallbackRegister.

Remarks

This is supported by only the devices with multi-block transfer feature. The "descriptorControl" parameter is a XDMAC hardware version specific requirement and may not be needed in some of the devices.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize. Linked list with descriptors in any available descriptor view formats must be defined.

Example

```
// Transfer 2 blocks of data to UART TX, each 10 bytes.

//Two blocks of data
char src1[11] = {"1234567890"};
char src2[11] = {"ABCDEFGHIJ"};

//Cache line aligned first descriptor control, "descptr_crtl"
//Cache line aligned linked list with descriptors in any supported view,
//"linkedList1[2]" with 2 descriptors of different source addresses.

// User registers an event handler with PLIB.
XDMAC_ChannelCallbackRegister(handle, APP_XDMAC_TransferEventHandler,
    NULL);

// Add linked list to the XDMAC PLIB, channel will be enabled by default.
if(XDMAC_ChannelLinkedListTransfer(XDMAC_CHANNEL_1, (uint32_t)&linkedList1[0], &descptr_crtl)
== true)
{
    // do something else
}
else
{
    // try again?
}
```

Parameters

Parameters	Description
channel	A specific XDMAC channel
firstDescriptorAddress	Address of the first descriptor of the linked list. All descriptors must be defined in the device specific view formats available as XDMAC_DESCRIPTOR_VIEW_x and must be cache line aligned.
firstDescriptorControl	Control settings for the first descriptor. This must be defined in device specific view format, XDMAC_DESCRIPTOR_CONTROL and must be cache line aligned.

Function

bool XDMAC_ChannelLinkedListTransfer

```
(
    XDMAC_CHANNEL channel,
uint32_t descriptor,
    XDMAC_DESCRIPTOR_CONTROL* descriptorControl
)
```

XDMAC_ChannelDisable Function

Disables the specified channel.

File

plib_xdmac.h

C

```
void XDMAC_ChannelDisable(XDMAC_CHANNEL channel);
```

Returns

None.

Description

This function disables the specified channel and resets it. Ongoing transfer will be aborted.

Remarks

None.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize. And transfer should have been initiated.

Example

```
XDMAC_ChannelDisable(XDMAC_CHANNEL_1);
```

Parameters

Parameters	Description
channel	A specific XDMAC channel

Function

void XDMAC_ChannelDisable (XDMAC_CHANNEL channel)

d) Status Functions

XDMAC_ChannelIsBusy Function

Returns the busy status of a specific XDMAC Channel.

File

plib_xdmac.h

C

```
bool XDMAC_ChannelIsBusy(XDMAC_CHANNEL channel);
```

Returns

Busy status of the specific channel.

- True - Channel is busy
- False - Channel is free

Description

This function returns the busy status of the XDMAC channel. XDMAC channel will be busy if any transfer is in progress.

This function can be used to check the status of the channel prior to submitting a transfer request. And this can also be used to check the status of the submitted request if callback mechanism is not preferred.

Remarks

None.

Preconditions

XDMAC should have been initialized by calling XDMAC_Initialize.

Example

```
// Transfer 10 bytes of data to UART TX using XDMAC channel 1
MY_APP_OBJ myAppObj;
uint8_t buf[10] = {0,1,2,3,4,5,6,7,8,9};
void *srcAddr = (uint8_t *) buf;
void *destAddr = (uin8_t*) &U1TXREG;
size_t size = 10;

if(false == XDMAC_ChannelIsBusy(XDMAC_CHANNEL_1))
{
    XDMAC_ChannelTransfer(XDMAC_CHANNEL_1, srcAddr, destAddr, size);
}
```

Parameters

Parameters	Description
channel	A specific XDMAC channel

Function

bool XDMAC_ChannelIsBusy (XDMAC_CHANNEL channel)

e) Data Types and Constants

XDMAC_CHANNEL Enumeration

This lists the set of channels available for data transfer using XDMAC.

File

plib_xdmac.h

C

```
typedef enum {
  XDMAC_CHANNEL_0,
  XDMAC_CHANNEL_1,
  XDMAC_CHANNEL_2,
  XDMAC_CHANNEL_3,
  XDMAC_CHANNEL_4,
  XDMAC_CHANNEL_5,
  XDMAC_CHANNEL_6,
  XDMAC_CHANNEL_7,
  XDMAC_CHANNEL_8,
  XDMAC_CHANNEL_9,
  XDMAC_CHANNEL_10,
  XDMAC_CHANNEL_11,
```

```
    XDMAC_CHANNEL_12,
    XDMAC_CHANNEL_13,
    XDMAC_CHANNEL_14,
    XDMAC_CHANNEL_15,
    XDMAC_CHANNEL_16,
    XDMAC_CHANNEL_17,
    XDMAC_CHANNEL_18,
    XDMAC_CHANNEL_19,
    XDMAC_CHANNEL_20,
    XDMAC_CHANNEL_21,
    XDMAC_CHANNEL_22,
    XDMAC_CHANNEL_23
} XDMAC_CHANNEL;
```

Description

XDMAC Channels

This lists the set of channels available for data transfer using XDMAC.

Remarks

None.

XDMAC_CHANNEL_CALLBACK Type

Pointer to a XDMAC Transfer Event handler function.

File

plib_xdmac.h

C

```
typedef void (* XDMAC_CHANNEL_CALLBACK)(XDMAC_TRANSFER_EVENT event, uintptr_t contextHandle);
```

Description

XDMAC Transfer Event Handler Function

This data type defines a XDMAC Transfer Event Handler Function.

A XDMAC PLIB client must register a transfer event handler function of this type to receive transfer related events from the PLIB.

If the event is XDMAC_TRANSFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is XDMAC_TRANSFER_EVENT_ERROR, this means that the data was not transferred successfully.

The contextHandle parameter contains the context handle that was provided by the client at the time of registering the event handler. This context handle can be anything that the client consider helpful or necessary to identify the client context object associated with the channel of the XDMAC PLIB that generated the event.

The event handler function executes in an interrupt context of XDMAC. It is recommended to the application not to perform process intensive operations with in this function.

Remarks

None.

XDMAC_CHANNEL_CONFIG Type

Defines the channel settings.

File

plib_xdmac.h

C

```
typedef uint32_t XDMAC_CHANNEL_CONFIG;
```

Description

DMA Channel Settings

This data type defines the channel settings and can be used to update the channel settings dynamically .

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

XDMAC_DESCRIPTOR_CONTROL Union

Defines the descriptor control for linked list operation.

File

plib_xdmac.h

C

```
typedef union {
  struct {
    uint8_t fetchEnable : 1;
    uint8_t sourceUpdate : 1;
    uint8_t destinationUpdate : 1;
    uint8_t view : 2;
  }
  uint8_t descriptorControl;
} XDMAC_DESCRIPTOR_CONTROL;
```

Members

Members	Description
uint8_t fetchEnable : 1;	Descriptor fetch enable. Zero in this field indicates the end of linked list.
uint8_t sourceUpdate : 1;	Enable/Disable source address update when the descriptor is retrieved
uint8_t destinationUpdate : 1;	Enable/Disable destination address update when the descriptor is retrieved
uint8_t view : 2;	Descriptor view type. Views can be changed when switching descriptors.

Description

DMA descriptor control

This data type defines the descriptor control for linked list operation. Descriptor control always applies to the next descriptor in the chain. That is, descriptor control parameters defined in descriptor 'n' applies to the transfer of descriptor "n+1".

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

XDMAC_DESCRIPTOR_VIEW_0 Structure

File

plib_xdmac.h

C

```
typedef struct {
  uint32_t mbr_nda;
  XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;
  uint32_t mbr_da;
} XDMAC_DESCRIPTOR_VIEW_0;
```

Members

Members	Description
uint32_t mbr_nda;	Next Descriptor Address number.
XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;	Micro-block Control Member.
uint32_t mbr_da;	Destination Address Member.

Description

View 0

XDMAC_DESCRIPTOR_VIEW_1 Structure

File

plib_xdmac.h

C

```
typedef struct {
  uint32_t mbr_nda;
  XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;
  uint32_t mbr_sa;
  uint32_t mbr_da;
} XDMAC_DESCRIPTOR_VIEW_1;
```

Members

Members	Description
uint32_t mbr_nda;	Next Descriptor Address number.
XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;	Micro-block Control Member.
uint32_t mbr_sa;	Source Address Member.
uint32_t mbr_da;	Destination Address Member.

Description

View 1

XDMAC_DESCRIPTOR_VIEW_2 Structure

File

plib_xdmac.h

C

```
typedef struct {
  uint32_t mbr_nda;
  XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;
  uint32_t mbr_sa;
  uint32_t mbr_da;
  uint32_t mbr_cfg;
} XDMAC_DESCRIPTOR_VIEW_2;
```

Members

Members	Description
uint32_t mbr_nda;	Next Descriptor Address number.
XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;	Micro-block Control Member.

uint32_t mbr_sa;	Source Address Member.
uint32_t mbr_da;	Destination Address Member.
uint32_t mbr_cfg;	Configuration Register.

Description

View 2

XDMAC_DESCRIPTOR_VIEW_3 Structure

File

plib_xdmac.h

C

```
typedef struct {
  uint32_t mbr_nda;
  XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;
  uint32_t mbr_sa;
  uint32_t mbr_da;
  uint32_t mbr_cfg;
  uint32_t mbr_bc;
  uint32_t mbr_ds;
  uint32_t mbr_sus;
  uint32_t mbr_dus;
} XDMAC_DESCRIPTOR_VIEW_3;
```

Members

Members	Description
uint32_t mbr_nda;	Next Descriptor Address number.
XDMAC_MICRO_BLOCK_CONTROL mbr_ubc;	Micro-block Control Member.
uint32_t mbr_sa;	Source Address Member.
uint32_t mbr_da;	Destination Address Member.
uint32_t mbr_cfg;	Configuration Register.
uint32_t mbr_bc;	Block Control Member.
uint32_t mbr_ds;	Data Stride Member.
uint32_t mbr_sus;	Source Micro-block Stride Member.
uint32_t mbr_dus;	Destination Micro-block Stride Member.

Description

View 3

XDMAC_MICRO_BLOCK_CONTROL Structure

Defines the control parameters for linked list operation.

File

plib_xdmac.h

C

```
typedef struct {
  uint32_t blockDataLength : 24;
  XDMAC_DESCRIPTOR_CONTROL nextDescriptorControl;
} XDMAC_MICRO_BLOCK_CONTROL;
```

Members

Members	Description
uint32_t blockDataLength : 24;	Size of block for the current descriptor.
XDMAC_DESCRIPTOR_CONTROL nextDescriptorControl;	Next Descriptor Control Setting

Description

DMA Micro Block Control

This data type defines the control parameters for linked list operation. Block length applies to the current descriptor and XDMAC_DESCRIPTOR_CONTROL applies to the next descriptor.

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

XDMAC_TRANSFER_EVENT Enumeration

Enumeration of possible XDMAC transfer events.

File

plib_xdmac.h

C

```
typedef enum {
  XDMAC_TRANSFER_COMPLETE,
  XDMAC_TRANSFER_ERROR
} XDMAC_TRANSFER_EVENT;
```

Members

Members	Description
XDMAC_TRANSFER_COMPLETE	Data was transferred successfully.
XDMAC_TRANSFER_ERROR	Error while processing the request

Description

XDMAC Transfer Events

This data type provides an enumeration of all possible XDMAC transfer events.

Remarks

None.

Files

Files

Name	Description
plib_xdmac.h	XDMAC peripheral library interface.

Description

This section will list only the library's interface header file(s).

plib_xdmac.h

XDMAC peripheral library interface.

Enumerations

	Name	Description
	XDMAC_CHANNEL	This lists the set of channels available for data transfer using XDMAC.
	XDMAC_TRANSFER_EVENT	Enumeration of possible XDMAC transfer events.

Functions

	Name	Description
	XDMAC_ChannelBlockLengthSet	Sets the channel Block Length of a specific XDMAC Channel.
	XDMAC_ChannelCallbackRegister	This function allows a XDMAC PLIB client to set an event handler.
	XDMAC_ChannelDisable	Disables the specified channel.
	XDMAC_ChannelIsBusy	Returns the busy status of a specific XDMAC Channel.
	XDMAC_ChannelLinkedListTransfer	Sets up a multi-block data transfer on a specified XDMAC channel using linked list feature.
	XDMAC_ChannelSettingsGet	Returns the channel settings of a specific XDMAC Channel.
	XDMAC_ChannelSettingsSet	Sets the channel settings of a specific XDMAC Channel.
	XDMAC_ChannelTransfer	Adds a data transfer to a XDMAC channel and enables the channel to start data transfer.
	XDMAC_Initialize	Initializes the XDMAC controller of the device.

Structures

	Name	Description
	XDMAC_DESCRIPTOR_VIEW_0	View 0
	XDMAC_DESCRIPTOR_VIEW_1	View 1
	XDMAC_DESCRIPTOR_VIEW_2	View 2
	XDMAC_DESCRIPTOR_VIEW_3	View 3
	XDMAC_MICRO_BLOCK_CONTROL	Defines the control parameters for linked list operation.

Types

	Name	Description
	XDMAC_CHANNEL_CALLBACK	Pointer to a XDMAC Transfer Event handler function.
	XDMAC_CHANNEL_CONFIG	Defines the channel settings.

Unions

	Name	Description
	XDMAC_DESCRIPTOR_CONTROL	Defines the descriptor control for linked list operation.

Description

XDMAC Peripheral Library Interface Header File

This file defines the interface to the XDMAC peripheral library. This library provides access to and control of the XDMAC controller.

Remarks

This header is for documentation only. The actual plib_xdmac header will be generated as required by the MCC.

File Name

plib_xdmac.h

Company

Microchip Technology Inc.

Applications Help

This topic provides help for the peripheral library applications that are available in the Microchip 32 bit Chip Support Package (CSP).

Description

Below Peripheral Library applications are provided to demonstrate typical or interesting usage models of one or more Peripheral libraries.

AC Applications

This Section provides help for AC PLIB Applications.

ac_sleepwalk_singleshot

This application shows how AC can be effectively used with RTC and event system modules to perform comparisons in standby sleep mode periodically and to wake the device only when the input signal crosses the threshold.

Description

This example application shows how to use AC Peripheral library to perform single shot conversion when the peripheral is triggered by an RTC compare event. The generation of trigger and compare operation are done while the CPU is in Standby sleep. The AC generates interrupt on end of comparison and it wakes the CPU. Messages displayed on the console shows whether the signal level at the input pin PA04 is above or below the voltage applied on the negative input of the comparator.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/ac/ac_sleepwalk_singleshot/firmware

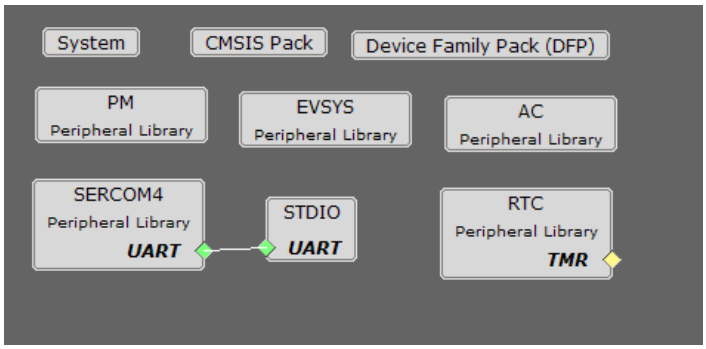
Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

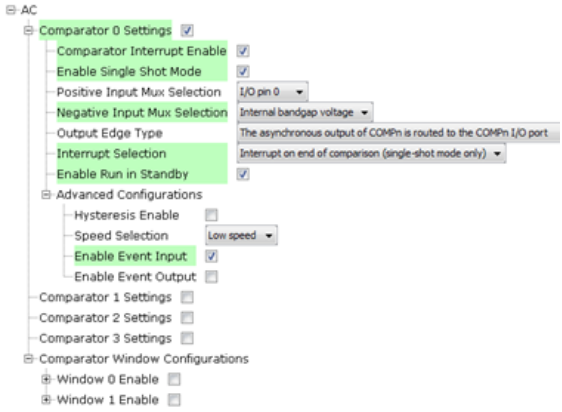
Description

The Project Graph diagram below shows the Harmony components that are included in this application.

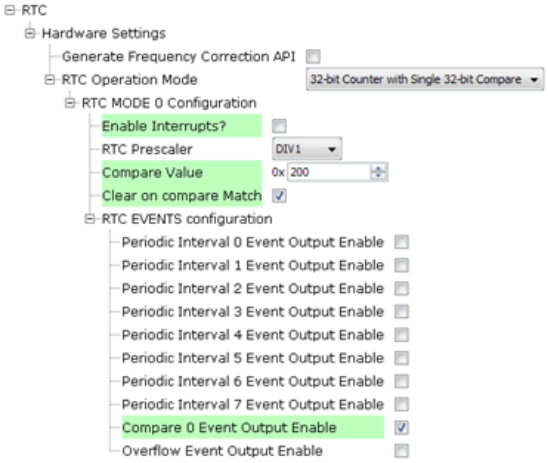


Components Used

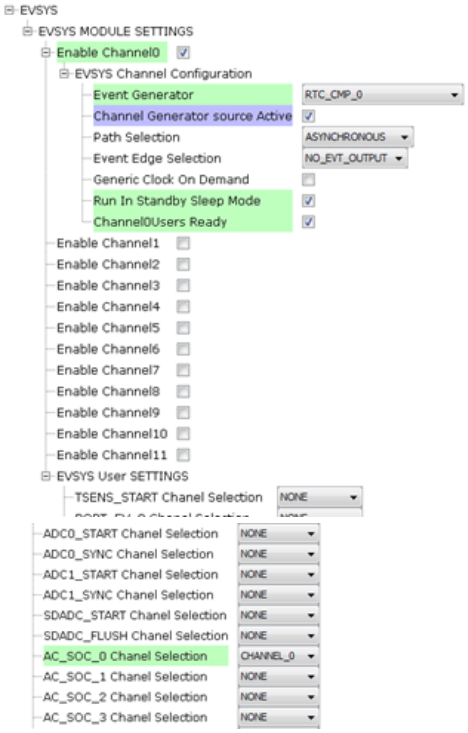
1. AC Peripheral Library



2. RTC Peripheral Library

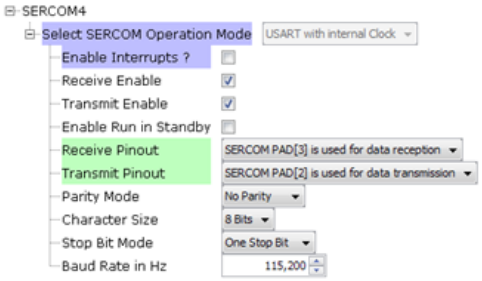


3. EVSYS Peripheral Library



4. PM Peripheral Library - No special configuration

5. STDIO - STDIO Utility uses SERCOM4 configured as below.



MHC Settings

1. Configure the components as shows in section 'Components Used'

Hardware Setup

This section describes how to configure the supported hardware.

Description

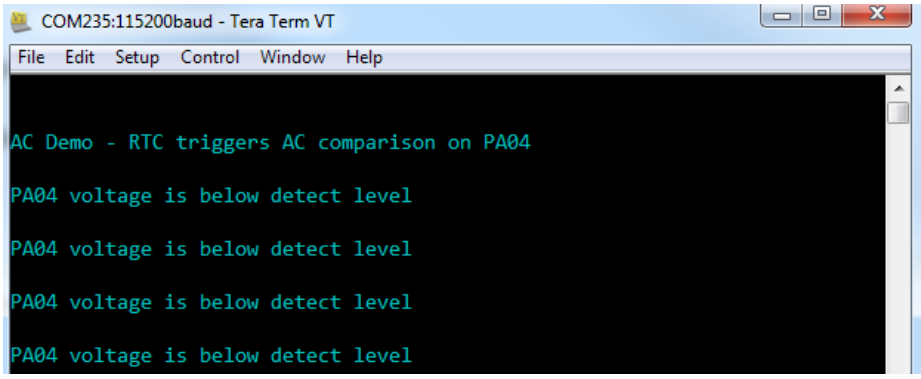
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit.
  - **Hardware Setup**
    - Connect a voltage below VDD to pin 3 of the 'VREF connector' (6-pin) on the SAM C21N Xplained Pro Evaluation Kit.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG.
3. Press the Reset button on the SAM C21N Xplained Pro Evaluation Kit.
4. The console output will be as given below when the voltage at PA04 is lower than the internal bandgap voltage since the bandgap voltage is connected to the negative input of COMP0.



ac_wav_gen

This example application shows how to use AC to generate 1KHz PWM wave.

Description

This example application shows how to use Analog comparator Peripheral library to generate analog 1KHz PWM wave by

comparing the sine wave generated from DAC with reference voltage selected as VDD scaler. The duty cycle of the analog PWM is changed by pressing a switch which changes the value of VDD scaler.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/ac/ac_wav_gen/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components that are included in this application.

Components Used

1. AC Peripheral Library
2. EIC Peripheral Library
3. SUPC Peripheral Library
4. DAC Peripheral Library
5. STDIO - STDIO Utility uses SERCOM4 configured as below.

MHC Settings

1. Configure “AC Peripheral Library”
2. Configure “EIC Peripheral Library” for Switch SW0
3. Configure “SERCOM Peripheral Library” as STDIO USART component
4. Configure “DAC Peripheral Library” to generate 1KHz wave sine wave.
5. Configure “SUPC Peripheral Library” to support Reference voltage level to 4.096V.

Hardware Setup

This section describes how to configure the supported hardware.

Description

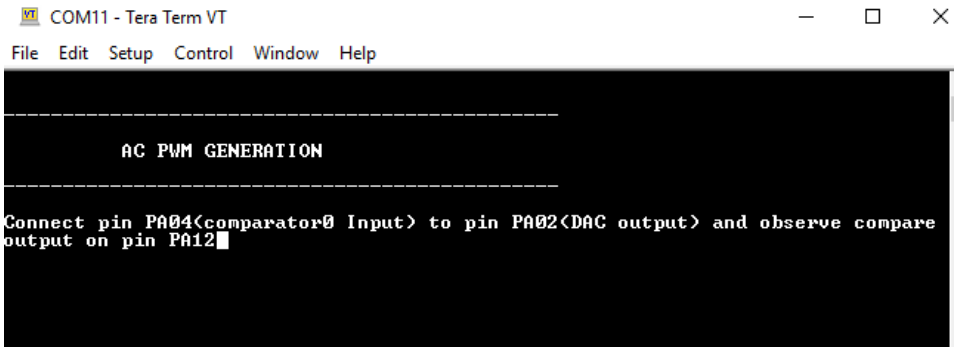
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit.
  - **Hardware Setup**
    - Connect a jumper between pin PA04(Comparator input) and PA02(DAC Output).
    - Use Logic Analyzer/Oscilloscope and probe on PA12 pin (Pin 14 of EXT1) to observe PWM output.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

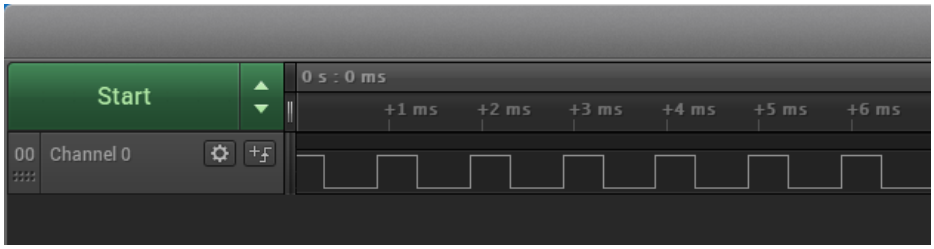
Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset button on the SAM C21N Xplained Pro Evaluation Kit.
4. Console displays the message like below,



5. Use Logic Analyzer/Oscilloscope and probe on PA12 pin (Pin 14 of EXT1) to observe PWM output as shown below.

6. The below Waveform after Switch press



ACC Applications

This Section provides help for ACC PLIB Applications.

acc_wav_gen

This example uses Analog comparator to generate analog PWM (5 KHz) by comparing the sine wave generated from DAC with external input voltage. The duty cycle of the analog PWM is inversely proportional to the input voltage.

Description

This example uses Analog comparator to generate analog PWM (5 KHz) by comparing the sine wave generated from DAC with external input voltage. The duty cycle of the analog PWM is inversely proportional to the input voltage.

Building the Application

This section provides information on how to build application using MPLAB X IDE

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/acc/acc_wav_gen/firmware/

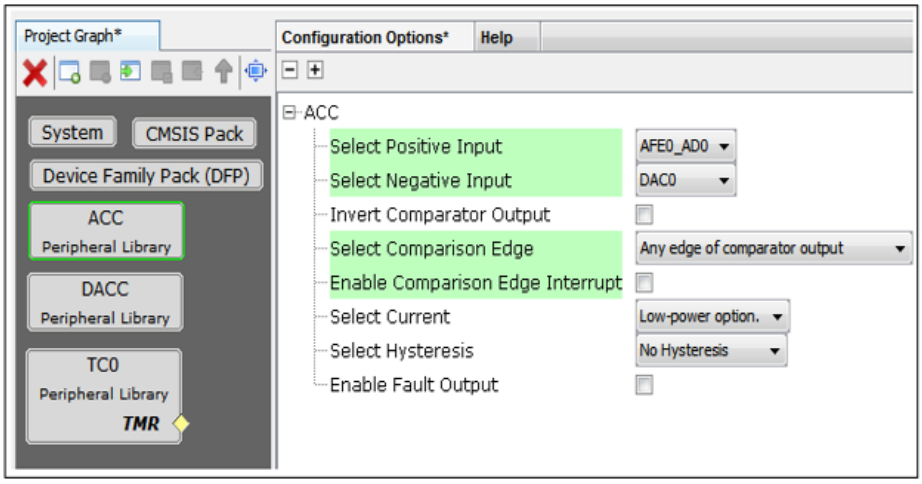
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. ACC Peripheral Library
2. DACC Peripheral Library
3. TC Peripheral Library

MHC Settings

1. Configure ACC, DACC and TC Peripherals

2. Configure Pins for DACC Peripheral in Pin Settings
3. Configure GPIO pin for Comparator output

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Comparator Positive Input: Connect the external voltage source to PD30(J506)
     - Comparator Negative Input: Sine wave from DAC Channel 0. DAC Channel 0 output is observable at PB13 (J502)
     - Comparator Output: Comparator output is observed at PB03 (J502)

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Comparator Positive Input: Connect the external voltage source to PD30 (J502).
     - Comparator Negative Input: Sine wave from DAC Channel 0. DAC Channel 0 output is observable at PB13 (J504).
     - Comparator Output: Comparator output is observed at PB03 (J504).

Note: Refer to SAM V71 Xplained Ultra User Guide to know information about Hardware connector overview.

Running the Application

This section provides information on how to run the application using MPLAB X IDE

Description

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as following:

Change the positive input of the comparator from 0 to 3.3V, and observe the duty cycle of square wave at comparator output, increases from 0% to 100%.

ADC Applications

This Section provides help for ADC PLIB Applications.

adc_sample

This example application shows how to sample analog input using ADC and display the converted samples on a serial terminal.

Description

This example application shows how to use the ADC Peripheral Library to sample analog input and display the converted data on a serial terminal.

In this application DAC output is fed as analog input for ADC conversion. The application starts conversion by calling ADC peripheral library and prints the sampled value on a serial terminal. On every user switch press on the development board, The DAC output is increased by 0.1V and the same will be reflected as ADC input. Therefore, on switch press the ADC sampled value is increased by 0.1V and the same is printed on the serial terminal.

ADC read sample output will always between 0.0V to 3.3V and it starts from 1.65V when device reset.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/adc/adc_sample/firmware

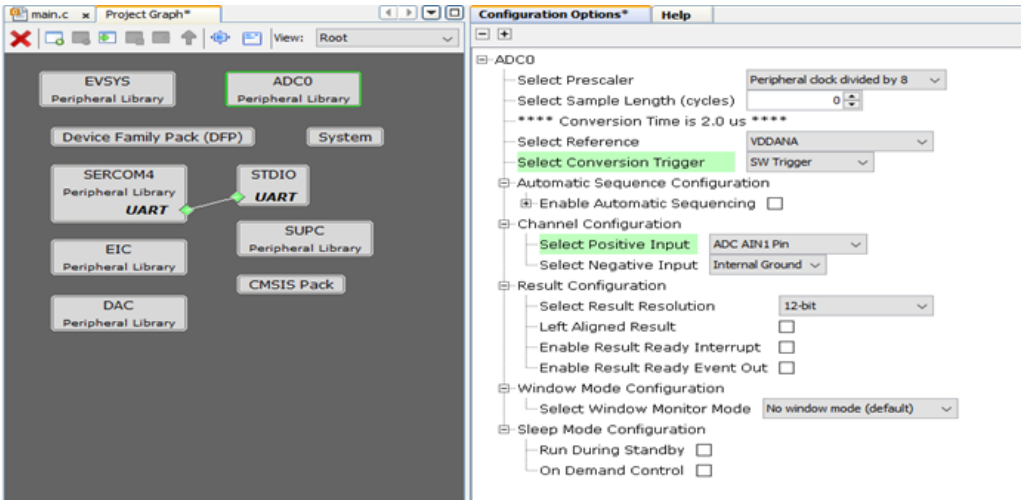
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. ADC Peripheral Library
2. DAC Peripheral Library
3. SERCOM Peripheral Library
4. EIC Peripheral Library
5. SUPC Peripheral Library
6. Systick Peripheral Library

MHC Settings

1. Configure ADC0
2. Configure DAC
3. Configure SUPC
4. Configure SERCOM4 as STDIO UART component

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Analog input (AIN1) is fed from DAC output. Analog input "AIN1" is mapped to Port Pin "PA03" that is routed to "Pin 4 of VREF header" and DAC output "DAC" is mapped to Port Pin "PA02" that is routed to "Pin 1 of DAC header". Use jumper wire to Connect "Pin 4 of VREF header" to "Pin 1 of DAC header"

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. Console displays the converted digital value.
5. Press Switch "SW0" to change the DAC value and observe the changed ADC value in console

adc_window_sleepwalking

This application shows how to configure the ADC in a sleepwalking operation, where the input voltage is measured at a fixed interval, but the device is not woken from sleep until the measured value is below a given threshold.

Description

This example application shows how to use ADC Peripheral library to perform sleepwalking. ADC0 on the device is configured to operate in window mode, to detect when the input voltage falls below the window upper threshold and report the voltage at the input pin when this happens. The ADC conversion is triggered by RTC at predefined interval. The generation of trigger and window detection operation are done while the CPU is in Standby sleep. The ADC generates interrupt if the voltage on AIN2 (PB08) is lower than half of VDDANA voltage (which is the selected reference for ADC0). The Window Monitor Interrupt wakes the CPU and then the CPU sends out a message via console to indicate that a window detection happened.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	**csp/apps/adc/adc_window_sleepwalking/firmware**

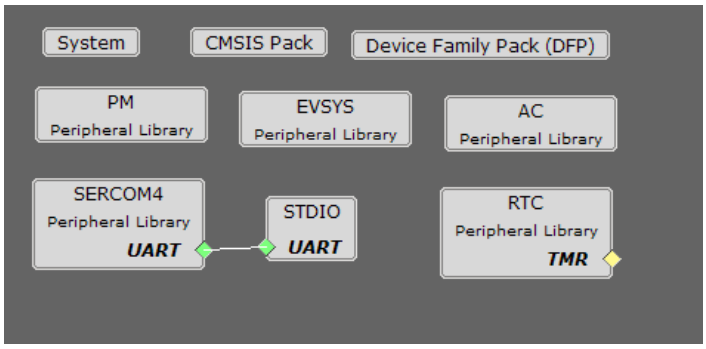
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

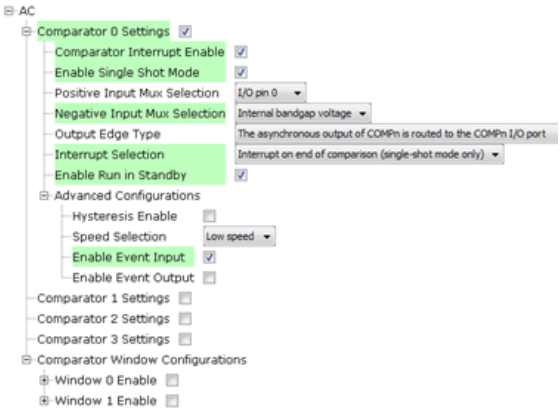
Description

The Project Graph diagram below shows the Harmony components that are included in this application.

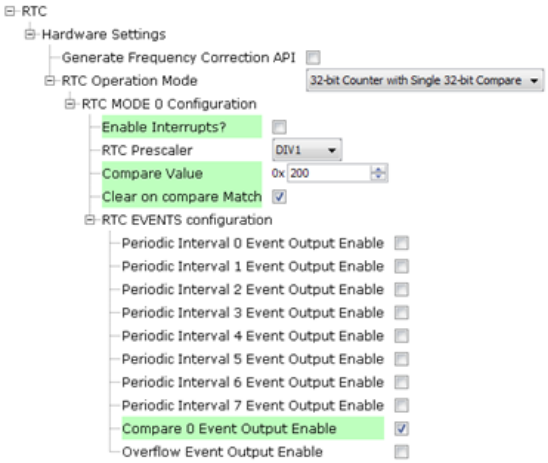


Components Used

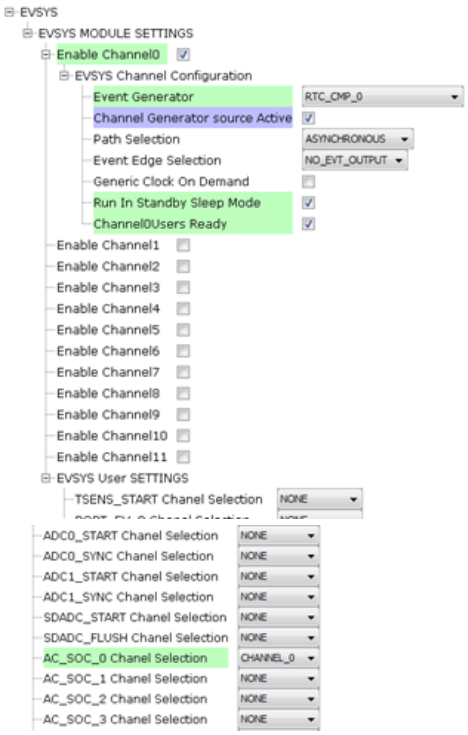
1. ADC Peripheral Library



2. RTC Peripheral Library



3. EVSYS Peripheral Library



4. PM Peripheral Library - No special configuration
5. STDIO Utility uses SERCOM4 configured as below.

MHC Settings

1. Configure the components as shows in section ‘Components Used’.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Connect a voltage below VDD to pin 4 (PB08 – ADC0_AIN2) of EXT1 connector on the SAM C21N Xplained Pro Evalution Kit.

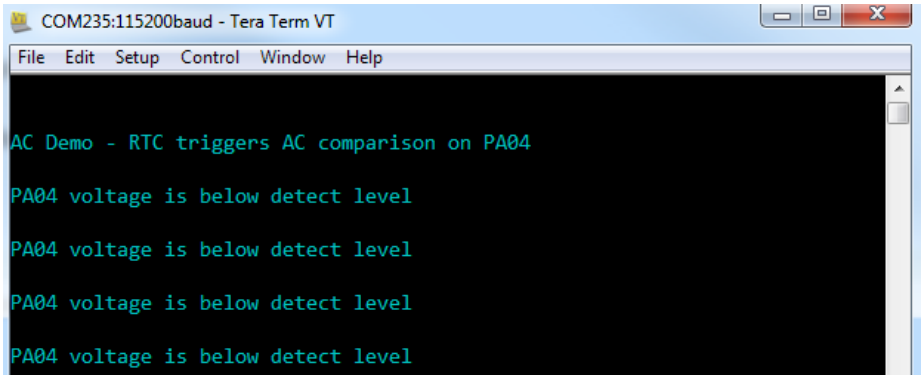
Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. The console output will be as given below when the voltage at PA04 is lower than half VDD since the window upper threshold

is set to 1024 and reference voltage for ADC0 is configured as VDDANA.



adc_dmac_sleepwalking

This application shows how to configure the ADC in a sleepwalking operation, where the input voltage is measured at a fixed interval, and then device is woken from sleep after N samples.

Description

This example application shows how to use Peripheral libraries of ADC, DMAC, RTC, EVSYS and PM to perform sleepwalking. ADC0 on the device is configured to perform conversion on incoming event from RTC. Event routing from RTC to ADC0 is handled via EVSYS. The ADC conversion is triggered by RTC at predefined interval. A DMAC channel is configured to copy a defined number of samples to an array in SRAM. In this application, this number is given as 16. The DMAC performs a 16-bit transfer on ADC conversion complete trigger. All these peripherals are operated while the CPU is in Standby sleep. The DMAC generates interrupt after it transfers 16 results into the array in SRAM. This wakes the CPU and then the CPU sends out a message via console to indicate that 16 samples are available in the SRAM array.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/adc/adc_dma_sleepwalking/firmware

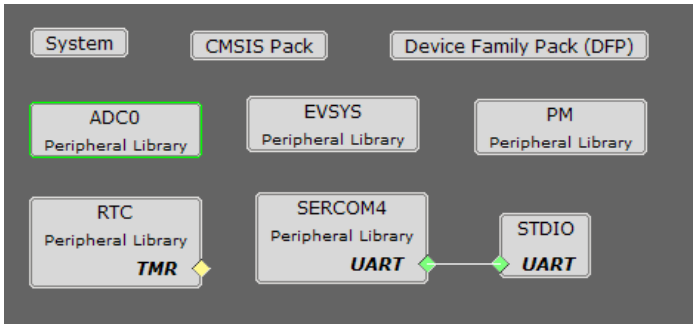
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

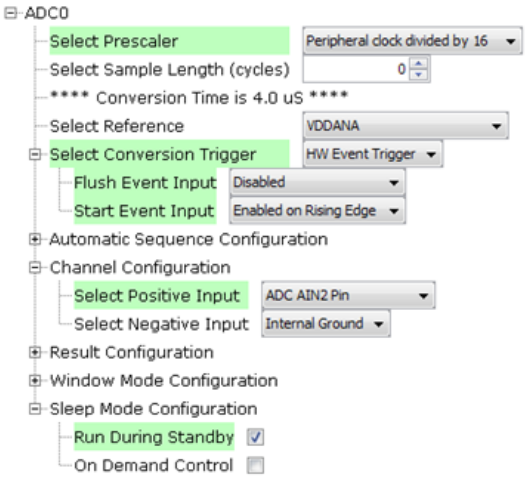
Description

The Project Graph diagram below shows the Harmony components that are included in this application.

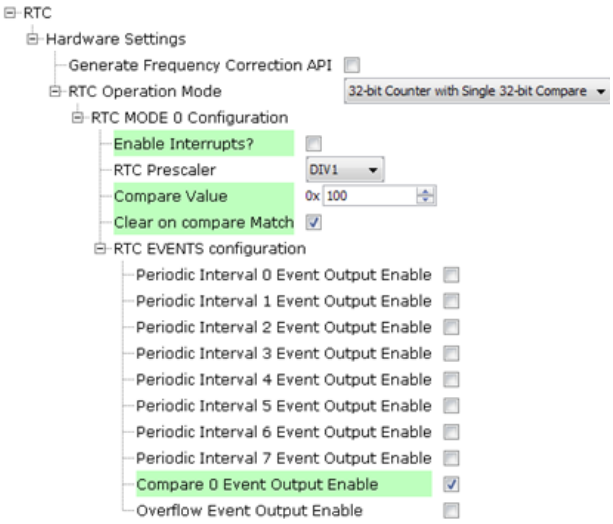


Components Used

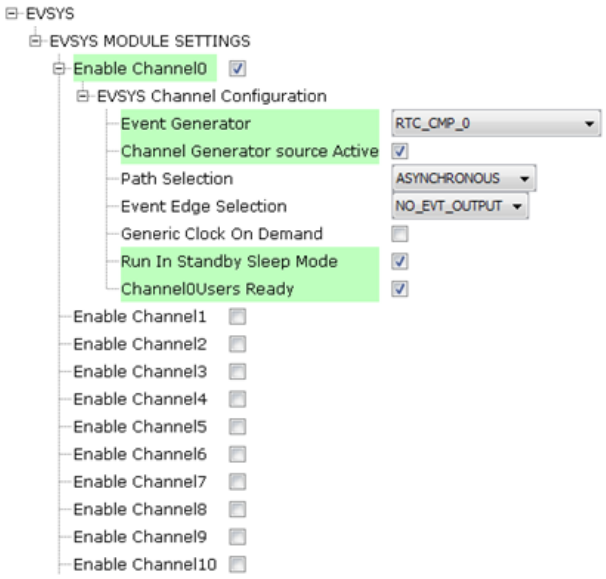
1. ADC Peripheral Library



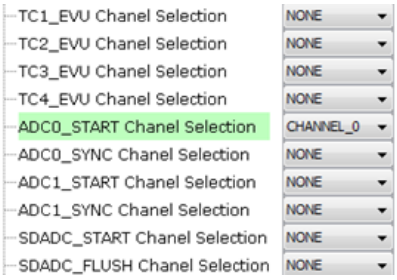
2. RTC Peripheral Library



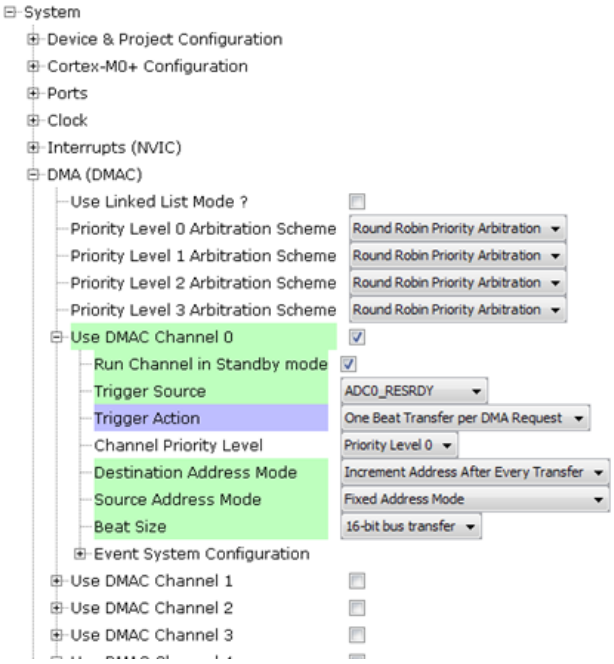
3. EVSYS Peripheral Library



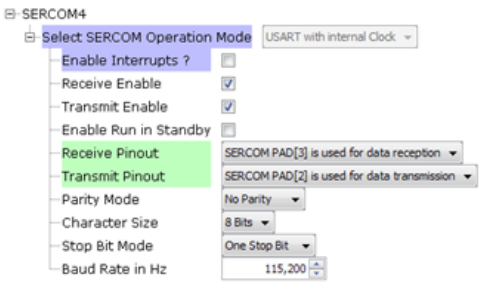
EVSYS User Settings



4. PM Peripheral Library - No special configuration
5. DMAC Peripheral Library



6. STDIO Utility uses SERCOM4 configured as below.



MHC Settings

1. Configure the components as shows in section 'Components Used'.

Hardware Setup

This section describes how to configure the supported hardware.

Description

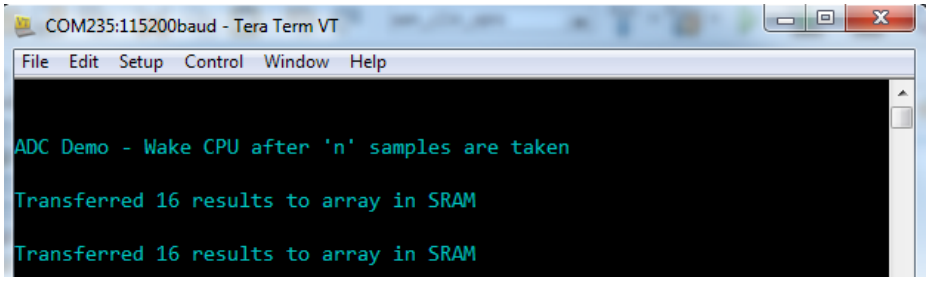
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Connect a voltage below the selected ADC reference voltage (VDDANA) to pin 4 (PB08 – ADC0_AIN2) of EXT1 connector on the SAM C21N Xplained Pro Evaluation Kit.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. The console output will be as given below after the DMAC transfers 16 conversion results into the array in SRAM.



5. To verify the ADC conversion result, enter debug session via the menu 'Debug -> Debug Main Project'. Set a break point inside the 'if condition' which checks whether the DMA transfer is done. Use the menu 'Window -> Debugging -> Variables' to see contents of 'adc_result_array'.

AFEC Applications

This Section provides help for AFEC PLIB Applications.

afec_polled_mode

This example demonstrates how to sample a analog input in polled mode and send the converted data to console.

Description

This example demonstrates how to sample a analog input (AD0) in polled mode and send the converted data to console.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

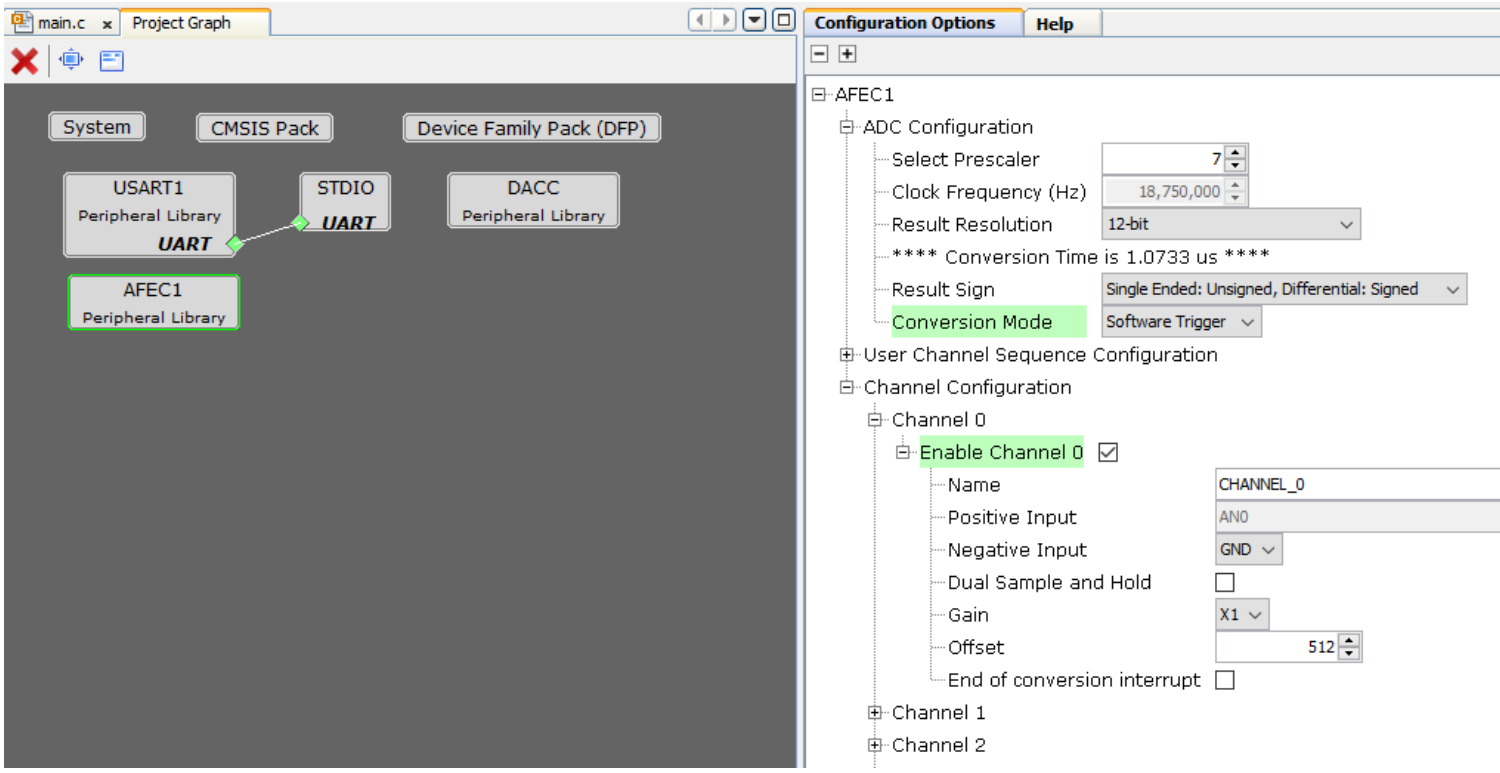
Application Path	csp/apps/afec/afec_polled_mode/firmware

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAMV71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description



Components Used

AFEC Peripheral Library

DACC Peripheral Library

Systick Peripheral Library

USART Peripheral Library

STDIO

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - ADC - input signals
       - Analog input (AD0) is feed from DAC output. DAC output is incremented by 0.1V on Switch (SW0) press.
       - Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
       - DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
       - Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

2. **Project sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - ADC - input signals
       - Analog input (AD0) is feed from DAC output. DAC output is incremented by 0.1V on Switch (SW0) press.

- Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
- DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
- Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port
2. Open Terminal application (Ex.:Tera term) on the computer
3. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Build and Program the application using MPLAB X IDE
5. Observe terminal application. It displays ADC Count and ADC Input Voltage.
6. Press switch to change DAC output by 0.1 V
7. And observe changed ADC count and ADC Input Voltage on the terminal application.
8. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: ADC count and ADC input voltage changes by 0.1 V when switch is pressed
sam_v71_xult.X	Success: ADC count and ADC input voltage changes by 0.1 V when switch is pressed

afec_user_sequence

This example demonstrates how to sample three analog inputs using user sequencer in triggered conversion mode and send the converted data to console.

Description

This example demonstrates how to sample three analog inputs using user sequencer in triggered conversion mode and send the converted data to console.

Conversion of three analog inputs (AD0, AD5, AD6) is triggered from TC peripheral every 250ms and user sequencer is used to define the order of conversion (AD5, AD6, AD0)

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/afec/afec_user_sequence/firmware

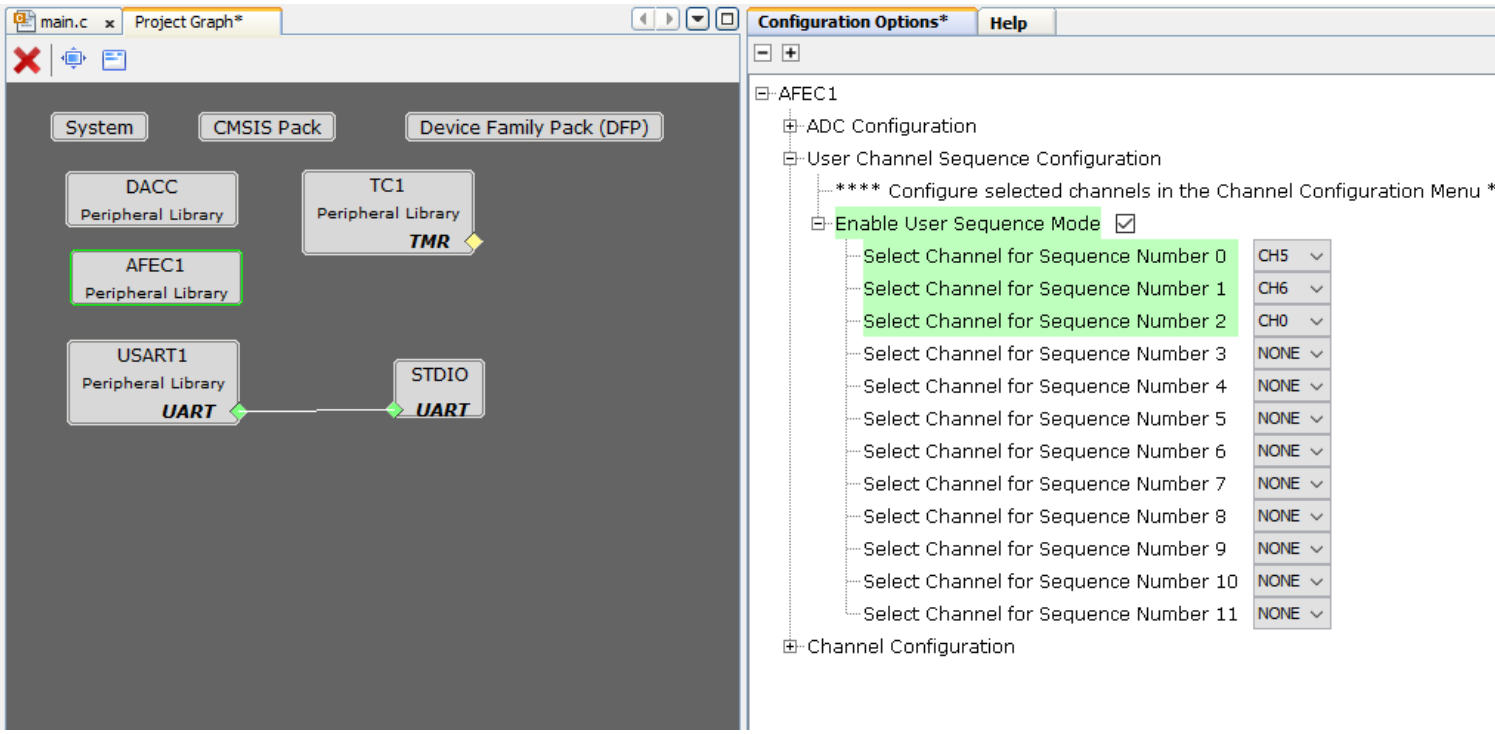
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAMV71 Xplained Ultra board

MPLAB Harmony Configurations

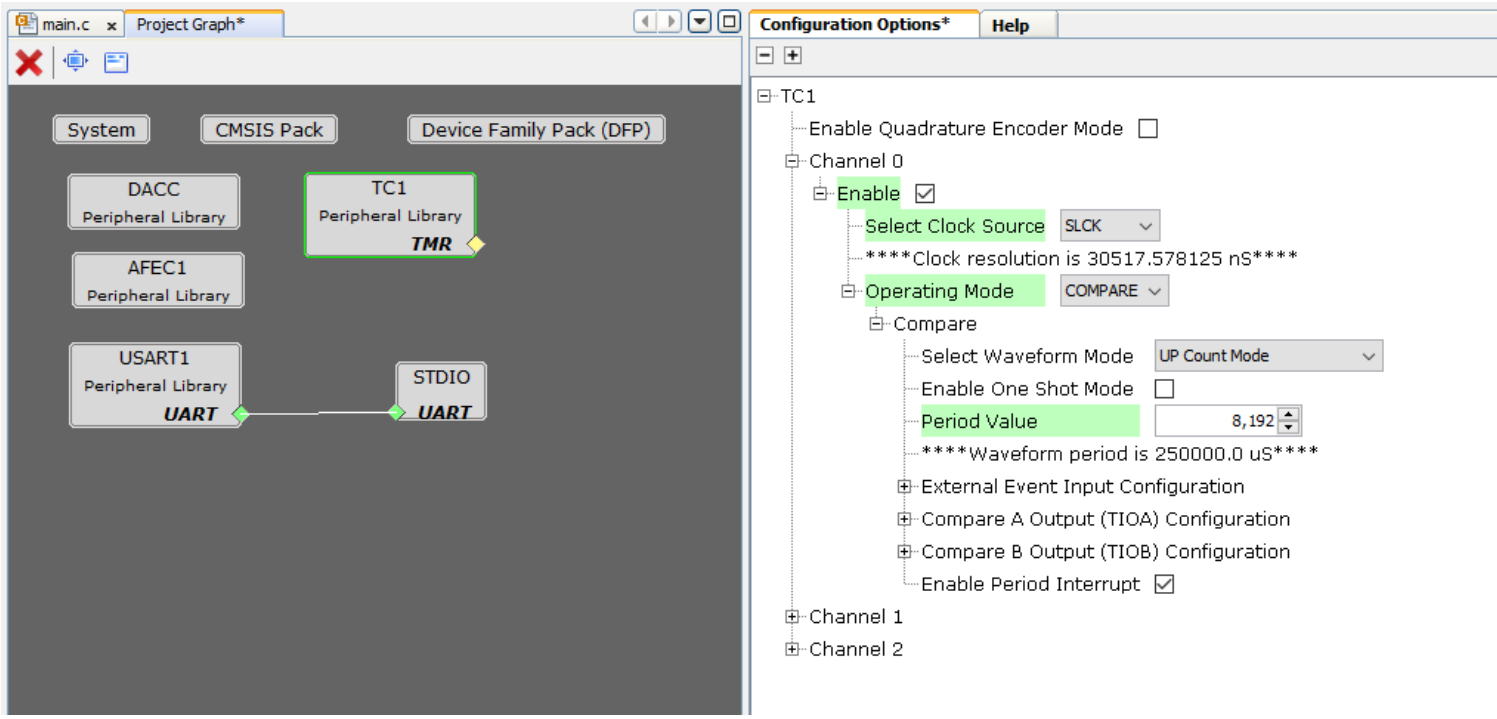
This topic provides information on MHC configurations

Description

AFEC Configurations:



TC Configurations:



Components Used

AFEC Peripheral Library

DACC Peripheral Library

TC Peripheral Library

USART Peripheral Library

STDIO

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Analog input (AD0) is feed from DAC output. DAC output is incremented by 0.1V on Switch (SW0) press.
       - Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
       - DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
       - Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

- Analog input (AD5) is connected to Vcc or user can supply input from external source
  - Analog input "AD5" is mapped to Port Pin "PC30" that is routed to "Pin 8 of EXT1 header".
  - Use jumper wire to connect "Pin 8 of EXT1 (AD5)" to Vcc or external voltage source

- Analog input (AD6) is connected to GND or user can supply input from external source
  - Analog input (AD6) is mapped to Port Pin "PC31" that is routed to "Pin 3 of EXT1 header".
  - Use jumper wire to connect "Pin 3 of EXT1" to GND or external voltage source

2. **Project sam_v71_xult.X**

- **Hardware Used**
  - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup**
  - Analog input (AD0) is feed from DAC output. DAC output is incremented by 0.1V on Switch (SW0) press.
    - Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
    - DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
    - Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

- Analog input (AD5) is connected to Vcc or user can supply input from external source
  - Analog input "AD5" is mapped to Port Pin "PC30" that is routed to "Pin 8 of EXT1 header".
  - Use jumper wire to connect "Pin 8 of EXT1 (AD5)" to Vcc or external voltage source

- Analog input (AD6) is connected to GND or user can supply input from external source
  - Analog input (AD6) is mapped to Port Pin "PC31" that is routed to "Pin 3 of EXT1 header".
  - Use jumper wire to connect "Pin 3 of EXT1" to GND or external voltage source

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port
2. Open Terminal application (Ex.:Tera term) on the computer
3. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Build and Program the application using MPLAB X IDE
5. Observe terminal application. Console displays the converted digital values and voltage
6. Press switch to change DAC output by 0.1 V
7. And observe changed ADC count and ADC Input Voltage on the terminal application.
8. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: Console displays ADC count and ADC input voltages for all three channels. CH0 values change when switch is pressed
sam_v71_xult.X	Success: Console displays ADC count and ADC input voltages for all three channels. CH0 values change when switch is pressed

afec_dma_striding

This example demonstrates taking 100 samples of three analog inputs at 5 KHz sampling rate in triggered conversion mode and then sort the data by channel using XDMAC.

Description

This example demonstrates taking 100 samples of three analog inputs at 5 KHz sampling rate in triggered conversion mode and then sort the data by channel using XDMAC. It sends the converted samples to console after all the samples are converted.

Conversion of three analog inputs (AD0, AD5, AD6) are triggered from TC peripheral every 200us and

it is converted in natural order (AD0, AD5, AD6).

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/afec/afec_dma_striding/firmware

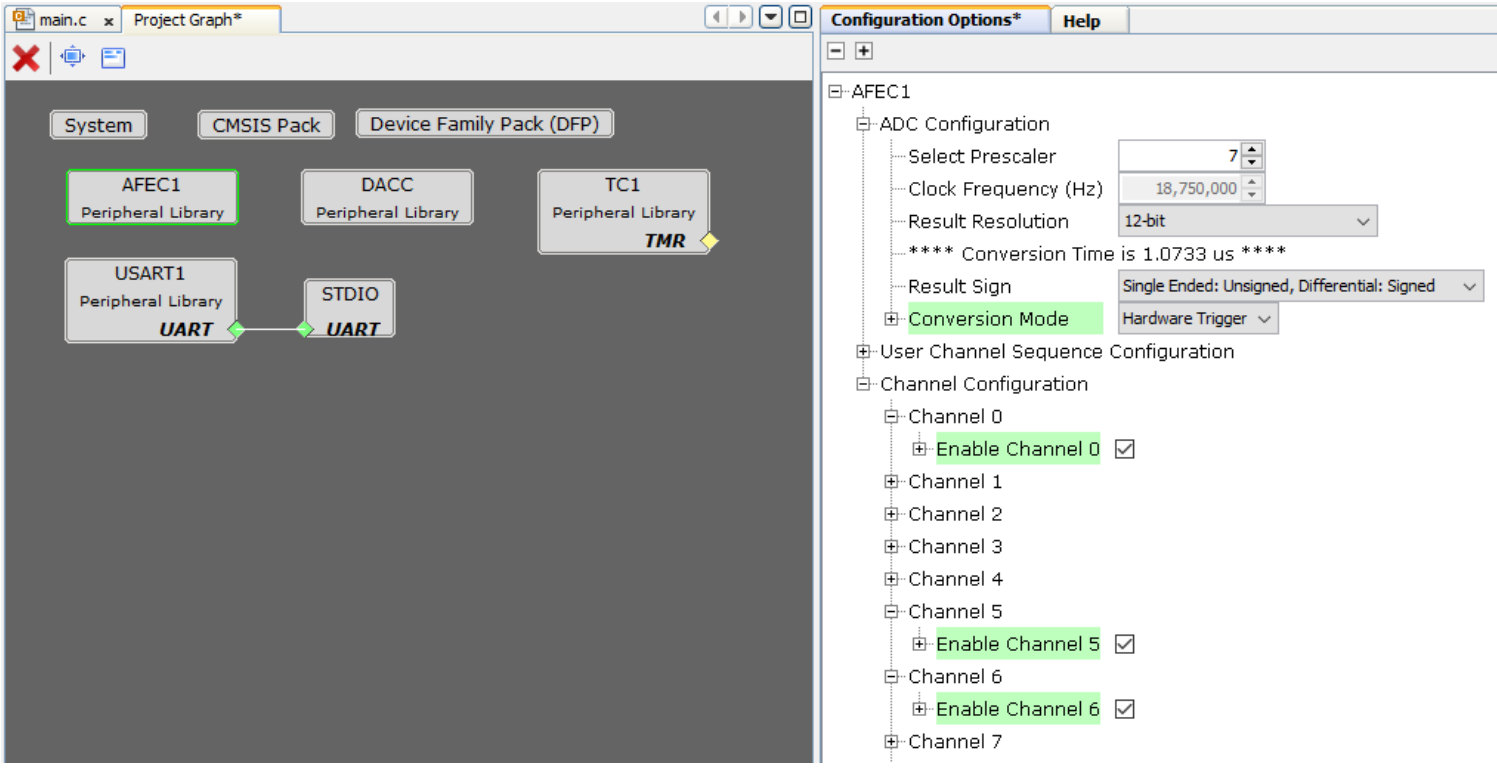
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAMV71 Xplained Ultra board

MPLAB Harmony Configurations

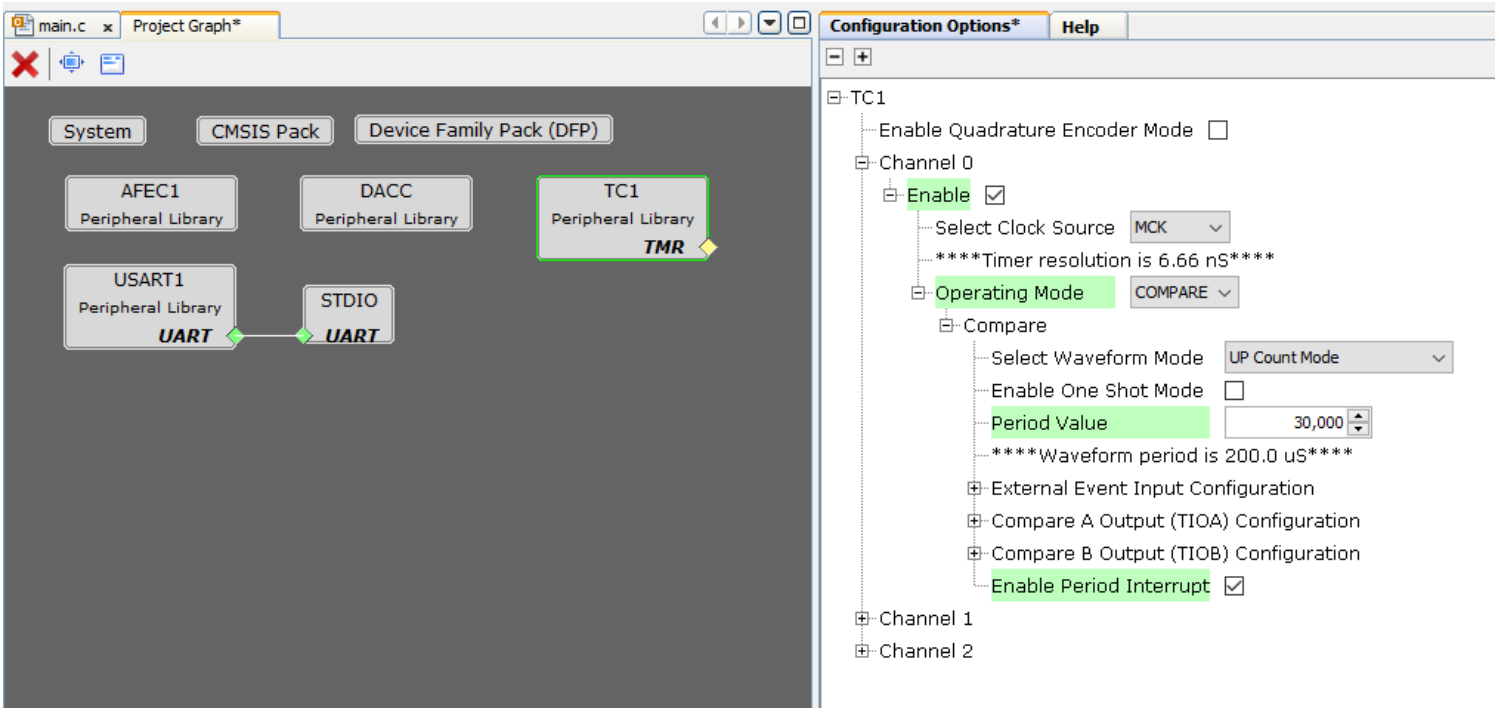
This topic provides information on MHC configurations

Description

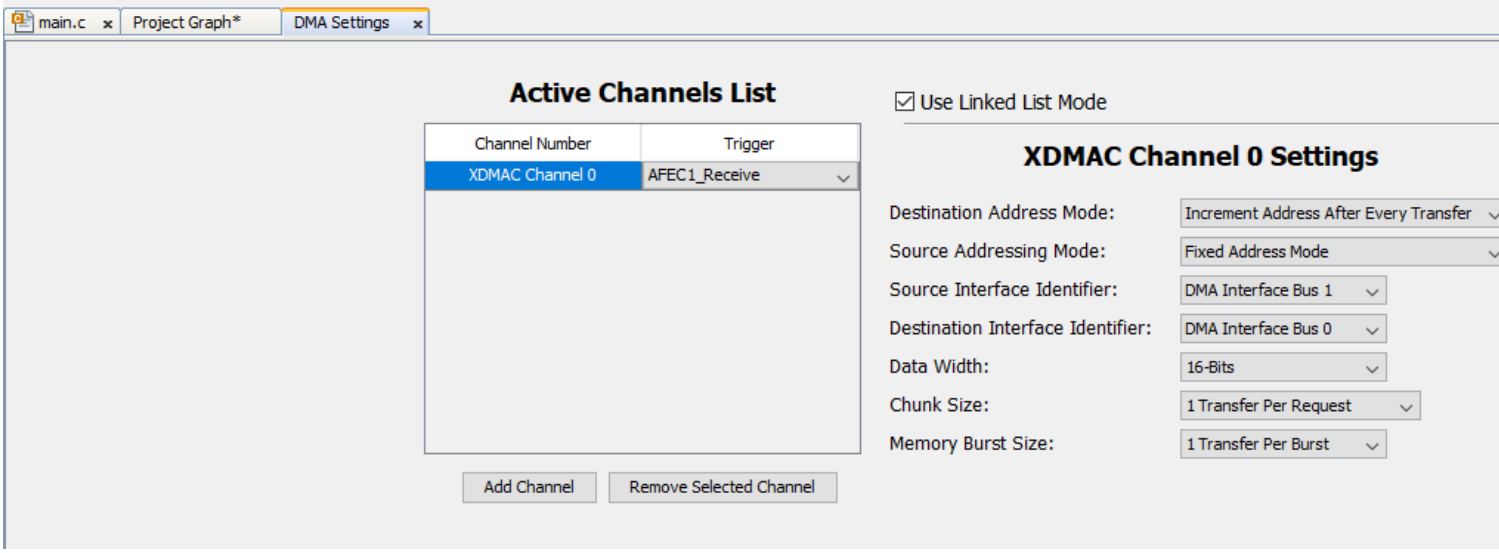
AFEC Configurations:



TC Configurations:



XDMAC Configurations:



Components Used

AFEC Peripheral Library

DACC Peripheral Library

TC Peripheral Library

USART Peripheral Library

XDMAC Peripheral Library

STDIO

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Analog input (AD0) is feed from DAC output. Sine wave output is generated on DAC output using 100 point lookup table.
       - Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
       - DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
       - Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

- Analog input (AD5) is connected to Vcc or user can supply input from external source
  - Analog input "AD5" is mapped to Port Pin "PC30" that is routed to "Pin 8 of EXT1 header".
  - Use jumper wire to connect "Pin 8 of EXT1 (AD5)" to Vcc or external voltage source

- Analog input (AD6) is connected to GND or user can supply input from external source
  - Analog input (AD6) is mapped to Port Pin "PC31" that is routed to "Pin 3 of EXT1 header".
  - Use jumper wire to connect "Pin 3 of EXT1" to GND or external voltage source

2. **Project sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Analog input (AD0) is feed from DAC output. Sine wave output is generated on DAC output using 100 point lookup table.
       - Analog input "AD0" is mapped to Port Pin "PB1" that is routed to "Pin 14 of EXT1 header".
       - DAC output "DAC0" is mapped to Port Pin "PB13" that is routed to "Pin 5 of J502 header".
       - Use jumper wire to Connect "Pin 14 of EXT1" to "Pin 5 of J502"

- Analog input (AD5) is connected to Vcc or user can supply input from external source
  - Analog input "AD5" is mapped to Port Pin "PC30" that is routed to "Pin 8 of EXT1 header".
  - Use jumper wire to connect "Pin 8 of EXT1 (AD5)" to Vcc or external voltage source

- Analog input (AD6) is connected to GND or user can supply input from external source
  - Analog input (AD6) is mapped to Port Pin "PC31" that is routed to "Pin 3 of EXT1 header".
  - Use jumper wire to connect "Pin 3 of EXT1" to GND or external voltage source

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port
2. Open Terminal application (Ex.:Tera term) on the computer
3. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Build and Program the application using MPLAB X IDE
5. Observe terminal application. Console displays the converted digital values and voltage by channel number
6. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: Console displays 100 converted values by channel number
sam_v71_xult.X	Success: Console displays 100 converted values by channel number

Cache Applications

This Section provides help for Cache maintenance Applications.

cache_maintenance

This example demonstrates cache maintenance operation by cleaning and invalidating cache for DMA buffers located in cacheable SRAM region

Description

This example application shows how to use the CMSIS cache maintenance APIs to avoid issues related to cache coherency when data cache is enabled.

The application uses USART and DMA PLIBs to demonstrate the cache maintenance APIs provided by CMSIS. It registers a callback with DMA transmit and receive channels. The application first transmits a message using DMA transmit channel and then schedules a read of 10 characters using DMA receive channel. Once the DMA read is complete, it reads the received data and echoes the same on the terminal using the DMA transmit channel.

The application calls SCB_CleanDCache_by_Addr API on the write buffer before transmitting it. Calling this API copies the data from cache memory to the main memory, thereby ensuring that the DMA peripheral uses the updated values in the write buffer.

The application calls SCB_InvalidateDCache_by_Addr API on the read buffer after reception of data is complete by the DMA receive channel. Calling this API invalidates the cache region corresponding to the read buffer thereby ensuring that the CPU reads the updated values in read buffer from the main memory and into the cache.

Also, since the cache maintenance operations are always performed on a cache line (1 cache line = 32 bytes), the read and write buffers must be aligned to 32 byte boundary and must be a multiple of 32 bytes. For the same reason, although in the application, the number of received and echoed back bytes is 10, the size of the receive and echo buffers is 32 bytes.

Building the Application

This section provides information on how to build and run the application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\cache\cache_maintenance\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

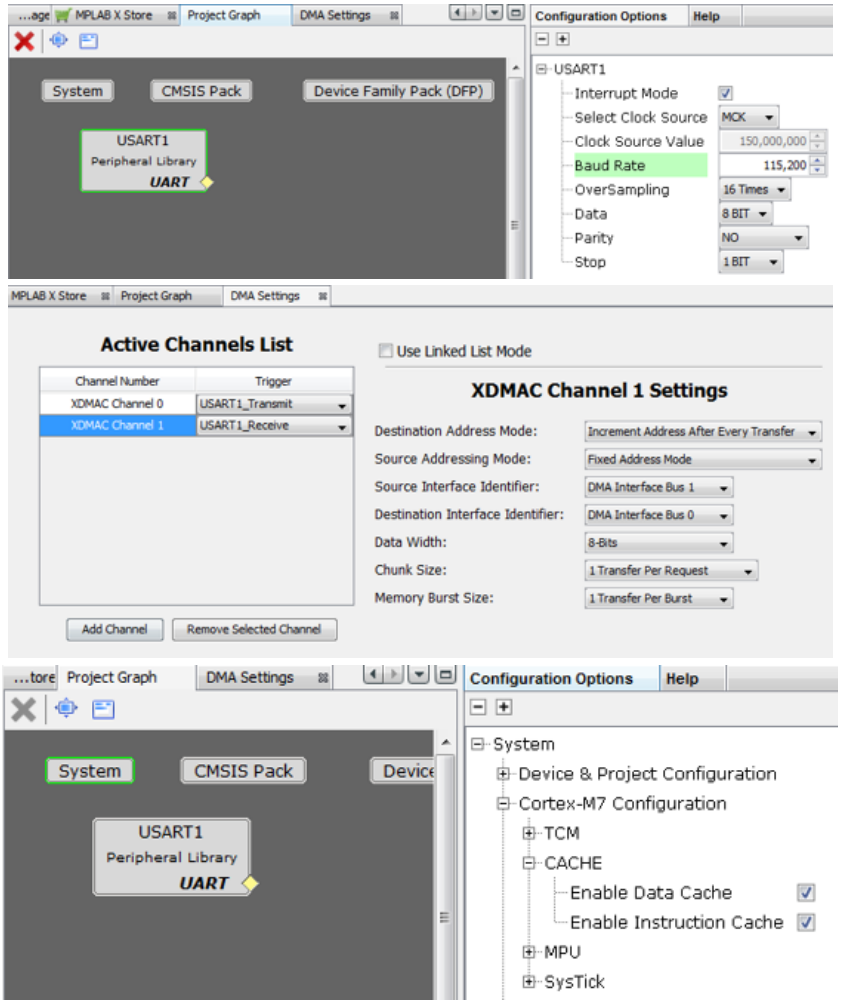
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Modules Used:

1. USART Peripheral Library
2. XDMAC Peripheral Library

MHC Settings:

1. Enable Data cache (Data cache is enabled by default)
2. Configure Pins for USART Peripheral in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**

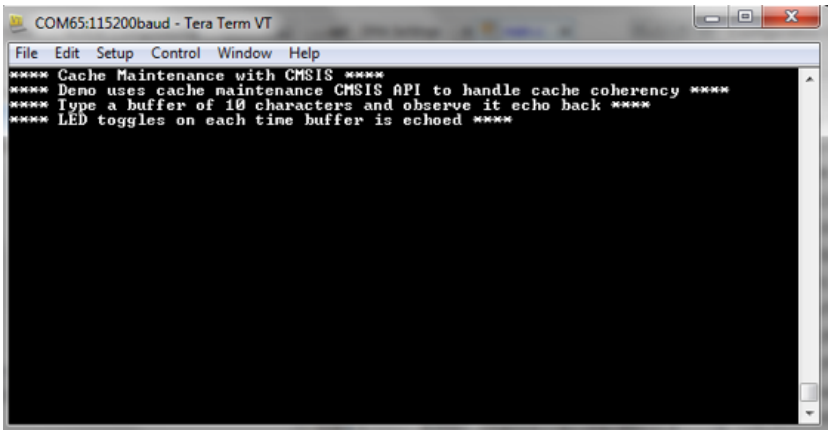
- Hardware Used:
  - SAM V71 Xplained Ultra Evaluation Kit
- Hardware Setup:
  - No special hardware setup is required

Running the Application

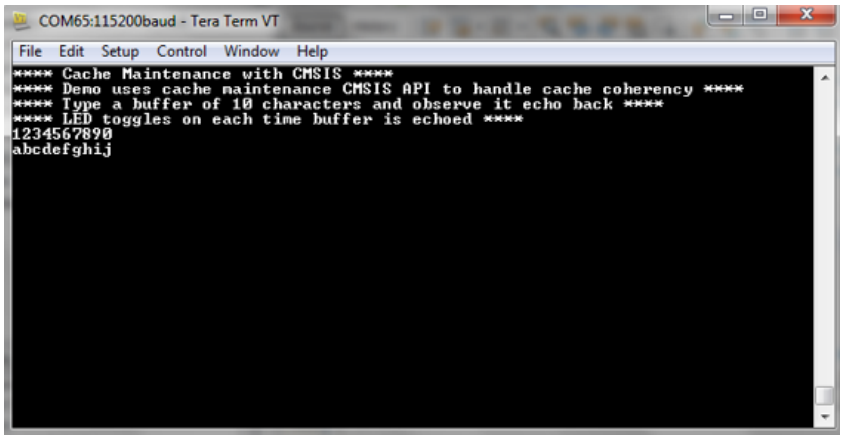
This section describes how to run the application.

Description

- **sam_e70_xult.X**
- **sam_v71_xult.X**
  1. Connect micro USB to DEBUG port.
  2. Build the cache_maintenance example code and program using MPLABX
  3. Open Terminal application (Ex.:Tera term) on the computer
  4. Connect to serial port and configure serial settings as follows
     - Baud : 115200
     - Data : 8 Bits
     - Parity : None
     - Stop : 1 Bit
     - Flow Control : None
  5. Console displays the following message



6. Enter 10 characters and observe it echo back. LED toggles each time the buffer is echoed back.



Clock Applications

This Section provides help for CLOCK PLIB Applications.

clock_config

This example application shows how to configure CLOCK to generate maximum frequency to the CPU. It also enables the Main clock to be output on a PORT pin.

Description

This example application shows how to use Clock Peripheral library to generate maximum frequency to the CPU. This project also enables the Main clock to be output on a PORT pin.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/clock/clock_config/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

Components Used

1. Clock Configuration
2. Pin Configuration

MHC Settings

Use "Clock Configuration" to setup maximum Main CPU clock and enable output clock signal on a PORT pin.

Configure PORT Pin function to output clock signal on an IO pin in "Pin Configuration".

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit

- **Hardware Setup**
  - Connect an oscilloscope to monitor PORT pin PB23 (Pin number #8 on EXT3 on SAM C21n Xplained Pro Evaluation Kit).

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Connect an oscilloscope to monitor PORT pin PA03 (J505 header on SAM E70 Xplained Ultra Evaluation Kit).

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Probe PORT pin PB23 (Pin number #8 on EXT3 on SAM C21n Xplained Pro Evaluation Kit) and observe a square waveform of 4 MHZ. Note: In this application demonstration the CPU is configured to run at 48 MHZ clock generated through GCLK Generator 0. The Input source to the CPU is also fed to GCLK Generator 1 and configured to generate an output clock of MainClock/12., i.e 48/12, 4 MHZ. The output of the GCLK Generator 1 is configured to be available on PORT pin PB23.
sam_e70_xult.X	Probe PORT pin PA03 (J505 header on SAM E70 Xplained Ultra Evaluation Kit) and observe a square waveform of 5 MHZ. Note: In this application demonstration the CPU is configured to run at 300 MHZ (HCLK) and master clock at 150 MHz (MCK). Master clock is used as peripheral clock. The operating frequency can be observed through the divided clock (300MHz/60 = 5 MHz) at PA03 pin.

DAC Applications

This Section provides help for DAC PLIB Applications.

dac_wav_gen

This example application shows how to use DAC to generate a 1 KHz Sine or Triangle wave with lookup table.

Description

This example application shows how to use DAC Peripheral library to generate a 1 KHz Sine or Triangle wave with lookup table. The DAC starts conversion on a trigger from a timer period interrupt configured to occur at every 10 microseconds. In this application the number of DAC samples per output wav (sine or triangle) sample is 100.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/dac/dac_wav_gen/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

1. DAC Peripheral Library
2. TC Peripheral Library
3. EIC Peripheral Library

MHC Settings

1. Use "TC0 Peripheral Library" component to configure timer period to 10 microseconds.
2. Use "EIC Peripheral Library" component to enable interrupts on switch press events.
3. Configure DAC to output the DAC VOUT pin using "DAC Peripheral Library" component.
4. Configure PORT Pin function to output DAC VOUT signal on an IO pin in "Pin Configuration".

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Connect an oscilloscope to monitor DAC pin PA02 (Pin DAC_VOUT on DAC Header on SAM C21n Xplained Pro Evaluation Kit).

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Probe PORT pin PA02 (Pin DAC_VOUT on DAC Header on SAM C21n Xplained Pro Evaluation Kit) and observe a sinusoidal waveform of 1 KHz. To observe a triangle waveform, Press the switch SW0. Subsequent pressing of switch SW0 toggles the output between sine wav and triangle wav.

dac_wav_gen_dma

This example application shows how to use DAC with DMA to generate 5 KHz sinusoidal waveform without CPU intervention.

Description

This example application shows how to use DAC Peripheral library to generate a 5 KHz sinusoidal waveform without CPU intervention. The DAC starts conversion on a timer period trigger configured to occur every 2 microseconds. The EVSYS (Event System) peripheral library is configured to set timer (TC0) as the generator and DAC as the user of an Event. At every 2 microseconds, a timer overflow event is generated and the DAC starts conversion of data in DAC "DATABUF" register at the reception of the timer overflow event. DMA is used to setup a linked list to transfer the sine wave samples from the lookup table to DAC "DATABUF" register. In this application the number of DAC samples per output wav (sine) sample is 100.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/dac/dac_wav_gen_dma/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

1. DAC Peripheral Library
2. TC Peripheral Library
3. EVSYS Peripheral Library

MHC Settings

1. Use "TC0 Peripheral Library" component to configure timer period to 2 microseconds and enable timer overflow event generation.
2. Use "EVSYS Peripheral Library" component to setup event generator and user.
3. Configure DAC to output the DAC VOUT pin using "DAC Peripheral Library" component and enable trigger conversion on event

detection.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Connect an oscilloscope to monitor DAC pin PA02 (Pin DAC_VOUT on DAC Header on SAM C21n Xplained Pro Evaluation Kit).

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Probe PORT pin PA02 (Pin DAC_VOUT on DAC Header on SAM C21n Xplained Pro Evaluation Kit) and observe a sinusoidal waveform of 5 KHz.

DACC Applications

This Section provides help for DACC PLIB Applications.

dacc_wav_gen

This example shows how to use DAC to generate 5 KHz Sinewave or Triangle with lookup table

Description

DACC Peripheral library is used to generate either Sine wave or Triangle wave, by writing Digital data to DACC in periodic callback on every Timer Counter (TC) Interrupt.

Building the Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/dacc/dacc_wav_gen/firmware

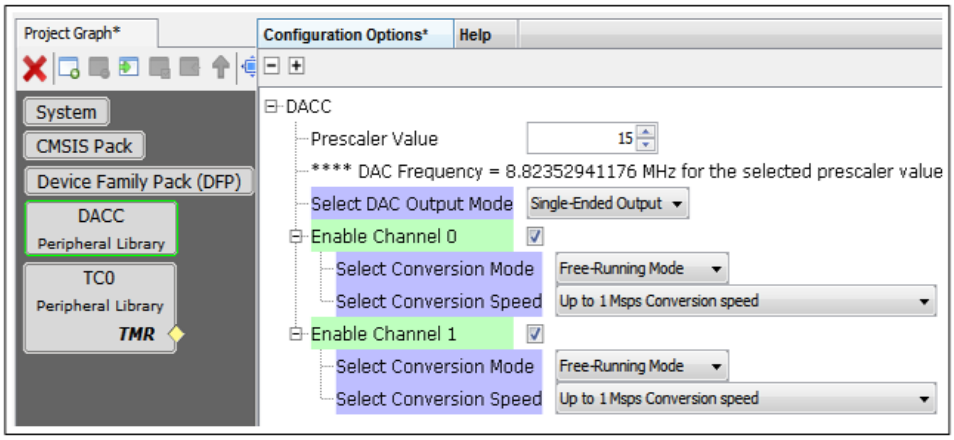
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This topic provides additional information how the library works.

Description

This topic provides information on MHC configurations



Components Used

1. DACC Peripheral Library
2. TC Peripheral Library

MHC Settings

1. Configure DACC and TC modules
2. Configure Pin for SWITCH

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Connect the DACC "Channel 0" and "Channel 1" output pin observable at PB13(J502) and PD00(J502) respectively to oscilloscope probe, to observe the analog output waveform.

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Connect the DACC "Channel 0" and "Channel 1" output pin observable at PB13(J504) and PD00(J504) respectively to

oscilloscope probe, to observe the analog output waveform.

Note: Refer to SAM V71 Xplained Ultra User Guide to know information about Hardware connector overview.

Running The Application

This section provides information on how to run the application

Description

1. Build and Program the application using MPLAB X IDE
2. Observe a Sine wave output in oscilloscope.
3. Press Switch on SAM E70 board, and observe a Triangle wave output.

dacc_wav_gen_dma

This example shows how to use DAC with DMA to generate sinusoidal waveform without CPU intervention

Description

DACC Peripheral library is used to in Trigger mode to generate a Sine wave.

On every Timer Counter (TC) Interrupt, DACC converts the digital data to analog data by reading it from memory using XDMAC.

Building the Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/dacc/dacc_wav_gen_dma/firmware

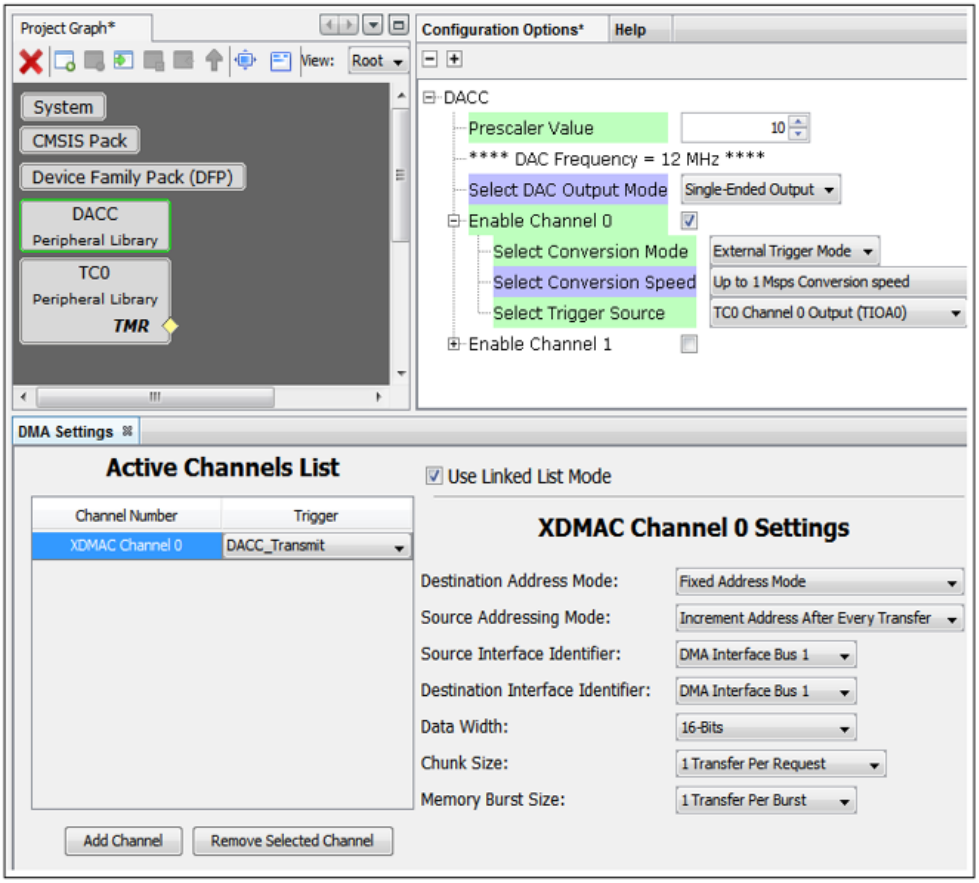
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This topic provides additional information how the library works.

Description

This topic provides information on MHC configurations



Components Used

1. DACC Peripheral Library
2. TC Peripheral Library
3. XDMAC Peripheral Library

MHC Settings

1. Configure DACC, TC and XDMAC modules

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Connect the DACC "Channel 0" output pin observable at PB13(J502) to oscilloscope probe, to observe the analog output waveform.

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - Connect the DACC "Channel 0" output pin observable at PB13(J504) respectively to oscilloscope probe, to observe the analog output waveform.

Note: Refer to SAM V71 Xplained Ultra User Guide to know information about Hardware connector overview.

Running The Application

This section provides information on how to run the application

Description

1. Build and Program the application using MPLAB X IDE
2. Observe a Sine wave output in oscilloscope.

DIVAS Applications

This Section provides help for DIVAS PLIB Applications.

divas_accelerate_computations

This example application demonstrates usage of DIVAS to accelerate computation.

Description

This example application shows how to use DIVAS Peripheral library to accelerate 32-bit signed, unsigned Integer division and square root computations using the hardware engine.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/divas/divas_accelerate_computations/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. DIVAS Peripheral Library
2. STDIO
3. SERCOM(USART) Peripheral Library

MHC Settings

1. Configure "DIVAS Peripheral Library".
2. Use "STDIO" and "SERCOM(USART) Peripheral Library" component to enable stdio functions.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	• Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. • The console output is as shown below. COM11 - Tera Term VT File Edit Setup Control Window Help ---------- SQUARE ROOT CALCULATION ---------- Input number = 144 Squareroot = 12 ---------- SIGNED INTEGER DIVSION ---------- Numerator = 10000 , Denominator = -2000 Quotient = -5 ---------- UNSIGNED INTEGER DIVSION ---------- Numerator = 10000 , Denominator = 2000 Quotient = 5 ---------- UNSIGNED INTEGER DIVSION WITH REMAINDER ---------- Numerator = 10001 , Denominator = 2000 Quotient = 5,remainder = 1

divas_c_operator_override

This example application demonstrates usage of DIVAS to override C division operator.

Description

This example application shows how to use DIVAS Peripheral library to override C division and modulus operator used in ARM, internally the EABI (Enhanced Application Binary Interface) calls are routed to DIVAS hardware engine to perform computations.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/divas/divas_c_operator_override/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. DIVAS Peripheral Library
2. STDIO
3. SERCOM(USART) Peripheral Library

MHC Settings

1. Configure "DIVAS Peripheral Library" and enable overload divide operator option.
2. Use "STDIO" and "SERCOM(USART) Peripheral Library" component to enable stdio functions.

Hardware Setup

This section describes how to configure the supported hardware.

Description

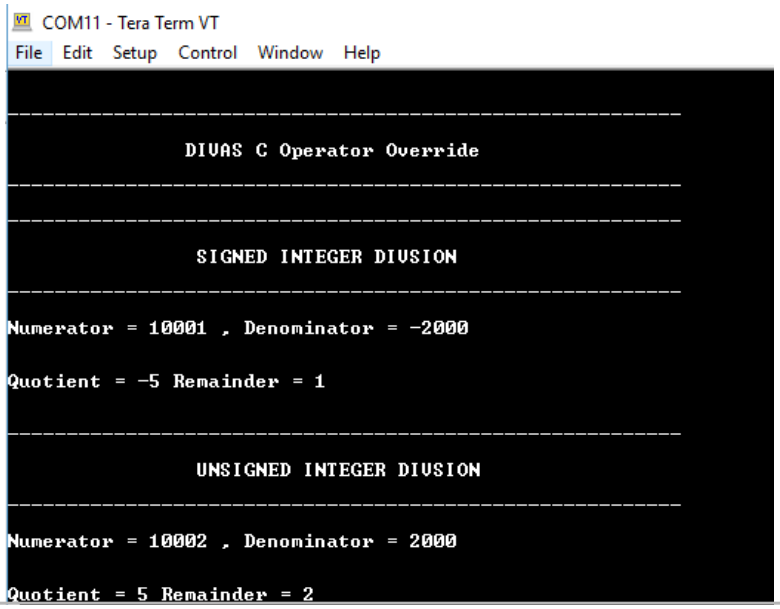
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	• Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. • The console output is as shown below. 

DMAC Applications

This Section provides help for DMAC PLIB Applications.

dmac_memory_transfer

This example application shows how to use DMA for memory to memory transfers.

Description

This example application shows how to use the DMAC Peripheral Library to perform a memory-memory transfer. The application uses software trigger to initiate a memory-memory transfer from source buffer to destination buffer 1 with Memory Beat Size set to 32-bits. Once the transfer is complete, it configures the DMAC PLIB for a Memory Beat Size of 16-bits and initiates another memory-memory transfer from source buffer to destination buffer 2. For each transfer, the application measures the time taken for the transfer in terms of SYS TICK counts and displays the time taken with Memory Beat Size 32-bits and Memory Beat Size 16-bits, on the console using the SERCOM4 USART PLIB and STDIO printf.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/dmac/dmac_memory_transfer/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit.

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

1. SERCOM Peripheral Library
2. DMAC Peripheral Library
3. STDIO Utility

MHC Settings

Use "SERCOM4 Peripheral Library" component to configure SERCOM as USART.

Use "DMAC Configuration" to setup DMAC for memory to memory data transfers.

Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.

Hardware Setup

This section describes how to configure the supported hardware.

Description

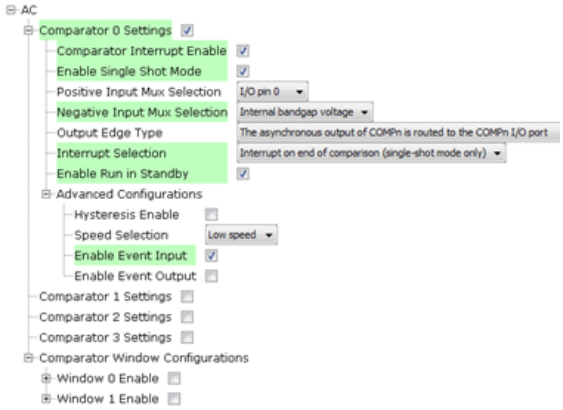
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware setup

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	-> Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. -> Console displays the following message, providing a comparison of time taken for the data transfer with beat size of 32-bits and 16-bits. 

dmac_usart_echo

This example application shows how to perform USART data transfer using DMAC.

Description

This example application shows how to perform serial data transfer using SERCOM (configured as USART) Peripheral Library and DMAC Peripheral Library. The application sets up 2 DMA channels. DMA channel 0 is setup to transfer data from memory to peripheral (SERCOM4 - USART Transmit) and DMA channel 1 is setup to transfer data from peripheral (SERCOM4 - USART receive) to memory.

This application displays a prompt message on the serial terminal, accepts a buffer of 10 characters and echoes it back to the serial terminal. An LED is toggled each time a buffer is echoed.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/dmac/dmac_usart_echo/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit.

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

1. SERCOM Peripheral Library
2. DMAC Peripheral Library

MHC Settings

Use "SERCOM4 Peripheral Library" component to configure SERCOM as USART.

Use "DMAC Configuration" to setup DMAC for memory to peripheral and vice versa data transfers.

Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.

Hardware Setup

This section describes how to configure the supported hardware.

Description

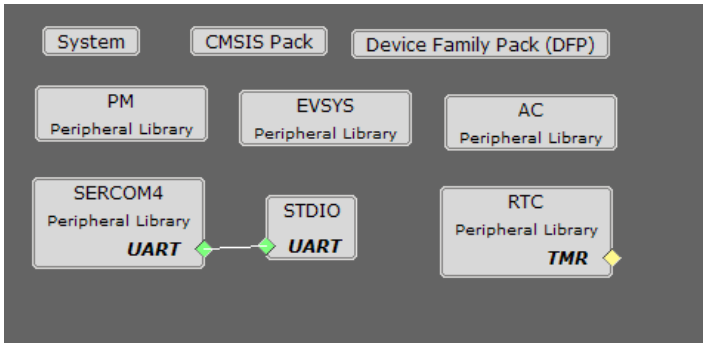
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware setup

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	-> Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. -> The console output is as shown below. 

DSU Applications

This Section provides help for DSU PLIB Applications.

dsu_crc32_generate

This example application shows how to use DSU Peripheral library to generate a CRC32 value on a memory block.

Description

This example application generates one CRC32 value using DSU Peripheral Libary and one CRC32 value using standard software logic on a constant array of random data and verify if both are matching.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/dsu/firmware

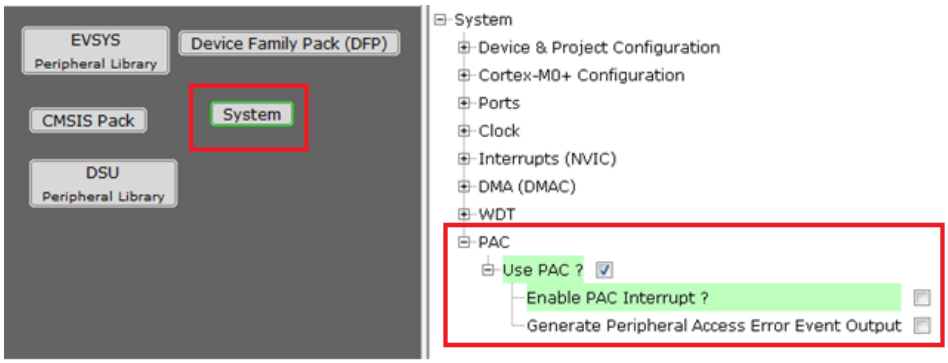
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. DSU Peripheral Library
2. PAC Peripheral Library

MHC Settings

1. Enable PAC Peripheral Library from system component.
2. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.
- **Project sam_d20_xpro.X**
  - **Hardware Used**
    - SAM D20 Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.
- **Project sam_d21_xpro.X**
  - **Hardware Used**
    - SAM D21 Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table
   1. **Success** : CRC32 Value generated by Software and DSU PLIB **matches**
   2. **Failure** : CRC32 Value generated by Software and DSU PLIB **Does not match**

Project Name	Output
sam_c21n_xult.X	Success : LED 0 **On** Failure : LED 0 **Off**
sam_d20_xpro.X	Success : LED 0 **On** Failure : LED 0 **Off**
sam_d21_xpro.X	Success : LED 0 **On** Failure : LED 0 **Off**

EFC Applications

This Section provides help for EFC PLIB Applications.

flash_read_write

This example application shows how to use EFC Peripheral library to Erase, write and read data on the On-Board flash memory.

Description

EFC Peripheral library is used to Erase, write and read data on the On-Board flash memory in Non-Interrupt Mode.

Building the Application

This section provides information on how to build application using MPLAB X IDE

Description

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/efc/flash_read_write/firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

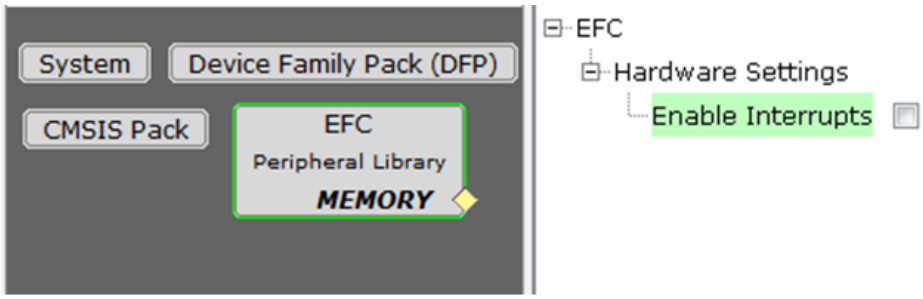
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This topic provides information on MHC configuration.

Description

The Project Graph below shows the MPLAB Harmony components that are included in this application for the all supported projects.



Main Components used in project configuration

1. EFC PLIB

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED 1 **ON**
sam_V71_xult.X	Success: LED 0 **ON**

EIC Applications

This Section provides help for EIC PLIB Applications.

eic_led_on_off

This example application shows how to generate interrupt using EIC on switch press and indicate it through LED.

Description

This example application shows how to generate interrupt using EIC Peripheral Library on switch press and indicate it through LED.

The application registers an event callback with EIC Peripheral library and toggles an LED for every switch press event.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/eic/eic_led_on_off/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

1. EIC Peripheral Library

MHC Settings

1. Use "EIC Peripheral Library" component to enable interrupts on switch press events.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	On every press of switch 'SW0' the application toggles the user LED 'LED0.

EVSYS Applications

This Section provides help for EVSYS PLIB Applications.

evsys_trigger

This example application shows how to use EVSYS Peripheral library to access and control of the associated peripheral instance.

Description

This example application shows how to use EVSYS Peripheral library to access and control of the associated peripheral instance.

In this application, when press and release switch for one time then LED will toggle continuously on development board. This is due to Switch routed GPIO pin will access and control LED routed GPIO pin through event system peripheral.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below,

Application Path	csp/apps/evsys/evsys_trigger/firmware

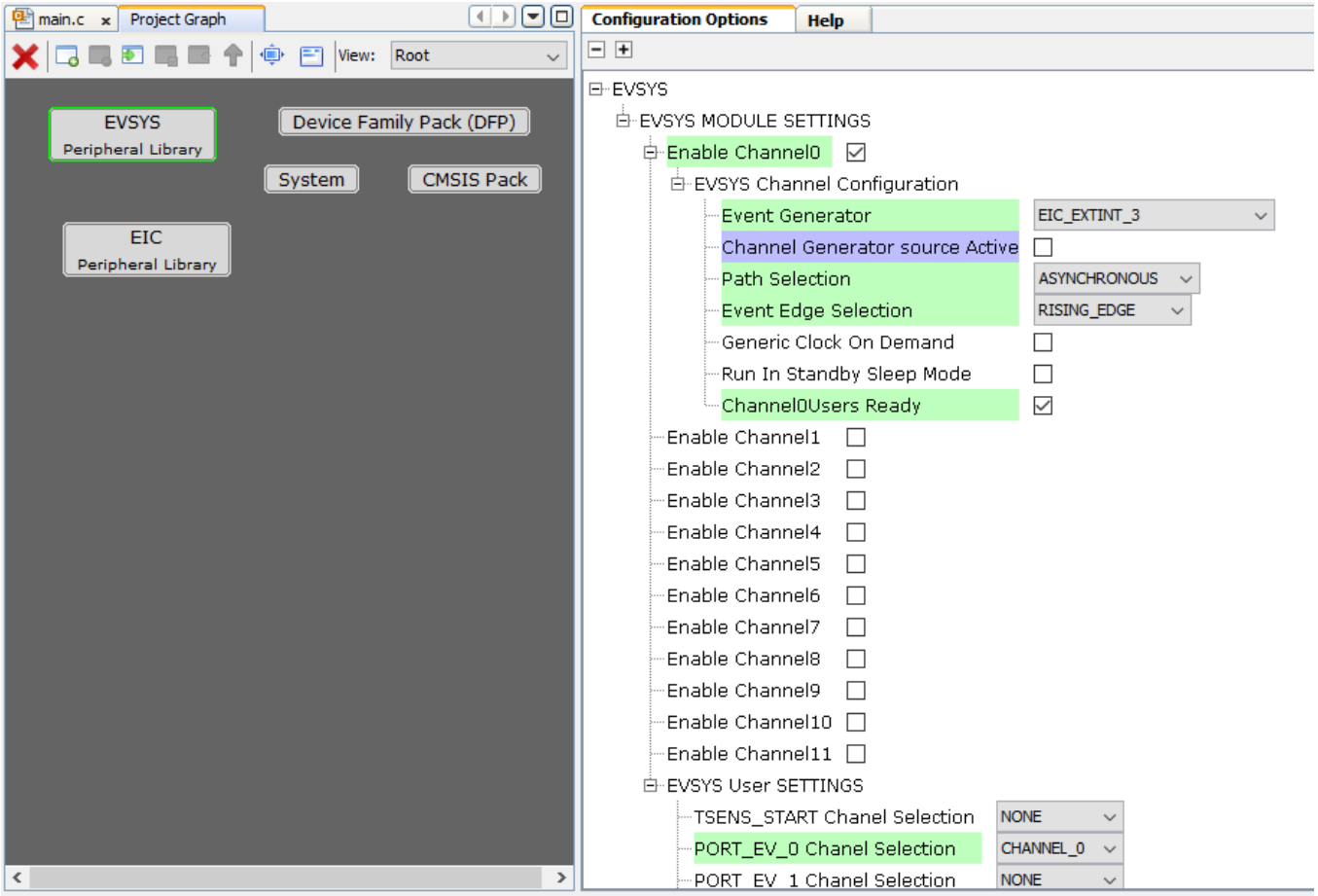
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

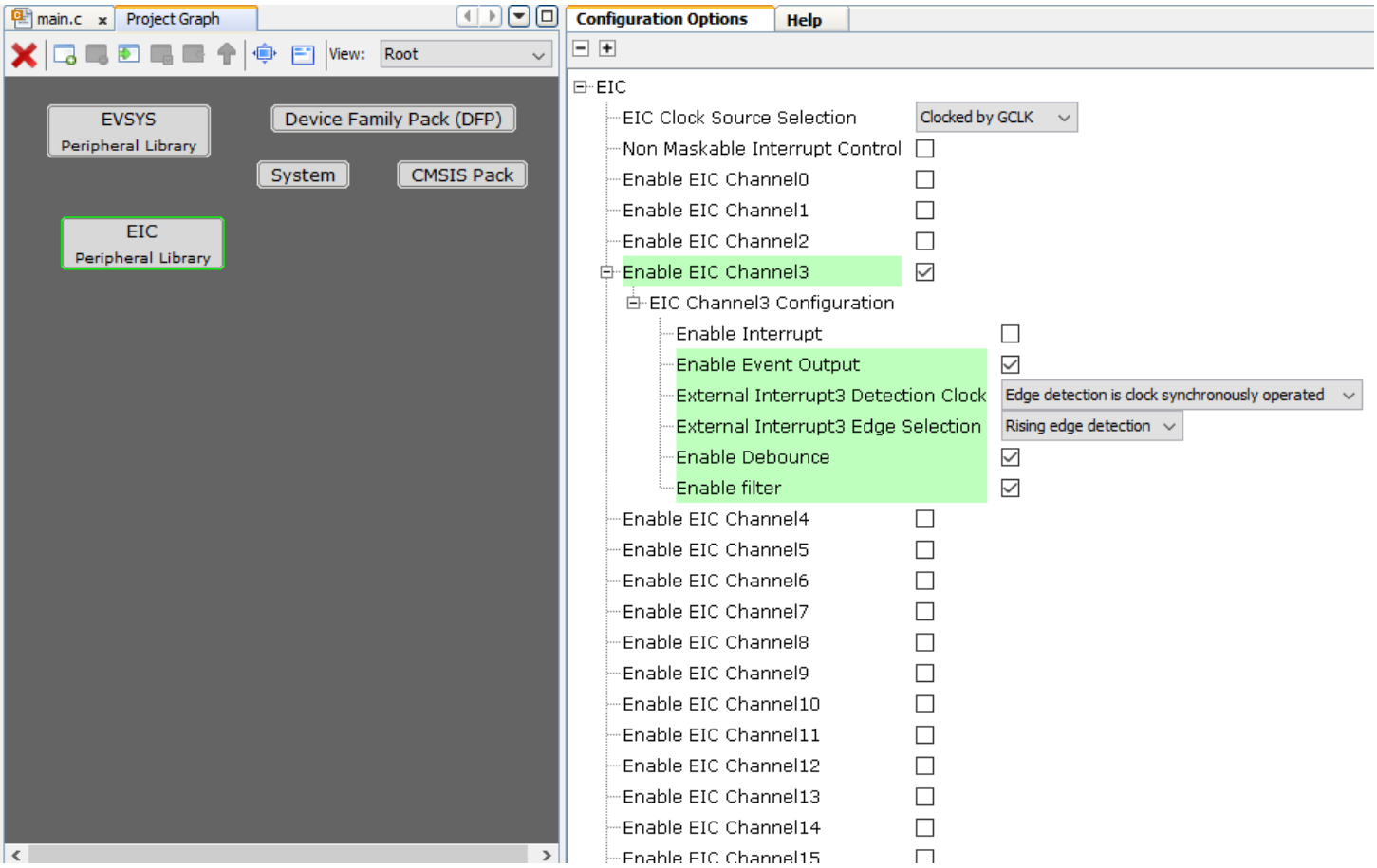
MPLAB Harmony Configurations

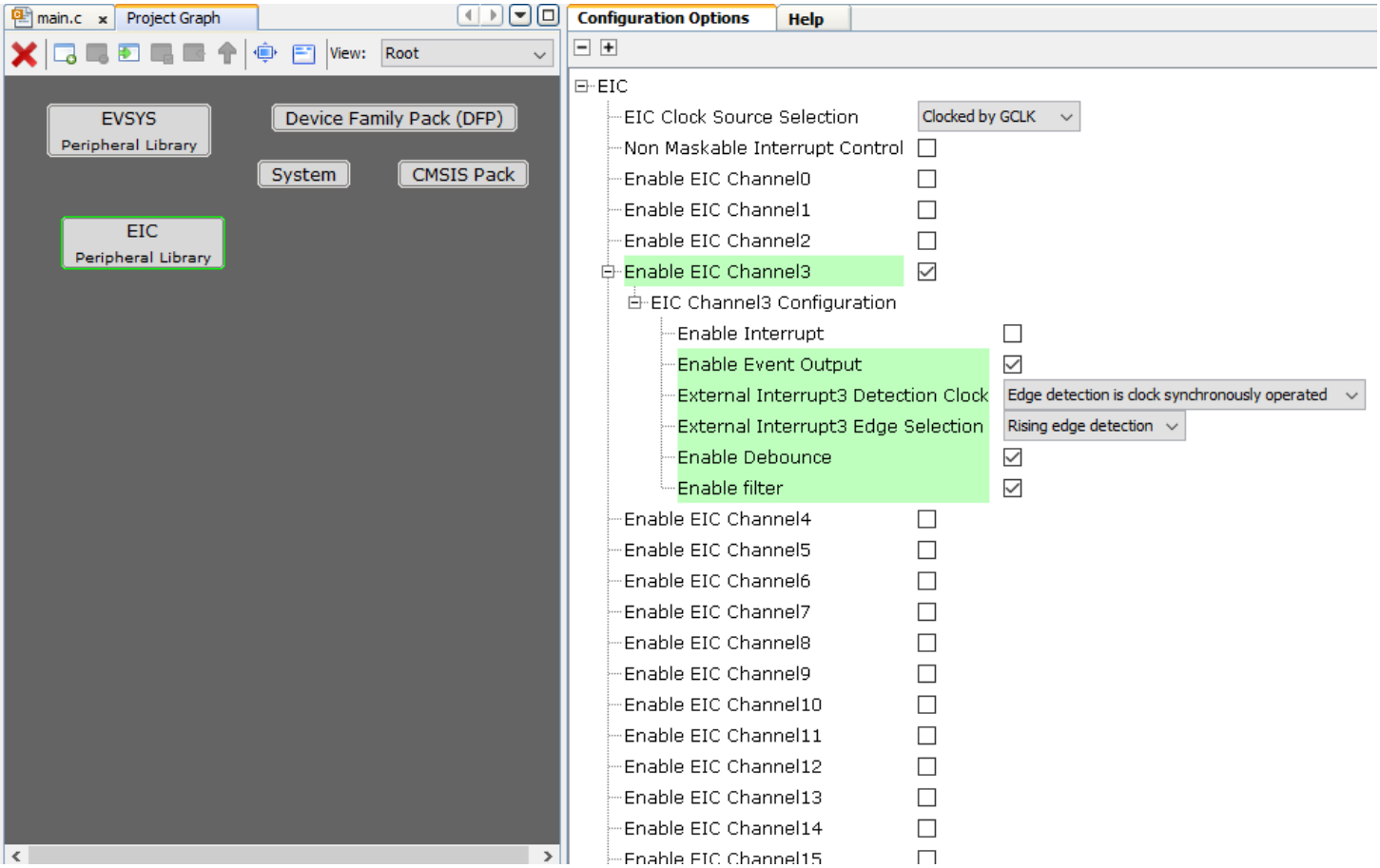
This topic provides information on MHC configurations.

Description

The Project Graph diagrams below show the Harmony components that are included in this application.


main.c
Project Graph
View: Root
EVSYS
Peripheral Library
Device Family Pack (DFP)
System
CMSIS Pack
EIC
Peripheral Library
Configuration Options
Help
EVSYS
EVSYS MODULE SETTINGS
Enable Channel0
EVSYS Channel Configuration
Event Generator
EIC_EXTINT_3
Channel Generator source Active
Path Selection
ASYNCHRONOUS
Event Edge Selection
RISING_EDGE
Generic Clock On Demand
Run In Standby Sleep Mode
Channel0Users Ready
Enable Channel1
Enable Channel2
Enable Channel3
Enable Channel4
Enable Channel5
Enable Channel6
Enable Channel7
Enable Channel8
Enable Channel9
Enable Channel10
Enable Channel11
EVSYS User SETTINGS
TSENS_START Chanel Selection
NONE
PORT_EV_0 Chanel Selection
CHANNEL_0
PORT_EV_1 Chanel Selection
NONE


main.c
Project Graph
View: Root
EVSYS
Peripheral Library
Device Family Pack (DFP)
System
CMSIS Pack
EIC
Peripheral Library
Configuration Options
Help
EIC
EIC Clock Source Selection
Clocked by GCLK
Non Maskable Interrupt Control
Enable EIC Channel0
Enable EIC Channel1
Enable EIC Channel2
Enable EIC Channel3
EIC Channel3 Configuration
Enable Interrupt
Enable Event Output
External Interrupt3 Detection Clock
Edge detection is clock synchronously operated
External Interrupt3 Edge Selection
Rising edge detection
Enable Debounce
Enable filter
Enable EIC Channel4
Enable EIC Channel5
Enable EIC Channel6
Enable EIC Channel7
Enable EIC Channel8
Enable EIC Channel9
Enable EIC Channel10
Enable EIC Channel11
Enable EIC Channel12
Enable EIC Channel13
Enable EIC Channel14
Enable EIC Channel15



Components Used

1. PORT Peripheral Library
2. EVSYS Peripheral Library
3. EIC Peripheral Library

MHC Settings

1. Configure GPIO pin for Switch.
2. Configure GPIO pin for LED.
3. Configure EIC for Switch port pin.
4. Configure EVSYS for Switch and LED port pins.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table,

Project Name	Output
sam_c21n_xpro.X	Success: LED Toggle Fail: LED OFF

FREQM Applications

This Section provides help for FREQM Applications.

freqm_measure_clock_freq

This example application shows how to use FREQM to measure the clock frequency.

Description

This example application shows how to use FREQM Peripheral library to measure 48MHz internal oscillator clock with respect to known 32KHz external clock.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/freqm/freqm_measure_clock_freq/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. FREQM Peripheral Library
2. STDIO
3. SERCOM (USART) Peripheral Library

MHC Settings

1. Configure and enable "FREQM Peripheral Library".
2. Use "STDIO" and "SERCOM4 (USART) Peripheral Library" components to enable stdio functions.
3. Configure 32KHz External Oscillator in clock section to generate 32KHz clock.

4. By Default 48MHz internal oscillator clock will be enabled.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

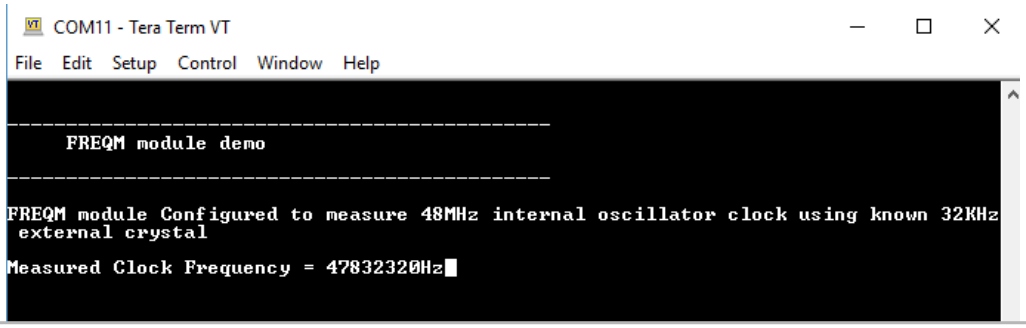
This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings

- Data : 8 Bits
- Baud :115200
- Parity : None
- Stop : 1 Bit
- Flow Control : None

3. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	• Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. • The console output is as shown below. 

MPU Applications

This Section provides help for MPU PLIB Applications.

mpu_coherent_region

This example application uses cache and demonstrates the cache maintenance using Memory Protection Unit.

Description

This example application uses cache and demonstrates the cache maintenance using MPU to create coherent or non-cacheable memory region.

This application accepts a buffer of characters and echoes it back over the console using USART Peripheral Library and XDMAC (DMA) PLIB. The transfer happens independent of CPU since it's a Direct Memory Access.

Building the Application

This section provides information on how to build application using MPLAB X IDE

Description

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/mpu/mpu_coherent_region/firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

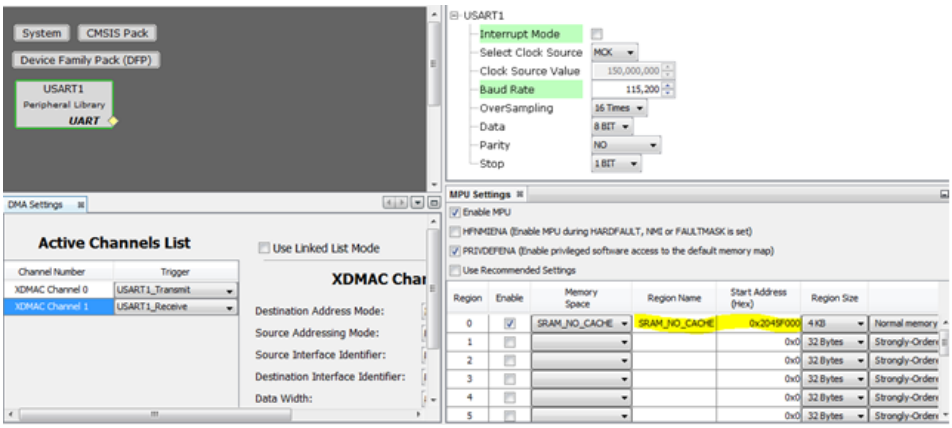
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This topic provides information on MHC configuration.

Description

The Project Graph below shows the MPLAB Harmony components that are included in this application for the all supported projects.



Main Components used in project configuration

1. USART PLIB
2. MPU PLIB
3. XDMAC PLIB

Other MHC Configurations:

1. Enable MPU for SRAM memory region
2. Enable DMA for USART Transmit and Receive.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup for above 2 projects**
  - Connect micro USB to DEBUG port
  - Open Terminal application (Ex.:Tera term) on the computer
  - Connect to serial port and configure serial settings as follows,
    - Baud : 115200
    - Data : 8 Bits
    - Parity : None
    - Stop : 1 Bit
    - Flow Control : None

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Program the example code using MPLABX
2. Console displays the following message

****** MPU Example to Create Coherent or Non-Cacheable Memory Region ******

****** Demo uses coherent or no-cache memory region to handle cache coherency ******

****** Type a buffer of 10 characters and observe it echo back ******

****** LED toggles on each time the buffer is echoed ******

3. Enter any 10 characters on the console
4. Entered characters will echo back and LED toggles
5. Repeat the steps

Project Name	Output
sam_e70_xult.X	Success: LED 1 Toggles
sam_V71_xult.X	Success: LED 0 Toggles

NVMCTRL Applications

This Section provides help for NVMCTRL PLIB Applications.

nvmctrl_flash_read_write

This example application shows how to use NVMCTRL to erase and program internal Flash memory.

Description

This example application shows how to use NVMCTRL to erase and program internal Flash memory. The application populates the RAM buffer with a known pattern and erases a flash row (256 bytes). It then writes a the RAM buffer to a flash page (64 bytes). The application compares the data in the RAM buffer with the data read from the flash page.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/nvmctrl/nvmctrl_flash_read_write/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

NVMCTRL Peripheral Library

MHC Settings

Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	Success: User LED 'LED0' glows ON. Failure: User LED 'LED0' glows OFF.

nvmctrl_flash_read_while_write

This example application shows programming RWWEE (Read While Write EEPROM emulation) area and reading user row.

Description

This example application shows programming RWWEE (Read While Write EEPROM emulation) area and reading user row.

The application performs the following steps to demonstrate RWEEE.

- > Populates SRAM buffer with a pattern 0x12 for writing into RWWEE page.
- > Erases the RWWEE row.
- > Program 64 byte page with pattern 0x12 to RWWEE section.
- > Populate another SRAM buffer with a pattern 0x04 for writing into main area page.
- > Erase the main row.
- > Program main area row witn data pattern 0x04.
- > Register callback to get event when erasing of RWWEE row is completed.
- > Perform erasing of RWWEE row
- > While the erasing of RWWEE row is under progress, perform read from main area
- > If the read from main area is successful, check if erase of RWWEE is completed.
- > Verifies that the data in RWWEE is erased, if the RWWEE erase is successful toggle LED ata every 1 second.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/nvmctrl/nvmctrl_flash_read_while_write/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The following Harmony components are used in this application.

Components Used

NVMCTRL Peripheral Library

SysTick Peripheral Library

MHC Settings

Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.

Use "NVMCTRL Peripheral Library" to enable callback for read completion events.

Use "System" to setup SysTick timer.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	Success: User LED 'LED0' toggles every second. Failure: User LED 'LED0' glows ON or OFF.

PIO Applications

This section provides help for PIO PLIB applications.

pio_led_on_off_interrupt

This example shows how to use PIO Peripheral library to turn on and off an LED in interrupt mode.

Description

After initialization, this application registers a callback function with PIO PLIB, enable the interrupt and wait for interrupt to occur i.e switch to press/release. Once switch is pressed (or released), interrupt occurs and the registered callback function is called. Application has also defined the callback function in which LED is toggled based on switch pressed or released state.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp/apps/pio/pio_led_on_off_interrupt/firmware

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This section provides information on MHC configuration used to create this application.

Description

Following Components are used to create this application:

1. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. GPIO pin is configured for LED in Pin Manager
2. GPIO pin is configured for switch and interrupt in Pin Manager

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	• Press Switch 1 on SAME70 Xplained Ultra board to see LED glow • When the switch is released, LED turns off • Repeat the above steps
sam_v71_xult.X	• Press Switch 1 on SAME70 Xplained Ultra board to see LED glow • When the switch is released, LED turns off • Repeat the above steps

pio_led_on_off_polling

This example shows how to use PIO Peripheral library to turn on and off an LED.

Description

This example demonstrates how to toggle an LED when a switch is pressed on the board. Whether switch is pressed or not, is continuously polled in the example.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp/apps/pio/pio_led_on_off_polling/firmware

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This section provides information on MHC configuration used to create this application.

Description

Following Components are used to create this application:

1. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. GPIO pin is configured for LED in Pin Manager
2. GPIO pin is configured for switch in Pin Manager

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**

    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	• Press Switch 1 on SAME70 Xplained Ultra board to see LED glow • When the switch is released, LED turns off • Repeat the above steps
sam_v71_xult.X	• Press Switch 1 on SAME70 Xplained Ultra board to see LED glow • When the switch is released, LED turns off • Repeat the above steps

PM Applications

This Section provides help for PM PLIB Applications.

pm_wakeup_eic

This application demonstrates entering low power modes (IDLE, STANDBY) and exiting it using EIC.

Description

This example application shows how to use PM (Power Manger) and EIC (External Interrupt controller) Peripheral library to enter into low power modes and exiting it using EIC.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/pm/pm_wakeup_eic/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. PM Peripheral Library
2. STDIO
3. SERCOM (USART) Peripheral Library
4. EIC Peripheral Library
5. SUPC Peripheral Library

MHC Settings

1. Configure and enable "PM Peripheral Library".
2. Use "STDIO" and "SERCOM4 (USART) Peripheral Library" components to enable stdio functions.
3. Configure "EIC Peripheral Library" to generate a interrupt whenever a switch(SW0) is pressed.
4. Use "Pin Configuration" to setup PORT pin PC05 to act as an LED and PB19 to act as External interrupt pin.
5. Configure voltage regulator to run in low power mode in "SUPC Peripheral Library".

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

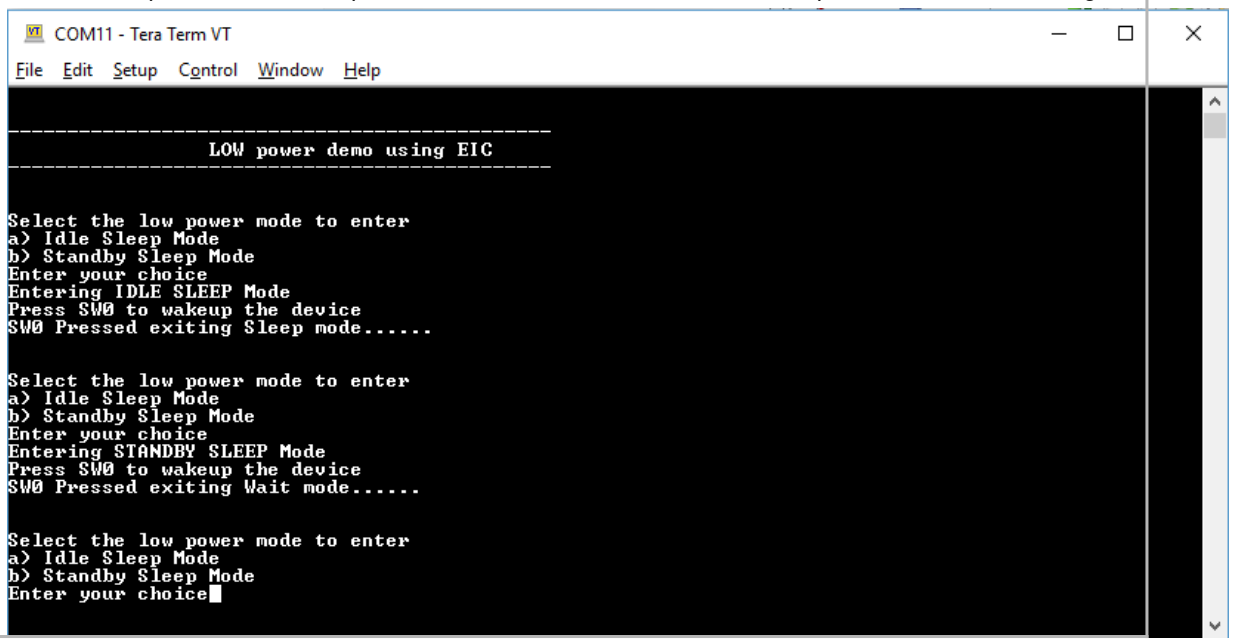
This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings

• Baud : 115200
• Data : 8 Bits
• Parity : None
• Stop : 1 Bit
• Flow Control : None

3. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	1. Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. 2. LED0 will continuously blink until device enters low power modes. 3. The console displays the available options as shown below. **Select the low power mode to enter** **a) Idle Sleep Mode** **b) Standby Sleep Mode** **Enter your choice** 4. Select 'a' or 'b' to enter low power mode. After entering low power mode LED0 will switch off. 5. To wakeup/exit from the sleep. Press SW0 on the board, device wakes up and LED0 starts blinking. 

pm_wakeup_rtc

This example demonstrates entering low power modes (IDLE, STANDBY) and exiting it using RTC

Description

This example application shows how to use PM (Power Manger) and RTC (Real Time Counter) Peripheral library to enter into low power modes and exiting it using RTC.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/pm/pm_wakeup_rtc/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. PM Peripheral Library
2. STDIO
3. SERCOM (USART) Peripheral Library
4. RTC Peripheral Library
5. SUPC Peripheral Library

MHC Settings

1. Configure and enable "PM Peripheral Library".
2. Use "STDIO" and "SERCOM4 (USART) Peripheral Library" components to enable stdio functions.
3. Configure "RTC Peripheral Library" to wakeup the device whenever a timer expires.
4. Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.
5. Configure voltage regulator to run in low power mode in "SUPC Peripheral Library".

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

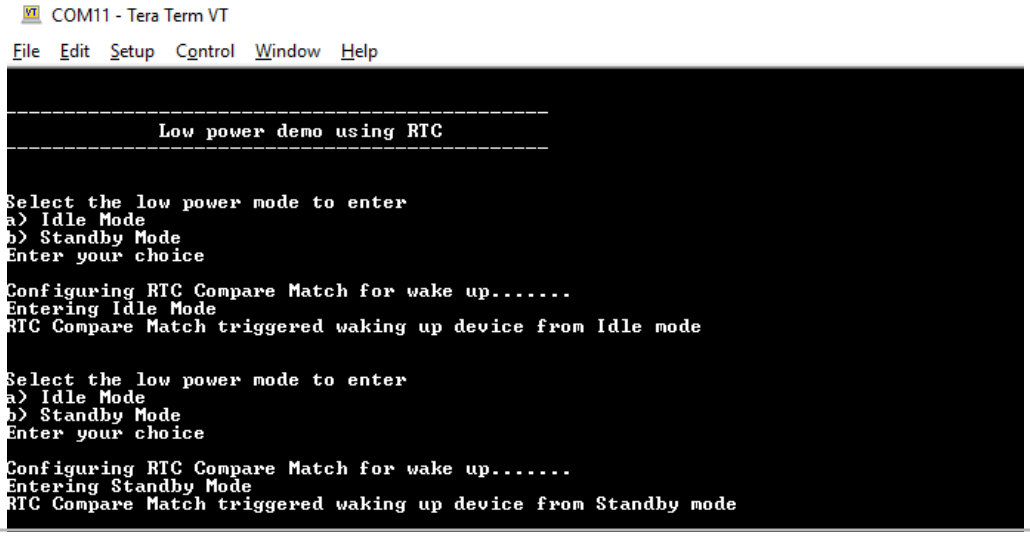
This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings

• Baud : 115200
• Data : 8 Bits
• Parity : None
• Stop : 1 Bit
• Flow Control : None

3. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	1. Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. 2. LED0 will continuously blink until device enters low power modes. 3. The console displays the available options as shown below. **Select the low power mode to enter** **a) Idle Sleep Mode** **b) Standby Sleep Mode** **Enter your choice** 4. Select 'a' or 'b' to enter low power mode. After entering low power mode LED0 will switch off. 5. After timer expires device wakes and LED0 Starts blinking. 

PORT Applications

This Section provides help for PORT PLIB Applications.

port_led_on_off_polling

This example shows how to read the switch status and indicate it through LED.

Description

This example application shows how to use PORT Peripheral library to control and interact with PORT I/O Pin controller. In this demonstration when ever user press switch on development board, LED0 glows ON and when releases the switch LED0 glows OFF. In this application, switch port pin status is read by polling.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below,

Application Path	csp/apps/port/port_led_on_off_polling/firmware

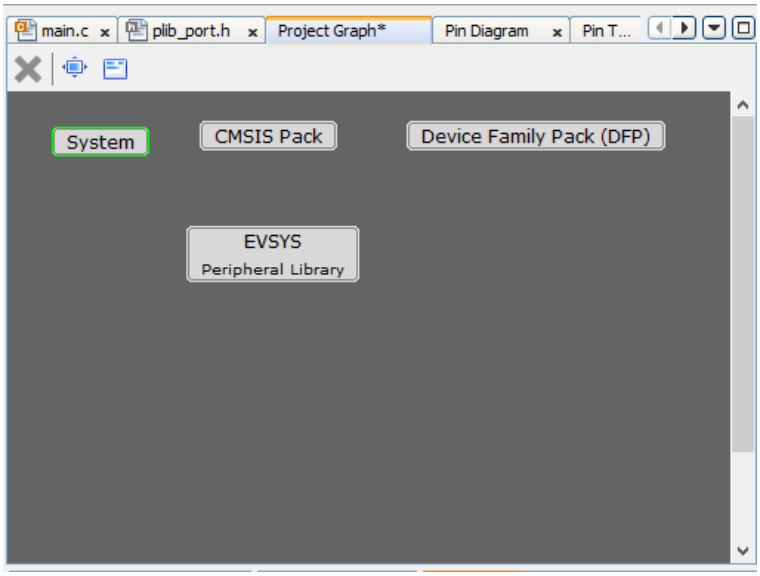
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. PORT Peripheral Library

MHC Settings

1. Configure GPIO pin for Button.
2. Configure GPIO pin for LED.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.

2. Observe Output as described in the below table,

Project Name	Output
sam_c21n_xpro.X	Switch Pressed State: LED0 ON Switch Released State: LED0 OFF

PWM Applications

This Section provides help for PWM PLIB Applications.

pwm_synchronous_channels

This section provides information on how the pwm_synchronous_channels application is configured, built and works.

Description

This example shows how to configure PWM to generate 3 PWM signals with dead time. (used for motor control)The duty cycle of the PWM is updated in the interrupt handler.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/pwm/pwm_synchronous_channels/firmware

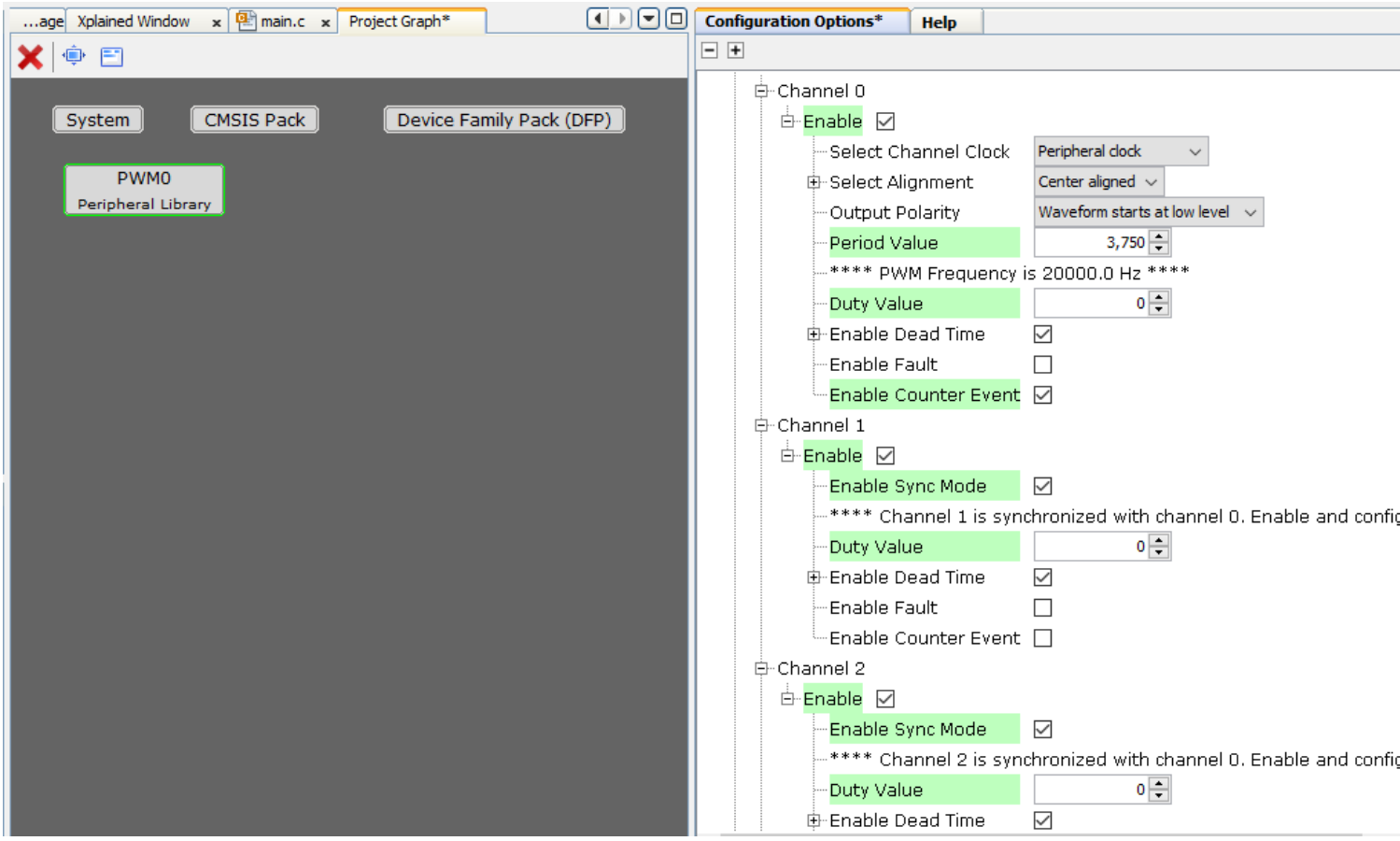
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

PWM Configurations:



Components Used

PWM Peripheral Library

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe high-side and low-side PWM waveforms on oscilloscope. Observe dead time between high side and low side.
   - Channel 0
     - Pin 7 of EXT1 - PWM0_PWMH0 (PA00)
     - Pin 4 of EXT1 - PWM0_PWML0 (PA19)
   - Channel 1
     - Pin 9 of EXT2 - PWM0_PWMH1 (PA02)
     - Pin 15 of EXT1 - PWM0_PWML1 (PD25)
   - Channel 2
     - Pin 7 of EXT2 - PWM0_PWMH2 (PC19)
     - Pin 8 of EXT2 - PWM0_PWML2 (PD26)
3. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: Observe high side and low side PWM waveforms. PWM frequency = 20KHz
sam_v71_xult.X	Success: Observe high side and low side PWM waveforms. PWM frequency = 20KHz

QSPI Applications

This Section provides help for QSPI PLIB Applications.

qspi_read_write

This example application shows how to use QSPI Peripheral library to Erase, write and read data on the On-Board QSPI flash memory device.

Description

QSPI Peripheral library is used to perform Erase, write and read operations on the On-Board SST26VF032B QSPI flash memory device.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/qspi/qspi_read_write/firmware/

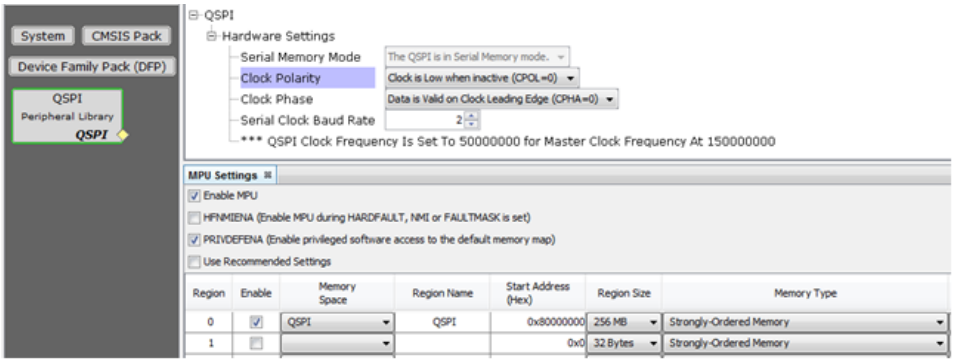
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. QSPI Periheral Library
2. Systick Peripheral Library
3. MPU Peripheral Library

MHC Settings

1. Enable MPU for QSPI Memory Region
2. Configure Pins for QSPI Peripheral in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED 1 **Blinks** Failure: LED 1 **ON**

qspi_xip

This example application shows how to use QSPI Peripheral library to perform execute in place(XIP) from the On-Board QSPI flash memory device.

Description

QSPI Peripheral library is used to perform execute in place(XIP) from the On-Board SST26VF032B QSPI flash memory device.

The application is demonstrated by using two different projects.

xip_image Application

This example is used to generate a **xip_image_pattern_hex** to run from the On-Board QSPI Flash (SST26VF032B) memory device.

It uses a custom linker file to place the binary generated into the QSPI Flash memory region.

xip_main Application

This example application shows how to use QSPI PLIB to program and execute **xip_image_pattern_hex** built to run from On-Board QSPI Flash (SST26VF032B) memory device.

Once program and verification is successfull, It jumps to the reset handler of **xip_image** application programmed in QSPI Flash memory to perform execute in place.

Building The Application

This section provides information on how to build the application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/qspi/qspi_xip/

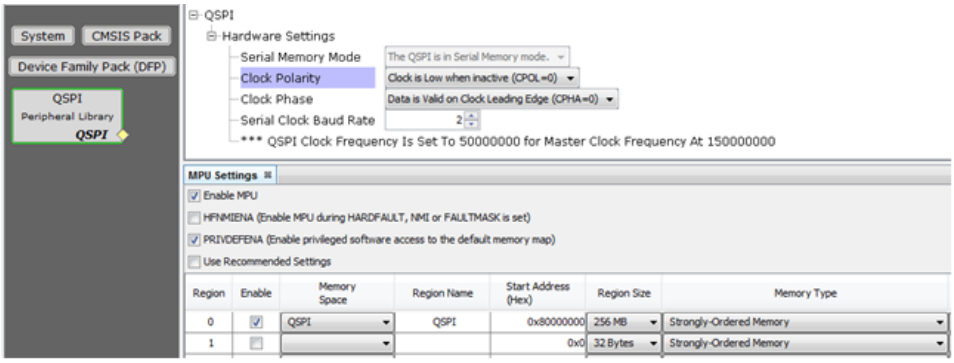
Project Name	Description
xip_main/firmware/sam_e70_xult.X	SAM E70 Xplained Ultra board
xip_image/firmware/sam_e70_xult.X	SAM E70 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in the **xip_main** application.



Components used by xip_image application

1. Systick Peripheral Library

Components used by xip_main application

1. QSPI Periheral Library
2. MPU Peripheral Library

MHC Settings for xip_main Application

1. Enable MPU for QSPI Memory Region
2. Configure Pins for QSPI Peripheral in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project xip_main/firmware/sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project xip_image/firmware/sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build the **xip_image** example using MPLAB X IDE and **do not program**.
2. Run **xip_image_pattern_gen.py** script to generate xip_image_pattern_hex.h
   1. python xip_image_pattern_gen.py ./xip_image/firmware/sam_e70_xult.X/dist/default/production/sam_e70_xult.X.production.bin

3. Copy the generated **xip_image_pattern_hex.h** to ./**xip_main**/firmware/src/config/sam_e70_xult to be programmed and executed from QSPI flash memory
4. Program and run the **xip_main** example code using MPLAB X IDE
5. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED 1 **Blinks** Failure: LED 1 **ON**

RSTC Applications

This Section provides help for RSTC PLIB Applications.

led_switcher

This example application shows how to use Reset Controller (RSTC) PLIB to generate an interrupt instead of generating a reset.

Description

RSTC Peripheral library is used to generate an interrupt instead of generating a reset, when the Reset switch is pressed on board.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/rstc/led_switcher/firmware

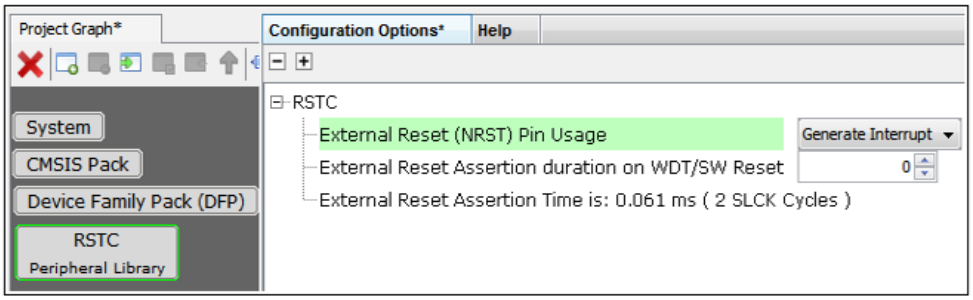
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. RSTC Peripheral Library

MHC Settings

1. Configure RSTC module
2. Configure Pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE
2. Press "RESET" Switch, and observe LED will toggle, instead of resetting the device.

rstc_reset_cause

This example shows how to use RSTC to identify the source of RESET.

Description

This example application shows how to use RSTC Peripheral library to identify the source of Device Reset and display the reset source type on a serial terminal.

In this demonstration below resets can able to emulate,

1. Press RESET button to emulate External Reset.
2. Press SW0 button to emulate Watchdog Timer Reset.
3. Load and run in debug mode to emulate System Reset.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below,

Application Path	csp/apps/rstc/rstc_reset_cause/firmware

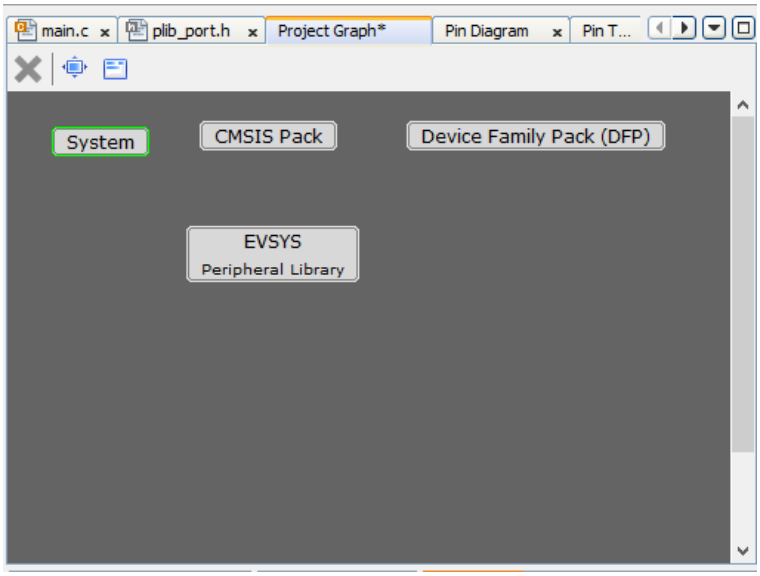
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. RSTC Peripheral Library
2. EIC Peripheral Library
3. SERCOM Peripheral Library

MHC Settings

1. Configure RSTC
2. Configure EIC for Switch SW0
3. Configure SERCOM as STDIO UART component

Hardware Setup

This section describes how to configure the supported hardware.

Description

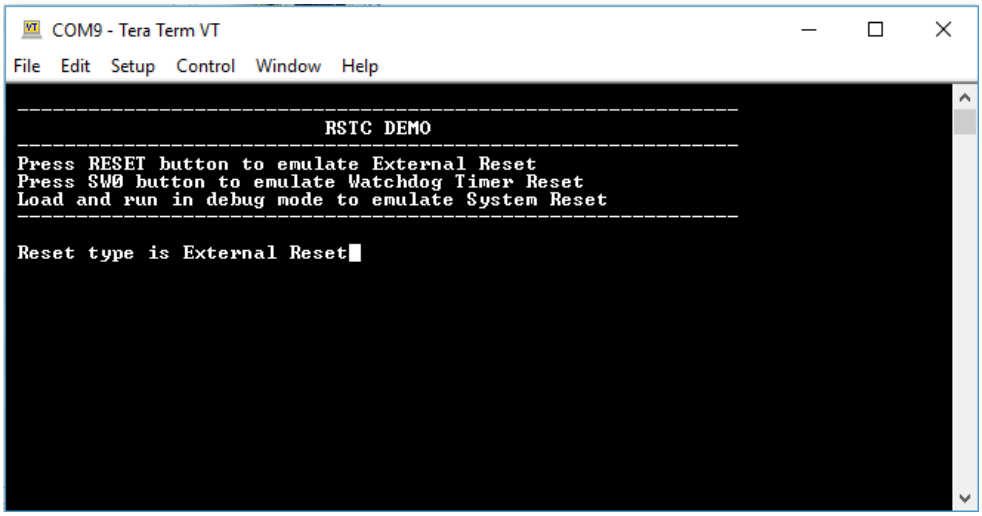
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. Console displays the reset cause like below,



RSWDT Applications

This Section provides help for RSWDT PLIB Applications.

rswdt_timeout

This example demonstrates Reinforced Safety Watchdog Timer reset by not feeding the watchdog.

Description

This example demonstrates Reinforced Safety Watchdog Timer reset by not feeding the watchdog on switch press

Building The Application

This section provides information on how to build the application using MPLAB X IDE.

Description

To build this application, open the below listed project file in MPLABX IDE.

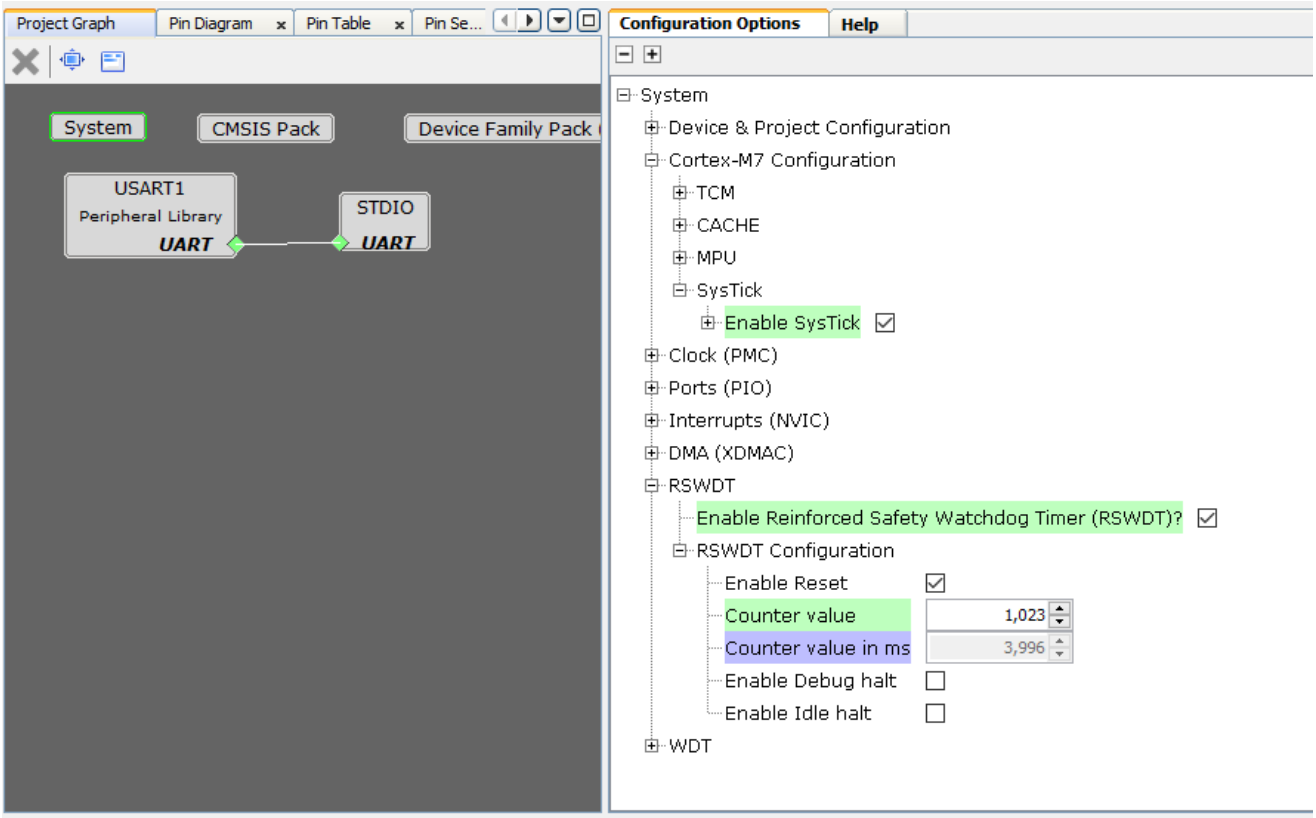
Project Path	Project Name	Description
\csp\apps\rswdt\rswdt_timeout\firmware	sam_e70_xult.X	SAM E70 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. RSWDT Periheral Library
2. Systick Peripheral Library
3. USART1/UART-STDIO Peripheral Library

MHC Settings

1. Configure Pins for USART1/UART Peripheral in Pin Settings
2. Configure GPIO pin for LED and SWITCH
3. Enable SysTick in Cortex-M7 Configuration

Hardware Setup

Hardware Used

1. SAM E70 Xplained Ultra Evaluation Kit

Setup

- No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Connect micro USB to DEBUG port of SAME70 Xplained Ultra
2. Open Terminal application (Ex.:Tera term) on the computer
   - Connect to serial port and configure serial settings as follows,
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Build and Program the application using MPLAB X IDE
4. Observe the on board LED and Terminal Application
   1. LED0 should start blinking
   2. Console displays the following message
      - ---
      - RSWDT DEMO
      - ---
   3. Press SW0 to emulate deadlock
   4. Press SW0 to put system in deadlock
   5. LED0 should stop blinking and console should print the following message
      - Emulating deadlock.....
      - RSWDT should reset device in 4 seconds
   6. RSWDT will reset the device in 4 seconds and demo should restart

RTC Applications

This section provides help for RTC PLIB Applications.

rtc_periodic_timeout

This example application shows how to use RTC to generate periodic interrupts.

Description

This example application shows how to use RTC Peripheral Library to generate periodic interrupts. The application registers a periodic timeout callback. It toggles an LED every time the callback is triggered.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/rtc/rtc_periodic_timeout/firmware/

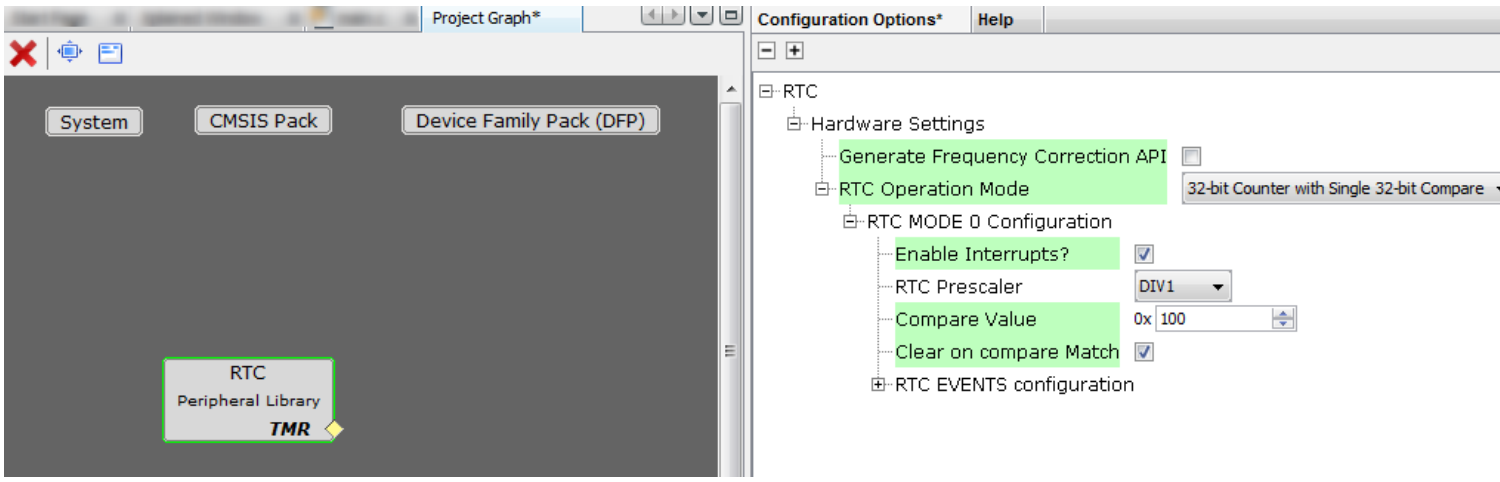
Project Name	Description
sam_c21n_xpro	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application for the **sam_c21n_xpro.X** project.



Components Used

1. RTC Periheral Library

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware setup

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

Running the Application

1. Build and Program the application using MPLAB X IDE.
2. Observe output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Success: User LED 'LED0' toggles at 10 millisecond rate. Success: User LED 'LED0' is off.

rtc_alarm

This example application shows how to use RTC to configure time and generate alarm.

Description

This example application shows how to use RTC Peripheral library to configure time and generate alarm. The application sets up system time and sets up an alarm to be generated after 20 seconds. The application displays a message on the alarm trigger.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/rtc/rtc_alarm/firmware/

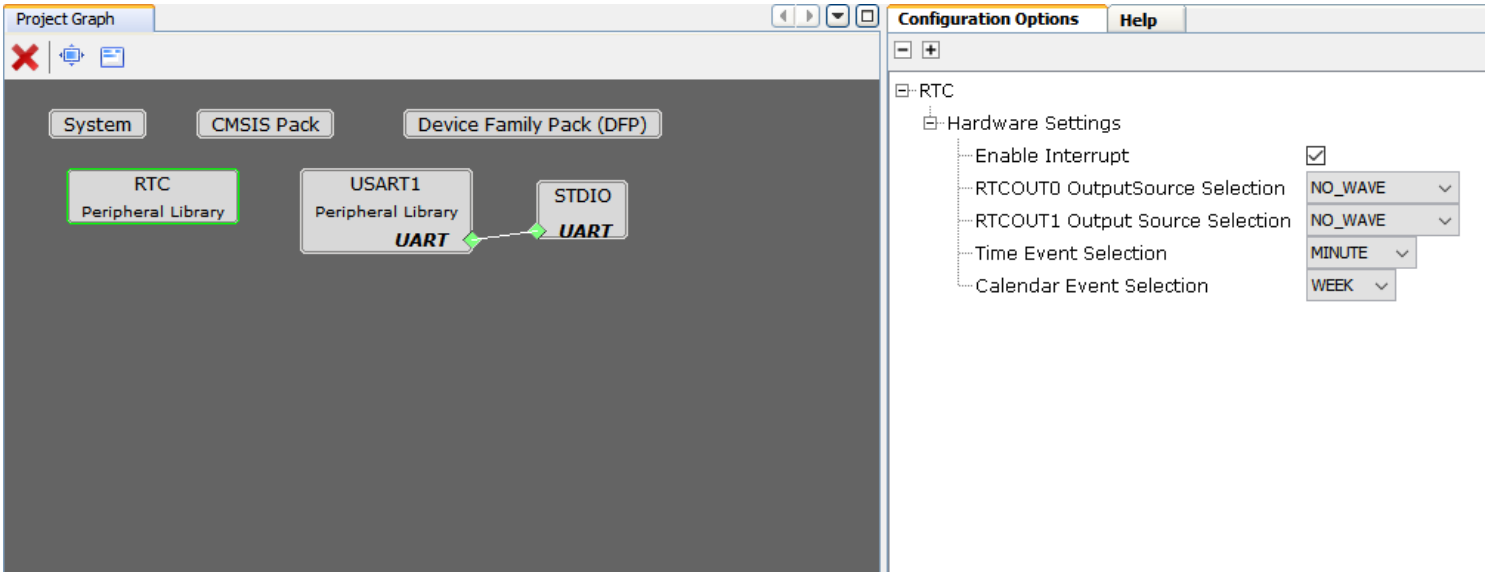
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_c21n_xpro	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

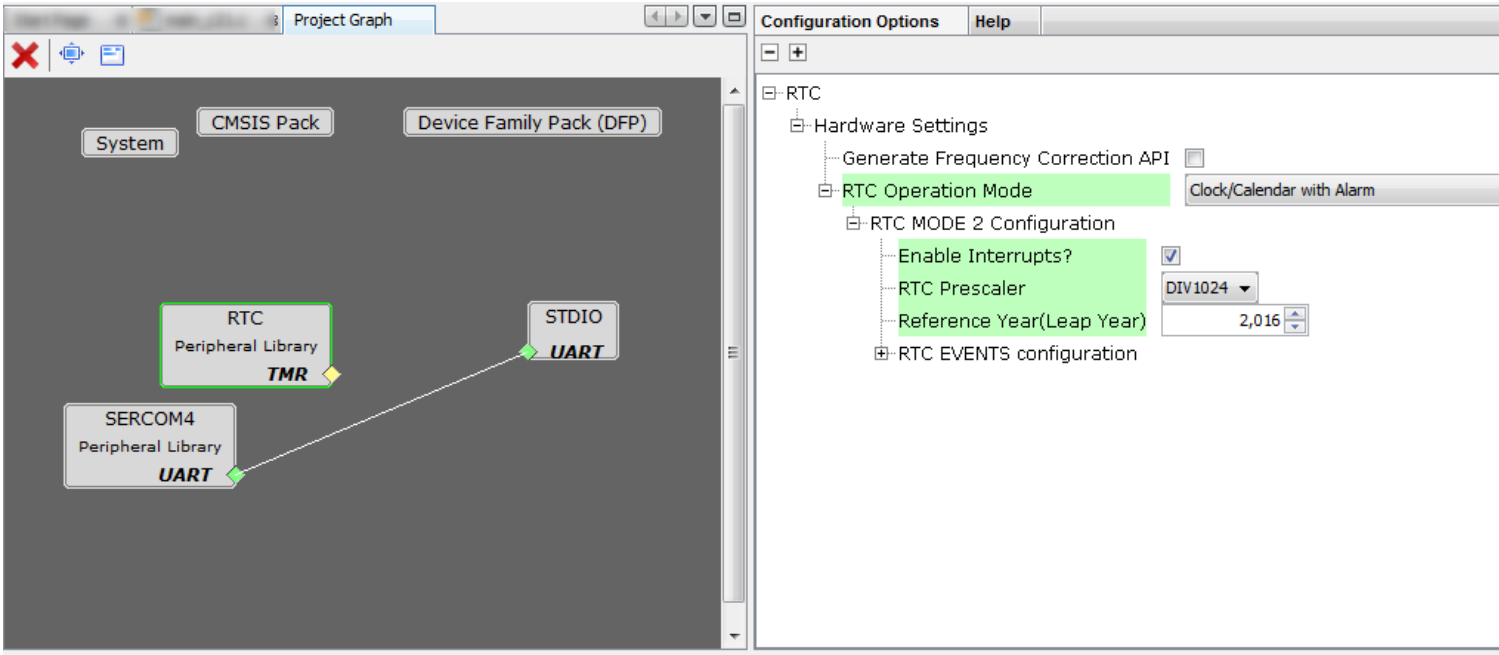
This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application for the **sam_e70_xult.X** project.



The Project Graph diagram below shows the Harmony components that are included in this application for the **sam_c21n_xpro.X** project.



Components Used

1. RTC Periheral Library
2. STDIO(USART)

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**

    - No Special hardware setup
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware setup

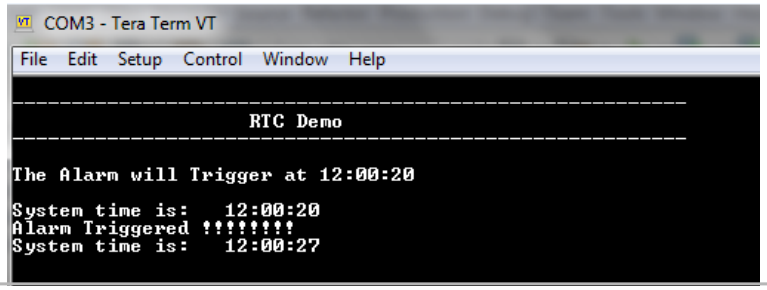
Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

Running the Application

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Observe output as described in the below table

Project Name	Output
sam_e70_xult.X	1. Console displays the following message ---------- RTC DEMO ---------- The ALARM will trigger at 12:00:20 System time is: HH:MM:SS 2. Once the system Time has reached 12:00:20 LED0 will turn on and following message is displayed:- Alarm Triggered !!!!!!!!
sam_c21n_xpro.X	1. Console displays the following message 

RTT Applications

This Section provides help for RTT PLIB Applications.

rtt_periodic_timeout

This example application shows how to use RTT Peripheral library to generate periodic interrupts.

Description

This example application shows how to use RTT Peripheral library to generate periodic interrupts.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/rtt/rtt_periodic_timeout/firmware/

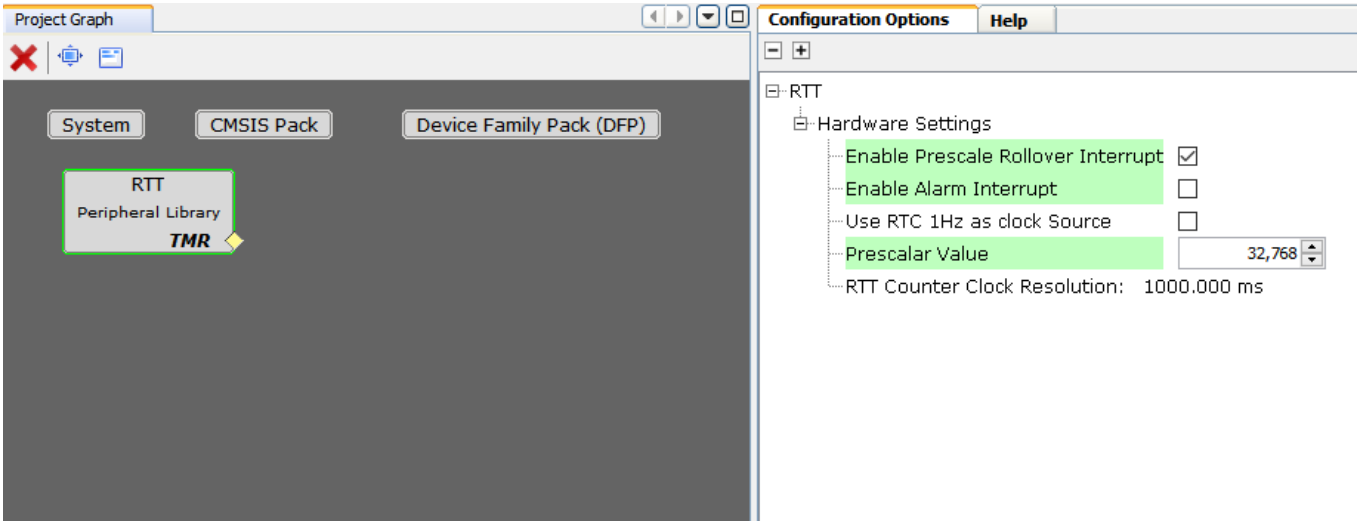
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. RTT Periheral Library

MHC Settings

1. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

Running the Application

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED 1 **Blinks**

SDADC Applications

This Section provides help for SDADC PLIB Applications.

sdadc_sample

This example application shows how to sample analog input using SDADC and display the converted samples on a serial terminal.

Description

This example application shows how to use the SDADC Peripheral Library to sample analog input and display the converted data on a serial terminal.

In this application DAC output is fed as analog input for SDADC conversion. The application starts conversion by calling SDADC peripheral library and prints the sampled value on a serial terminal. On every user switch press on the development board, the DAC output is increased by 0.1V and the same will be reflected as SDADC input. Therefore, on switch press the SDADC sampled value is increased by 0.1V and the same is printed on the serial terminal.

SDADC read sample output will always between 0.0V to 3.3V and it starts from 1.65V when device reset.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/sdadc/sdadc_sample/firmware

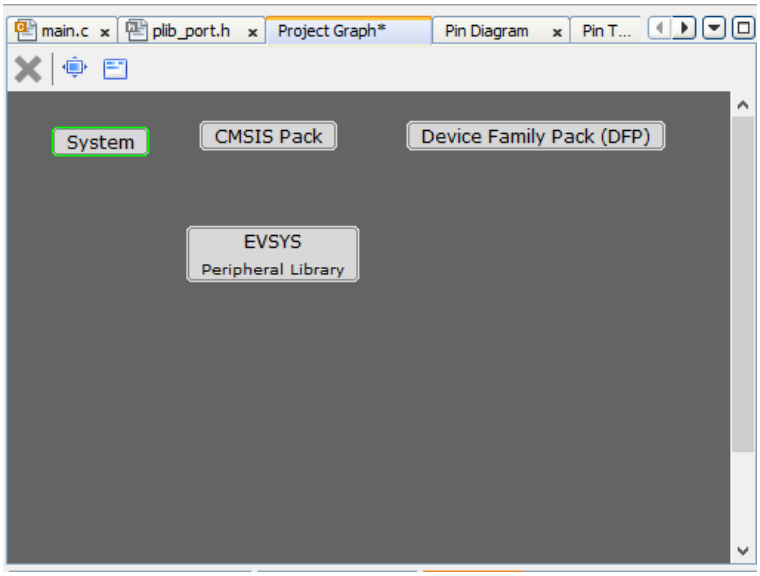
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SDADC Peripheral Library
2. DAC Peripheral Library
3. SERCOM Peripheral Library
4. EIC Peripheral Library
5. SUPC Peripheral Library
6. Systick Peripheral Library

MHC Settings

1. Configure SDADC
2. Configure DAC
3. Configure SUPC

4. Configure SERCOM4 as STDIO UART component

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Analog input (AIN1) of SDADC is fed from DAC output. These are routed like below on evaluation kit,
      - Analog input (AIN1) need to provide between two pins i.e, "INP1" and "INN1" .
      - "INP1" is mapped to Port Pin "PB09" is routed to "Pin 3 of EXT1 header".
      - "INN1" is mapped to Port Pin "PB08" is routed to "Pin4 of EXT1 header".
      - DAC output "DAC" is available between Port Pin "PA02" and GND pin these are routed to "Pin 1 and Pin 2 of DAC header".
    - Use jumper wires to connect below pins,
      - "Pin 3 of EXT1 header" to "Pin 1 of DAC header".
      - "Pin 4 of EXT1 header" to "Pin 2 of DAC header".

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. Console displays the converted digital value.
5. Press Switch "SW0" to change the DAC value and observe the changed SDADC value in console

SDRAMC Applications

This Section provides help for SDRAMC PLIB Applications.

sdramc_read_write

This section provides information on how the **sdramc_read_write** application is configured, built and works.

Description

This example application shows how to use SDRAM PLIB to WRITE-READ 2MB of memory on the On-Chip SDRAM memory device.

Building The Application

This section provides information on how to build and run the application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\sdramc\sdramc_read_write\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.

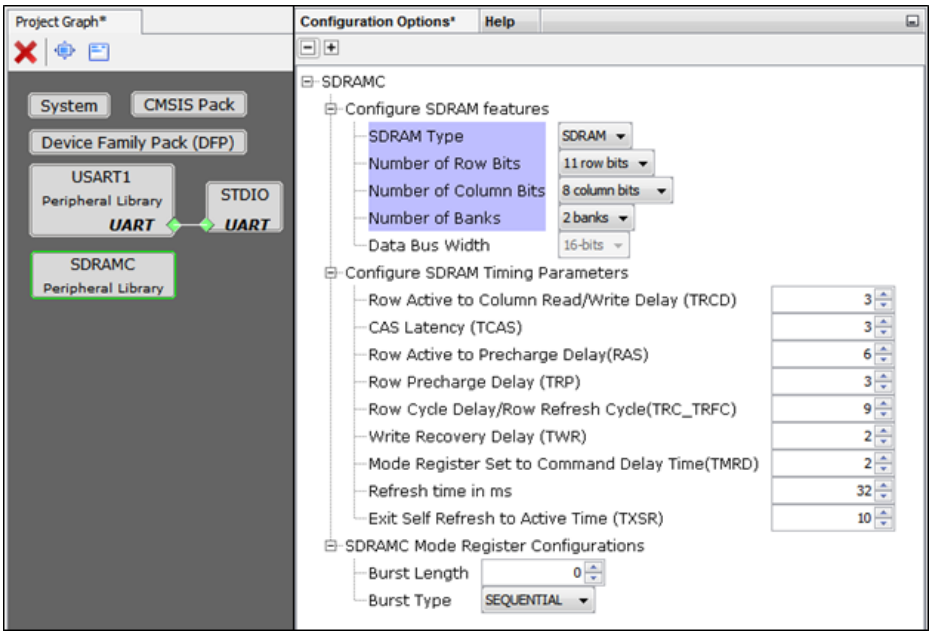


Figure 1: Project Graph and SDRAMC Configuration Option

Modules Used:

1. SDRAMC Peripheral Library
2. SDIO Module
3. USART Peripheral Library

MHC Settings:

1. Enable MPU for SDRAMC Memory Region
2. Configure Pins for SDRAMC Peripheral in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - Hardware Used:
     - SAM V71 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

Running The Application

This section describes how to run the application.

Description

1. Connect micro USB to DEBUG port.
2. Build the **sdramc_read_write** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
-----------------------------------------------------
SDRAM WRITE - READ Example
-----------------------------------------------------
Writing 0 to 256 KB data to SDRAM.
Writing 256 to 512 KB data to SDRAM.
Writing 512 to 768 KB data to SDRAM.
Writing 768 to 1024 KB data to SDRAM.
Writing 1024 to 1280 KB data to SDRAM.
Writing 1280 to 1536 KB data to SDRAM.
Writing 1536 to 1792 KB data to SDRAM.
Writing 1792 to 2048 KB data to SDRAM.

Reading 0 to 256 KB data from SDRAM.
Comparing 0 to 256 KB data.
Reading 256 to 512 KB data from SDRAM.
```

```
Comparing 256 to 512 KB data.
Reading 512 to 768 KB data from SDRAM.
Comparing 512 to 768 KB data.
Reading 768 to 1024 KB data from SDRAM.
Comparing 768 to 1024 KB data.
Reading 1024 to 1280 KB data from SDRAM.
Comparing 1024 to 1280 KB data.
Reading 1280 to 1536 KB data from SDRAM.
Comparing 1280 to 1536 KB data.
Reading 1536 to 1792 KB data from SDRAM.
Comparing 1536 to 1792 KB data.
Reading 1792 to 2048 KB data from SDRAM.
Comparing 1792 to 2048 KB data.

Test Pass....
```

sdramc_read_write_perf

This section provides information on how the **sdramc_read_write_perf** application is configured, built and works.

Description

This example application measures the performance of SDRAM PLIB with the help of best suitable DMA configuration by performing Write and Read operation between SRAM to SDRAM and SDRAM to SRAM respectively. Write and Read Time in CPU cycles are captured with the help of Systick module.

Building The Application

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\sdramc\sdramc_read_write_perf\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.

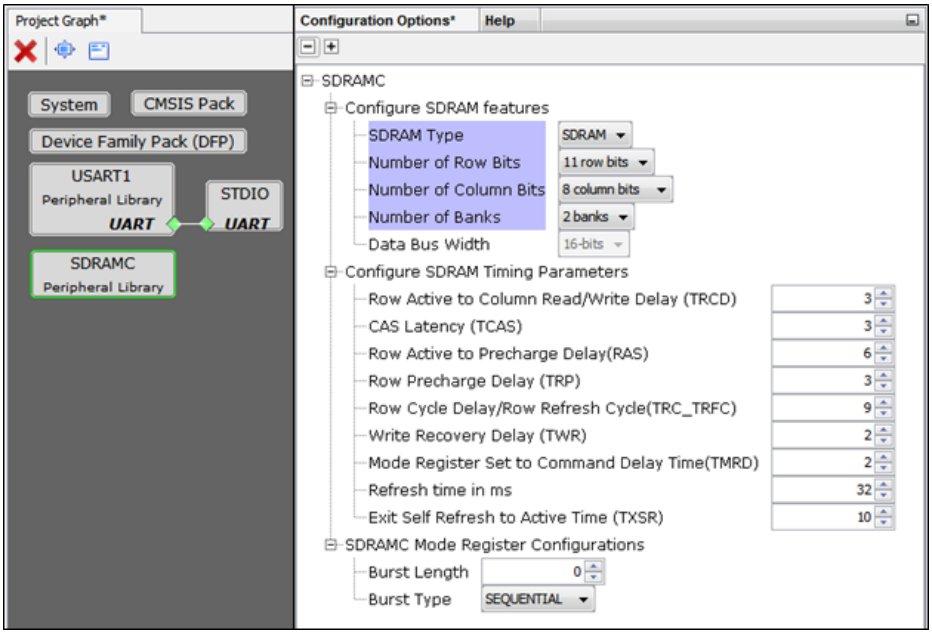


Figure 2: Project Graph and SDRAMC Configuration Option

Modules Used:

1. SDRAMC Peripheral Library
2. SYSTICK Peripheral Library
3. DMA Peripheral Library, See Figure 3.
4. USART Peripheral Library
5. SDIO Module

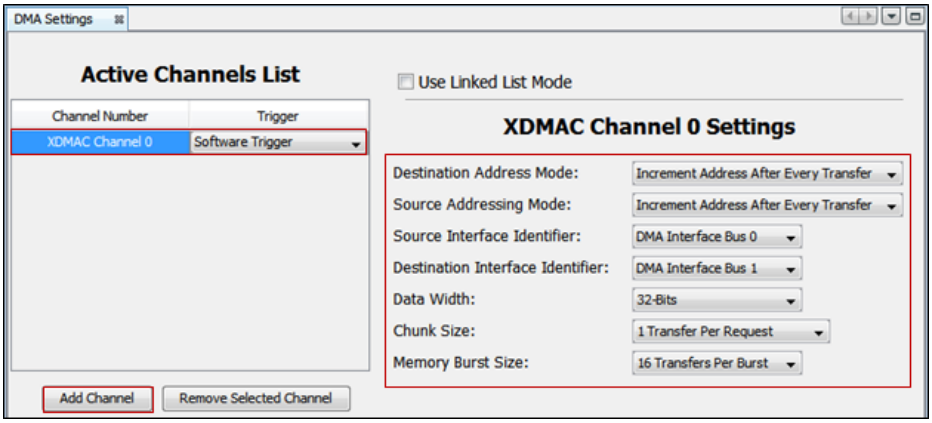


Figure 3: MPU Settings

MHC Settings:

1. Enable MPU for SDRAMC Memory Region
2. Enable SysTick and set SysTick Clock to Processor Clock
3. Configure Pins for SDRAMC Peripheral in Pin Settings
4. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - Hardware Used:
     - SAM V71 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

Running The Application

This section describes how to run the application.

Description

1. Connect micro USB to DEBUG port
2. Build the **sdramc_read_write_perf** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
-----------------------------------------------------
SDRAM Performance Benchmark Example
-----------------------------------------------------
Test Passed....!



SDRAM Performance Benchmark results based on the below DMA configurations
Data Width = 32-bits
Chunk Size = 1-transfers per request
Burst Size = 16-transfers per burst

Write Time in CPU cycles = 336140
Read Time in CPU cycles = 368382
```

SERCOM Applications

This Section provides help for SERCOM PLIB Applications.

SERCOM I2C Applications

This Section provides help for SERCOM I2C PLIB Applications.

i2c_eeprom

This example application shows how to use SERCOM I2C to write and read from Serial EEPROM memory.

Description

This example application shows how to use SERCOM I2C Peripheral library to write and read from external AT24 Serial EEPROM memory device(EEPROM 3 CLICK Board).

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/sercom/i2c/master/i2c_eeprom/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components are used in this application.

Components Used

1. SERCOM (SERCOM I2C Master) Peripheral Library

MHC Settings

1. Use "SERCOM1 Peripheral Library" component to configure SERCOM as SERCOM I2C Master.
2. Use "Pin Configuration" to setup PORT pin PC05 to act as an LED.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
    - MikroElektronika EEPROM 3 click board.

- mikroBUS Xplained Pro board
- **Hardware Setup**
  - Connect MikroElektronika EEPROM 3 click board to mikroBUS Xplained Pro board.
  - Connect mikroBUS Xplained Pro board to extension header 2(EXT2) of SAM C21n Xplained Pro kit.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	Press the Reset button on the SAM C21n Xplained Pro Evaluation Kit. Success: LED0 Will be ON. Failure: LED0 will be OFF.

SERCOM SPI Applications

This Section provides help for SERCOM SPI PLIB Applications.

spi_eeprom_write_read

This example application shows how to write and read on Serial EEPROM memory using SERCOM SPI.

Description

This example demonstrates how to use SERCOM SPI PLIB to write and read on Serial EEPROM memory.

The example first registers a callback with SERCOM5 SPI PLIB which is defined in the example and will be called by SERCOM5 SPI PLIB once the requested transfer completes. Then SERCOM5_SPI_WriteRead API is used to request SERCOM5 SPI PLIB to send data. The transfer will start happening in interrupt context. Once the transfer is completed, registered callback function is called and then check read status of EEPROM. Once EEPROM is ready to read, then read data from EEPROM and compare with written data. Status is indicated with LED.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/sercom/spi/spi_eeprom_write_read/firmware

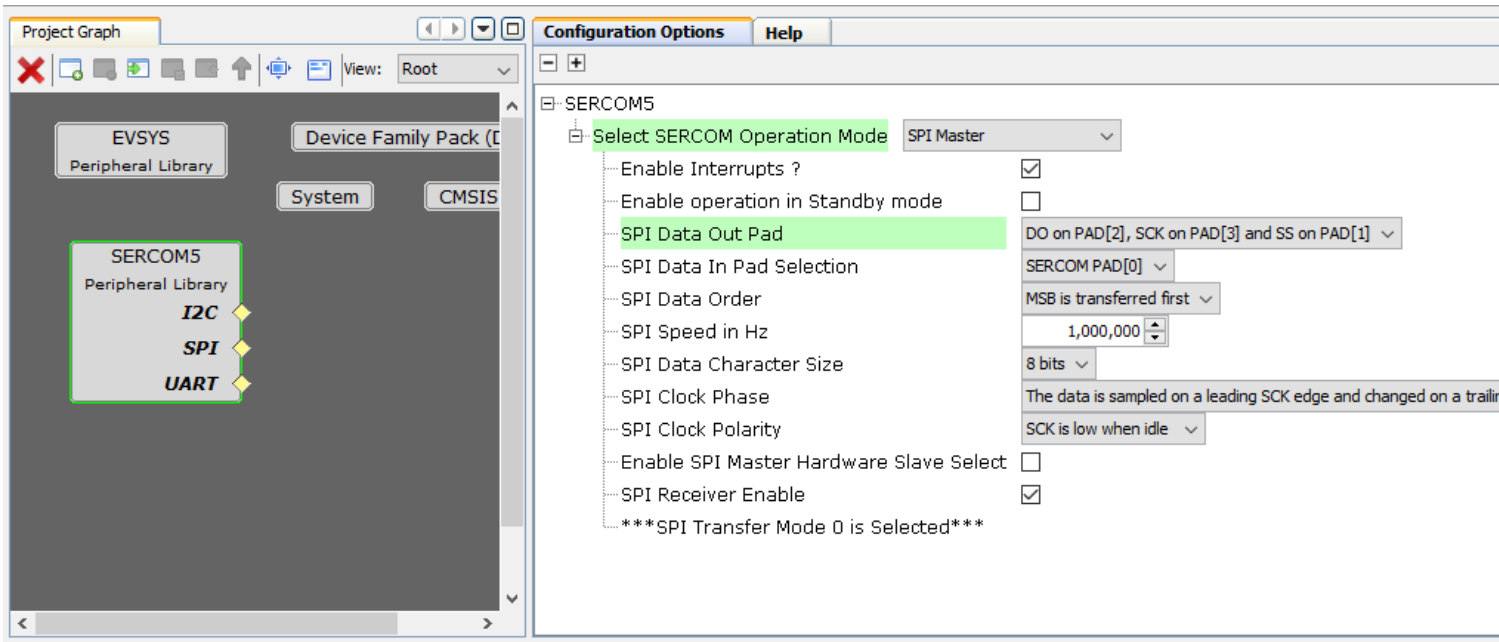
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SERCOM5 Peripheral Library

MHC Settings

1. Configure SERCOM5 as SPI Master by enabling Interrupts.
2. Configure GPIO pin for LED.
3. Configure GPIO pins for SERCOM5-SPI.
4. Configure GPIO pins for EEPROM Write Protection, Hold and Chip Select as per "EEPROM 4 Click Board(MIKROE-2536)"
   - EEPROM Write Protection signal is mapped to PA20 that is routed to "Pin 5 of EXT2 header"
   - EEPROM Hold signal is mapped to PB30 that is routed to "Pin 7 of EXT2 header"
   - EEPROM Chip Select signal is mapped to PB03 that is routed to "Pin 15 of EXT2 header"

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - Install EEPROM 4 Click Board on the EXT2 click slot provided on SAM C21N Xplained Pro Evaluation Kit.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Test Pass - LED0 ON Test Fail - LED0 OFF

spi_ping_pong_with_dma

This example application shows how to use SERCOM SPI in DMA mode to continuously transfer data between two buffers.

Description

This example demonstrates how to use DMA in linked list mode to continuously transfer data from two buffers one by one.

Two sets of data are stored (or populated dynamically) in two buffers by name "txPing" and "txPong". Using DMA, the data in "txPing" buffer is first transferred to SERCOM SPI transmit register, followed by data in "txPong" buffer, then again data in "txPing" buffer and so on, the cycle continues forever.

To verify the correctness of the transmitted data, similar setup is done at SERCOM SPI receive end as well and received data is saved in two different buffers by name "rxPing" and "rxPong".

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/sercom/spi/spi_ping_pong_with_dma/firmware

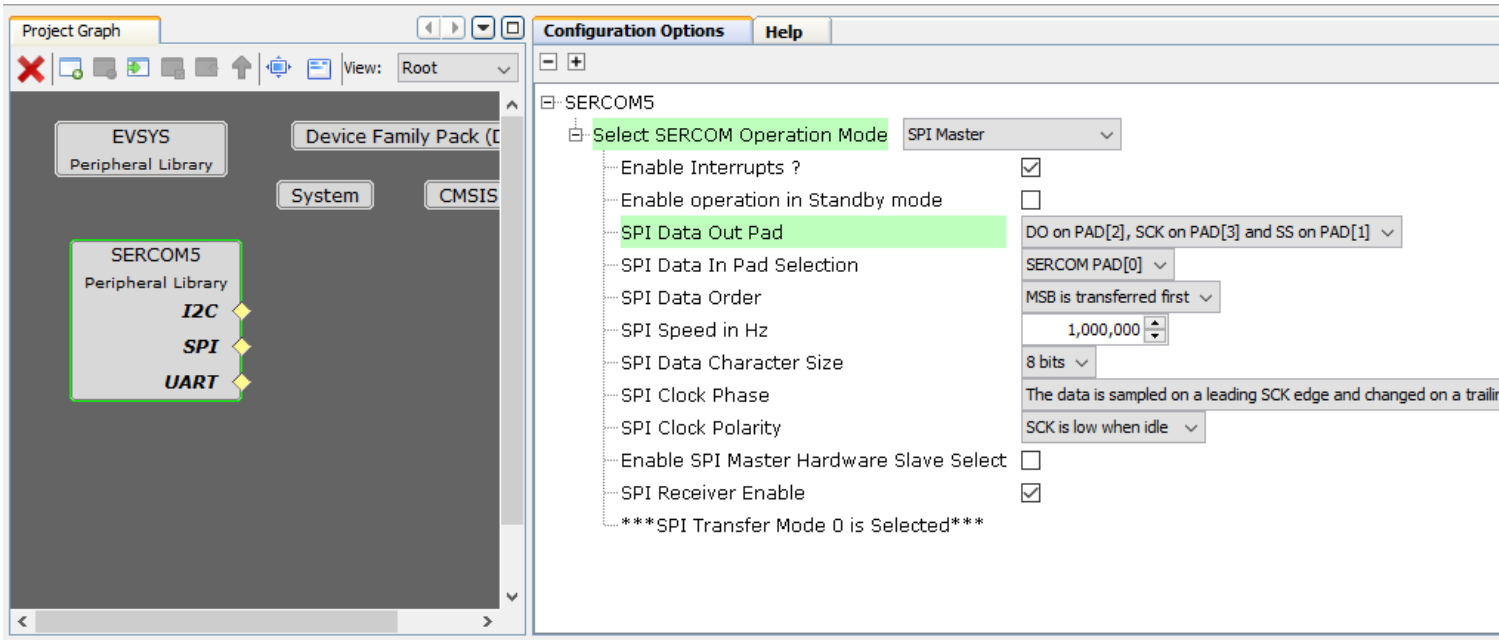
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

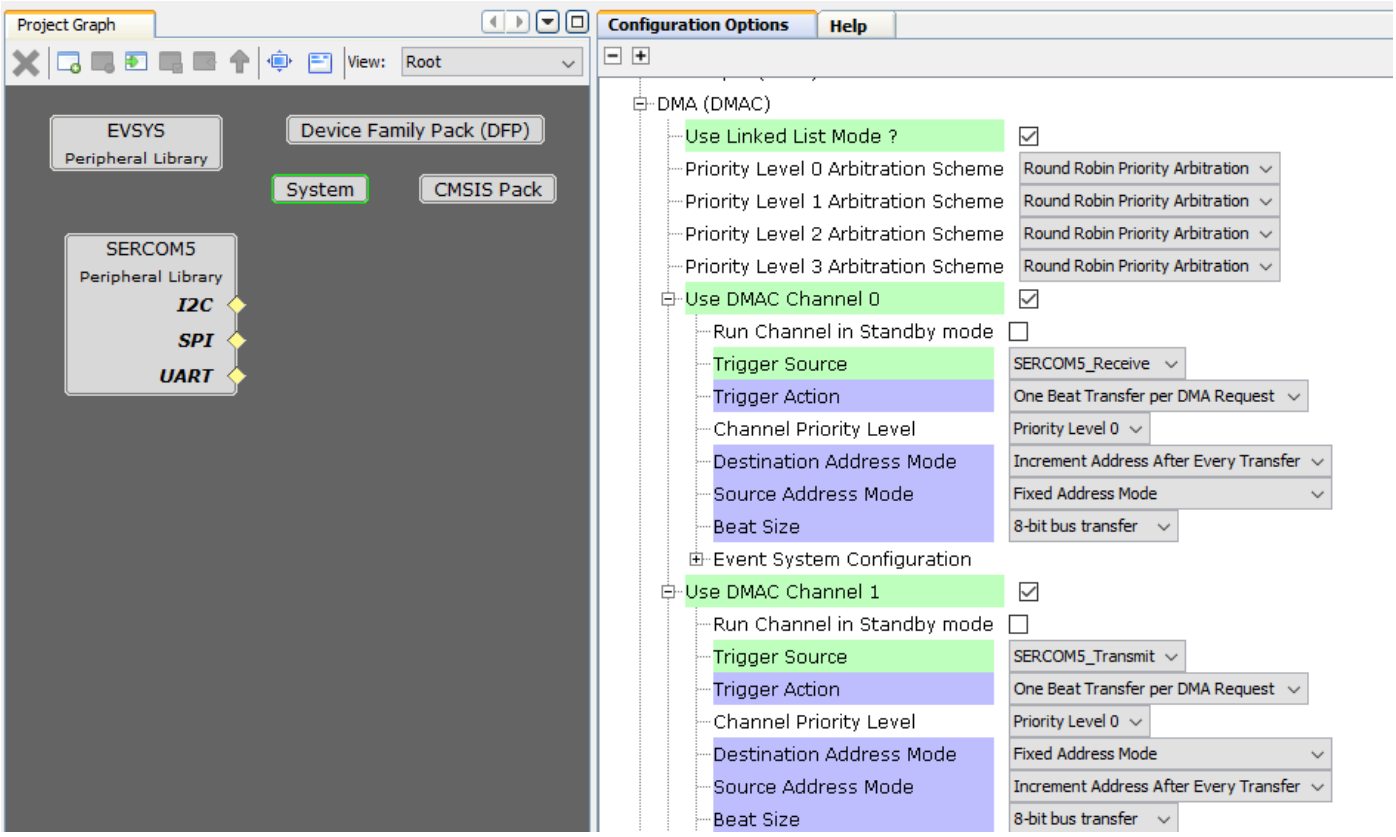
MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.





Components Used

1. SERCOM5 Peripheral Library
2. DMA Peripheral Library

MHC Settings

1. Configure SERCOM5 Operation Mode as SPI Master with enabling Interrupts
2. Configure DMA CH0 and CH1 as SERCOM5 Receive and Transmit respectively
3. Enable Linked List Mode in DMA configuration.
4. Configure GPIO pin for LED
5. Configure GPIO pins for SERCOM5-SPI

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - SERCOM SPI0 MOSI signal is mapped to PB00 that is routed to "Pin 16 of EXT2 header".
    - SERCOM SPI0 MISO signal is mapped to PB02 that is routed to "Pin 17 of EXT2 header".
    - Use jumper wire to Connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header".

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Test Pass - LED0 ON Test Fail - LED0 OFF

spi_self_loopback_blocking

This example application shows how to use SERCOM SPI with external loop back to write and then read back an array of data in blocking mode.

Description

This example demonstrates how to use SERCOM SPI PLIB to write and then read back an array of data in blocking mode.

The example uses SERCOM5_SPI_WriteRead API to send an array of data over SERCOM5_SPI_MOSI line and the same data is received over SERCOM5_SPI_MISO line. CPU can not do any other task until this transfer completes. Once the transfer is completed, received data is compared with transmitted data and status is indicated with LED.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/sercom/spi/spi_self_loopback_blocking/firmware

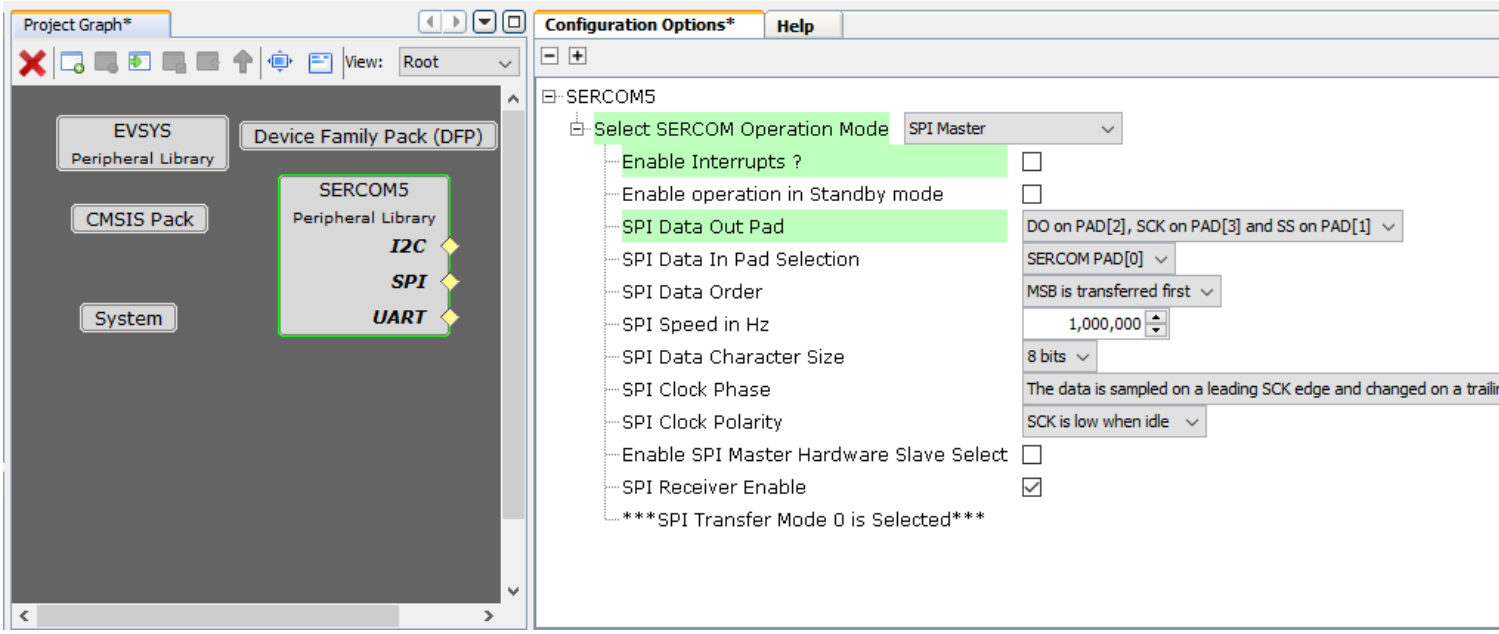
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SERCOM5 Peripheral Library

MHC Settings

1. Configure SERCOM5 as SPI Master by disabling Interrupts.
2. Configure GPIO pin for LED.
3. Configure GPIO pins for SERCOM5-SPI.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - SPI0 MOSI signal is mapped to PB00 that is routed to "Pin 16 of EXT2 header".
    - SPI0 MISO signal is mapped to PB02 that is routed to "Pin 17 of EXT2 header".
    - Use jumper wire to Connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header".

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.

2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Test Pass - LED0 ON Test Fail - LED0 OFF

spi_self_loopback_interrupt

This example application shows how to use SERCOM SPI PLIB with external loop back to write and then read back an array of data in interrupt mode.

Description

This example demonstrates how to use SERCOM SPI PLIB to write and then read back an array of data in interrupt mode.

The example first registers a callback with SERCOM5 SPI PLIB which is defined in the example and will be called by SERCOM5 SPI PLIB once the requested transfer completes. Then SERCOM5_SPI_WriteRead API is used to request SERCOM5 SPI PLIB to send and receive data. The transfer will start happening in interrupt context. Once the transfer is completed, registered callback function is called and then check for correctness of the transfer.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/sercom/spi/spi_self_loopback_interrupt/firmware

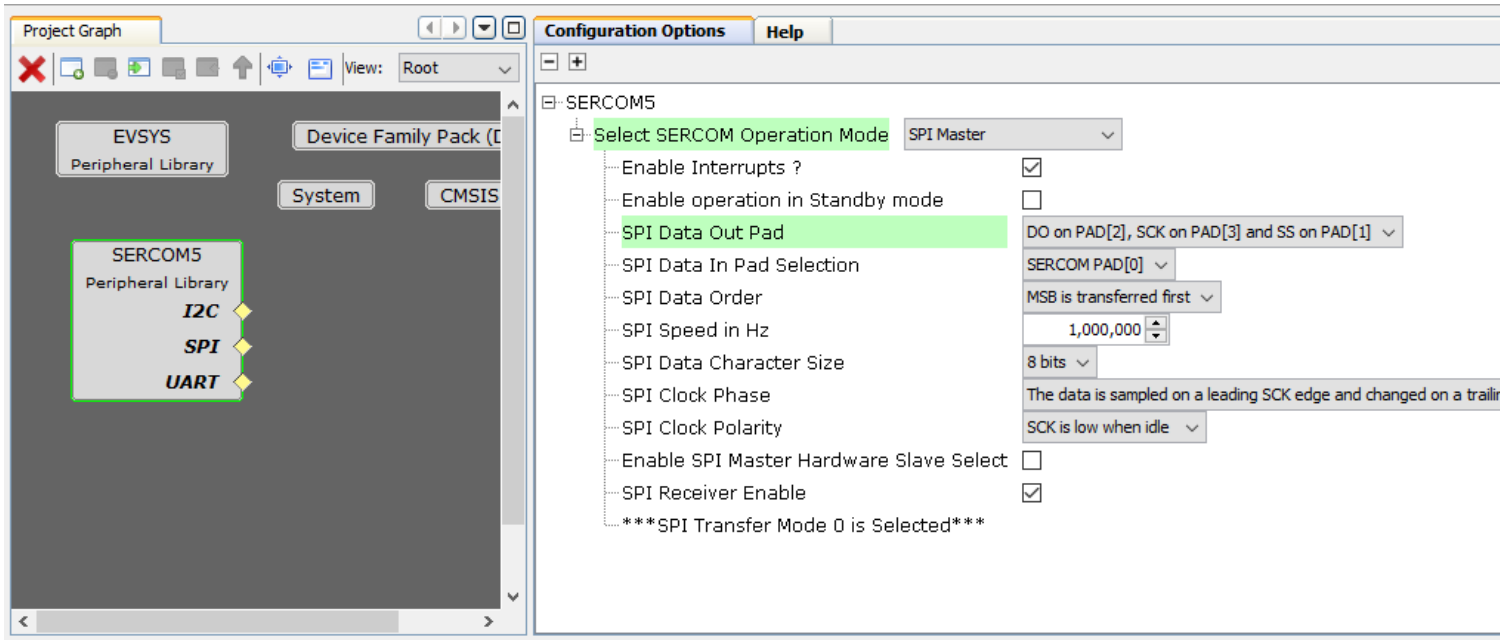
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SERCOM5 Peripheral Library

MHC Settings

1. Configure SERCOM5 as SPI Master by enabling Interrupts.
2. Configure GPIO pin for LED.
3. Configure GPIO pins for SERCOM5-SPI.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - SPI0 MOSI signal is mapped to PB00 that is routed to "Pin 16 of EXT2 header".
    - SPI0 MISO signal is mapped to PB02 that is routed to "Pin 17 of EXT2 header".
    - Use jumper wire to Connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header".

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_c21n_xpro.X	Test Pass - LED0 ON Test Fail - LED0 OFF

SERCOM USART Applications

This Section provides help for SERCOM USART PLIB Applications.

usart_echo_blocking

This example application shows how to use SERCOM USART to do basic read and write operation over USART in blocking mode.

Description

This example application shows how to use SERCOM USART peripheral library to do basic read and write operation over USART in blocking mode. It echoes back every line received from terminal window.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below,

Application Path	csp/apps/sercom/usart/usart_echo_blocking/firmware

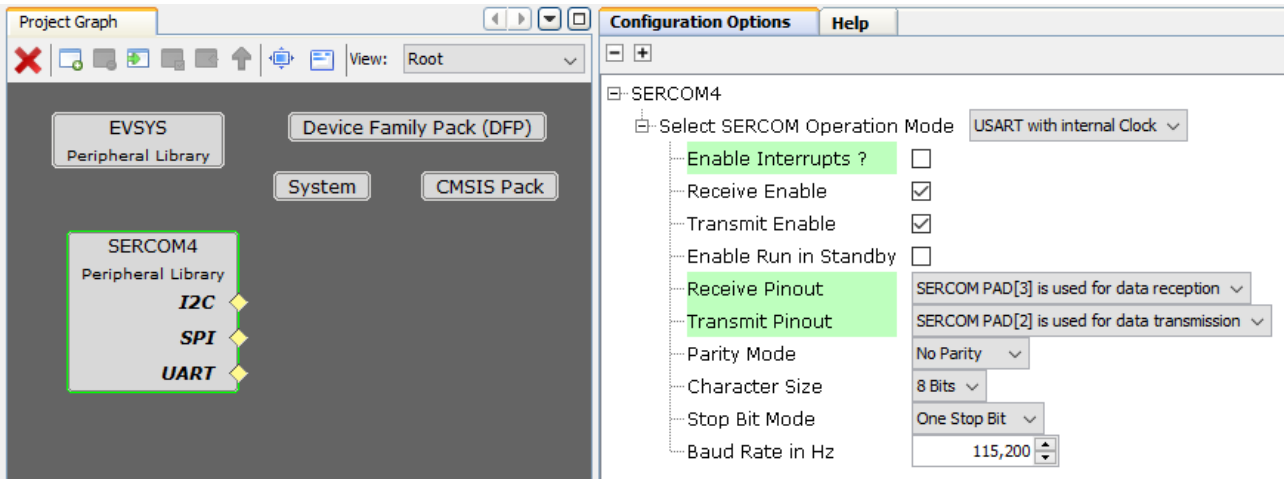
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SERCOM Peripheral Library

MHC Settings

1. Configure SERCOM4 as USART by disabling Interrupts.

2. Configure GPIO pin for LED.
3. Configure GPIO pins for SERCOM4-USART.

Hardware Setup

This section describes how to configure the supported hardware.

Description

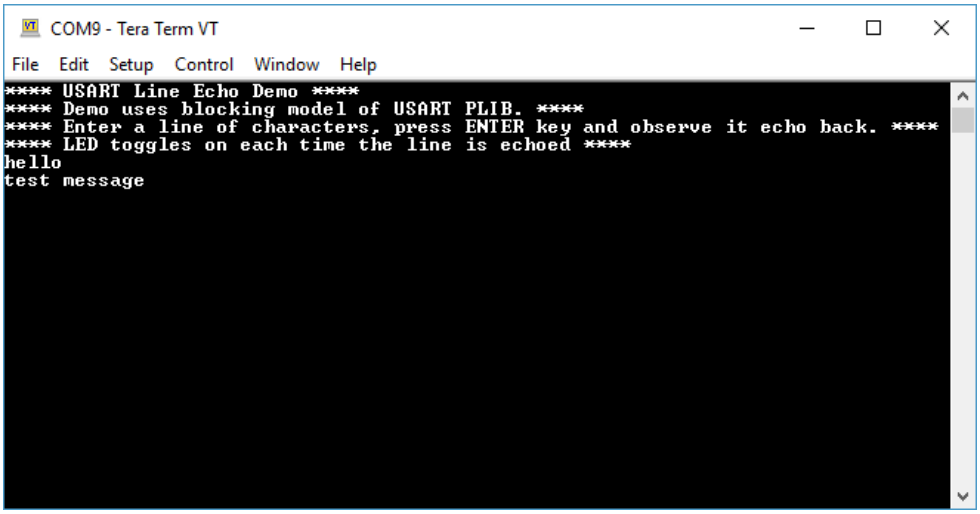
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. Enter a line of characters and press ENTER key.
5. Entered line will echo back and **LED0** will toggle.
6. Console displays the echo data as shown below.



usart_echo_interrupt

This example application shows how to use SERCOM USART to do basic read and write operation over USART in interrupt mode.

Description

This example application shows how to use SERCOM USART to do basic read and write operation over USART in interrupt

mode. It receives 10 characters from terminal window and echoes them back.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

The parent folder for all the MPLABX projects for this application is given below,

Application Path	csp/apps/sercom/usart/usart_echo_interrupt/firmware

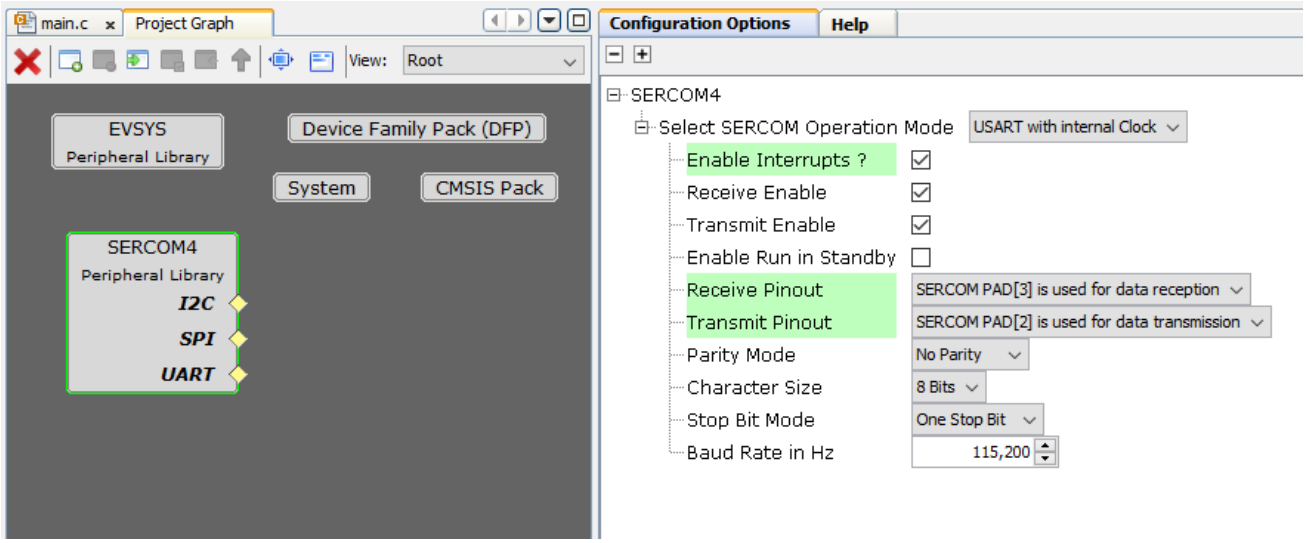
Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SERCOM Peripheral Library

MHC Settings

1. Configure SERCOM4 as USART by enabling Interrupts.
2. Configure GPIO pin for LED.
3. Configure GPIO pins for SERCOM4-USART.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_c21n_xpro.X**

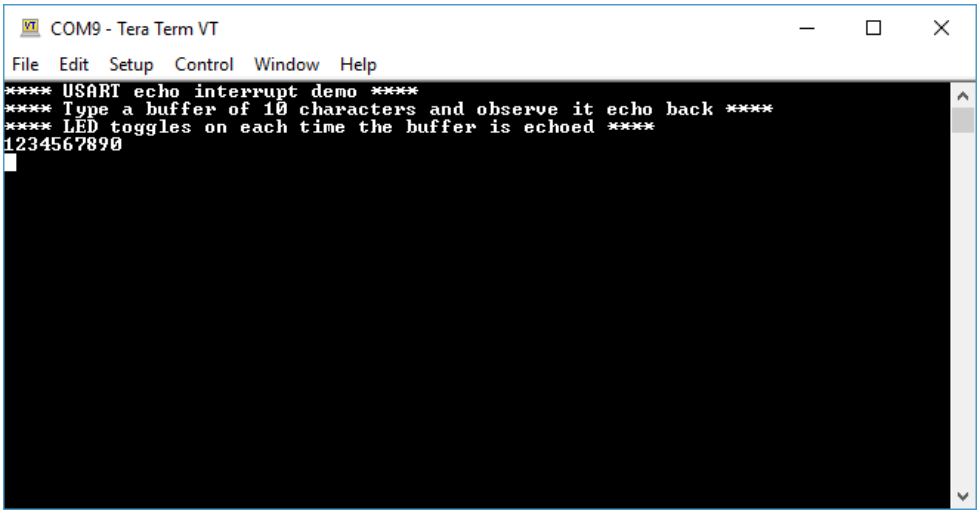
- **Hardware Used**
  - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup**
  - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Build and Program the application using MPLAB X IDE.
2. Open a terminal application and connect it to the Virtual COM of EDBG with the following settings
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Press the Reset switch on the SAM C21N Xplained Pro Evaluation Kit.
4. Enter a line of characters and press ENTER key.
5. Entered line will echo back and **LED0** will toggle.
6. Console displays the echo data as shown below.



SPI Applications

This section provides help for SPI PLIB applications.

spi_eeprom_write_read

This example shows how to use SPI Peripheral library to write and read data on an external EEPROM device.

Description

This example writes a string into starting location of EEPROM device, then read the same string back and at the end compare the read data with written data in order to ensure the correctness of the transfer. It indicates the transfer result through on board LED.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	csp/apps/spi/spi_eeprom_write_read/firmware/

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This section provides information on MHC configuration used to create this application.

Description

Following Components are used to create this application:

1. SPI 0 Peripheral Library in Interrupt mode
2. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. SPI 0 PLIB configuration is used as default
2. SPI 0 related pins are configured in Pin Manager
3. GPIO pins are configured in Pin Manager for EEPROM Chip Select, Hold and Write Protect.
4. GPIO pin is configured for LED in Pin Manager

Hardware Setup

This section describes which are the hardware used and how to set them up to run this application.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
    - EEPROM 4 Click Board - 1 nos (MIKROE-2536)
  - **Hardware Setup**
    - Install EEPROM 4 Click Board on the click slot provided on SAM E70 Xplained Ultra board.
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
    - EEPROM 4 Click Board - 1 nos (MIKROE-2536)
  - **Hardware Setup**
    - Install EEPROM 4 Click Board on the click slot provided on SAM V71 Xplained Ultra board.

Running The Application

This section provides information on how to run the application and observe the output.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table:

Project Name	Output Success	Output Failure
sam_e70_xult.X	LED 1 ON	LED 1 OFF
sam_v71_xult.X	LED 1 ON	LED 1 OFF

spi_self_loopback_interrupt

This example demonstrates how to use SPI PLIB to write and then read back an array of data in interrupt mode.

Description

The example first registers a callback with SPI PLIB which is defined in the example and will be called by SPI PLIB once the requested transfer completes. Then SPI0_WriteRead API is used to request SPI PLIB to send and receive data. The transfer will start happening in interrupt context. Once the transfer is completed, registered callback function is called and then correctness of the transfer is checked.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	csp/apps/spi/spi_self_loopback_interrupt/firmware/

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This section provides information on MHC configuration used to create this application.

Description

Following Components are used to create this application:

1. SPI 0 Peripheral Library in Interrupt mode
2. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. SPI 0 PLIB configuration is used as default
2. SPI 0 related pins are configured in Pin Manager
3. GPIO pin is configured for LED in Pin Manager

Hardware Setup

This section describes which are the hardware used and how to set them up to run this application.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".

Running The Application

This section provides information on how to run the application and observe the output.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table:

Project Name	Output Success	Output Failure
sam_e70_xult.X	LED 1 ON	LED 1 OFF
sam_v71_xult.X	LED 1 ON	LED 1 OFF

spi_self_loopback_blocking

This example demonstrates how to use SPI PLIB to write and then read back an array of data in blocking mode.

Description

The example uses SPI0_WriteRead API to send an array of data over SPI0_MOSI line and the same data is received over SPI0_MISO line. CPU can not do any other task until this transfer completes. Once the transfer is completed, received data is compared with transmitted data and status is indicated through LED.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	csp/apps/spi/spi_self_loopback_blocking/firmware/

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This section provides information on MHC configuration used to create this application.

Description

Following Components are used to create this application:

1. SPI 0 Peripheral Library in Blocking mode
2. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. Default SPI 0 PLIB configuration is changed to Non-Interrupt mode
2. SPI 0 related pins are configured in Pin Manager
3. GPIO pin is configured for LED in Pin Manager

Hardware Setup

This section describes which are the hardware used and how to set them up to run this application.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".

Running The Application

This section provides information on how to run the application and observe the output.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table:

Project Name	Output Success	Output Failure
sam_e70_xult.X	LED 1 ON	LED 1 OFF
sam_v71_xult.X	LED 1 ON	LED 1 OFF

spi_ping_pong_with_dma

This example demonstrates how to use SPI PLIB along with DMA PLIB to achieve data transfer from two different sources without any lag.

Description

This example demonstrates how to use DMA in linked list mode to continuously transfer data from two buffers one by one. SPI peripheral is used to show the feature as there can be such real application need with SPI. Two sets of data are stored (or populated dynamically) in two buffers by name "txPing" and "txPong". Using DMA, the data in "txPing" buffer is first transferred to SPI transmit register, followed by data in "txPong" buffer, then again data in "txPing" buffer and so on, the cycle continues forever.

To verify the correctness of the transmitted data, similar setup is done at SPI receive end as well and received data is saved in two different buffers by name "rxPing" and "rxPong".

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	csp/apps/spi/spi_ping_pong_with_dma/firmware/

Following table indicates which MPLAB X project should be used for which board:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This section provides information on MHC configuration used to create this application.

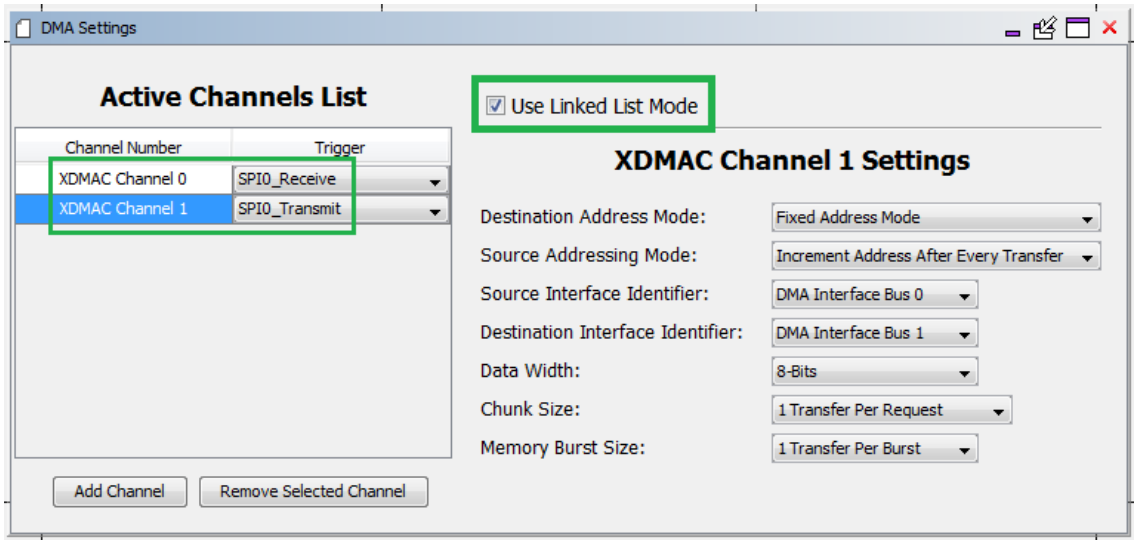
Description

Following Components are used to create this application:

1. SPI 0 Peripheral Library
2. XDMAC PLIB
3. Clock, Interrupt (NVIC) and Port (PIO) Peripheral libraries are by default added

Following configuration is done in MHC to create this application:

1. SPI 0 PLIB configuration is used as default
2. SPI 0 related pins are configured in Pin Manager
3. GPIO pin is configured for LED in Pin Manager
4. Couple of DMA channels are configured for SPI transfer and DMA is configured in linked list mode as shown below:



Hardware Setup

This section describes which are the hardware used and how to set them up to run this application.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".
- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
      - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header".
      - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header".

Running The Application

This section provides information on how to run the application and observe the output.

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table:

Project Name	Output Success	Output Failure
sam_e70_xult.X	LED 1 ON	LED 1 OFF
sam_v71_xult.X	LED 1 ON	LED 1 OFF

SUPC Applications

This Section provides help for SUPC PLIB Applications.

supc_wakeup_rtc

This example demonstrates entering different low power modes and exiting it using RTC alarm.

Description

This example demonstrates entering different low power modes and exiting it using RTC alarm. It also demonstrates retention of general purpose backup registers (GPBR) and backup SRAM on wakeup from backup mode through device reset.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	**csp/apps/supc/supc_wakeup_rtc/firmware**

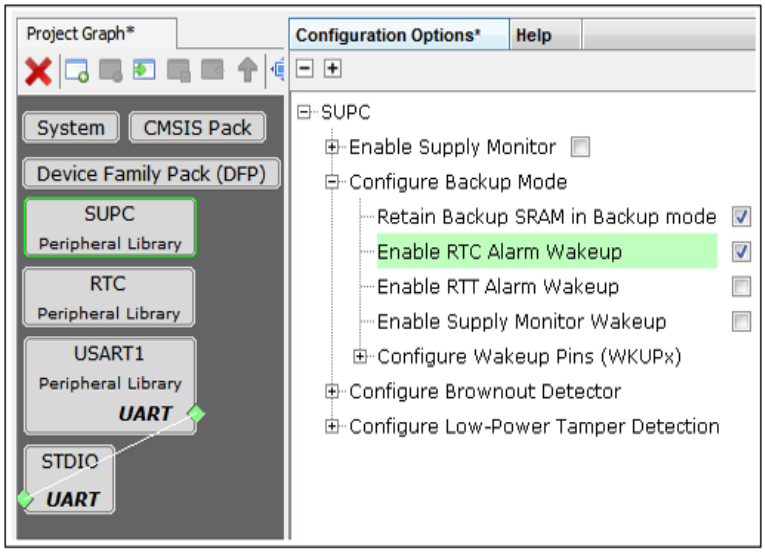
Project Name	**Description**
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SUPC Peripheral Library
2. RTC Peripheral Library
3. USART Peripheral Library
4. SYSTICK Peripheral Library
5. STDIO module

MHC Settings

1. Configure SUPC, RTC, USART and STDIO modules
2. Configure Pins for USART module
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

Running The Application

This section provides information on how to run the application

Description

1. Connect micro USB to DEBUG port.
2. Build the **supc_wakeup_rtc** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
-----------------------------------------------
LOW power demo
-----------------------------------------------
Select the low power mode to enter
a) Sleep Mode
b) Wait Mode
c) Backup Mode

Enter your choice
```

6. Select the option for device to enter into Low power mode
7. After selecting the option, device enters the Low power mode and wakes up after RTC wakeup event.

supc_wakeup_rtt

This example demonstrates entering different low power modes and exiting it using RTT alarm

Description

This example demonstrates entering different low power modes and exiting it using RTT alarm

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/supc/supc_wakeup_rtt/firmware

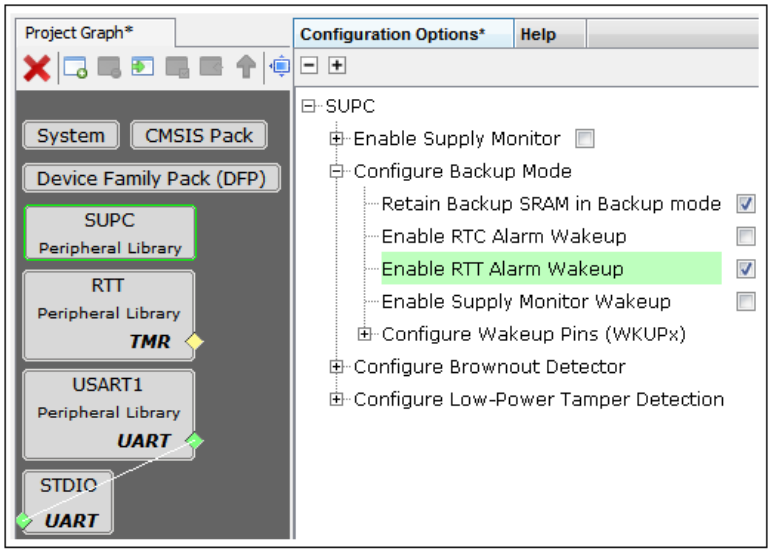
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SUPC Peripheral Library
2. RTT Peripheral Library
3. USART Peripheral Library
4. SYSTICK Peripheral Library
5. STDIO module

MHC Settings

1. Configure SUPC, RTT, USART and STDIO modules
2. Configure Pins for USART module
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - **Hardware Used**
     - SAM V71 Xplained Ultra Evaluation Kit
   - **Hardware Setup**
     - No special hardware setup is required

Running The Application

This section provides information on how to run the application

Description

1. Connect micro USB to DEBUG port.
2. Build the **supc_wakeup_rtt** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
-----------------------------------------------
LOW power demo
-----------------------------------------------
Select the low power mode to enter
a) Sleep Mode
b) Wait Mode
c) Backup Mode

Enter your choice
```

6. Select the option for device to enter into Low power mode
7. After selecting the option, device enters the Low power mode and wakes up after RTT wakeup event.

supc_wakeup_wkup

This example demonstrates entering different low power modes and exiting it using wake up pin.

Description

This example demonstrates entering different low power modes and exiting it using wake up pin.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/supc/supc_wakeup_wkup/firmware

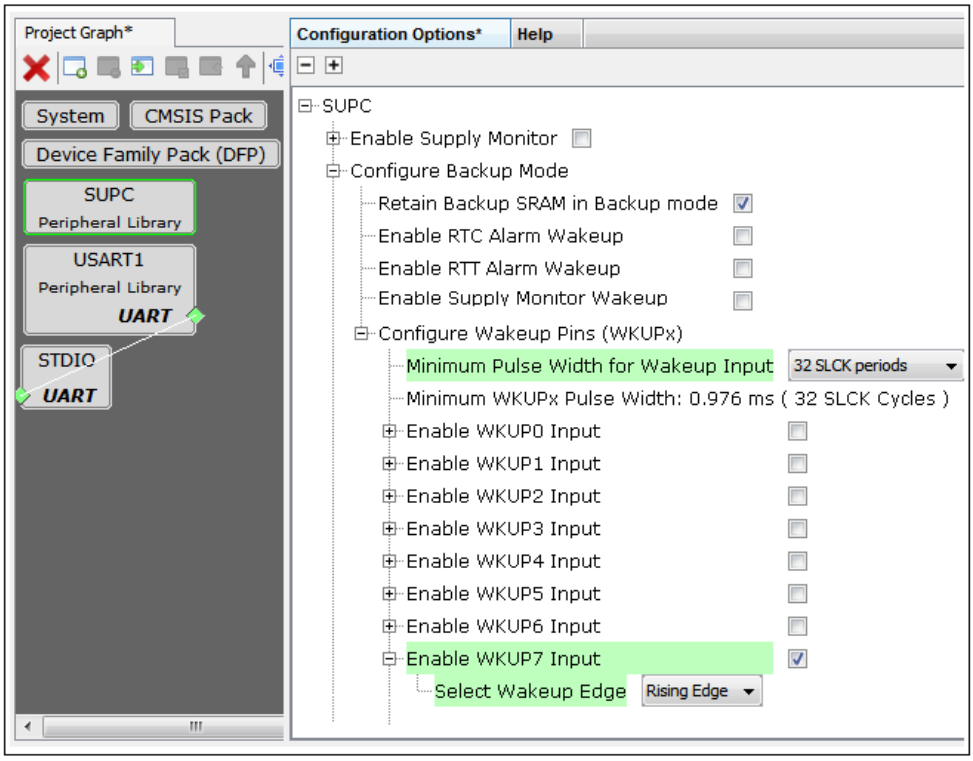
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. SUPC Peripheral Library
2. USART Peripheral Library
3. SYSTICK Peripheral Library
4. STDIO module

MHC Settings

1. Configure SUPC, USART and STDIO modules
2. Configure Pins for USART module and Wakeup Pin 7
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - **Hardware Used**
     - SAM E70 Xplained Ultra Evaluation Kit
   - **Hardware Setup**

- No special hardware setup is required

Running The Application

This section provides information on how to run the application

Description

1. Connect micro USB to DEBUG port.
2. Build the **supc_wakeup_wkup** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
---------------------------------------------
LOW power demo
---------------------------------------------
Select the low power mode to enter
a) Sleep Mode
b) Wait Mode
c) Backup Mode

Enter your choice
```

6. Select the option for device to enter into Low power mode
7. After selecting the option, device enters the Low power mode and wakes up wakes up when Switch is pressed on board, which is configured as Wakeup pin (WKUP7)

SYSTICK Applications

This section provides help for SYSTICK PLIB Applications.

systick_periodic_timeout

This example application shows how to use SysTick to generate periodic interrupts.

Description

This example application shows how to use SysTick Peripheral Library to generate periodic interrupts. The application registers a periodic timeout callback. It toggles an LED every time the callback is triggered.

Building The Application

This section provides information on how to build application using MPLAB X IDE.

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp/apps/systick/systick_periodic_timeout/firmware/

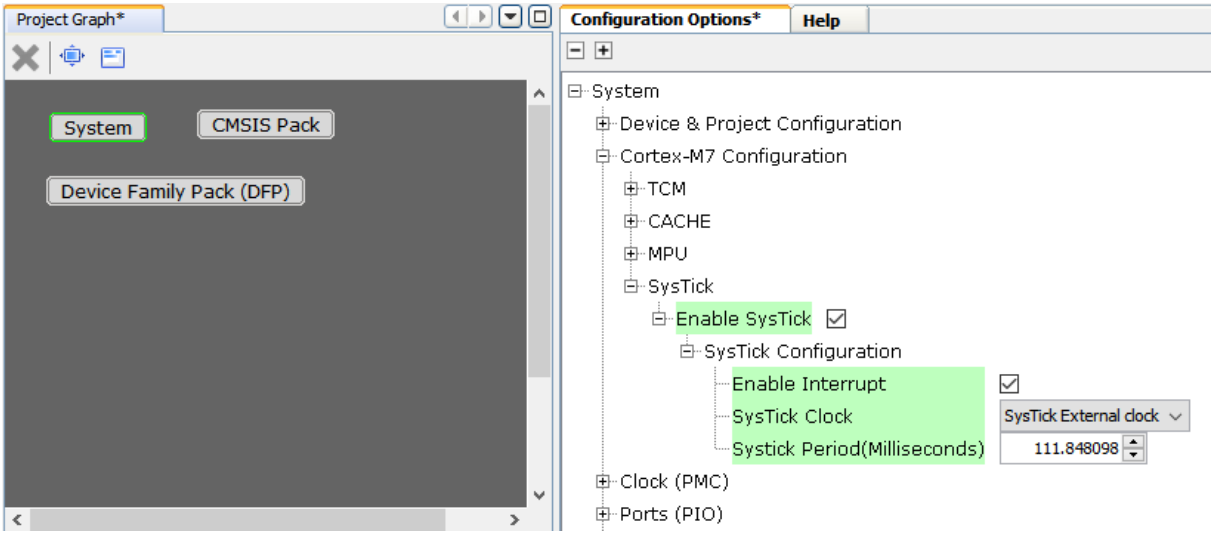
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_c21n_xpro.X	SAM C21N Xplained pro Evaluation Kit

MPLAB Harmony Configurations

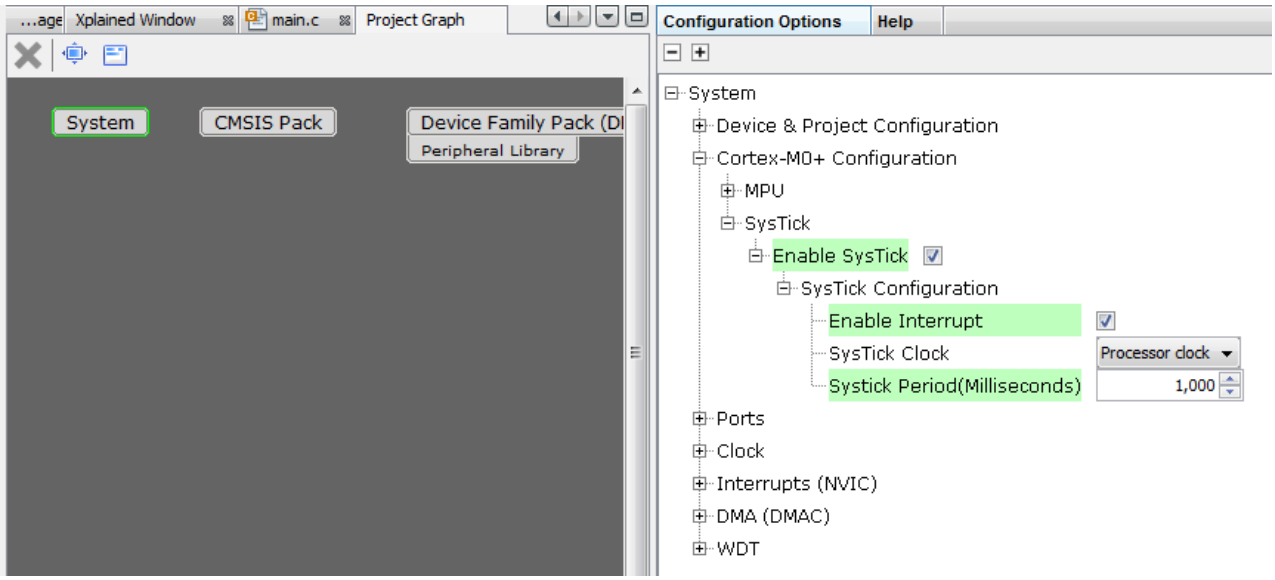
This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application for **sam_e70_xult.X**.



The Project Graph diagram below shows the Harmony components that are included in this application for **sam_c21n_xpro.X**.



Components Used

1. SysTick Peripheral Library

MHC Settings

1. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

Running the Application

1. Build and Program the application using MPLAB X IDE.
2. Observe Output as described in the below table.

Project Name	Output
sam_e70_xult.X	Success: LED 1 **Blinks**
sam_c21n_xpro.X	Success: LED 0 **Blinks**

TC Applications

This Section provides help for TC PLIB Applications.

tc_timer_mode

This example demonstrates how to use TC module in timer mode to generate periodic interrupt.

Description

This example demonstrates how to use TC module in timer mode to generate periodic interrupt. LED in toggle to indicate periodic callback.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/tc/tc_timer_mode/firmware

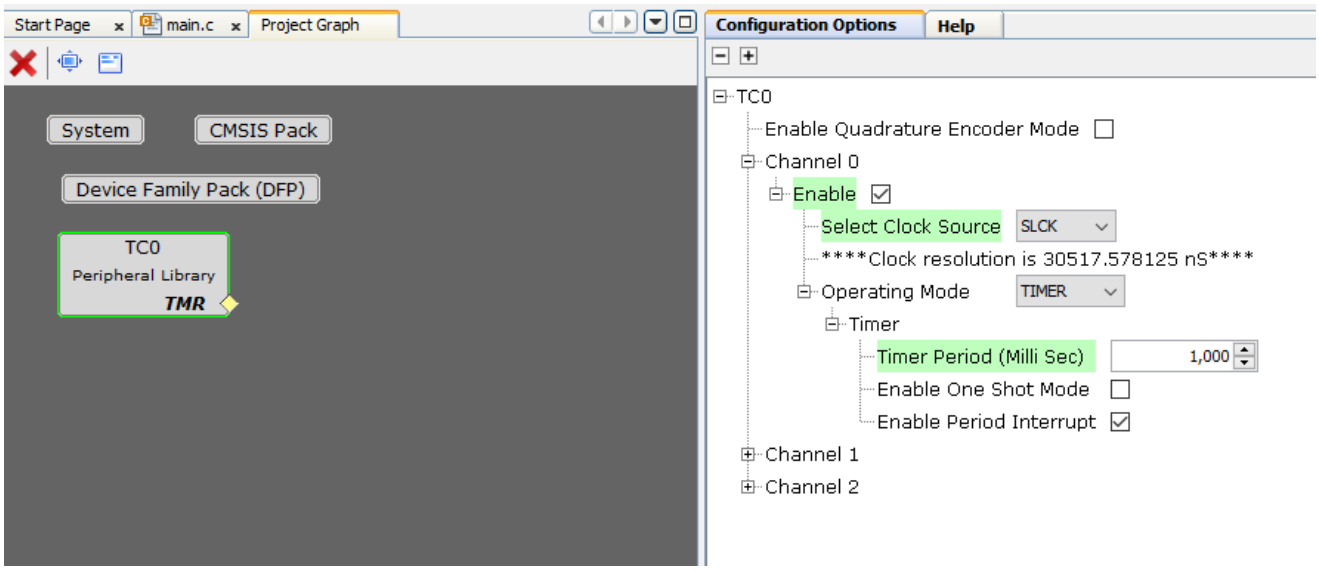
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit.

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

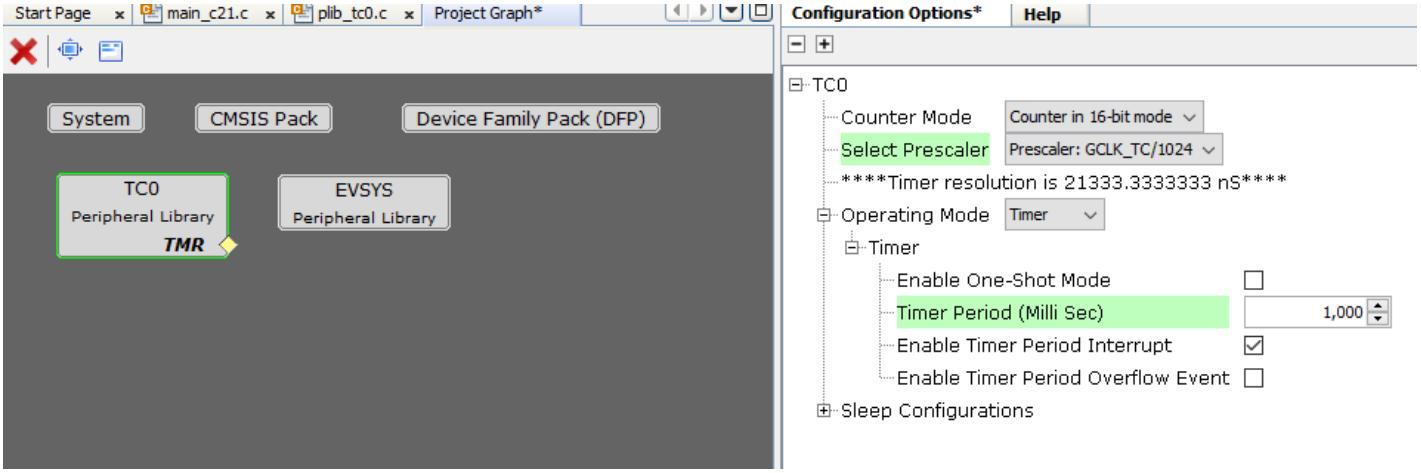
SAM E70 Xplained Ultra board and SAM V71 Xplained Ultra board MHC configurations:



Components Used

TC Peripheral Library

SAM C21n Xplained Pro Evaluation Kit MHC configurations:



Components Used

TC Peripheral Library

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit.
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED blinks at 1 sec rate
sam_v71_xult.X	Success: LED blinks at 1 sec rate

sam_c21n_xpro.X	Success: LED blinks at 1 sec rate

tc_compare_mode

This example demonstrates how to use TC module in compare mode to generate active low, active high and toggle output on compare match

Description

This example demonstrates how to use TC module in compare mode to generate active low, active high and toggle output on compare match.

SAM E70 Xplained Ultra board and SAM V71 Xplained Ultra board :

Active low output: Output sets high at compare match A and sets low at compare match C

Active high output: Output sets low at compare match A and sets high at compare match C

Toggle output: Output toggles at compare match A

SAM C21n Xplained Pro Evaluation Kit :

Active low output: Output sets high at compare match.

Active high output: Output sets low at compare match.

Toggle output: Output toggles at compare match.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/tc/tc_compare_mode/firmware

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

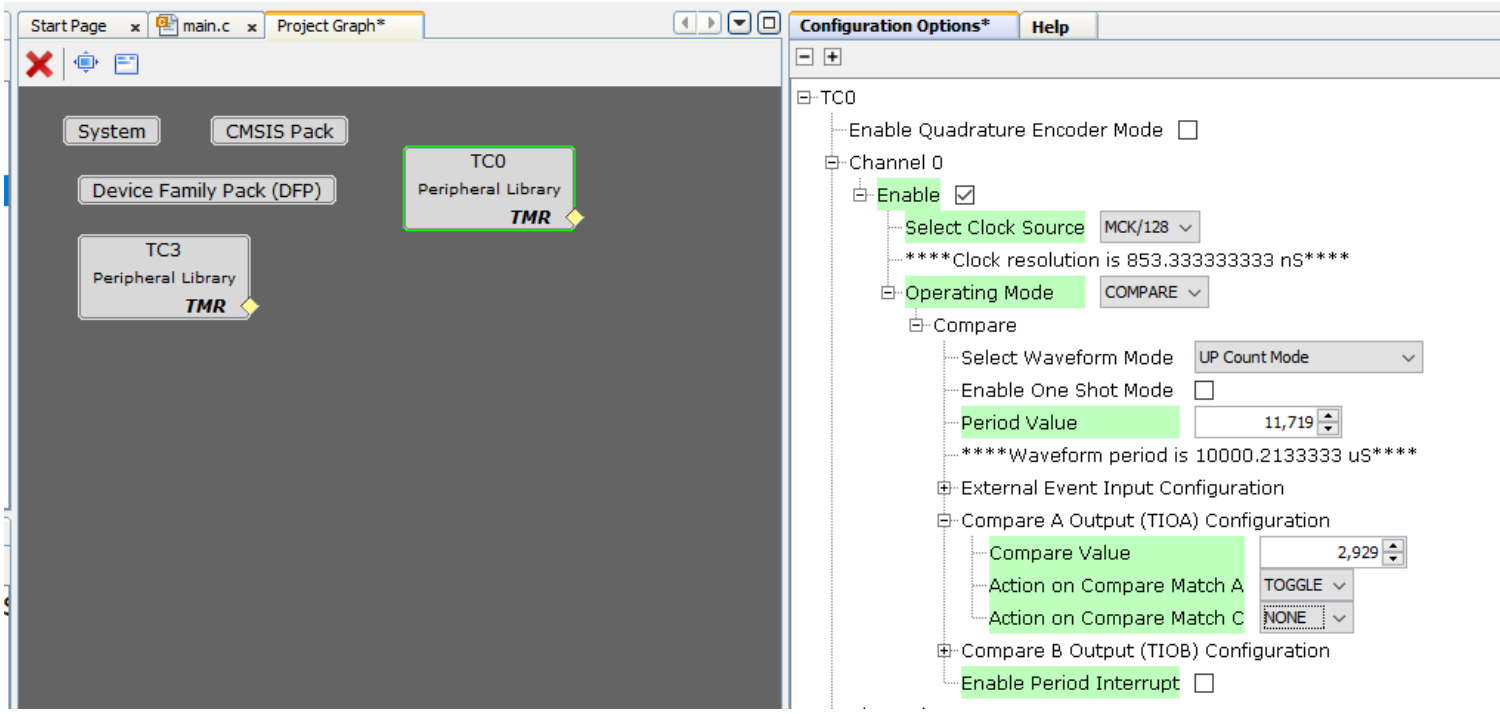
MPLAB Harmony Configurations

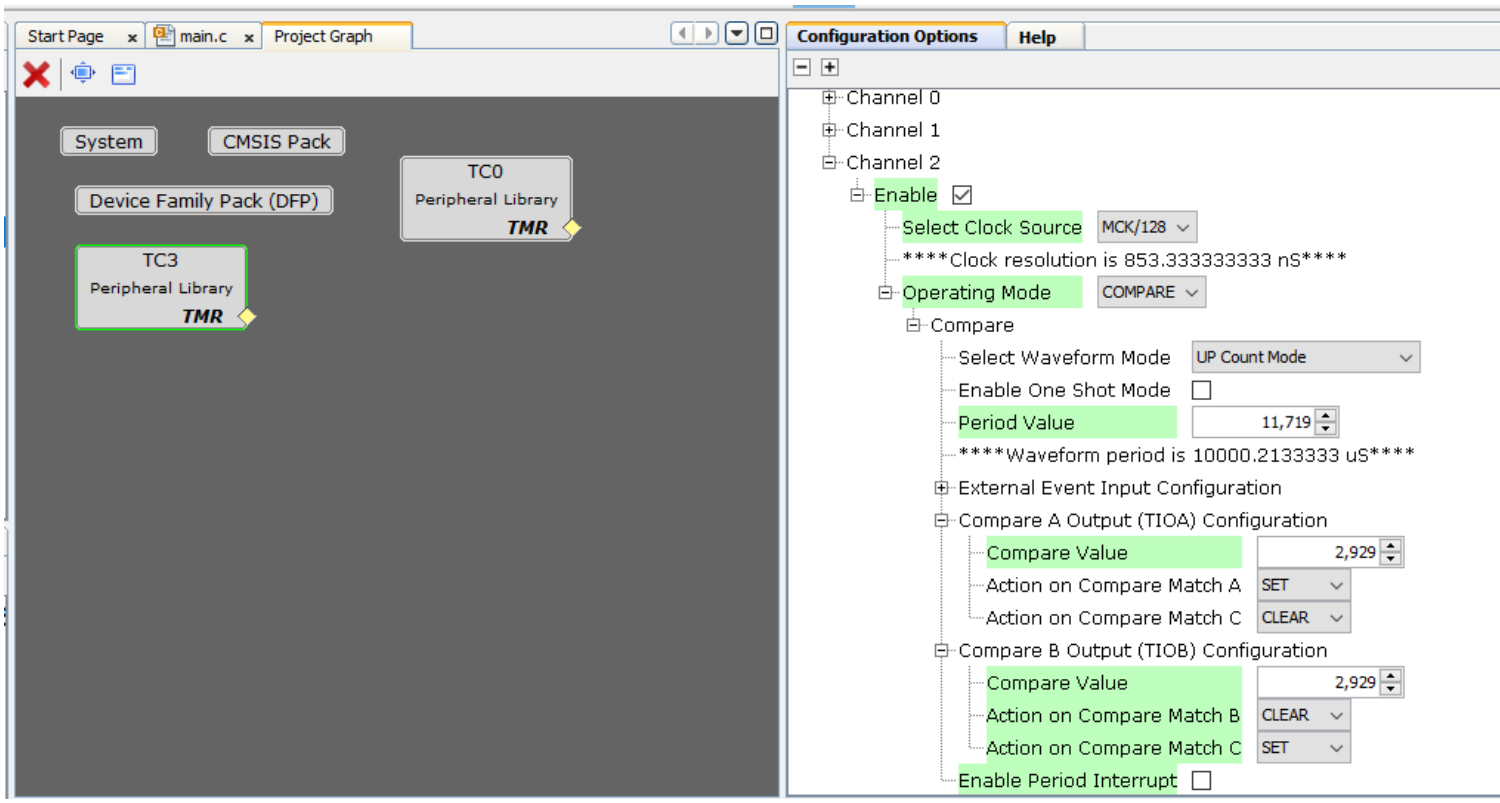
This topic provides information on MHC configurations

Description

SAM E70 Xplained Ultra board and SAM V71 Xplained Ultra board MHC configurations:

TC Configurations:



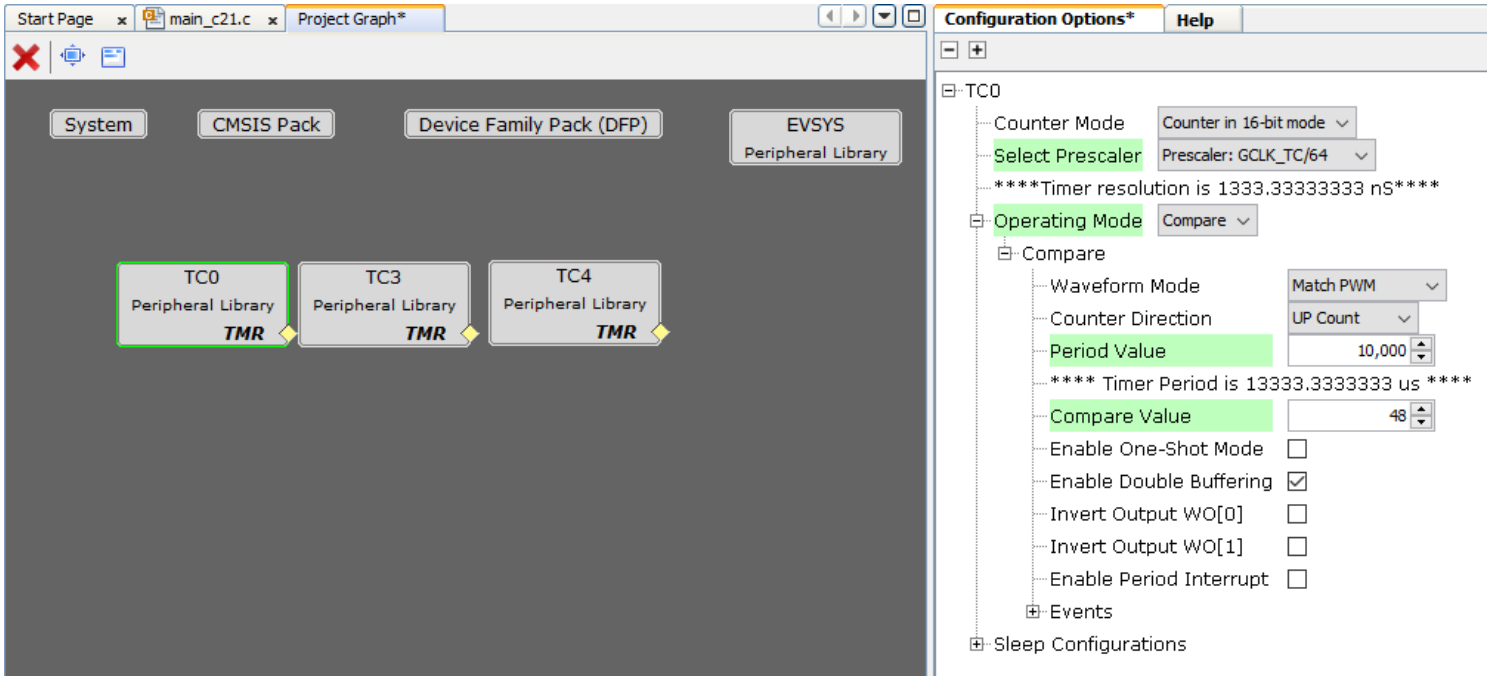


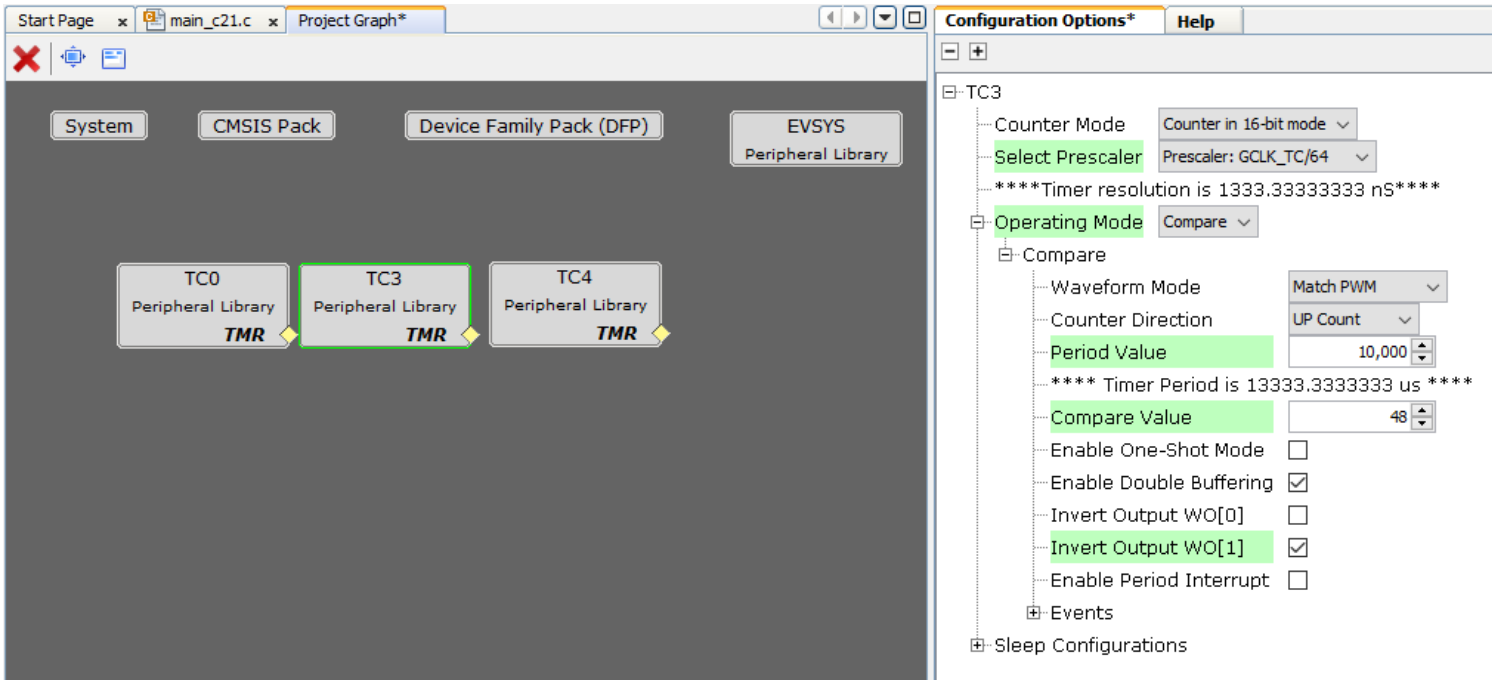
Components Used

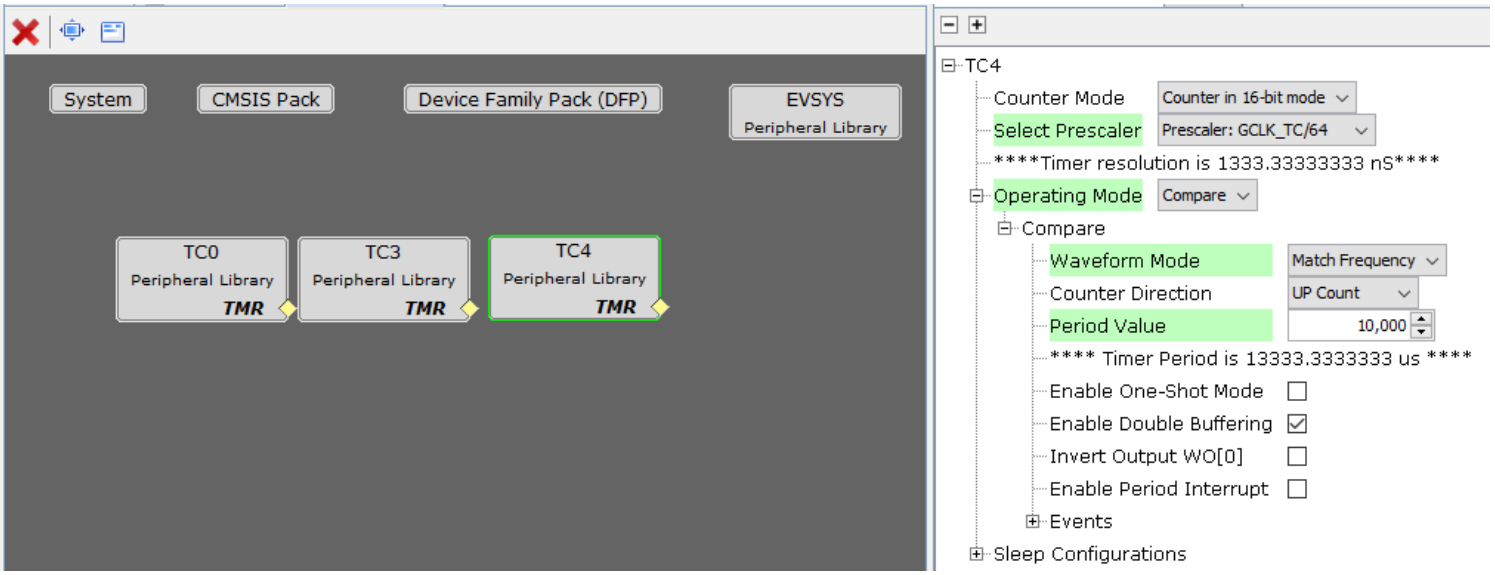
TC Peripheral Library

SAM C21n Xplained Pro Evaluation Kit MHC configurations:

TC Configurations:


Start Page
main_c21.c
Project Graph*
System
CMSIS Pack
Device Family Pack (DFP)
EVSYS
Peripheral Library
TC0
Peripheral Library
TMR
TC3
Peripheral Library
TMR
TC4
Peripheral Library
TMR
Configuration Options*
Help
TC0
Counter Mode
Counter in 16-bit mode
Select Prescaler
Prescaler: GCLK_TC/64
****Timer resolution is 1333.33333333 nS****
Operating Mode
Compare
Compare
Waveform Mode
Match PWM
Counter Direction
UP Count
Period Value
10,000
**** Timer Period is 13333.3333333 us ****
Compare Value
48
Enable One-Shot Mode
Enable Double Buffering
Invert Output WO[0]
Invert Output WO[1]
Enable Period Interrupt
Events
Sleep Configurations


Start Page
main_c21.c
Project Graph*
System
CMSIS Pack
Device Family Pack (DFP)
EVSYS
Peripheral Library
TC0
Peripheral Library
TMR
TC3
Peripheral Library
TMR
TC4
Peripheral Library
TMR
Configuration Options*
Help
TC3
Counter Mode
Counter in 16-bit mode
Select Prescaler
Prescaler: GCLK_TC/64
****Timer resolution is 1333.33333333 nS****
Operating Mode
Compare
Compare
Waveform Mode
Match PWM
Counter Direction
UP Count
Period Value
10,000
**** Timer Period is 13333.3333333 us ****
Compare Value
48
Enable One-Shot Mode
Enable Double Buffering
Invert Output WO[0]
Invert Output WO[1]
Enable Period Interrupt
Events
Sleep Configurations



Components Used

TC Peripheral Library

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit.
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

- SAM E70 Xplained Ultra Evaluation Kit and SAM V71 Xplained Ultra Evaluation Kit
  1. Build and Program the application using MPLAB X IDE
  2. Observe "Pin 7 of EXT1 (TC0_TIOA0)" on oscilloscope which generates toggle output

3. Observe "Pin 16 of EXT1 (TC3_TIOA11)" on oscilloscope which generates active low output
4. Observe "Pin 18 of EXT1 (TC3_TIOB12)" on oscilloscope which generates active high output
5. Observe Output as described in the below table

- SAM C21n Xplained Pro Evaluation Kit.
  1. Build and Program the application using MPLAB X IDE
  2. Observe "Pin 2 of J801 (TC0_WO1)" on oscilloscope which generates active low output
  3. Observe "Pin 6 of J804 (TC3_WO1)" on oscilloscope which generates active high output
  4. Observe "Pin 7 of EXT1 (TC4_WO0)" on oscilloscope which generates toggle output
  5. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: Observe waveforms on oscilloscope
sam_v71_xult.X	Success: Observe waveforms on oscilloscope
sam_c21n_xpro.X	Success: Observe waveforms on oscilloscope

tc_capture_mode

This example demonstrates how to use TC module in capture mode to measure duty cycle and frequency of external input.

Description

This example demonstrates how to configure TC channel in capture mode to measure duty cycle and frequency of external PWM input which is generated by another TC channel in compare mode.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/tc/tc_capture_mode/firmware

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

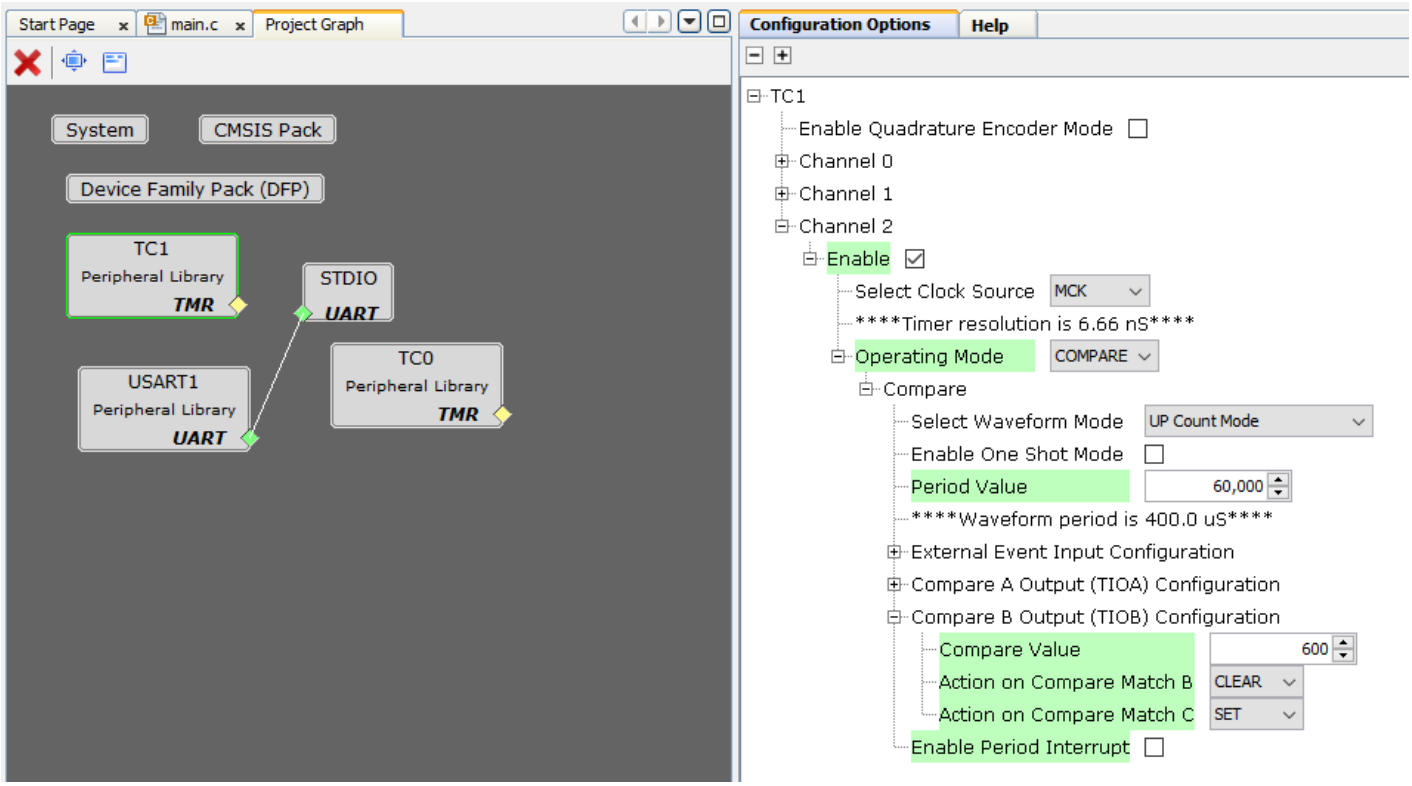
MPLAB Harmony Configurations

This topic provides information on MHC configurations

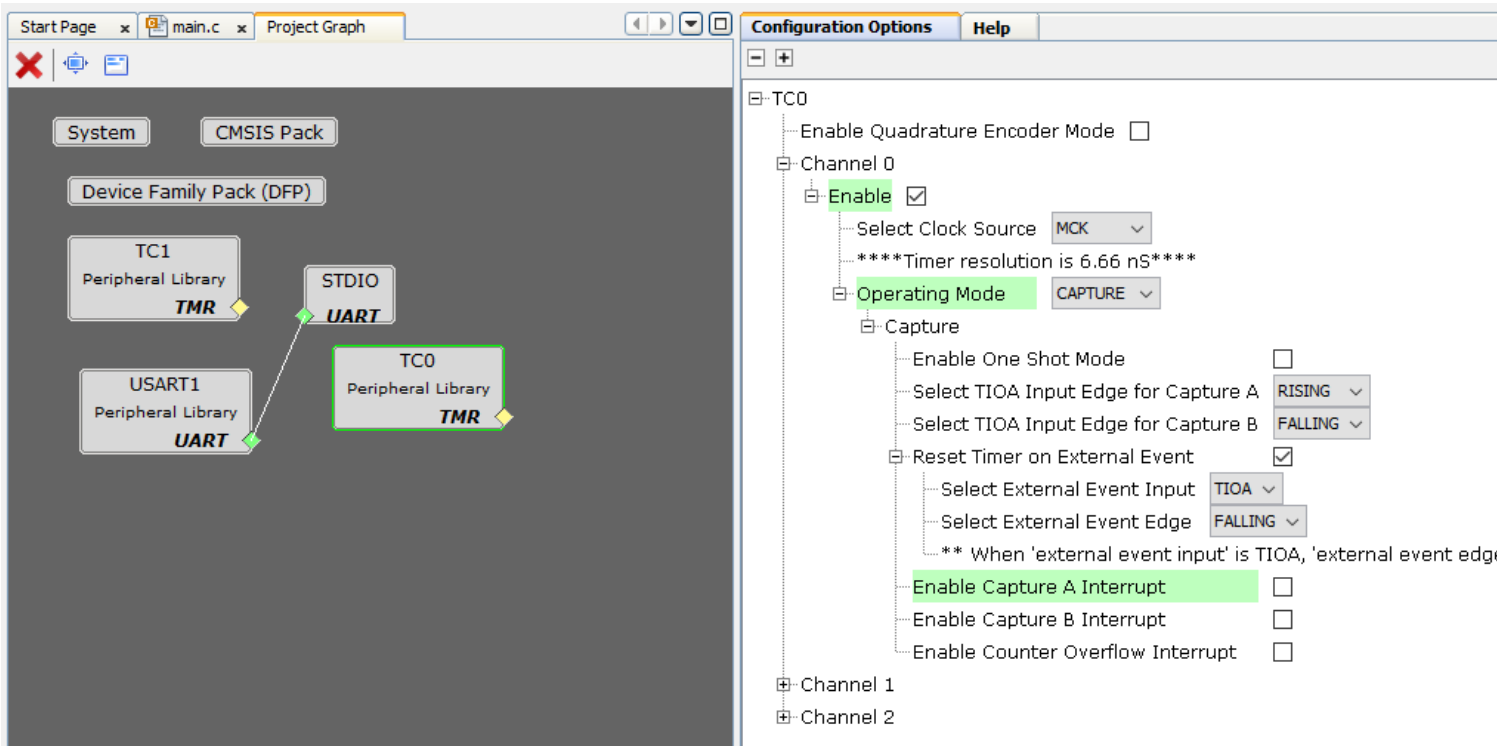
Description

SAM E70 Xplained Ultra board and SAM V71 Xplained Ultra board MHC configurations:

TC Compare Configurations:



TC Capture Configurations:



Components Used:

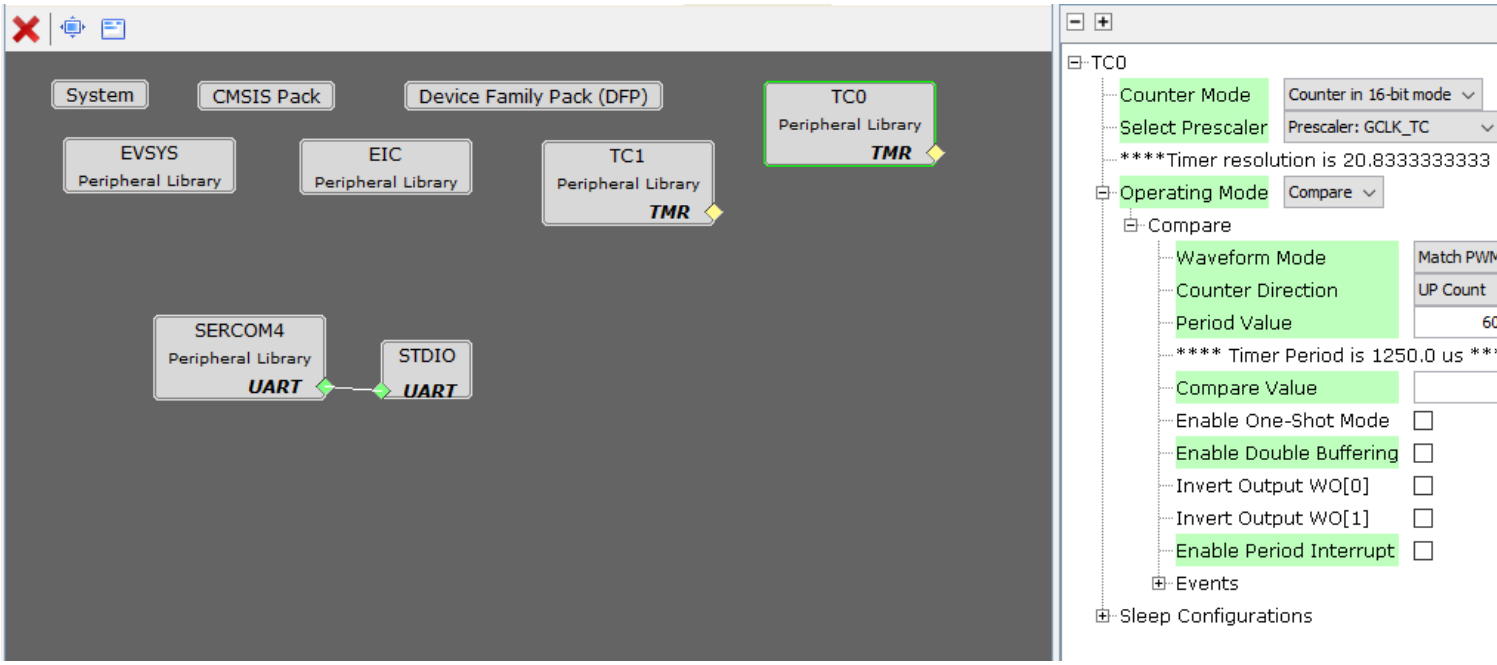
TC Peripheral Library

USART Peripheral Library

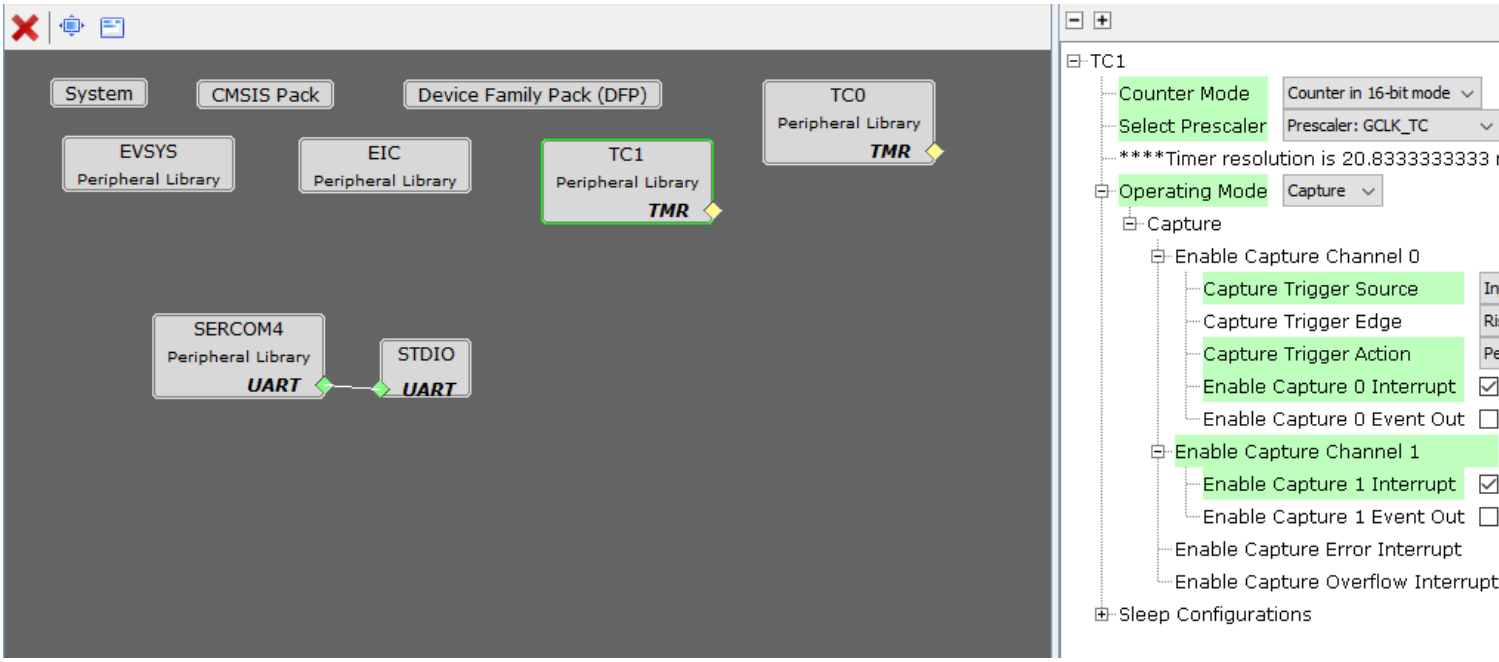
STDIO

SAM C21n Xplained Pro Evaluation Kit MHC configurations:

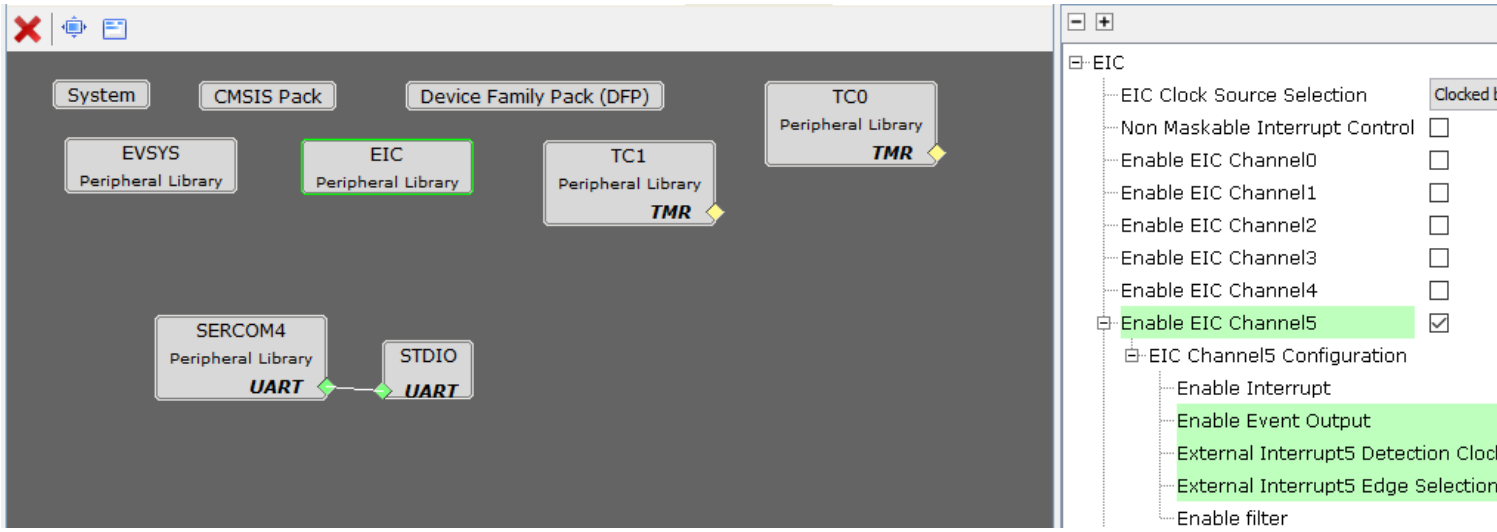
TC Compare Configurations:



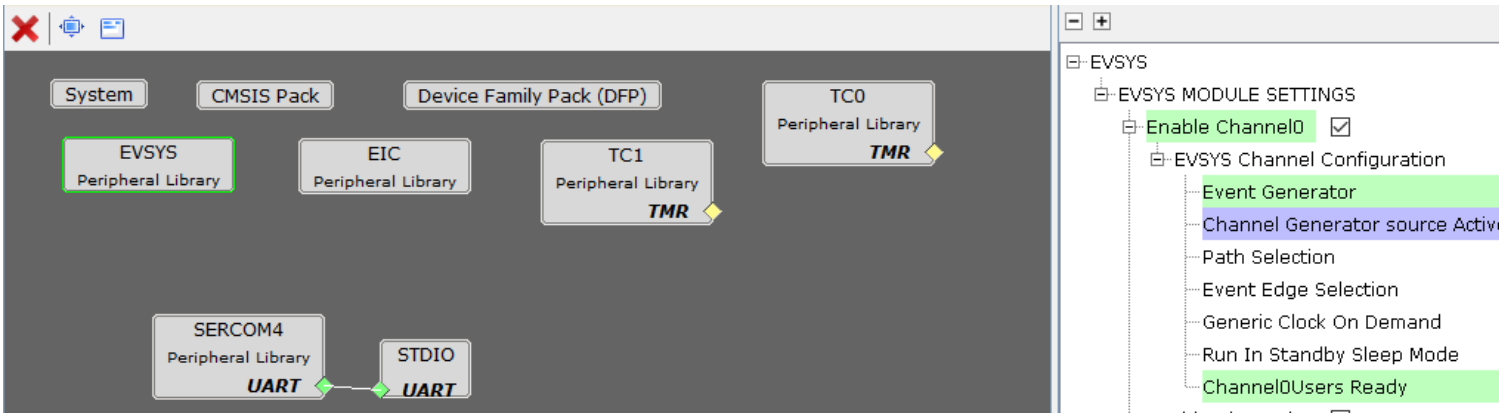
TC Capture Configurations:



EIC Configurations:



Event System Configurations:



Components Used:

TC Peripheral Library

USART Peripheral Library

STDIO

EIC Peripheral Library

EVSYS(Event System)

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - TC1 generates PWM waveform on pin PC30
    - TC0 is in capture mode with input signal connected to pin PA00
    - Use jumper wire to connect "Pin 8 of EXT1 (TC1_TIOB5)" to "Pin 7 of EXT1 (TC0_TIOA0)"

- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit

- **Hardware Setup**
  - TC1 generates PWM waveform on pin PC30
  - TC0 is in capture mode with input signal connected to pin PA00
  - Use jumper wire to connect "Pin 8 of EXT1 (TC1_TIOB5)" to "Pin 7 of EXT1 (TC0_TIOA0)"

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit.
  - **Hardware Setup**
    - TC0 generates PWM waveform on pin PA09
    - TC1 is in capture mode with input signal from TC0 is triggered using EIC and EVSYS
    - Use jumper wire to connect "Pin 4 of EXT2 (TC0_W01)" to "Pin 6 of EXT2 (EIC_EXTINT5)"

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port
2. Open Terminal application (Ex.:Tera term) on the computer
3. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Build and Program the application using MPLAB X IDE
5. Console displays frequency and duty cycle of the input signal.

```
-------------------------------------------------------
                    TC Capture Demo
-------------------------------------------------------
Connect input signal to TC0_CH0 PA00 Pin

Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 1.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 1.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 2.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 3.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 4.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 5.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 6.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 7.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 8.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 9.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 10.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 11.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 12.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 13.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 14.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 15.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 16.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 17.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 18.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 19.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 20.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 21.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 22.00 %
Frequency of waveform: 2500.00 Hz        Duty Cycle of waveform: 23.00 %
```

6. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: Console displays frequency and duty cycle of input signal. Frequency is constant and duty cycle changes by 1%
sam_v71_xult.X	Success: Console displays frequency and duty cycle of input signal. Frequency is constant and duty cycle changes by 1%
sam_c21n_xpro.X	Success: Console displays frequency and duty cycle of input signal. Frequency is constant and duty cycle changes by 1%

TCC Applications

This Section provides help for TCC PLIB Applications.

tcc_synchronous_pwm_channels

This example application shows how to use TCC to generate 3 phase PWM for motor control with dead time.

Description

This example application shows how to use TCC Peripheral library to generate a 3 phase PWM with dead time enabled used for motor control. The duty cycle of the PWM is updated in the interrupt handler.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	/csp/apps/tcc/tcc_synchronous_pwm_channels/firmware

Project Name	Description
sam_c21n_xpro.X	SAM C21n Xplained Pro Evaluation Kit

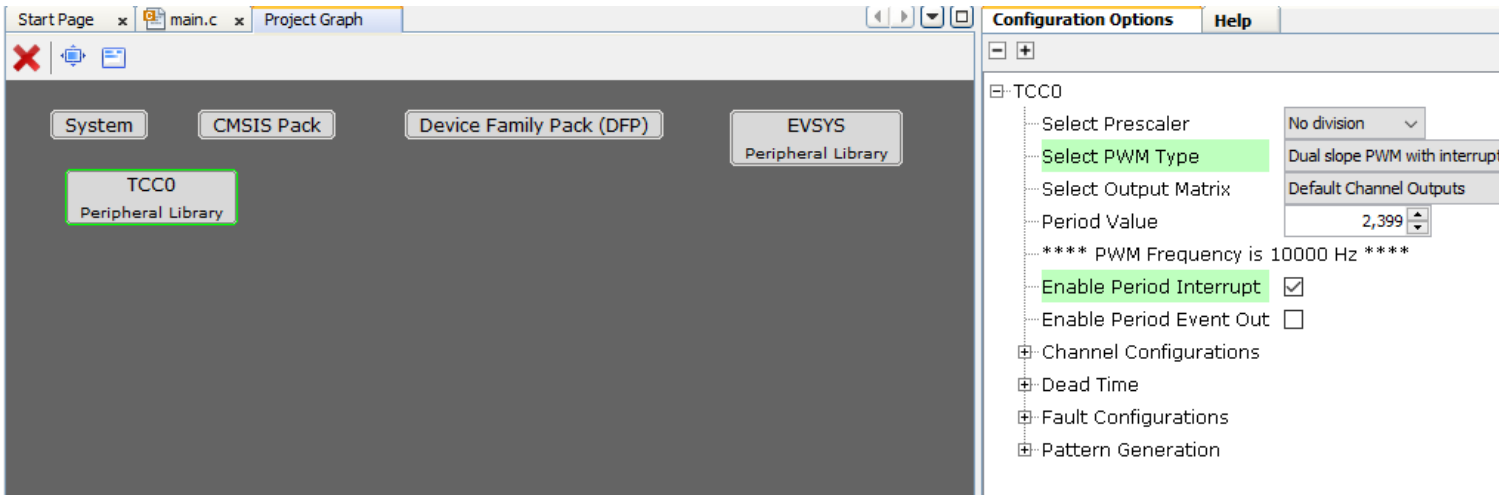
MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The following Harmony components and configurations are used in this application.

TCC Configurations:



Components Used

1. TCC Peripheral Library

MHC Settings

1. Configure "TCC0 Peripheral library" to generate PWM.

Hardware Setup

This section describes how to configure the supported hardware.

Description

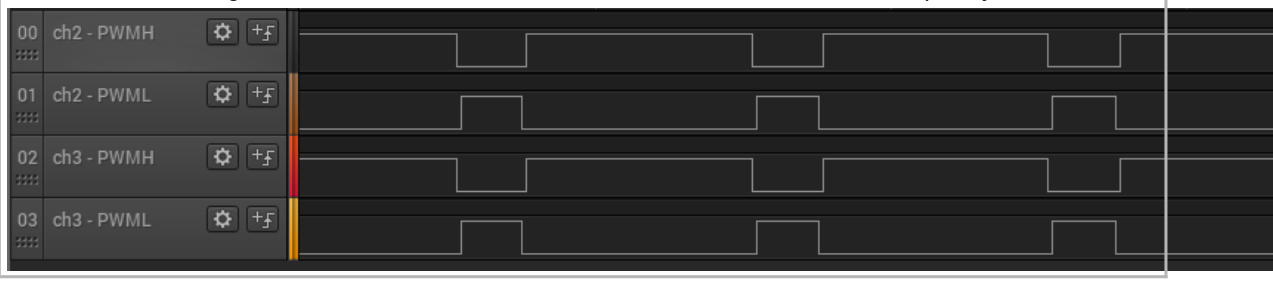
- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21n Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE.
2. Observe high-side and low-side PWM waveforms on oscilloscope. Observe dead time between high side and low side.
   - Channel 2
     - Pin 5 of EXT1 - PWMH (PA10)
     - Pin 7 of EXT1 - PWML (PB12)
   - Channel 3
     - Pin 6 of EXT1 - PWMH (PA11)
     - Pin 8 of EXT1 - PWML (PB13)
3. Observe Output as described in the below table

Project Name	Output
sam_c21n_xpro.X	Success: Observe high side and low side PWM waveforms as shown below. PWM frequency = 10KHz 

TCM Applications

This Section provides help for TCM Applications.

data_tcm

This example demonstrates maintaining cache coherency by allocating DMA buffers in Tightly Coupled Memory (TCM), which are not cacheable.

Description

This example application shows how to use the data TCM to ensure a high-speed, low latency, and deterministic access to data. TCM is a memory accessed by a dedicated connection from the core. Access to the TCM memory is at full processor clock speed and can be accessed both by CPU and by DMA. The TCM provides deterministic behavior as it does not suffer from cache hit and cache miss. There is no explicit/extra memory for TCM. It is the internal SRAM of the microcontroller that gets partitioned as TCM + SRAM.

This application uses USART and DMA PLIBs to demonstrate the TCM usage. It registers a callback with DMA transmit and receive channels. The application first transmits a message using DMA transmit channel and then schedules a read of 10 characters using DMA receive channel. Once the DMA read is complete, it reads the received data and echoes the same on the terminal using the DMA transmit channel.

Unlike the cache maintenance APIs, using TCM does not pose restrictions on aligning the read/write buffers to 32-byte boundary and the size to be a multiple of 32 bytes. A XC32 compiler attribute can be used to place the read/write buffers into the TCM memory region to avoid the issues related to cache coherency when DMA is used with data cache enabled.

Building The Application

This section provides information on how to build and run the application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\tcm\data_tcm\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

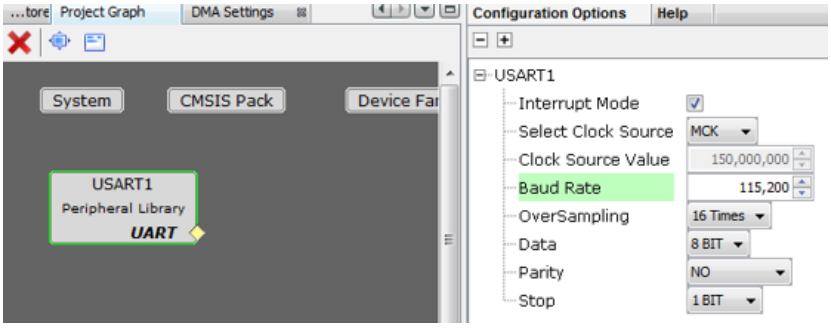
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

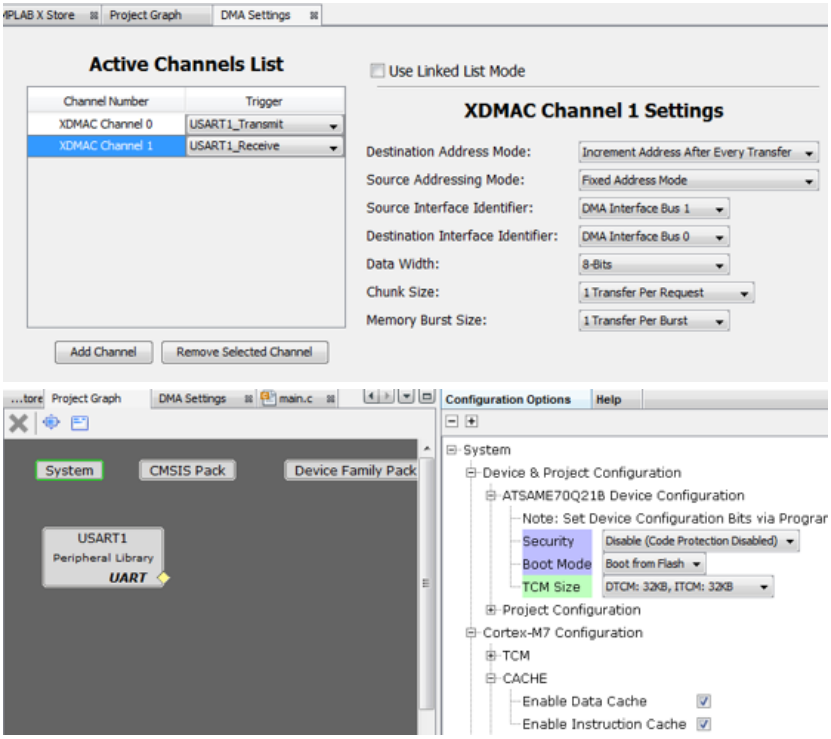
MPLAB Harmony Configuration

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.





Modules Used:

1. USART Peripheral Library
2. XDMAC Module

MHC Settings:

1. Enable TCM (DTCM = 32KB, ITCM = 32KB)
2. Configure Pins for USART Peripheral in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

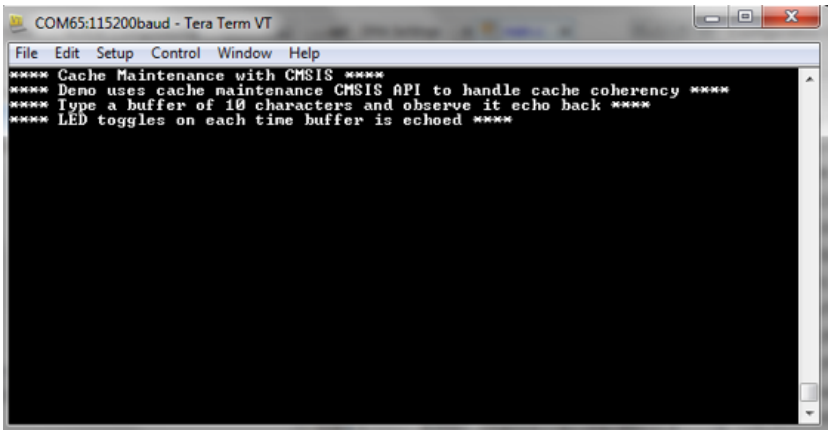
2. **Project: sam_v71_xult.X**
   - Hardware Used:
     - SAM V71 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

Running The Application

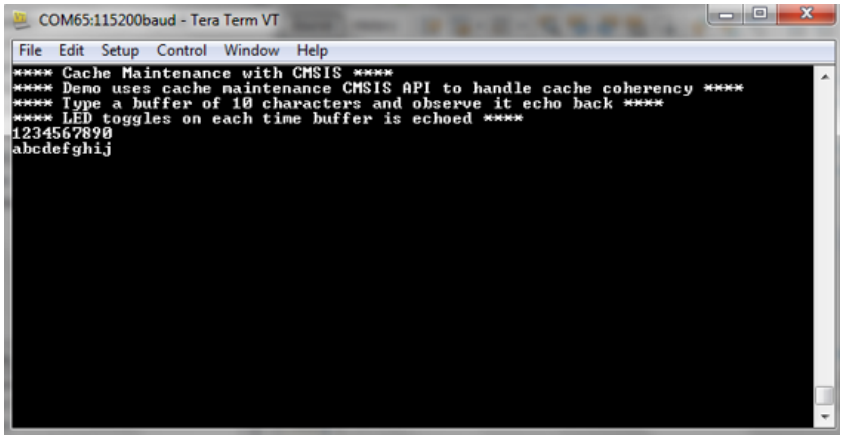
This section describes how to run the application.

Description

- **sam_e70_xult.X**
- **sam_v71_xult.X**
  1. Connect micro USB to DEBUG port.
  2. Build the **data_tcm** example code and program using MPLABX
  3. Open Terminal application (Ex.:Tera term) on the computer
  4. Connect to serial port and configure serial settings as follows
     - Baud : 115200
     - Data : 8 Bits
     - Parity : None
     - Stop : 1 Bit
     - Flow Control : None
  5. Console displays the following message



6. Enter 10 characters and observe it echo back. LED 1 toggles each time the buffer is echoed back.



TRNG Applications

This Section provides help for TRNG PLIB Applications.

random_number

This example application shows how to use TRNG Peripheral library to generate and read random number.

Description

This example application shows how to use TRNG Peripheral library to generate and read random number using true random number generator peripheral.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

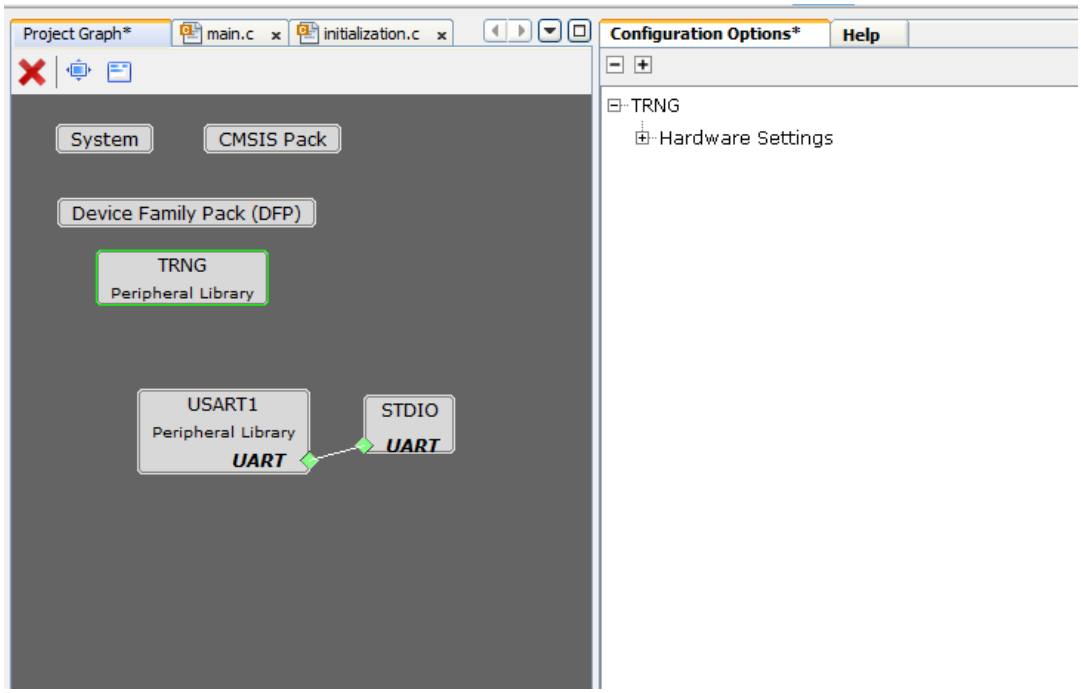
1. Connect micro USB to DEBUG port of SAME70 Xplained Ultra
2. Open Terminal application (Ex.:Tera term) on the computer
   - Connect to serial port and configure serial settings as follows,
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
3. Build and Program the application using MPLAB X IDE
4. Observe the on board LED and Terminal Application
   1. Console displays the following message
      - ----------
      - TRNG DEMO
      - ----------
      - Press SW0 to generate a 32-bit random number
   2. Press SW0 to generate the 32-bit True random Number
   3. LED0 should stop blinking and console should print the following message
      - The generated 32-bit random number is :- 0xYYYYYYYY
   4. Press SW0 to generate another random number

MPLAB Harmony Configurations

This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. TRNG Periheral Library
2. USART1/UART-STDIO Peripheral Library

MHC Settings

1. Configure Pins for USART1/UART Peripheral in Pin Settings
2. Configure GPIO pin for LED and SWITCH

Hardware Setup

Hardware Used

1. SAM E70 Xplained Ultra Evaluation Kit

Setup

- No Special hardware Setup Required.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

To build this application, open the below listed project file in MPLABX IDE.

Project Path	Project Name	Description
\csp\apps\trng\random_number\firmware	sam_e70_xult.X	SAM E70 Xplained Ultra board

TWIHS Applications

This Section provides help for TWIHS PLIB Applications.

twihs_eeprom

This example demonstrates how to write and read from I2C based Serial EEPROM memory.

Description

This example demonstrates how to write and read from the on-board AT24MAC402 I2C based Serial EEPROM memory. The application writes 16 bytes of data "ATSAM TWIHS Demo" starting from EEPROM memory address 0x00.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

The parent folder for all the MPLABX projects for this application is given below

Application Path	csp/apps/twihs/master/twihs_eeprom/firmware

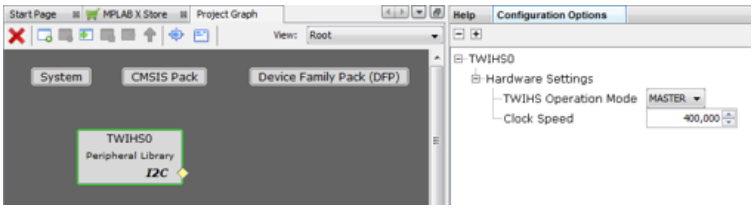
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Components Used

1. TWIHS Periheral Library

MHC Settings

1. Configure Pins for TWIHS Peripheral in Pin Settings
2. Configure GPIO pin for EEPROM write-protect in Pin Settings
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_v71_xult.X**
  - **Hardware Used**
    - SAM V71 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Build and Program the application using MPLAB X IDE
2. Observe Output as described in the below table

Project Name	Output
sam_e70_xult.X	Success: LED 1 ON. Failure: LED 1 OFF
sam_v71_xult.X	Success: LED 0 ON. Failure LED 0 OFF

USART Applications

This Section provides help for USART PLIB Applications.

usart_echo_blocking

This application shows how to use USART PLIB to read-write data over console in blocking mode.

Description

This application accepts a line of characters and echoes it back over the console using USART PLIB in the blocking mode.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\usart\usart_echo_blocking\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.

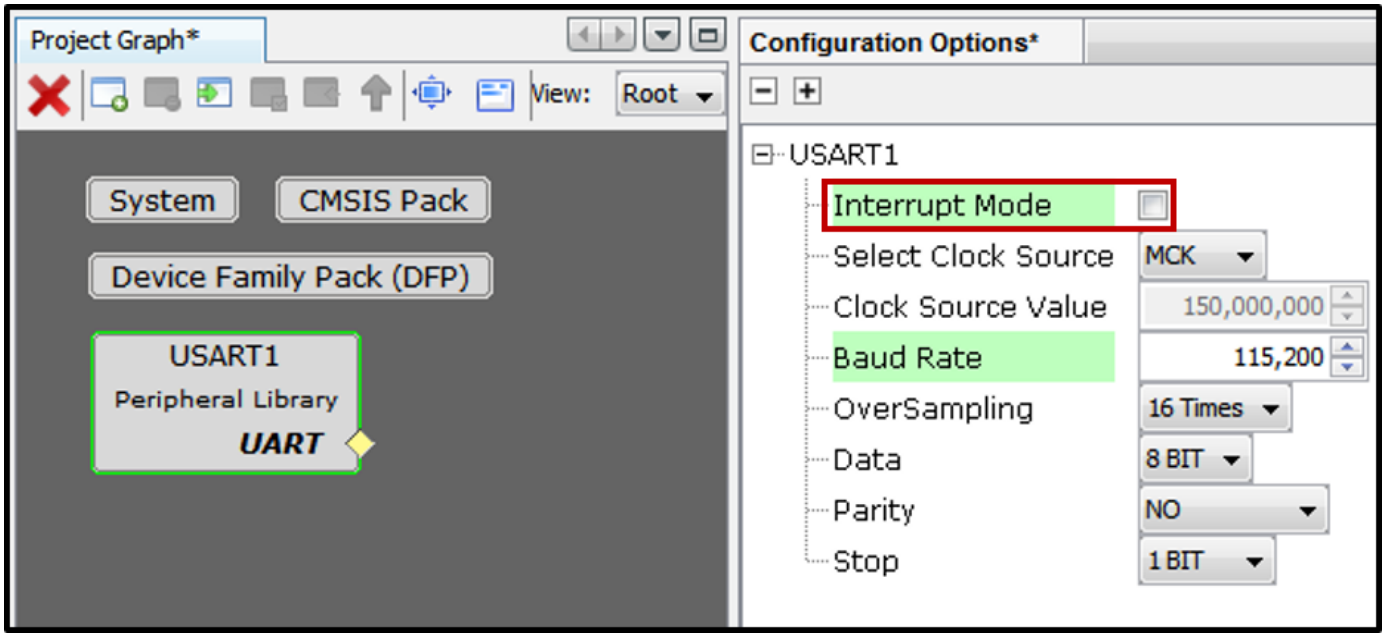


Figure 1: Project Graph and USART Configuration Option

Modules Used:

1. USART Peripheral Library

MHC Settings:

1. Configure Pins for USART Peripheral in Pin Settings
2. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:

- No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - Hardware Used:
     - SAM V71 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port.
2. Build the **usart_echo_blocking** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
- Console displays the following message,
**** USART Line Echo Demo ****
**** Demo uses blocking model of USART PLIB. ****
**** Enter a line of characters, press ENTER key and observe it echo back. ****
**** LED toggles on each time the line is echoed ****
- Enter a line of characters and press ENTER key.
- Entered line will echo back and LED1 toggles
- Repeat the steps
```

usart_echo_interrupt

This application shows how to use USART PLIB to read-write data over console in interrupt mode.

Description

This demonstration accepts a buffer of characters and echoes it back over the console using USART PLIB in the interrupt mode.

Building The Application

This section provides information on how to build application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	csp\apps\usart\usart_echo_interrupt\firmware

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configurations

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.

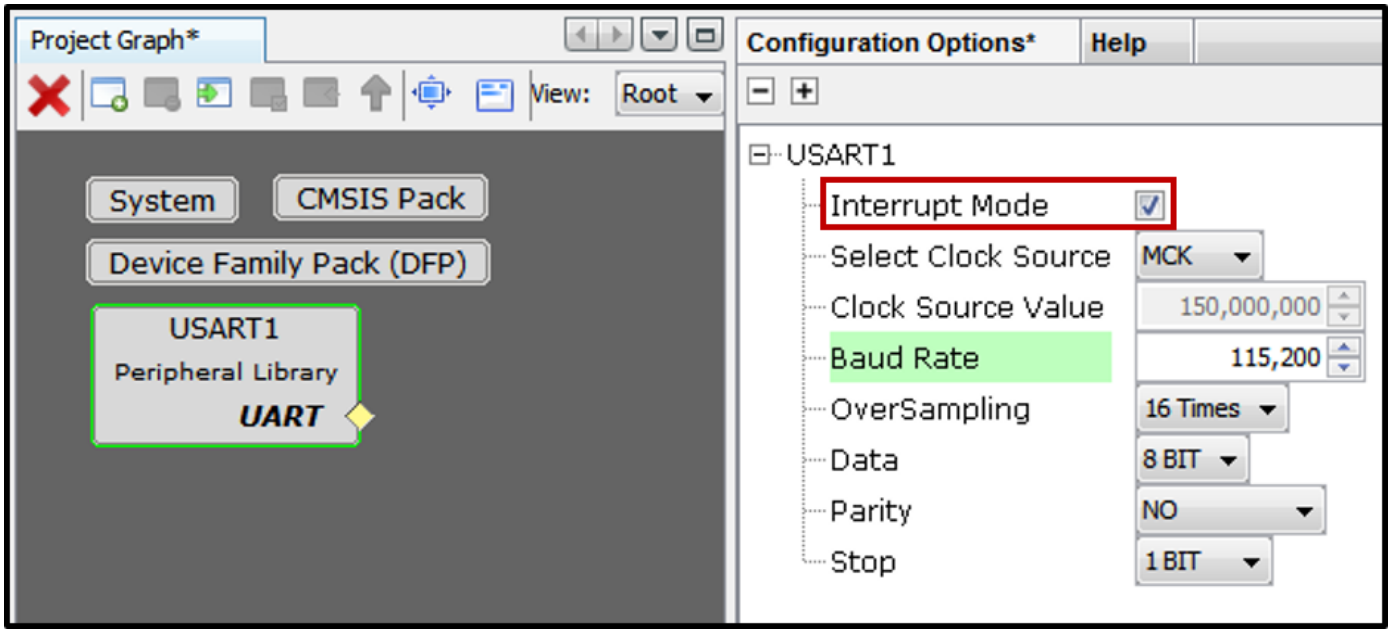


Figure 1: Project Graph and USART Configuration Option

Modules Used:

1. USART Peripheral Library

MHC Settings:

1. Configure Pins for USART Peripheral in Pin Settings
2. Enable Interrupt Mode
3. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

2. **Project: sam_v71_xult.X**
   - Hardware Used:
     - SAM V71 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

Running The Application

This section provides information on how to run the application using MPLAB X IDE

Description

1. Connect micro USB to DEBUG port.
2. Build the **usart_echo_interrupt** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message

```
- Console displays the following message,
**** USART Echo Interrupt Demo ****
**** Type a buffer of 10 characters and observe it echo back ****
**** LED toggles on each time the buffer is echoed ****
- Enter the requested number of characters.
- Entered characters will echo back and LED1 toggles
- Repeat the steps
```

WDT Applications

This Section provides help for WDT PLIB Applications.

wdt_timeout

This example application shows how the watchdog timer resets by not feeding the watchdog on switch press.

Description

This example application shows how the WDT peripheral library resets the watchdog timer by not feeding the watchdog on switch press. The application sets up the watchdog to reset at 4 seconds. The application also sets up the SysTick timer to clear the watchdog reset every 100 milliseconds. A user switch press disables the feeding of watchdog resulting in watchdog reset after 4 seconds of switch press.

Building The Application

This section provides information on how to build the application using MPLAB X IDE.

Description

Building The Application

The parent folder for all the MPLABX projects for this application is given below.

Application Path	csp\apps\wdt\wdt_timeout\firmware

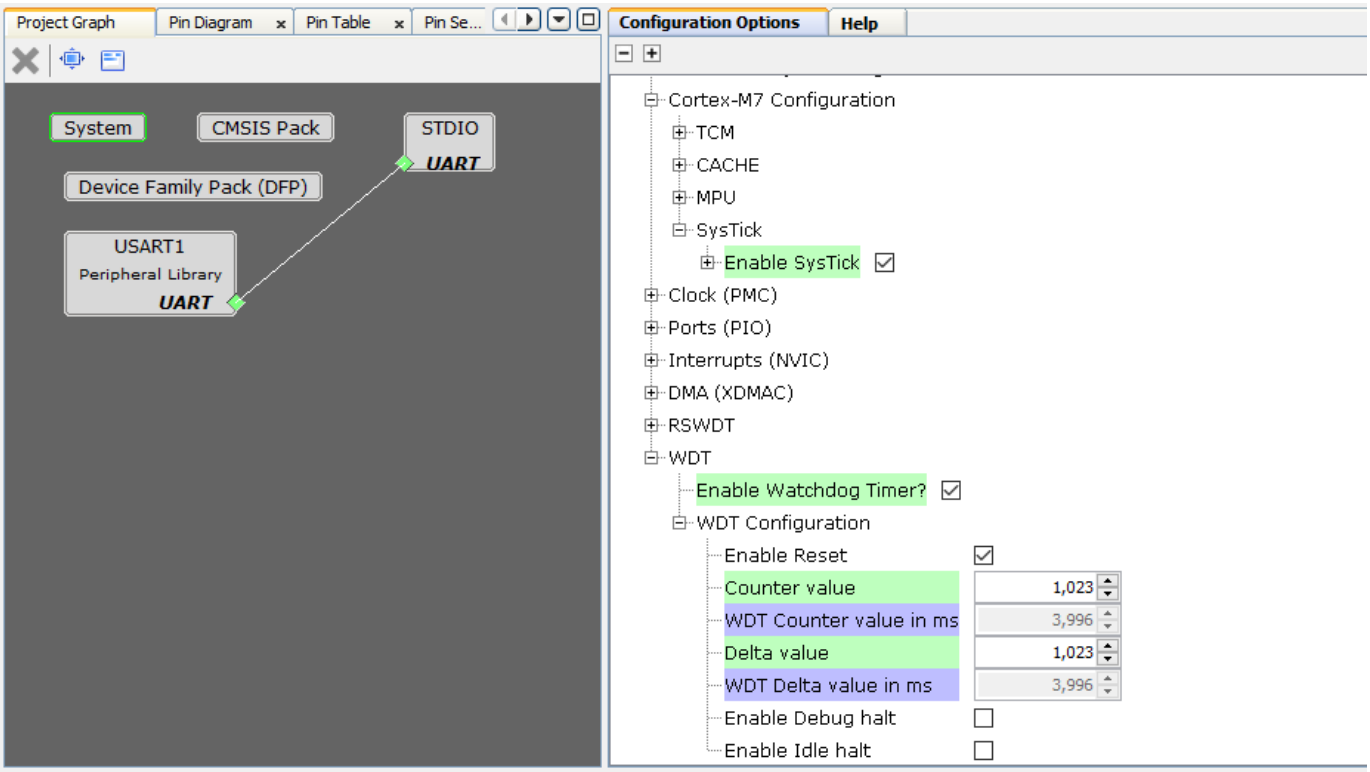
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_c21n_xpro.X	SAM C21N Xplained pro Evaluation Kit

MPLAB Harmony Configurations

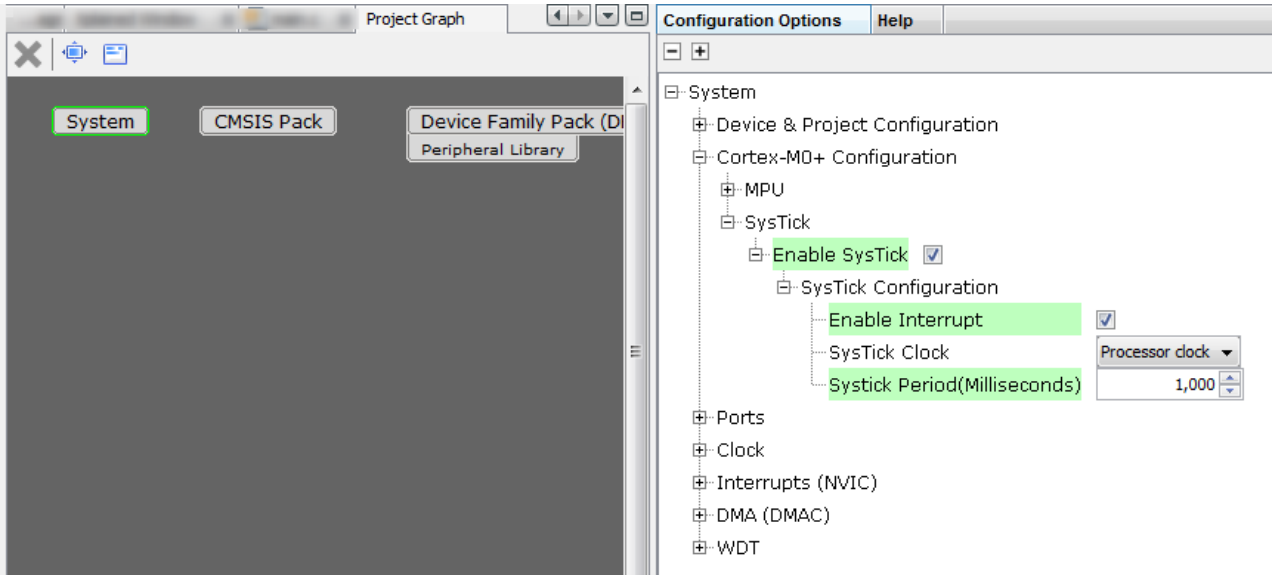
This topic provides information on MHC configurations.

Description

The Project Graph diagram below shows the Harmony components that are included in this application for **sam_e70_xult.X**.



The Project Graph diagram below shows the Harmony components that are included in this application for **sam_c21n_xpro.X**



Components Used

1. WDT Periheral Library
2. SysTick Peripheral Library
3. USART1-STDIO Peripheral Library on **sam_e70_xult.X**
4. SERCOM 4(USART) -STDIO Peripheral Library on **sam_c21n_xpro.X**

MHC Settings

1. Configure Pins for SERCOM/USART Peripheral in Pin Settings.
2. Configure GPIO pin for LED and SWITCH.
3. Enable SysTick in Cortex-M7/Cortex-M0 Configuration.

Hardware Setup

This section describes how to configure the supported hardware.

Description

- **Project sam_e70_xult.X**
  - **Hardware Used**
    - SAM E70 Xplained Ultra Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

- **Project sam_c21n_xpro.X**
  - **Hardware Used**
    - SAM C21N Xplained Pro Evaluation Kit
  - **Hardware Setup**
    - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Connect micro USB to DEBUG port of SAM E70 Xplained Ultra or SAM C21N
2. Open Terminal application (Ex.:Tera term) on the computer. Connect to serial port and configure serial settings as follows,
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
4. Build and Program the application using MPLAB X IDE.
5. Observe the on board LED and Terminal Application.
   1. LED0 should start blinking.
   2. Console displays the following message.

WDT DEMO

Press SW0 to emulate deadlock

3. Press SW0 to put system in deadlock.
4. LED0 should stop blinking and console should print the following message.

Emulating deadlock.....

WDT should reset device in 4 seconds

5. WDT will reset the device in 4 seconds and demo should restart.

XDMAC Applications

This Section provides help for XDMAC PLIB Applications.

xdmac_memory_transfer

This example demonstrates how to use DMA for memory to memory transfers and illustrates usage of burst size to transfer data more efficiently.

Description

This example application shows how to use the XDMAC PLIB to perform a memory-memory transfer. The application uses software trigger to initiate a memory-memory transfer from source buffer to destination buffer 1 with Memory Burst Size set to 1. Once the transfer is complete, it configures the XDMAC PLIB for a Memory Burst Size of 16 and initiates another memory-memory transfer from source buffer to destination buffer 2. For each transfer, the application measures the time taken for the transfer in terms of SYS TICK counts and displays the time taken with Memory Burst Size 1 and Memory Burst Size 16, on the console using the USART PLIB and STDIO printf.

Building The Application

This section provides information on how to build and run the application using MPLAB X IDE

Description

Building the application:

This application has multiple MPLABX projects. The parent folder for all the MPLABX projects for this application is given below:

Application Path	**csp\apps\xdmac\xdmac_memory_transfer\firmware**

To build the application, refer the below table and open the appropriate project file in MPLABX IDE.

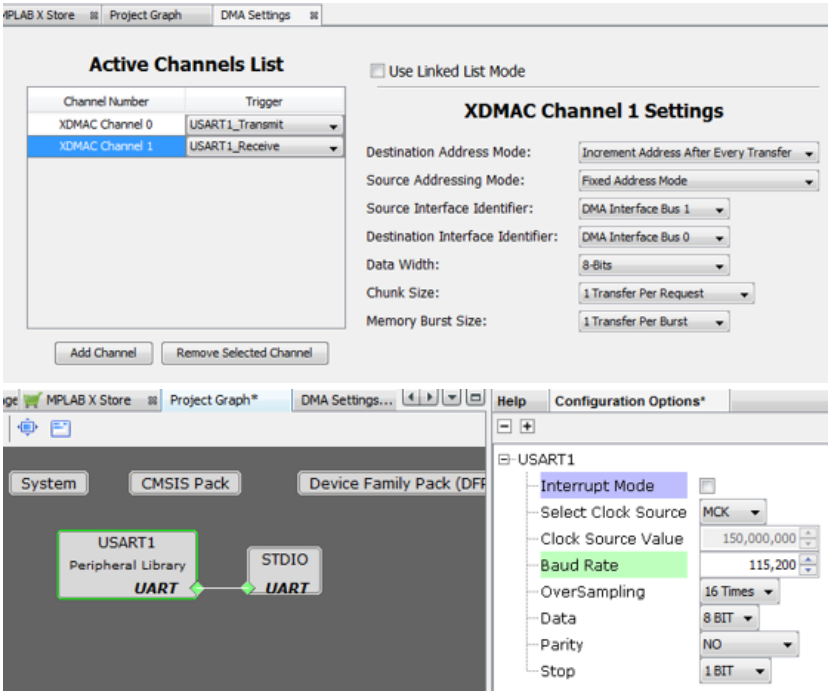
Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra board
sam_v71_xult.X	SAM V71 Xplained Ultra board

MPLAB Harmony Configuration

This topic provides information on MHC configurations

Description

The Project Graph diagram below shows the Harmony components that are included in this application.



Modules Used:

1. XDMAC Peripheral Library
2. STDIO Module
3. USART Peripheral Library

MHC Settings:

1. Configure Pins for USART Peripheral in Pin Settings
2. Configure GPIO pin for LED

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project: sam_e70_xult.X**
   - Hardware Used:
     - SAM E70 Xplained Ultra Evaluation Kit
   - Hardware Setup:
     - No special hardware setup is required

2.

Project: sam_v71_xult.X

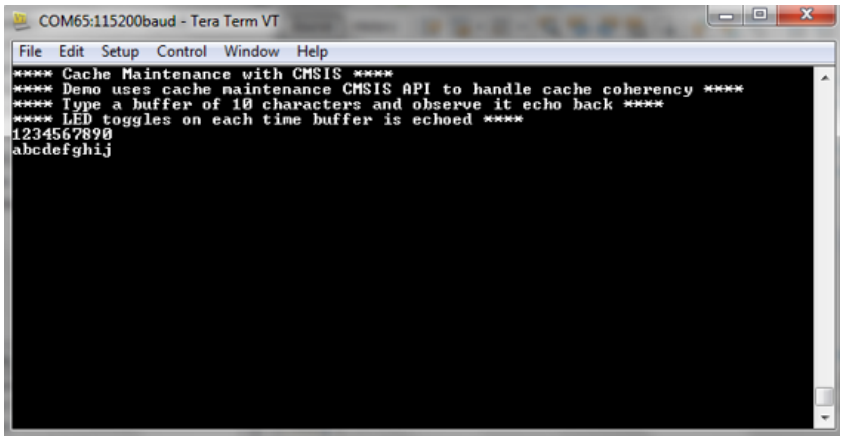
- Hardware Used:
  - SAM V71 Xplained Ultra Evaluation Kit
- Hardware Setup:
  - No special hardware setup is required

Running The Application

This section describes how to run the application.

Description

1. Connect micro USB to DEBUG port.
2. Build the **xdmac_memory_transfer** example code and program using MPLABX
3. Open Terminal application (Ex.:Tera term) on the computer
4. Connect to serial port and configure serial settings as follows
   - Baud : 115200
   - Data : 8 Bits
   - Parity : None
   - Stop : 1 Bit
   - Flow Control : None
5. Console displays the following message, providing a comparison of time taken for the data transfer with burst size of 1 and 16.



Index

A

a) Setup Functions 31
AC Applications 277
ac_sleepwalk_singleshot 277
ac_wav_gen 279
ACC Applications 281
ACC Peripheral Library Help 3
ACC_CALLBACK type 6
ACC_STATUS_SOURCE enumeration 7
acc_wav_gen 281
ACCx_CallbackRegister function 5
ACCx_Initialize function 4
ACCx_StatusGet function 6
ADC Applications 283
adc_dmac_sleepwalking 288
adc_sample 283
adc_window_sleepwalking 285
AFEC Applications 292
AFEC Peripheral Library Help 8
AFEC_CALLBACK type 19
AFEC_CALLBACK_OBJECT structure 19
AFEC_CHANNEL_GAIN enumeration 20
AFEC_CHANNEL_MASK enumeration 20
AFEC_CHANNEL_NUM enumeration 21
afec_dma_striding 297
AFEC_INTERRUPT_MASK enumeration 21
afec_polled_mode 292
afec_user_sequence 294
AFECx_CallbackRegister function 11
AFECx_ChannelGainSet function 12
AFECx_ChannelOffsetSet function 13
AFECx_ChannelResultGet function 13
AFECx_ChannelResultIsReady function 18
AFECx_ChannelsDisable function 15
AFECx_ChannelsEnable function 14
AFECx_ChannelsInterruptDisable function 15
AFECx_ChannelsInterruptEnable function 16
AFECx_ConversionSequenceSet function 17
AFECx_ConversionStart function 17
AFECx_Initialize function 11
Applications Help 276

B

b) Setup Functions 162
b) Transaction Functions 32, 208
Building the Application 282, 301, 308, 310, 322, 330
Building The Application 277, 280, 284, 285, 288, 292, 294, 298, 304, 305, 307, 312, 314, 316, 318, 320, 323, 325, 328, 332, 333, 335, 336, 337, 339, 341, 343, 345, 347, 349, 350, 352, 355, 356, 359, 361, 363, 365, 368, 369, 371, 373, 375, 377, 379, 381, 382, 383, 385, 387, 389, 391, 393, 396, 398, 402, 407, 409, 413, 414, 415, 417, 419, 422

C

c) Status Functions 36
Cache Applications 301
cache_maintenance 301
Capture 189
CLK_Initialize function 24
Clock Applications 304
Clock Peripheral Library Help 23
clock_config 304
Compare 180
Configuring the Library 3, 9, 23, 25, 31, 41, 45, 47, 48, 69, 85, 95, 102, 108, 118, 129, 131, 133, 146, 151, 160, 171, 180, 189, 197, 206, 211, 220, 238, 256, 260

D

DAC Applications 305
dac_wav_gen 305
dac_wav_gen_dma 307
DACC Applications 308
DACC Peripheral Library Help 25
DACC_CHANNEL_NUM enumeration 29
dacc_wav_gen 308
dacc_wav_gen_dma 310
DACCx_DataWrite function 28
DACCx_Initialize function 26
DACCx_IsReady function 27
data_tcm 409
DIVAS Applications 312
divas_accelerate_computations 312
divas_c_operator_override 314
DMAC Applications 316
dmac_memory_transfer 316
dmac_usart_echo 318
DSU Applications 320
dsu_crc32_generate 320

E

EFC Applications 322
EFC Peripheral Library Help 30
EFC_CALLBACK type 37
EFC_ERROR enumeration 38
EFCx_CallbackRegister function 31
EFCx_ErrorGet function 37
EFCx_IsBusy function 36
EFCx_PageWrite function 34
EFCx_QuadWordWrite function 35
EFCx_Read function 35
EFCx_RegionLock function 32
EFCx_RegionUnlock function 33
EFCx_SectorErase function 33
EIC Applications 323
eic_led_on_off 323
EVSYS Applications 324
evsys_trigger 325

F

Files 8, 22, 25, 29, 38, 44, 68, 83, 93, 101, 107, 117, 127, 130, 132, 144, 149, 159, 169, 204, 210, 219, 236, 254, 259, 275
flash_read_write 322
FREQM Applications 328
freqm_measure_clock_freq 328

H

Hardware Setup 279, 280, 283, 285, 287, 291, 293, 296, 299, 302, 304, 306, 308, 309, 311, 313, 315, 317, 319, 321, 323, 324, 327, 329, 331,

332, 334, 335, 336, 338, 340, 342, 344, 346, 348, 350, 351, 354, 355, 357, 360, 362, 364, 367, 368, 370, 373, 374, 376, 378, 379, 381, 383, 384, 386, 388, 390, 392, 395, 397, 401, 405, 408, 410, 413, 415, 416, 418, 421, 423

I

i2c_eeprom 368
I2SC Peripheral Library Help 39
I2SCx_Initialize function 43
I2SCx_LRCLK_Get function 43
Introduction 3, 8, 23, 25, 30, 39, 44, 46, 48, 69, 84, 94, 102, 108, 118, 128, 130, 132, 145, 150, 160, 170, 171, 180, 189, 196, 206, 211, 220, 237, 256, 260

L

led_switcher 349
Library Interface 4, 10, 24, 26, 31, 42, 46, 47, 49, 70, 85, 95, 103, 110, 119, 129, 131, 134, 147, 151, 161, 171, 180, 190, 197, 207, 212, 222, 239, 256, 261

M

MPLAB Harmony Configuration 309, 310, 322, 330, 381, 382, 384, 385, 409, 423
MPLAB Harmony Configurations 277, 280, 282, 284, 286, 288, 292, 295, 298, 301, 304, 306, 307, 312, 315, 317, 318, 320, 324, 325, 328, 332, 333, 335, 336, 338, 340, 342, 343, 345, 347, 349, 351, 353, 355, 356, 359, 361, 363, 365, 368, 370, 371, 374, 375, 377, 379, 387, 389, 392, 394, 396, 398, 402, 407, 412, 414, 416, 418, 420
MPU Applications 329
MPU Peripheral Library Help 44
mpu_coherent_region 329
MPU_Initialize 46

N

NVIC Peripheral Library Help 46
NVIC_Initialize 47
NVMCTRL Applications 331
nvmctrl_flash_read_while_write 333
nvmctrl_flash_read_write 332

P

Peripheral Libraries 3
Peripheral Libraries Help 2
Peripheral Library Overview 2
PIO Applications 334
PIO Peripheral Library Help 48
PIO_Initialize function 49
pio_led_on_off_interrupt 334
pio_led_on_off_polling 336
PIO_PIN enumeration 65
PIO_PIN_CALLBACK type 64
PIO_PinClear function 50
PIO_PinInputEnable function 51
PIO_PinInterruptCallbackRegister function 56
PIO_PinInterruptDisable function 51
PIO_PinInterruptEnable function 52
PIO_PinLatchRead function 52
PIO_PinOutputEnable function 53
PIO_PinRead function 54
PIO_PinSet function 54
PIO_PinToggle function 55
PIO_PinWrite function 55
PIO_PORT enumeration 67
PIO_PortClear function 57
PIO_PortInputEnable function 58
PIO_PortInterruptDisable function 58
PIO_PortInterruptEnable function 59
PIO_PortLatchRead function 60
PIO_PortOutputEnable function 61
PIO_PortRead function 61
PIO_PortSet function 62
PIO_PortToggle function 63
PIO_PortWrite function 63
plib_acc.h 8
plib_afec.h 22
plib_clk.h 25
plib_dacc.h 29
plib_efc.h 38
plib_i2sc.h 44
plib_pio.h 68
plib_pwm.h 83
plib_qspi.h 94
plib_rstc.h 101
plib_rswdt.h 107
plib_rtc.h 117
plib_rtt.h 128
plib_sdramc.h 130
plib_smc.h 132
plib_spi.h 144
plib_ssc.h 150
plib_supc.h 159
plib_systick.h 169
plib_tc.h 204
plib_trng.h 210
plib_twihs.h 219
plib_uart.h 236
plib_usart.h 254
plib_wdt.h 260
plib_xdmac.h 275
PM Applications 337
pm_wakeup_eic 337
pm_wakeup_rtc 339
PORT Applications 341
port_led_on_off_polling 341
PWM Applications 343
PWM Peripheral Library Help 69
PWM_CALLBACK type 80
PWM_CALLBACK_OBJECT structure 81
PWM_CHANNEL_MASK enumeration 81
PWM_CHANNEL_NUM enumeration 82
PWM_COMPARE enumeration 82
PWM_FAULT_ID enumeration 83
pwm_synchronous_channels 343
PWMx_CallbackRegister function 72
PWMx_ChannelCounterEventDisable function 72
PWMx_ChannelCounterEventEnable function 73
PWMx_ChannelCounterEventStatusGet function 80
PWMx_ChannelDeadTimeSet function 74
PWMx_ChannelDutySet function 74
PWMx_ChannelPeriodGet function 75

PWMx_ChannelPeriodSet function 76
PWMx_ChannelsStart function 76
PWMx_ChannelsStop function 77
PWMx_CompareValueSet function 78
PWMx_FaultStatusClear function 79
PWMx_Initialize function 71
PWMx_SyncUpdateEnable function 78

Q

QSPI Applications 345
QSPI Peripheral Library Help 84
QSPI_ADDRESS_LENGTH enumeration 91
qspi_command_xfer_t structure 92
QSPI_LANE_WIDTH enumeration 91
qspi_memory_xfer_t structure 93
QSPI_OPTION_LENGTH enumeration 92
qspi_read_write 345
qspi_register_xfer_t structure 93
qspi_xip 347
QSPIx_CommandWrite function 86
QSPIx_Initialize function 86
QSPIx_MemoryRead function 89
QSPIx_MemoryWrite function 90
QSPIx_RegisterRead function 87
QSPIx_RegisterWrite function 88
Quadrature Decoder 196

R

random_number 411
RSTC Applications 349
RSTC Peripheral Library Help 94
RSTC_CALLBACK type 99
rstc_reset_cause 350
RSTC_RESET_CAUSE enumeration 100
RSTC_RESET_TYPE enumeration 100
RSTCx_CallbackRegister function 96
RSTCx_Initialize function 96
RSTCx_NRSTPinRead function 98
RSTCx_Reset function 97
RSTCx_ResetCauseGet function 98
RSWDT Applications 352
RSWDT Peripheral Library Help 102
RSWDT_CALLBACK type 106
RSWDT_CallbackRegister function 104
RSWDT_Clear function 106
RSWDT_Disable function 105
RSWDT_Enable function 104
RSWDT_Initialize function 103
RSWDT_MODE enumeration 107
rswdt_timeout 352
RTC Applications 354
RTC Peripheral Library Help 108
rtc_alarm 356
RTC_ALARM_MASK enumeration 115
RTC_CALLBACK type 116
RTC_INT_MASK enumeration 116
rtc_periodic_timeout 354
RTCx_AlarmSet function 112
RTCx_CallbackRegister function 111
RTCx_Initialize function 110
RTCx_InterruptDisable function 113
RTCx_InterruptEnable function 114
RTCx_TimeGet function 114
RTCx_TimeSet function 112
RTT Applications 358
RTT Peripheral Library Help 118
RTT_CALLBACK type 126
RTT_INTERRUPT_TYPE enumeration 126
RTT_OBJECT structure 127
rtt_periodic_timeout 359
RTTx_AlarmValueSet function 121
RTTx_CallbackRegister function 120
RTTx_Disable function 122
RTTx_DisableInterrupt function 123
RTTx_Enable function 123
RTTx_EnableInterrupt function 124
RTTx_FrequencyGet function 124
RTTx_Initialize function 120
RTTx_PrescalarUpdate function 125
RTTx_TimerValueGet function 125
Running the Application 283, 303
Running The Application 279, 281, 285, 287, 291, 294, 297, 300, 305, 306, 308, 310, 312, 313, 315, 317, 319, 321, 323, 324, 327, 329, 331, 333, 334, 335, 337, 338, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 360, 362, 364, 367, 369, 371, 373, 374, 376, 378, 380, 382, 383, 384, 386, 388, 391, 393, 395, 397, 401, 406, 408, 410, 412, 415, 417, 419, 421, 424

S

SAM C20/C21 Peripheral Libraries 2
SAM E70/S70/V70/V71 Peripheral Libraries 2
SDADC Applications 360
sdadc_sample 360
SDRAMC Applications 362
SDRAMC Peripheral Library Help 128
SDRAMC_Initialize function 129
sdramc_read_write 362
sdramc_read_write_perf 365
SERCOM Applications 367
SERCOM I2C Applications 368
SERCOM SPI Applications 369
SERCOM USART Applications 377
SMC Peripheral Library Help 130
SMC_Initialize function 131
SPI Applications 380
SPI Peripheral Library Help 132
SPI_CALLBACK type 141
SPI_CLOCK_PHASE enumeration 141
SPI_CLOCK_POLARITY enumeration 142
SPI_DATA_BITS enumeration 142
spi_eeprom_write_read 369, 380
spi_ping_pong_with_dma 371, 385
spi_self_loopback_blocking 373, 383
spi_self_loopback_interrupt 375, 382
SPI_TRANSFER_SETUP structure 143
SPIx_CallbackRegister function 136
SPIx_Initialize function 135

SPIx_IsBusy function 140
SPIx_Read function 137
SPIx_TransferSetup function 135
SPIx_Write function 138
SPIx_WriteRead function 139
SSC Peripheral Library Help 145
SSC_Initialize function 148
SSC_LRCLK_Get function 148
SSCx_BaudSet function 149
SUPC Applications 387
SUPC Peripheral Library Help 150
SUPC_CALLBACK type 157
supc_wakeup_rtc 387
supc_wakeup_rtt 389
supc_wakeup_wkup 391
SUPCx_BackupModeEnter function 153
SUPCx_CallbackRegister function 153
SUPCx_GPBRRead function 154
SUPCx_GPBRWrite function 155
SUPCx_Initialize function 152
SUPCx_SleepModeEnter function 155
SUPCx_WaitModeEnter function 156
SYSTICK Applications 393
SYSTICK Peripheral Library Help 160
SYSTICK_CALLBACK type 168
SYSTICK_DelayMs function 167
SYSTICK_OBJECT structure 169
systick_periodic_timeout 393
SYSTICK_TimerCallbackSet function 162
SYSTICK_TimerCounterGet function 163
SYSTICK_TimerFrequencyGet function 164
SYSTICK_TimerInitialize function 162
SYSTICK_TimerPeriodGet function 164
SYSTICK_TimerPeriodHasExpired function 167
SYSTICK_TimerPeriodSet function 166
SYSTICK_TimerRestart function 166
SYSTICK_TimerStart function 165
SYSTICK_TimerStop function 165

T

TC Applications 395
TC Peripheral Library Help 170
TC_CAPTURE_CALLBACK type 195
TC_CAPTURE_CALLBACK_OBJECT structure 196
tc_capture_mode 402
TC_CAPTURE_STATUS enumeration 195
TC_COMPARE_CALLBACK type 188
TC_COMPARE_CALLBACK_OBJECT structure 188
tc_compare_mode 398
TC_COMPARE_STATUS enumeration 187
TC_QUADRATURE_CALLBACK type 203
TC_QUADRATURE_CALLBACK_OBJECT structure 204
TC_QUADRATURE_STATUS enumeration 203
TC_TIMER_CALLBACK type 179
TC_TIMER_CALLBACK_OBJECT structure 179
tc_timer_mode 395
TC_TIMER_STATUS enumeration 178
TCC Applications 407
tcc_synchronous_pwm_channels 407
TCM Applications 408
TCx_CHy_CaptureAGet function 192
TCx_CHy_CaptureBGet function 192
TCx_CHy_CaptureCallbackRegister function 191
TCx_CHy_CaptureInitialize function 190
TCx_CHy_CaptureStart function 193
TCx_CHy_CaptureStatusGet function 194
TCx_CHy_CaptureStop function 194
TCx_CHy_CompareASet function 182
TCx_CHy_CompareBSet function 183
TCx_CHy_CompareCallbackRegister function 182
TCx_CHy_CompareFrequencyGet function 184
TCx_CHy_CompareInitialize function 181
TCx_CHy_ComparePeriodGet function 184
TCx_CHy_ComparePeriodSet function 185
TCx_CHy_CompareStart function 185
TCx_CHy_CompareStatusGet function 187
TCx_CHy_CompareStop function 186
TCx_CHy_TimerCallbackRegister function 173
TCx_CHy_TimerCompareSet function 177
TCx_CHy_TimerCounterGet function 174
TCx_CHy_TimerFrequencyGet function 174
TCx_CHy_TimerInitialize function 172
TCx_CHy_TimerPeriodGet function 175
TCx_CHy_TimerPeriodHasExpired function 178
TCx_CHy_TimerPeriodSet function 175
TCx_CHy_TimerStart function 176
TCx_CHy_TimerStop function 176
TCx_QuadratureCallbackRegister function 198
TCx_QuadratureInitialize function 198
TCx_QuadraturePositionGet function 200
TCx_QuadratureRevolutionsGet function 199
TCx_QuadratureSpeedGet function 200
TCx_QuadratureStart function 201
TCx_QuadratureStatusGet function 202
TCx_QuadratureStop function 201
Timer 170
TRNG Applications 411
TRNG Peripheral Library Help 206
TRNG_CALLBACK type 210
TRNGx_CallbackRegister function 207
TRNGx_RandomNumberGenerate function 209
TRNGx_ReadData function 208
TWIHS Applications 414
TWIHS Peripheral Library Help 211
TWIHS_CALLBACK type 218
twihs_eeprom 414
TWIHS_ERROR enumeration 219
TWIHSx_CallbackRegister function 214
TWIHSx_ErrorGet function 217
TWIHSx_Initialize function 213
TWIHSx_IsBusy function 217
TWIHSx_Read function 214
TWIHSx_Write function 215
TWIHSx_WriteRead function 216

U

UART Peripheral Library Help 220
UART_CALLBACK type 234
UART_ERROR enumeration 235
UART_PARITY enumeration 235
UART_SERIAL_SETUP structure 236
UARTx_ErrorGet function 229
UARTx_Initialize function 223
UARTx_Read function 226
UARTx_ReadByte function 228
UARTx_ReadCallbackRegister function 225
UARTx_ReadCountGet function 230
UARTx_ReadIsBusy function 231
UARTx_ReceiverIsReady function 233
UARTx_SerialSetup function 224
UARTx_TransmitterIsReady function 233
UARTx_Write function 227
UARTx_WriteByte function 229
UARTx_WriteCallbackRegister function 225
UARTx_WriteCountGet function 231
UARTx_WriteIsBusy function 232
USART Applications 415
USART Peripheral Library Help 237
USART_CALLBACK type 251
USART_DATA_BIT enumeration 252
usart_echo_blocking 377, 415
usart_echo_interrupt 378, 417
USART_ERROR enumeration 253
USART_PARITY enumeration 253
USART_SERIAL_SETUP structure 252
USART_STOP_BIT enumeration 254
USARTx_ErrorGet function 249
USARTx_Initialize function 240
USARTx_Read function 244
USARTx_ReadByte function 245
USARTx_ReadCallbackRegister function 241
USARTx_ReadCountGet function 250
USARTx_ReadIsBusy function 247
USARTx_ReceiverIsReady function 249
USARTx_SerialSetup function 243
USARTx_TransmitterIsReady function 248
USARTx_Write function 244
USARTx_WriteByte function 246
USARTx_WriteCallbackRegister function 242
USARTx_WriteCountGet function 251
USARTx_WriteIsBusy function 247
Using the Library 3, 9, 24, 26, 30, 40, 45, 47, 48, 69, 85, 95, 102, 108, 118, 129, 131, 133, 146, 151, 161, 171, 180, 189, 197, 206, 211, 221, 238, 256, 260

W

WAITMODE_FLASH_STATE enumeration 158
WAITMODE_WKUP_SOURCE enumeration 158
WDT Applications 419
WDT Peripheral Library Help 255
WDT_CALLBACK type 259
WDT_CallbackRegister function 257
WDT_Clear function 258
WDT_Initialize function 257
wdt_timeout 419

X

XDMAC Applications 422
XDMAC Peripheral Library Help 260
XDMAC_CHANNEL enumeration 270
XDMAC_CHANNEL_CALLBACK type 271
XDMAC_CHANNEL_CONFIG type 271
XDMAC_ChannelBlockLengthSet function 265
XDMAC_ChannelCallbackRegister function 263
XDMAC_ChannelDisable function 269
XDMAC_ChannelIsBusy function 269
XDMAC_ChannelLinkedListTransfer function 267
XDMAC_ChannelSettingsGet function 264
XDMAC_ChannelSettingsSet function 265
XDMAC_ChannelTransfer function 266
XDMAC_DESCRIPTOR_CONTROL union 272
XDMAC_DESCRIPTOR_VIEW_0 structure 272
XDMAC_DESCRIPTOR_VIEW_1 structure 273
XDMAC_DESCRIPTOR_VIEW_2 structure 273
XDMAC_DESCRIPTOR_VIEW_3 structure 274
XDMAC_Initialize function 262
xdmac_memory_transfer 422
XDMAC_MICRO_BLOCK_CONTROL structure 274
XDMAC_TRANSFER_EVENT enumeration 275