MPLAB Harmony PIC32CX-BZ System Services

Table of Contents

1. MPLAB® Harmony PIC32CX-BZ System Services

The PIC32CX-BZ System services provides several services to protocol developers and application users which use PIC32CX-BZ products.

It gives common system services like device driver library, persistence data server library, standalone bootloader and Bootloader services. These services are created as components in Harmony 3.

Services Provided

Service	Description
Device Support Library	This service provides help on the Device Support library that can be used as interface with RF System, PMU System, Info Block and Sleep System
Persistent Data Server Library	This service provides help on the PDS library that can be used for storing and restoring of important data in non-volatile memory using wear levelling mechanism
Standalone Bootloader	This service provides help on the Standalone Bootloader component that can be used to upgrade firmware on a target device without the need for an external programmer or debugger
Bootloader Services	This service provides help on the OTA Services that can be used to create signed firmware image for OTA with the provided header and OTA header information

2. PIC32CX-BZ2 Device Support Component Library Help

The PIC32CX-BZ Device Support Component Library provides software API's for various subsystems in the PIC32CX-BZ.

These systems include:

- RF System - Helps in initialization, calibration and other activities for using RF Sub system
- PMU System - Helps in setting various power modes available in the PMU System
- Information Block - Helps in getting the factory default values of information like device/MAC address, ADC value, temperature, etc
- Sleep System - Helps to enter or exit the sleep modes supported by the system

Additionally device component includes the following two services as sources

- OSAL(Operating System Abstraction Layer) Extension for FreeRTOS
- Application Idle task service

2.1 Device Support Library Usage

Configuring the library

There is no configuration of this library.

Device Support Libraries groups classification

- RF System
- PMU System
- Information Base
- Sleep System

Using the library - An Example

When the device is first powered on, the system will be calibrated with its factory settings by calling:

SYS_Load_Cal();

Similarly other API’s can be used . Refer API information for more details on its description and usage

2.1.1 RF System

RF System helps in initialization, calibration and other activities for using RF Sub system

The major routines of RF System.

- Configuring RF system clock that will be fed into the MCU clock
- Loading system calibration values for RF system
- Helps to check whether the RF needs to be calibrated or not
- RF Calibration based on timer event
- Reading current temperature ADC value

2.1.1.1 wssEnable_t Enum

2.1.1.1.1 C

```
typedef enum
{
    WSS_ENABLE_NONE, //None
    WSS_ENABLE_BLE,  // BLE Enable
    WSS_ENABLE_ZB,   // Zigbee Enable
    WSS_ENABLE_BLE_ZB //BLE and Zigbee Enable
}wssEnable_t;
```

2.1.1.1.2 Description

Wireless Subsystem Enable Flag

2.1.1.2 SYS_ClkGen_Config Function

2.1.1.2.1 C

```
void SYS_ClkGen_Config(void);
```

2.1.1.2.2 Description

This routine will configure the RF system clock that feeds the MCU.

2.1.1.2.3 Parameters

None

2.1.1.2.4 Returns

None

2.1.1.3 SYS_Load_Cal Function

2.1.1.3.1 C

```
void SYS_Load_Cal(wssEnable_t wssEnable);
```

2.1.1.3.2 Description

This routine will load the System calibration values for the RF, and PMU subsystems. The main functions are RF initialization, BLE Modem initialization, load Calibration data from IB, VCO coarse tune (ACLB) and Initialize Arbiter

2.1.1.3.3 Parameters

Param	Description
wssEnable_t wssEnable	wireless subsystem enable flag 0: None 1: BLE enable 2: ZB enable 3: BLE enable & ZB enable

2.1.1.3.4 Returns

None

2.1.1.4 RF_NeedCal Function

2.1.1.4.1 C

```
bool RF_NeedCal(void);
```

2.1.1.4.2 Description

This routine will decide if RF need to be calibrated. current method is a default 60 sec timer to trigger calibration

2.1.1.4.3 Parameters

None

2.1.1.4.4 Returns

A boolean value: True means RF need to be calibrated. False means RF doesn't need to be calibrated

2.1.1.5 RF_Timer_Cal Function

2.1.1.5.1 C

```
void RF_Timer_Cal(wssEnable_t wssEnable);
```

2.1.1.5.2 Description

This routine will calibrate the RF from a Timer event condition. The main functions are TX_Power_Compensation and RSSI_ED_Compensation, which are compensation over temperature. Since temperature may change over time, user can choose do RF_Timer_Cal over time.

2.1.1.5.3 Parameters

Param	Description
wssEnable_t wssEnable	wireless subsystem enable flag 0: None 1: BLE enable 2: ZB enable 3: BLE enable & ZB enable

2.1.1.5.4 Returns

None

2.1.1.6 RF_SetIdleMode Function

2.1.1.6.1 C

```
void RF_SetIdleMode(void);
```

2.1.1.6.2 Description

This routine will the RF subsystem as idle mode.

2.1.1.6.3 Parameters

None

2.1.1.6.4 Returns

None

2.1.1.7 Temperature_Reading Function

2.1.1.7.1 C

```
uint16_t Temperature_Reading(void);
```

2.1.1.7.2 Description

This routine will read current temperature ADC value. The functionality is provided by RF IP.

2.1.1.7.3 Parameters

None

2.1.1.7.4 Returns

uint16 The ADC value of current temperature

2.1.2 PMU System

The PMU (Power Management Unit) subsystem helps in setting various power modes available in the system

Power Management Unit Subsystem is a complex power controller that requires specific configuration and handling by the software for correct, safe operation of the SOC. The software routines help in handles the startup and operation of the PMU.

2.1.2.1 PMU_Mode_T Enum

2.1.2.1.1 C

```
typedef enum
{
    // Linear mode - This is the default mode when CPU and peripherals are running.
    PMU_MODE_MLDO = 1,
    // Buck (DC-DC/switching) mode; supports High Power (PWM) - The most efficient mode
    //when the CPU and peripherals are running. In this mode, the SoC is powered by the DC-DC
converter
    PMU_MODE_BUCK_PWM,   // 2
    // Buck (DC-DC/switching) mode; supports Low Power (PSK) mode
    PMU_MODE_BUCK_PSM   // 3
} PMU_Mode_T;
```

2.1.2.1.2 Description

Supported PMU Modes

2.1.2.2 PMU_Get_Mode Function

2.1.2.2.1 C

```
PMU_Mode_T PMU_Get_Mode(void);
```

2.1.2.2.2 Description

This routine will get the power mode of the system.

2.1.2.2.3 Parameters

None

2.1.2.2.4 Returns

PMU_Mode_T - PMU_MODE_MLDO // 1 PMU_MODE_BUCK_PWM // 2 PMU_MODE_BUCK_PSM // 3

2.1.2.3 PMU_Set_Mode Function

2.1.2.3.1 C

```
uint8_t PMU_Set_Mode(PMU_Mode_T mode);
```

2.1.2.3.2 Description

This routine will set the power mode of the system.

2.1.2.3.3 Parameters

Param	Description
PMU_Mode_T	PMU_MODE_MLDO // 1 PMU_MODE_BUCK_PWM // 2 PMU_MODE_BUCK_PSM // 3

2.1.2.3.4 Returns

uint8_t 0 is success 1 is fail which means no BDADDR IB

2.1.2.4 PMU_ConfigCurrentSensor Function

2.1.2.4.1 C

```
bool PMU_ConfigCurrentSensor(bool enable);
```

2.1.2.4.2 Description

This routine will configure the BUCK current sensor. It only can be configured when the power mode is set as PMU_MODE_BUCK_PSM. Disable current sensor can improve the current consumption of sleep mode.

2.1.2.4.3 Parameters

Param	Description
bool enable	Enable/Disable BUCK current sensor false: Disable true : Enable

2.1.2.4.4 Returns

bool - true is success false is fail due to power mode is not in PMU_MODE_BUCK_PSM

2.1.3 Info Block

Information block helps in getting factory set values from the information base

Information block/base helps in.

- loading all settings from the Info Block into the appropriate sub-systems for proper chip operation
- getting the Bluetooth Device Address
- getting the Zigbee MAC Address
- getting factory temperature ADC value
- getting Battery voltage 3.0 V ADC value
- getting Battery voltage 2.2 V ADC value

- getting RSSI compensation offset value
- getting antenna gain value

2.1.3.1 InformationBlockLoad Function

2.1.3.1.1 C

```
uint8_t InformationBlockLoad(uint8_t checkIb, uint8_t * checkIbExist, wssEnable_t wssEnable);
```

2.1.3.1.2 Description

This routine will load all settings from the Info Block into the appropriate sub-systems for proper chip operation.
NOTE: This is done automatically by RF_SYS_Initialize(uint8_t wssEnable)

2.1.3.1.3 Parameters

None

2.1.3.1.4 Returns

uint8_t 0 is success 1 is fail

2.1.3.2 IB_GetBdAddr Function

2.1.3.2.1 C

```
bool IB_GetBdAddr(uint8_t * p_bdAddr);
```

2.1.3.2.2 Description

This routine will get Bluetooth Device Address from IB. If there exists BDADDR IB, then it will return the BDADDR stored in IB. Otherwise, it returns FAILS (1)

2.1.3.2.3 Parameters

Param	Description
uint8_t "p_Addr"	the pointer to the memory stores BDADDR

2.1.3.2.4 Returns

A boolean value, True means valid BDADDR IB

2.1.3.3 IB_GetMACAddr Function

2.1.3.3.1 C

```
bool IB_GetMACAddr(uint8_t * p_addr);
```

2.1.3.3.2 Description

This routine will get ZB MAC Address from IB. If there exists MAC_ADDR IB, then it will return the MAC_ADDR stored in IB. Otherwise, it returns FAILS (1))

2.1.3.3.3 Parameters

Param	Description
uint8_t "p_Addr"	the pointer to the memory stores MAC_ADDR

2.1.3.3.4 Returns

A boolean value, True means valid ZB MAC Address IB

2.1.3.4 IB_GetAntennaGain Function

2.1.3.4.1 C

```
bool IB_GetAntennaGain(int8_t * p_antennaGain);
```

2.1.3.4.2 Description
This routine will get antenna gain value from IB. The value can be used to calculate radiative power. radiative power = conductive power + antenna gain

2.1.3.4.3 Parameters

Param	Description
int8_t p_antennaGain	the memory to store antenna gain value

2.1.3.4.4 Returns
A boolean value, True means vaild antenna gain IB

2.1.3.5 IB_GetBatVoltage2v2Sar Function

2.1.3.5.1 C

```
bool IB_GetBatVoltage2v2Sar(int16_t * p_batVoltageSar);
```

2.1.3.5.2 Description
This routine will get Battery voltage 2.2 V ADC value from IB. The value can be used for Battery voltage slope and detect battery voltage

2.1.3.5.3 Parameters
IB_BatteryCalSar_T p_batVoltageSar - the pointer to the memory stores Battery voltage ADC value

2.1.3.5.4 Returns
A boolean value, True means valid Battery voltage ADC IB

2.1.3.6 IB_GetBatVoltage3v0Sar Function

2.1.3.6.1 C

```
bool IB_GetBatVoltage3v0Sar(int16_t * p_batVoltageSar);
```

2.1.3.6.2 Description
This routine will get Battery voltage 3.0 V ADC value from IB. The value can be used for Battery voltage slope and detect battery voltage

2.1.3.6.3 Parameters
IB_BatteryCalSar_T p_batVoltageSar - the pointer to the memory stores Battery voltage ADC value

2.1.3.6.4 Returns
A boolean value, True means valid Battery voltage ADC IB

2.1.3.7 IB_GetRssiOffset Function

2.1.3.7.1 C

```
bool IB_GetRssiOffset(int8_t * p_rssiOffset);
```

2.1.3.7.2 Description
This routine will get RSSI compensation offset value from IB. The value can be used to compensation RSSI detection

2.1.3.7.3 Parameters

Param	Description
int8_t p_rssiOffset	the memory to store RSSI compensation offset value

2.1.3.7.4 Returns
A boolean value, True means vaild RSSI compensation offset IB

2.1.4 Sleep System

Sleep system helps to enter and exit sleep modes

Two routines are defined

- Device Enter Sleep Mode routine
- Device Exit Sleep Mode routine

2.1.4.1 DEVICE_SLEEP_ConfigAclbClk Function

2.1.4.1.1 C

```
void DEVICE_SLEEP_ConfigAclbClk(bool enable);
```

2.1.4.1.2 Description

The API is used to enable/disable ACLB clock .

2.1.4.1.3 Parameters

bool enable Set as true to enable ACLB; set as false to disable.

2.1.4.1.4 Returns

None

2.1.4.2 DEVICE_SLEEP_ConfigRetRam Function

2.1.4.2.1 C

```
void DEVICE_SLEEP_ConfigRetRam(bool enable);
```

2.1.4.2.2 Description

The API is used to enable/disable retention RAM

2.1.4.2.3 Parameters

bool enable Set as true to enable retention RAM; set as false to disable.

2.1.4.2.4 Returns

None

2.1.4.3 DEVICE_SLEEP_ConfigRfClk Function

2.1.4.3.1 C

```
void DEVICE_SLEEP_ConfigRfClk(bool enable);
```

2.1.4.3.2 Description

The API is used to enable/disable RF clock.

2.1.4.3.3 Parameters

bool enable Set as true to enable RF clock; set as false to disable.

2.1.4.3.4 Returns

None

2.1.4.4 DEVICE_SLEEP_ConfigRfMbs Function

2.1.4.4.1 C

```
void DEVICE_SLEEP_ConfigRfMbs(bool enable);
```

2.1.4.4.2 Description

The API is used to enable/disable RF MBS.

2.1.4.4.3 Parameters

bool enable Set as true to enable RF MBS; set as false to disable.

2.1.4.4.4 Returns

None

2.1.4.5 DEVICE_SLEEP_ConfigRfXtal Function

2.1.4.5.1 C

```
void DEVICE_SLEEP_ConfigRfXtal(bool enable);
```

2.1.4.5.2 Description

The API is used to enable/disable RF crystal .

2.1.4.5.3 Parameters

bool enable Set as true to enable RF crystal; set as false to disable.

2.1.4.5.4 Returns

None

2.1.4.6 DEVICE_SLEEP_ConfigSubSysPllLock Function

2.1.4.6.1 C

```
void DEVICE_SLEEP_ConfigSubSysPllLock(bool enable);
```

2.1.4.6.2 Description

The API is used to enable/disable BT/ZB sub-system bypass PLL lock.

2.1.4.6.3 Parameters

bool enable Set as true to enable BT/ZB sub-system bypass PLL lock; set as false to disable.

2.1.4.6.4 Returns

None

2.1.4.7 DEVICE_SLEEP_ConfigSubSysXtalReady Function

2.1.4.7.1 C

```
void DEVICE_SLEEP_ConfigSubSysXtalReady(bool enable);
```

2.1.4.7.2 Description

The API is used to enable/disable BT/ZB sub-system crystal .

2.1.4.7.3 Parameters

bool enable Set as true to enable BT/ZB sub-system crystal; set as false to disable.

2.1.4.7.4 Returns

None

2.1.4.8 DEVICE_SLEEP_DisableDebugBus Function

2.1.4.8.1 C

```
void DEVICE_SLEEP_DisableDebugBus(void);
```

2.1.4.8.2 Description

The API is used to disable debug bus for power saving purpose.

2.1.4.8.3 Parameters

None

2.1.4.8.4 Returns

None

2.1.4.9 DEVICE_DeepSleepWakeSrc_T Enum

2.1.4.9.1 C

```
typedef enum DEVICE_DeepSleepWakeSrc_T
{
    DEVICE_DEEP_SLEEP_WAKE_NONE,   // The device is not waken from deep sleep.
```

```
    DEVICE_DEEP_SLEEP_WAKE_INT0,   // The device is waken from deep sleep by interrupt 0.
    DEVICE_DEEP_SLEEP_WAKE_RTC,    // The device is waken from deep sleep by RTC.
    DEVICE_DEEP_SLEEP_WAKE_DSWDT,  // The device is waken from deep sleep by Deep Sleep Watch
Dog Timeout.
    DEVICE_DEEP_SLEEP_WAKE_OTHER,  // The device is waken from deep sleep by the other reason.

    DEVICE_DEEP_SLEEP_WAKE_END
} DEVICE_DeepSleepWakeSrc_T;
```

2.1.4.9.2 Description

Various available deep Sleep Wakeup Sources

2.1.4.10 DEVICE_ClearDeepSleepReg Function

2.1.4.10.1 C

```
bool DEVICE_ClearDeepSleepReg(void);
```

2.1.4.10.2 Description

The API is used to clear the deep sleep related register. If the device is waken from deep sleep mode, the related register will be cleared. If it's a normal Power-on Reset. no register will be cleared.

2.1.4.10.3 Parameters

None

2.1.4.10.4 Returns

A boolean value, True means the device is waken from deep sleep mode.

2.1.4.11 DEVICE_deepSleepIntervalCal Function

2.1.4.11.1 C

```
uint32_t DEVICE_deepSleepIntervalCal(uint32_t expectedInt);
```

2.1.4.11.2 Description

The API is used to perform the deep sleep interval calibration to exclude the HW preparation time of advertising.

2.1.4.11.3 Parameters

Param	Description
expectedInt	The expected deep sleep interval.

2.1.4.11.4 Returns

The deep sleep interval after calibration.

2.1.4.12 DEVICE_EnterDeepSleep Function

2.1.4.12.1 C

```
void DEVICE_EnterDeepSleep(bool enableRetRam,  uint32_t interval);
```

2.1.4.12.2 Description

The API is used to enter deep sleep mode.

2.1.4.12.3 Parameters

Param	Description
enableRetRam	Enable/Disable retention ram.

2.1.4.12.4 Parameters

Param	Description
interval	The interval of deep sleep mode (unit: ms). Set as 0 will keep the device in the deep sleep mode until the INT0 (PB4) is triggered.

2.1.4.12.5 Parameters

None

2.1.4.12.6 Returns

None

2.1.4.13 DEVICE_GetDeepSleepWakeUpSrc Function

2.1.4.13.1 C

```
void DEVICE_GetDeepSleepWakeUpSrc(DEVICE_DeepSleepWakeSrc_T *wakeUpSrc);
```

2.1.4.13.2 Description

The API is used to get the wake up source of deep sleep mode.

2.1.4.13.3 Parameters

Param	Description
wakeUpSrc	Pointer to the wake up source. See @ref DEVICE_DeepSleepWakeSrc_T.

2.1.4.13.4 Returns

None.

2.1.4.14 DEVICE_SetDeepSleepWakeUpSrc Function

2.1.4.14.1 C

```
void DEVICE_SetDeepSleepWakeUpSrc(DEVICE_DeepSleepWakeSrc_T wakeUpSrc);
```

2.1.4.14.2 Description

The API is used to set the wake up source of deep sleep mode.

2.1.4.14.3 Parameters

Param	Description
wakeUpSrc	Tthe wake up source. See @ref DEVICE_DeepSleepWakeSrc_T.

2.1.4.14.4 Returns

None.

2.2 OSAL Extension for FreeRTOS

OSAL (Operating System Abstraction Layer) Extension for FreeRTOS provide extension for OSAL mappings for the FreeRTOS Real-time operating system. The following table defines OSAL routines.

Table 2-1.

Fuction Name	Short Description
OSAL_QUEUE_Create	Creates a new queue instance
OSAL_QUEUE_CreateSet	Creates a new queue set instance
OSAL_QUEUE_AddToSet	Adds the queues and semaphores to the set

.....continued	
Fuction Name	**Short Description**
OSAL_QUEUE_SelectFromSet	Block to wait for something to be available from the queues or semaphore that have been added to the set
OSAL_QUEUE_Send	Post an item into an OSAL Queue
OSAL_QUEUE_SendISR	Post an item into an OSAL Queue from ISR
OSAL_QUEUE_Receive	Receive an item from an OSAL Queue
OSAL_QUEUE_IsFullISR	Query if an OSAL Queue is full

2.2.1 OSAL_QUEUE_Create Function

2.2.1.1 C

```
OSAL_RESULT OSAL_QUEUE_Create(OSAL_QUEUE_HANDLE_TYPE *queID, uint32_t queueLength, uint32_t
itemSize);
```

2.2.1.2 Description

Creates a new queue instance, and returns a handle by which the new queue can be referenced

2.2.1.3 Parameters

Param	Description
queID	A pointer to the queue ID
queueLength	The maximum number of items that the queue can contain.
itemSize	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item in the queue must be the same size.

2.2.1.4 Returns

OSAL_RESULT_TRUE - A queue had been created

OSAL_RESULT_FALSE - Queue creation failed

2.2.2 OSAL_QUEUE_CreateSet Function

2.2.2.1 C

```
OSAL_RESULT OSAL_QUEUE_CreateSet(OSAL_QUEUE_SET_HANDLE_TYPE *queID, uint32_t queueLength);
```

2.2.2.2 Description

Creates a new queue set instance, and returns a handle by which the new queue can be referenced

2.2.2.3 Parameters

Param	Description
queID	A pointer to the queue ID
queueLength	The maximum number of items that the queue can contain.

2.2.2.4 Returns

OSAL_RESULT_TRUE - A queue set had been created

OSAL_RESULT_FALSE - Queue creation failed

2.2.3 OSAL_QUEUE_AddToSet Function

2.2.3.1 C

```
OSAL_RESULT OSAL_QUEUE_AddToSet(OSAL_QUEUE_SET_MEMBER_HANDLE_TYPE *queSetMember,
OSAL_QUEUE_SET_HANDLE_TYPE *queSetID);
```

2.2.3.2 Description

Add the queues and semaphores to the set.Reading from these queues and semaphore can only be performed after a call to xQueueSelectFromSet() has returned the queue or semaphore handle from this point on.

2.2.3.3 Parameters

Param	Description
queSetMember	Member queue or semaphore to be added in the set
queSetID	A pointer to the queue ID

2.2.3.4 Returns

OSAL_RESULT_TRUE - A queue had been created

OSAL_RESULT_FALSE - Queue creation failed

2.2.4 OSAL_QUEUE_SelectFromSet Function

2.2.4.1 C

```
OSAL_RESULT OSAL_QUEUE_SelectFromSet(OSAL_QUEUE_SET_MEMBER_HANDLE_TYPE *queSetMember,
OSAL_QUEUE_SET_HANDLE_TYPE *queSetID, uint16_t waitMS);
```

2.2.4.2 Description

Block to wait for something to be available from the queues or semaphore that have been added to the set.

2.2.4.3 Parameters

Param	Description
queSetMember	Member queue or semaphore to be added in the set
queSetID	A pointer to the queue ID
waitMS	wait time in milliseconds. other value OSAL_WAIT_FOREVER

2.2.4.4 Returns

OSAL_RESULT_TRUE - A queue had been created

OSAL_RESULT_FALSE - Queue creation failed

2.2.5 OSAL_QUEUE_Send Function

2.2.5.1 C

```
OSAL_RESULT OSAL_QUEUE_Send(OSAL_QUEUE_HANDLE_TYPE *queID, void *itemToQueue, uint16_t
waitMS);
```

2.2.5.2 Description

Post an item into an OSAL Queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See OSAL_QUEUE_SendISR() for an alternative which may be used in an ISR.

2.2.5.3 Parameters

Param	Description
queID	A pointer to the queue ID
itemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from itemToQueue into the queue storage area.
waitMS	Time limit to wait in milliseconds.
0	do not wait
OSAL_WAIT_FOREVER	return only when semaphore is obtained
Other values	timeout delay

2.2.5.4 Returns

OSAL_RESULT_TRUE - Item copied to the queue

OSAL_RESULT_FALSE - Item not copied to the queue or timeout occurred

2.2.6 OSAL_QUEUE_SendISR Function

2.2.6.1 C

```
OSAL_RESULT OSAL_QUEUE_SendISR(OSAL_QUEUE_HANDLE_TYPE *queID, void  *itemToQueue);
```

2.2.6.2 Description

Post an item into an OSAL Queue. The item is queued by copy, not by reference. The highest priority task currently blocked on the queue will be released and made ready to run. This form of the send function should be used witin an ISR.

2.2.6.3 Parameters

Param	Description
queID	A pointer to the queue ID
itemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from itemToQueue into the queue storage area.
waitMS	Time limit to wait in milliseconds.
0	do not wait
OSAL_WAIT_FOREVER	return only when semaphore is obtained
Other values	timeout delay

2.2.6.4 Returns

OSAL_RESULT_TRUE - Item copied to the queue

OSAL_RESULT_FALSE - Item not copied to the queue or timeout occurred

2.2.7 OSAL_QUEUE_Receive Function

2.2.7.1 C

```
OSAL_RESULT OSAL_QUEUE_Receive(OSAL_QUEUE_HANDLE_TYPE *queID, void  *pBuffer, uint16_t
waitMS);
```

2.2.7.2 Description

Receive an item from an OSAL Queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created. Successfully received items are removed from the queue. This function must not be used in an interrupt service routine.

2.2.7.3 Parameters

Param	Description
queID	A pointer to the queue ID
buffer	A pointer to the buffer into which the received item will be copied. The size of the items the queue hold was defined when the queue was created, so this many bytes will be copied from the queue storage area into the buffer.
waitMS	Time limit to wait in milliseconds.
0	do not wait
OSAL_WAIT_FOREVER	return only when semaphore is obtained
Other values	timeout delay

2.2.7.4 Returns

OSAL_RESULT_TRUE - An item was successfully received from the queue

OSAL_RESULT_FALSE - An item was not successfully received from the queue or timeout occurred

2.2.8 OSAL_QUEUE_IsFullISR Function

2.2.8.1 C

```
OSAL_RESULT OSAL_QUEUE_IsFullISR(OSAL_QUEUE_HANDLE_TYPE *queID);
```

2.2.8.2 Description

Query if an OSAL Queue is full. These function should be used only from witin an ISR, or within a critical section.

2.2.8.3 Parameters

Param	Description
queID	A pointer to the queue ID

2.2.8.4 Returns

OSAL_RESULT_TRUE - The queue is Full

OSAL_RESULT_FALSE - The queue is not Full

2.3 App Idle Task

App Idle task service provides idle task routines which will be called from FreeRTOS Idle hook function. It executes the idle activities. The following activities may be carried out

- RF Calibration if RF needs calibration
- Storing of PDS Items by calling PDS Store Item Handler
- Checking whether the Zigbee stack is ready to sleep
- Requesting BLE to enter sleep mode

2.3.1 app_idle_task Function

2.3.1.1 C

```
void app_idle_task( void );
```

2.3.1.2 Description

This function performs the activities like PDS store, Sleep and other RF system idle activities which can be performed when the complete system is idle.

2.3.1.3 Parameters

None

2.3.1.4 Returns

None

2.3.2 app_idle_updateRtcCnt Function

2.3.2.1 C

```
void app_idle_updateRtcCnt(uint32_t cnt);
```

2.3.2.2 Description

RTC based tickless idle mode Hook function records RTC counter value in each tick interrupt to ensure the real time RTC counter value be recorded during system is active. Then RTC tickless idle mode can use this value to calculate how much time passed during system sleep.

2.3.2.3 Parameters

Param	Description
cnt	RTC counter value

2.3.2.4 Returns

None

3. PIC32CX-BZ2 Persistent Data Server Component Library Help

The PIC32CX-BZ2 Persistent Data Server Component Library provides interface for storing and restoring items into the Non volatile memory using wear leveling mechanism

PERSISTENT DATA SERVER (PDS)

The Persistent Data Server (PDS) component implements interfaces and functionality for storing and restoring data in a non-volatile (NV) memory storage.

In PDS particular pieces of persistent data are called files and groups of parameters are called directories.Following section describes how to define such persistent items and overview of PDS API functions that can be used to store and restore them.

User can specify parameters he(she) wants to backup in a non-volatile memory and restore in case of power failure. This service is provided by Persistent Data Server (PDS) module. The BitCloud (TM) stack also uses the same service for its internal structures.

The main feature behind the wear leveling PDS is the mechanism designed to extend the lifetime of the NV storage as well as to protect data from being lost when a reset occurs during writing to the NV. This mechanism is based on writing data evenly through the dedicated area, so that the storage's lifetime in not limited by the number of reading and writing operations performed with more frequently used parameters. For this purpose, the non-volatile storage organized as a cyclic log with new versions of data being written at the end of the log, not in place where the previous versions of the same data are stored.

3.1 PDS Library Usage

PIC32CX-BZ2 Persistent Data Server(PDS) Library Usage

Configuring the library

There is no configuration of this library.

Using the library

Defining Files and Directories

In PDS particular pieces of persistent data are called files (or items), and groups of files are called directories. Note that directories are just the way to refer to particular files, and a file can belong to several directories at once. Files and directories contain the meta-information about the data that allows its maintenance within the NV – file descriptors and directory descriptors.

The PDS component defines a number of file units for individual stack parameters and directories to group files, for more subtle control. The application may define its own items.

File and File Descriptor

Each item which user wants to backup in a non-volatile memory and restore in case of power failure is treated as a FILE - actual item value with associated service information, FILE DESCRIPTOR. Each file could be accessed by it's ID a unique 16-bit value associated with a file. File descriptor keeps information about item's size and it's displacement in RAM and inside the NV storage.

All file descriptors should be placed in a special segment inside the MCU Flash memory - [PDS_FF]. The PDS_FILE_DESCR() macro is used to initialize descriptor and PDS_DECLARE_FILE() macro is used to place descriptor to required segment.

A file descriptor consists of the following parts:

- memoryId: memory identifier associated with a file
- size: the size of file's data
- ramAddr: the pointer to item's entity in RAM (that is, to a variable holding file's data), if one exists, or NULL otherwise
- fileMarks: file marks, specifying specific characteristics of the file.
  File marks may be set either to following values:

- SIZE_MODIFICATION_ALLOWED: indicates that size of the file can be different in new firmware image after over-the-air upgrade. Usually is set for files storing table data, such as binding table, group table and others.
- ITEM_UNDER_SECURITY_CONTROL: no impact, works same as NO_FILE_MARKS
- NO_FILE_MARKS: no special characteristics for the file

A file descriptor tied to some data in RAM is defined by using the PDS_DECLARE_FILE macro in the code that may be used by both the stack and the application:

```
PDS_DECLARE_FILE(memoryId, size, ramAddr, fileMarks)
```

Directory and Directory Descriptor

PDS is able to operate with separate files or with file lists - DIRECTORIES. Directory nesting is allowed. Each directory should be provided with DIRECTORY DESCRIPTOR which keeps information about associated items. Directories could be accessed by 16-bit ID, different from already associated with files.

All directory descriptors should be placed in a special segment inside the MCU Flash memory - [PDS_FD]. The PDS_DECLARE_DIR() macro is used to place a directory to required segment.

Directory descriptors are special entities describing a group of file. A directory descriptor is defined in the code (the stack's or the application's one) and is placed to the separate flash memory segment.

The directory descriptor consists of the following parts:

- list: pointer to the list of files IDs associated with the directory. This list should be placed in the flash memory (by the use of the PROGMEM_DECLARE macro – see an example below).
- filesCount: the amount of files associated with the directory
- memoryId: memory identifier associated with the directory

A directory is declared via the PDS_DECLARE_DIR macro in the following way:

```
PDS_DECLARE_DIR(const PDS_DirDescr_t csGeneralParamsDirDescr) =
{
.list = CsGeneralMemoryIdsTable,
.filesCount = ARRAY_SIZE(CsGeneralMemoryIdsTable),
.memoryId = BC_GENERAL_PARAMS_MEM_ID
};
//Where CsGeneralMemoryIdsTable is the list of objects defined in the following way:
PROGMEM_DECLARE(const PDS_MemId_t CsGeneralMemoryIdsTable[]) =
{
CS_UID_MEM_ID,
CS_RF_TX_POWER_MEM_ID,
//other parameters in this list
}
```

Table 3-1. Major Functions

Name	Description
PDS_Init	initializes the Persistence Data Server
PDS_InitItems	initializes the Persistence Data Server Items
PDS_Restore	Restores data from non-volatile storage
PDS_Store	Stores data in non-volatile memory in background, not blocking other processes
PDS_DeleteAll	deletes data from non-volatile storage
PDS_AddItemExcpetionFromDeleteAll	extempts the item from the Delete All command
PDS_Delete	removes specified file records from NV Storage

.....continued

Name	Description
PDS_IsAbleToRestore	Checks if the specified PDS file or directory can be restored from non-volatile memory
PDS_RegisterWriteCompleteCallback	registers the callback for the Item Write completion
PDS_RegisterUpdateMemoryCallback	registers the callback for the Item update memory
PDS_StoreItemTaskHandler	task that handles the store items into NV memory
PDS_GetPendingItemsCount	gets the number of items pending in the PDS write queue
PDS_GetItemDescr	gets the item descriptor for the given item ID

3.1.1 ITEM_ID_TO_MEM_MAPPING Macro

3.1.1.1 C

```
/** PDS Item to memory mapping Definition */
#define ITEM_ID_TO_MEM_MAPPING(item, size, pointer, func, flag) \
  {.itemId = item, .itemSize = size, .itemData = pointer, .filler = func, .flags = flag}
```

3.1.1.2 Description

PDS Item to memory mapping Definition

3.1.2 ITEM_UNDER_SECURITY_CONTROL Macro

3.1.2.1 C

```
#define ITEM_UNDER_SECURITY_CONTROL 0x02U
```

3.1.2.2 Description

PDS Item under security control Flag

3.1.3 PDS_DECLARE_ITEM Macro

3.1.3.1 C

```
#define PDS_DECLARE_ITEM(item, size, pointer, func, flag) \
  PDS_FF_OBJECT(ItemIdToMemoryMapping_t pds_ff_##item) = \
    ITEM_ID_TO_MEM_MAPPING(item, size, pointer, func, flag);
```

3.1.3.2 Description

PDS Declare Item Definition

To declare an itemfile in the PDS

item : Item ID (Unique Indentifier number)for that particular ItemFile

size: size of the item (Maximum allowed size for an Item is 2K (2048 Bytes)

pointer: RAM address of the Item

func: filler function, can be set to NULL, (will be called during the store operation). Shall be kept to min size.

flag: NO_ITEM_FLAGS SIZE_MODIFICATION_ALLOWED ITEM_UNDER_SECURITY_CONTROL

3.1.4 NO_ITEM_FLAGS Macro

3.1.4.1 C

```
#define NO_ITEM_FLAGS               0x00U
```

3.1.4.2 Description

PDS Item No Item Flags

3.1.5 NO_FILE_MARKS Macro

3.1.5.1 C

```
#define NO_FILE_MARKS 0U
```

3.1.5.2 Description

No File Marks

3.1.6 PDS_MAX_FILE_SIZE Macro

3.1.6.1 C

```
#define PDS_MAX_FILE_SIZE          (2 * 1024)   //bytes
```

3.1.6.2 Description

Ensure that the max size of the item is less than or equal to PDS_MAX_FILE_SIZE

3.1.7 PDS_DIRECTORY_ID_MASK Macro

3.1.7.1 C

```
#define  PDS_DIRECTORY_ID_MASK      0x0C00U // (Bit 11 and Bit 10)
```

3.1.7.2 Description

Use this Directory mask on declaring directory item id's along with module specific offsets

3.1.8 SIZE_MODIFICATION_ALLOWED Macro

3.1.8.1 C

```
#define SIZE_MODIFICATION_ALLOWED   0x01U
```

3.1.8.2 Description

PDS Item Size Modificaiton Allowed Flag

3.1.9 PDS_MODULE OFFSET Macro

3.1.9.1 C

```
#define PDS_MODULE_APP_OFFSET      (1 << 12)
#define PDS_MODULE_BT_OFFSET       (1 << 13)
#define PDS_MODULE_ZB_OFFSET       (1 << 14)
#define PDS_MODULE_RES_OFFSET      (1 << 15)
```

3.1.9.2 Description

These offsets CAN be used(OR'ed) to define the range and also to differentiate

the module specific IDs, so the same item ID will not be used across the stacksmodules.

The IDs ranges are required to maintain the backward compatibility during an SW upgrade

with newly added item(s) in any module.

Note: These offset were not used anywhere inside PDS implementation(Library).

This is purely to enable the application to use specific IDs across modules.

3.1.10 PDS_DataServerState_t Enum

3.1.10.1 C

```
typedef enum
{
    PDS_SUCCESS,            //!< Command completed successfully
} PDS_DataServerState_t;
```

3.1.10.2 Description

PDS Data Server State Enumeration

3.1.11 ItemIdToMemoryMapping_t Struct

3.1.11.1 C

```
typedef struct
{
    // Item identifier
    PDS_MemId_t       itemId;
    // Size of the item
    uint16_t          itemSize;
    // Pointer to the item data
    void              *itemData;
    // Function which gets called upon operation completion.
    void              (*filler)(void);
    // Flags.
    uint8_t           flags;
} ItemIdToMemoryMapping_t;
```

3.1.11.2 Description

PDS Item to memory mapping structure

3.1.12 PDS_DirDescr_t Struct

3.1.12.1 C

```
typedef struct
{
    // PDS Memory identifier record list
    PDS_MemIdRec_t list;
    // Number of files in the directory
    uint16_t    filesCount;
    // Memory identifier
    PDS_MemId_t memoryId;
} PDS_DirDescr_t;
```

3.1.12.2 Description

PDS Directory Descriptor structure

3.1.13 PDS_Operation_Offset_t Struct

3.1.13.1 C

```
typedef struct
{
    // Item identifier
    PDS_MemId_t id;
    // Item offset
    uint16_t    offset;
    // size of item
    uint16_t    size;
    // corresponding ram address for the item
    uint8_t     *ramAddr;
}PDS_Operation_Offset_t;
```

3.1.13.2 Description

PDS Operation offset structure

3.1.14 PDS_UpdateMemory_t Struct

3.1.14.1 C

```
typedef struct
{
    // Item identifier
    PDS_MemId_t id;
    // Data pointer
    void *data;
    // current item size
    uint16_t size;
    // last size of item
    uint16_t oldSize;

}PDS_UpdateMemory_t;
```

3.1.14.2 Description

PDS Update Memory structure

3.1.15 PDS_AddItemExcpetionFromDeleteAll Function

3.1.15.1 C

```
bool PDS_AddItemExcpetionFromDeleteAll (PDS_MemId_t itemID);
```

3.1.15.2 Description

This routine extempts the item from the Delete All command. Makes the item double persistent

3.1.15.3 Parameters

Param	Description
PDS_MemId_t	Item ID to be exempted

3.1.15.4 Returns

bool - True

3.1.16 PDS_Delete Function

3.1.16.1 C

```
PDS_DataServerState_t PDS_Delete(PDS_MemId_t memoryId);
```

3.1.16.2 Description

This routine removes specified file records from NV Storage

3.1.16.3 Parameters

Param	Description
memoryId	an identifier of PDS file or directory to be removed from NV memory

3.1.16.4 Returns

PDS state as an operation result.

3.1.17 PDS_DeleteAll Function

3.1.17.1 C

```
PDS_DataServerState_t PDS_DeleteAll(bool includingPersistentItems);
```

3.1.17.2 Description

This routine deletes data from non-volatile storage except the Persistent items depending on the parameter passed

3.1.17.3 Parameters

Param	Description
includingPersistentItems	deletes Persistent items if TRUE deletes all other items except Persistent items if FALSE

3.1.17.4 Returns

PDS_DataServerState_t - status of PDS DeleteAll

3.1.18 PDS_GetItemDescr Function

3.1.18.1 C

```
bool PDS_GetItemDescr(PDS_MemId_t memoryId,
```

3.1.18.2 Description

This routine gets the item descriptor for the given item ID

3.1.18.3 Parameters

Param	Description
memoryId	item id
itemDescrToGet	pointer to item descriptor to be loaded

3.1.18.4 Returns

true if descriptor is found out for the given item ID, false - otherwise

3.1.19 PDS_GetPendingItemsCount Function

3.1.19.1 C

```
uint8_t PDS_GetPendingItemsCount (void);
```

3.1.19.2 Description

This routine returns no of items pending in the PDS write queue

3.1.19.3 Parameters

None

3.1.19.4 Returns

uint8_t value, No of items waiting in the queue, zero if no items are pending..

3.1.20 PDS_Init Function

3.1.20.1 C

```
void PDS_Init(PDS_MemId_t maxItems, PDS_MemId_t maxDirectories);
```

3.1.20.2 Description

This routine initializes Persistent Data Server

3.1.20.3 Parameters

Param	Description
maxItems	Total number of individual PDS items used in the entire system

.....continued	
Param	**Description**
maxDirectories	Total number of PDS directories used in the entire system

3.1.20.4 Returns

None

3.1.21 PDS_InitItems Function

3.1.21.1 C

```
void PDS_InitItems(uint16_t memIdStart, uint16_t memIdEnd);
```

3.1.21.2 Description

This routine initializes PDS items, Initializes an item with default data if it doesn't exist yet, or reads it when it does exist.

3.1.21.3 Parameters

Param	Description
memIdStart	The start memory identifier
memIdEnd	The end memory identifier

3.1.21.4 Returns

true, if all expected files have been restored, false - otherwise

3.1.22 PDS_IsAbleToRestore Function

3.1.22.1 C

```
bool PDS_IsAbleToRestore(PDS_MemId_t memoryId);
```

3.1.22.2 Description

This routine checks if the specified PDS file or directory can be restored from non-volatile memory

3.1.22.3 Parameters

Param	Description
memoryId	an identifier of PDS file or directory to be checked

3.1.22.4 Returns

true, if the specified memory can be restored; false - otherwise.

3.1.23 PDS_RegisterUpdateMemoryCallback Function

3.1.23.1 C

```
void PDS_RegisterUpdateMemoryCallback (bool (*callbackFn)(PDS_UpdateMemory_t *));
```

3.1.23.2 Description

This routine register the callback for the Item update memory. Updates BC parameters after restoring taking into account possible size

3.1.23.3 Parameters

Param	Description
PDS_UpdateMemory_t	PDS_UpdateMemory_t
callbackFn	pointer to callback functions

3.1.23.4 Returns

None.

3.1.24 PDS_RegisterWriteCompleteCallback Function

3.1.24.1 C

```
void PDS_RegisterWriteCompleteCallback (void (*callbackFn)(PDS_MemId_t));
```

3.1.24.2 Description

This routine registers the callback for the Item Write completion

3.1.24.3 Parameters

Param	Description
callbackFn	callback which gets called after write complete

3.1.24.4 Returns

None.

3.1.25 PDS_Restore Function

3.1.25.1 C

```
bool PDS_Restore(PDS_MemId_t memoryId);
```

3.1.25.2 Description

This routine restores data from non-volatile storage.PDS files not included in the current build configuration will be ignored. Restoring process will be performed only if all files, expected for actual configuration, are presented in NV storage

3.1.25.3 Parameters

Param	Description
memoryId	an identifier of PDS file or directory to be restored from non-volatile memory

3.1.25.4 Returns

true, if all expected files have been restored, false - otherwise.

3.1.26 PDS_Store Function

3.1.26.1 C

```
bool PDS_Store(PDS_MemId_t memoryId);
```

3.1.26.2 Description

This routine stores data in non-volatile memory in background, not blocking other processes. Make sure the item/file size is less than MAX_FILE_SIZE

3.1.26.3 Parameters

Param	Description
memoryId	an identifier of PDS file or directory to be stored from non-volatile memory

3.1.26.4 Returns

True, if storing process has begun, false - otherwise.

3.1.27 PDS_StoreItemTaskHandler Function

3.1.27.1 C

```
void PDS_StoreItemTaskHandler(void);
```

3.1.27.2 Description

This routine is the PDS store item handler.Actual flash write operation of a particular Item ID Can be called when system is Idle or when necessary

3.1.27.3 Returns

None.

4. PIC32CX-BZ2 Standalone Bootloader Component Help

PIC32CX-BZ2 bootloader is a standalone harmony component, Using it, one can generate bootloader which is a program that is loaded on internal flash memory and gets executed every time the device powers up or resets.

Bootloader can be used to upgrade firmware on a target device without the need for an external programmer or debugger. It does not fully operate on the device, but can perform various functions prior to starting the main application

Functionality of Bootloader

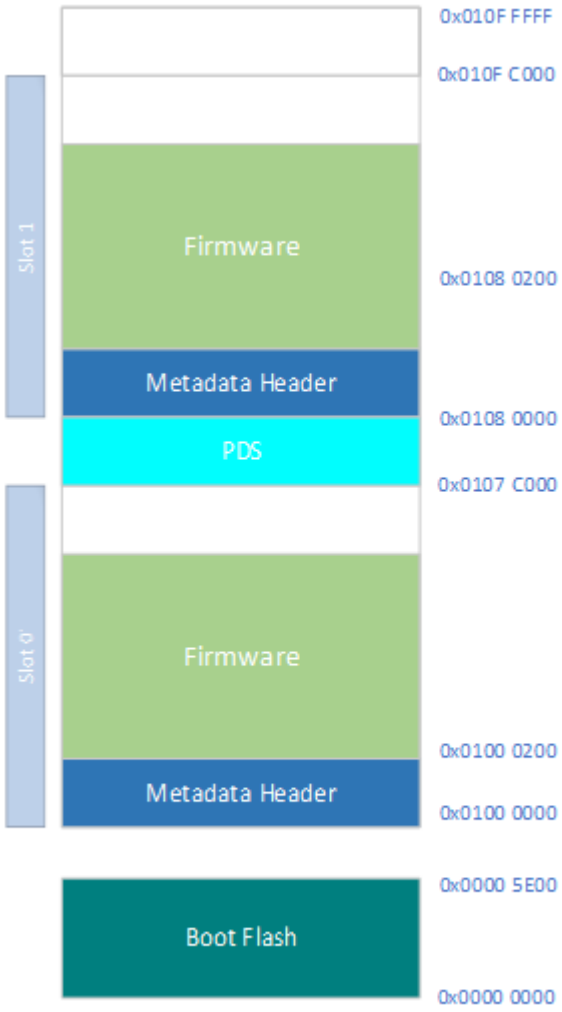
- Loads firmware images to flash over the serial connection using a tool or python script, known as Device Firmware Upgrade (DFU)
- Provides application protection for firmware
- Replaces application firmware
- Starts the application
- If the device wakes from deep sleep, then bootloader directly jumps to the application

Memory Information and Layout of PIC32CXBZ2 device

Table 4-1.

Memory Area	Purpose
Boot Flash Size is 0x0 to 0x5E00 – (~24KB)	The memory where bootloader code runs
Slot 0 base address - 0x01000000	The memory where the application firmware runs and the memory where the bootloader copies the new application firmware
Slot 1 base address - 0x01080000	The memory where the received image(Either through DFU or any other way) gets stored. and the memory where bootloader looks for the new image to copy into the Slot 0

The full memory layout can be found here



Boot Flash

In PIC32CXBZ2, 24KB boot flash memory is separated from the main execution memory. This boot flash is used for bootloader code

Metadata Header

Format of image meta-data header are described below. Notice that some of the elements in the header is reserved for future expansion and they are not used in this version of bootloader implementation

Offset	Name	Description
Metadata Header		
0x00:0x07	Filler	Set to all zero
0x08:0x0D	MANU_IDENTIFIER *	“MCHP” ASCII String Identifier
0x0C:0x0F	Filler	Set to all zero
0x10:0x13	SEQ_NUM *	Metadata Sequence Number of the image (little endian). Monotonically decreasing image index. Values of 0 or 0xFFFFFFFF indicate that the image is invalid.

0x14	MD_REV	Metadata Revision. This field must be set to 0x02 for this version of metadata header
0x15	CONT_IDX	Container Index 1: Plain firmware image 2: Encrypted firmware image
0x16	MD_AUTH_MTHD *	Metadata authentication method. 0: None 1: SHA-256 2: ECDSA p256 + SHA-256
0x17	MD_AUTH_KEY	Key index for authenticating metadata 0: IB Key 1: Key #2 Reserved for future use. Not for MD_REV 0x01
0x18	PL_DEC_MTHD	Payload Decryption method 0: Plain 1: AES
0x19	PL_DEC_KEY	Key index for decrypting payload 0: Key #1 1: Key #2
0x1A:0x1B	PL_LEN	Metadata payload length. The payload length for this version should be 0x55
Metadata Payload = Firmware Image Header		
0x1C:0x1F	FW_IMG_REV *	Firmware Image revision (little endian).
0x20:0x23	FW_IMG_LEN	Firmware Image length
0x24	FW_IMG_AUTH_MTHD *	Firmware Image authentication method. For Chimera the acceptable method are ECDSA p256 + SHA-256 0: None 1: SHA-256 2: ECDSA p256 + SHA-256
0x25	FW_IMG_AUTH_KEY	Firmware Image authentication key index 0: IB key 1: Key #2
0x26	FW_IMG_DEC_MTHD	Firmware Image decryption method. 0: Plain 1: AES Reserved for future use. Not for MD_REV 0x01

0x27	FW_IMG_DEC_KEY	Firmware Image decryption key index 0: Key #1 1: Key #2 Reserved for future use. Not for MD_REV 0x01
0x28	FW_IMG_SIG_SZ	Firmware Image signature size
0x29:0x70	FW_IMG_SIG	Firmware Image Signature. The concatenated R and S term of the ECDSA signature (P-256) of the SHA-256 hash of the firmware image
Metadata Footer		
0x1B7	MD_SIG_SZ	Metadata payload signature size
0x1B8:0x1FF	MD_SIG	Metadata payload signature The concatenated R and S term of the ECDSA signature (P-256r1) of the SHA-256 hash of the metadata payload (firmware image header).

* Configurable by user

Working of Bootloader

when the application receives new image from a server or via a tool, it will/should store in the Slot 1 location with meta data header & firmware and it should trigger software reset so that bootloader code runs

Bootloader checks for valid image in Slot 1 by reading the meta data header and firmware and authenticating the same with the selected authentication method. If valid image is found and successfully authenticated, then

- Erases Slot 0
- Copies the Slot 1 image to Slot 0
- Verifies the Copy

After that it checks for valid image in Slot 0 by validating it, it jumps to the application if valid image is found

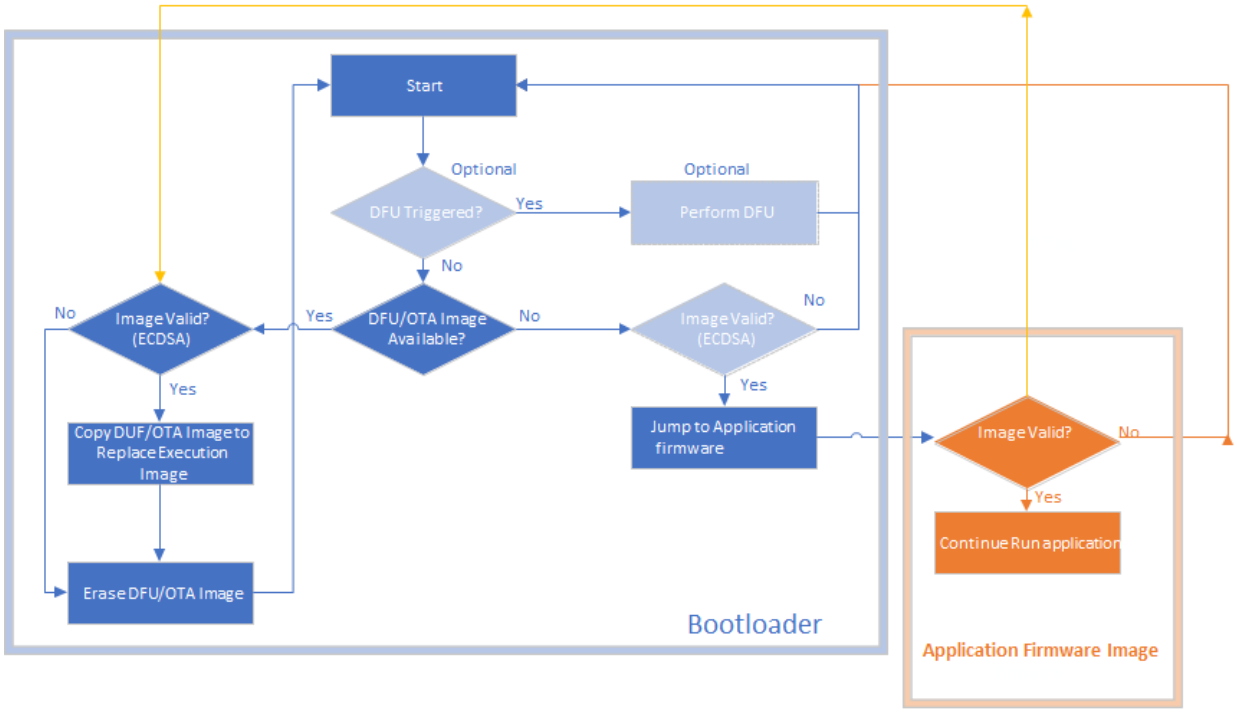
Bootloader usage with DFU

bootloader programmed to the MCU (in DFU mode) receives an application image from the host over serial interface and writes it to the internal flash in Slot 1 . After loading new image on slot 1, if reset is triggered, then bootloader erases the slot 0, copies the image from slot 1 to slot 0, verifies the copy procedure, erases the slot 1. Then bootloader jumps to application which is the new application image

Bootloader stores information about the image in Meta Data Header. This is basically size of 0x200 bytes and gets stored at the start of slot. Bootloader reads this meta data header and does the authentication procedure based on the information in meta data header

Flow Diagram of Bootloader

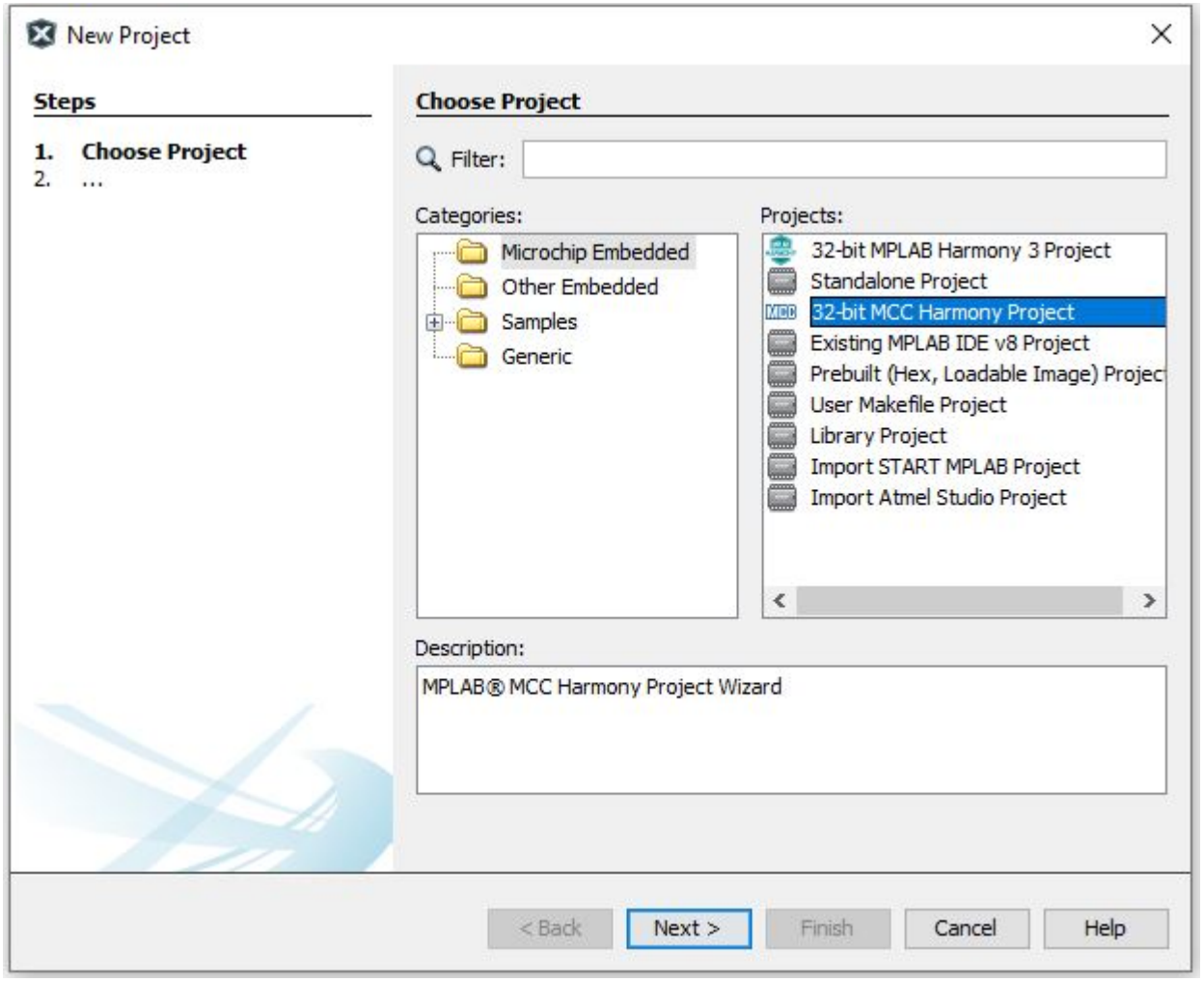
The detailed flow diagram of bootloader with the optional DFU block can be found below



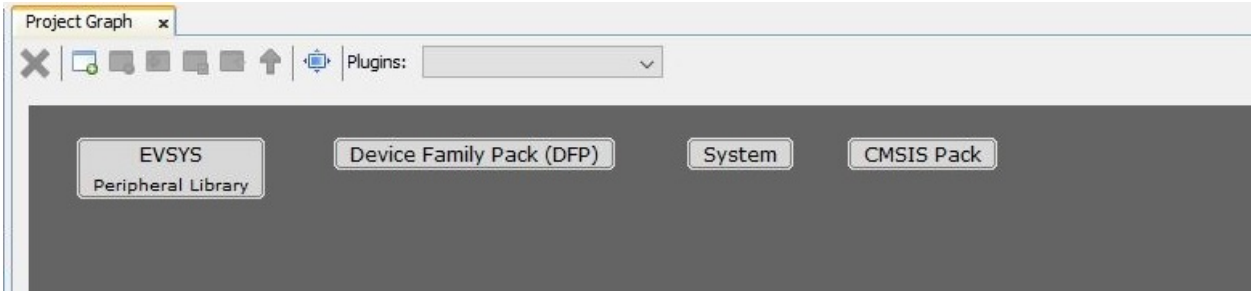
4.1 Create and generate bootloader standalone project for bootloader image

For Bootloader image, a new project has to be created using bootloader component. The following steps describes steps for creating a bootloader standalone MPLAB project

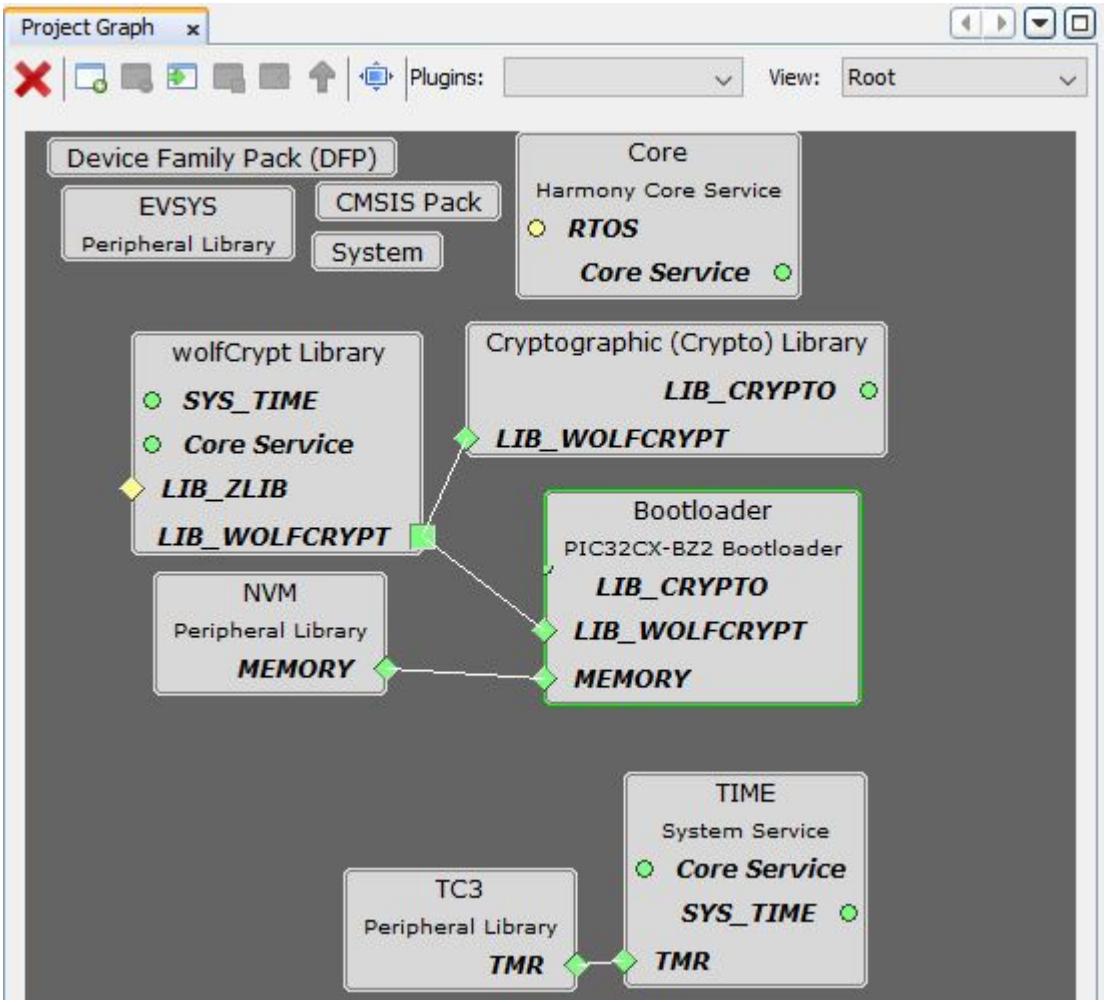
Step 1 : Create a new MCC project for PIC32CX1012BZ25048 Device

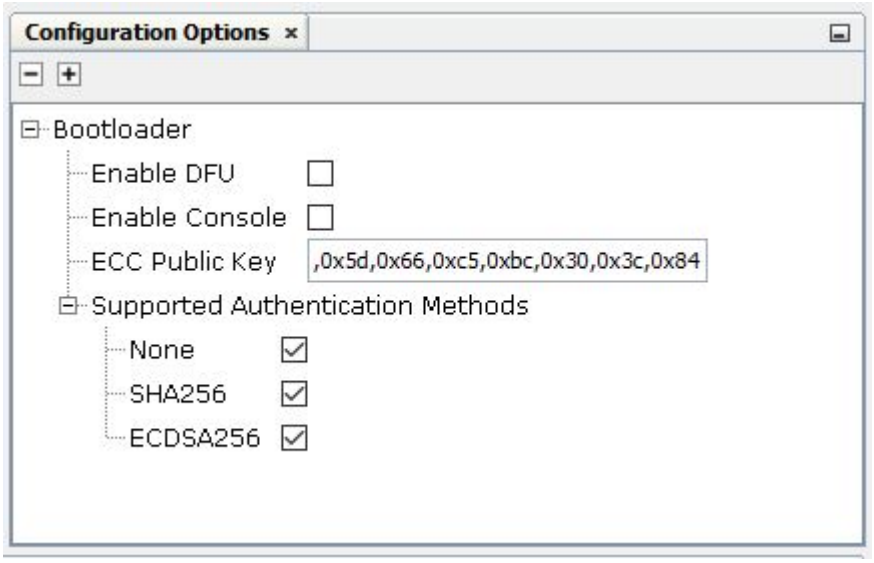


Step 2: Open MPLAB Code Configurator (MCC) . The default project graph will look like the below



Step 3: Add the Boot loader component in MCC Project Graph. Accept auto activation and auto connection of components by giving 'Yes' to popups. The project graph and configuration options will look like this after adding bootloader component.





If User can use the default configurations , then SKIP to Step 7 to generate code with the default configuration options.

Configuration Options in Bootloader

Enable DFU option

By checking/Enabling this option, DFU (Device Firmware Upgrade) functionality will be enabled in the bootloader. DFU can be used to upgrade firmware on a target device through serially (UART). By Default, this option is disabled. If customer wants to use it, they can enable it.

Enable Console option

By checking/Enabling this option, Information on bootloader activities will printed into console using SERCOM UART. By Default, this option is disabled. If customer wants to use it, they can enable it.

ECC Public Key

By default, ECC Public key is
0xc2,0x81,0x8f,0xbb,0x28,0x61,0x47,0x8b,0xa2,0x53,0x37,0x79,0xd4,0x63,0x18,0x7c,0x8b,0x41,0x59,0xa9,0x5f,0x0b,0x6b,0x94,0x4e,0xb9,0x57,0xa1,0x03,0xfe,0x20,0xbf,0x2b,0xb8,0x14,0x2a,0x64,0xb5,0xae,0x4a,0x83,0x80,0xdd,0xe6,0xee,0x29,0x89,0xdd,0xa0,0x9a,0xc7,0xda,0x82,0xeb,0x56,0x62,0x90,0x5d,0x66,0xc5,0xbc,0x30,0x3c,0x84

User should be able to change this key by changing the text box.

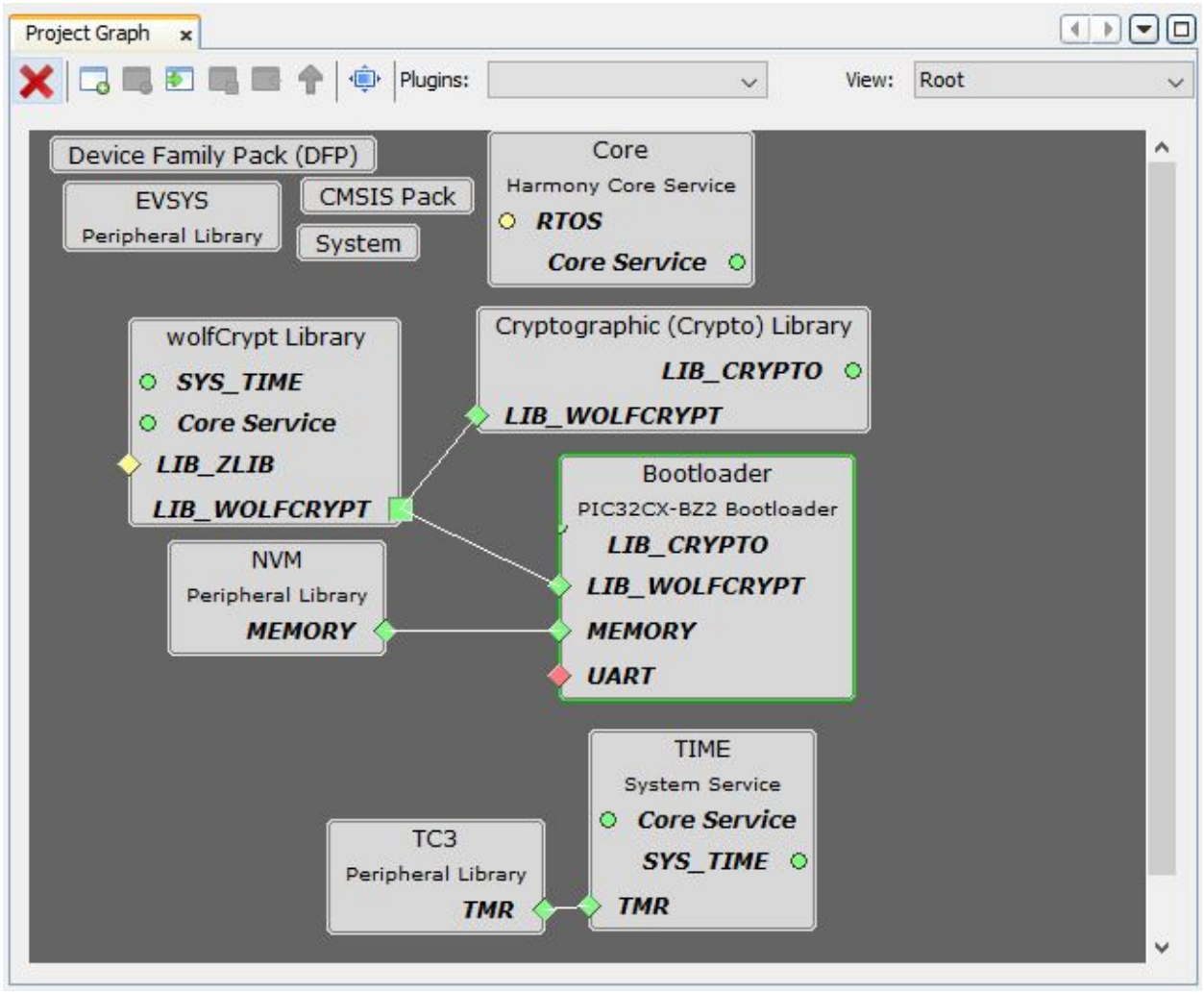
Supported Authentication Methods

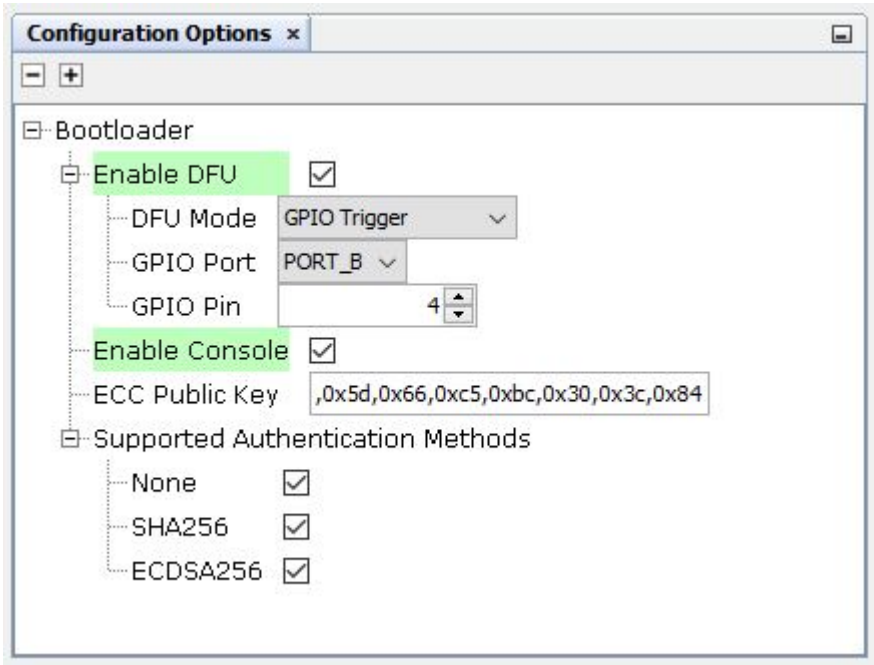
Three methods of authentication is supported in the bootloader

- None
- SHA-256
- ECDSA-256

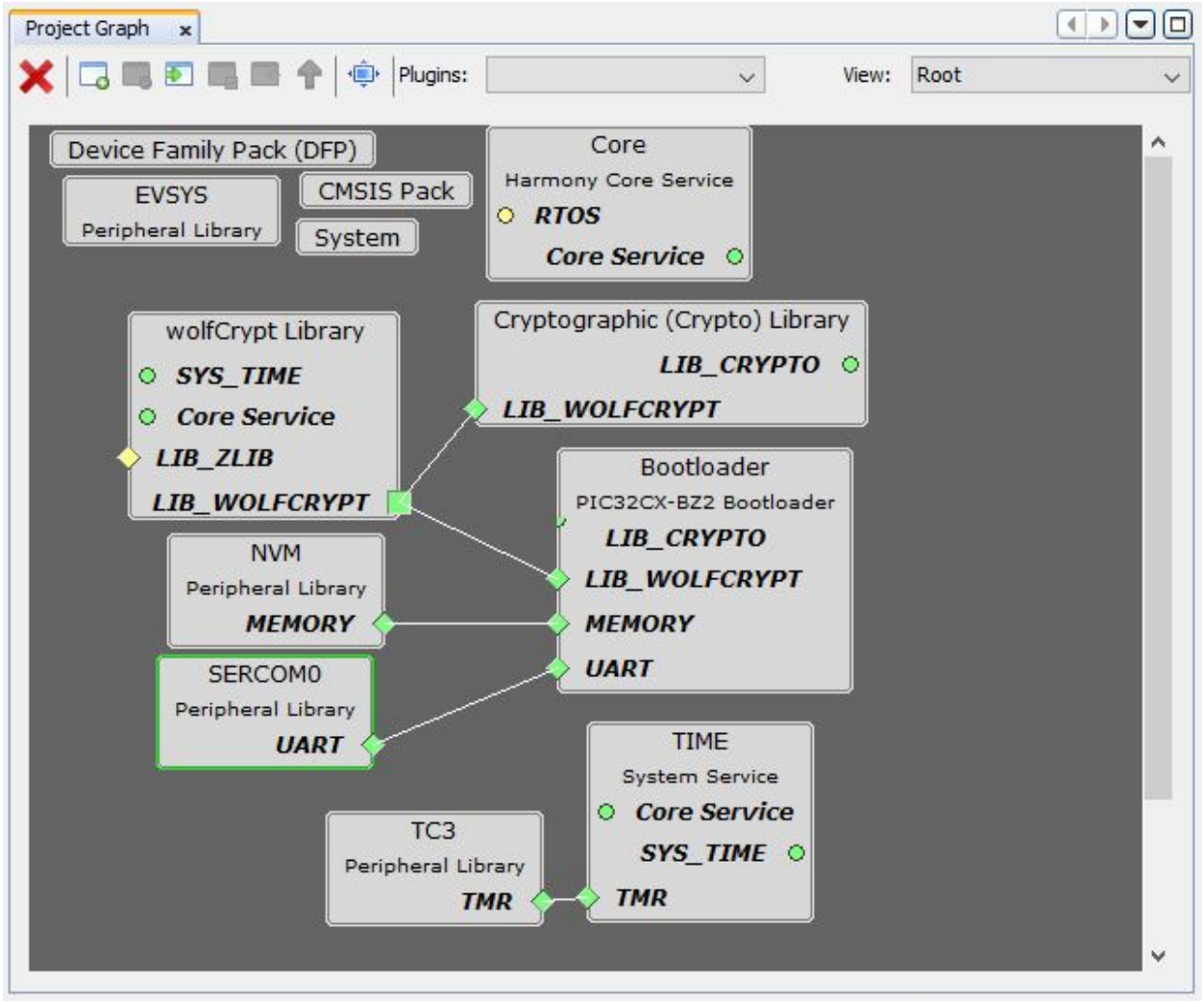
By Default, all these three modes are enabled. If customer want to use a specific authentication alone for security aspect, they can choose that alone and unselect other two's.

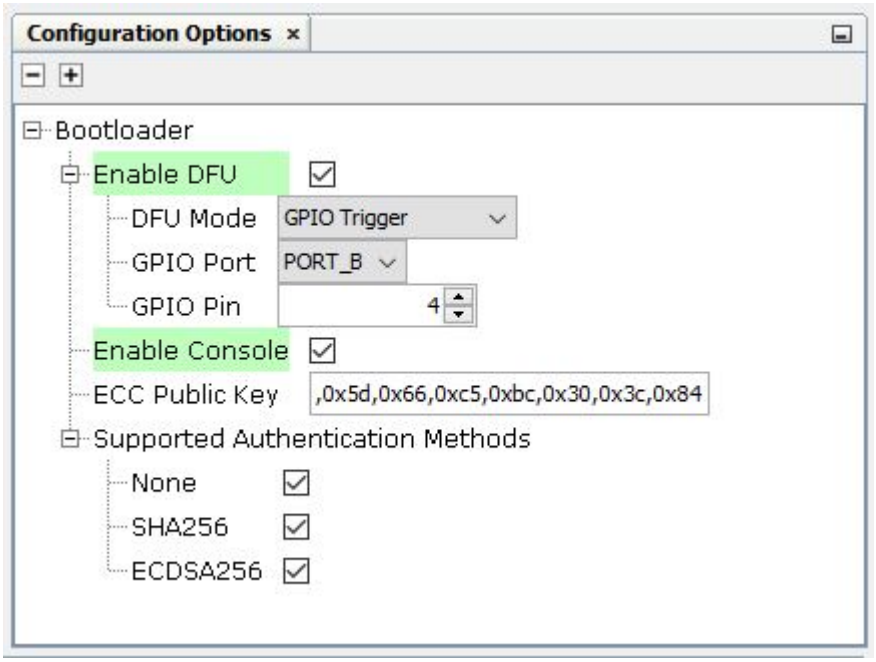
Step 4: By enabling DFU or Console or both, UART dependency will be enabled.


Project Graph
Plugins:
View:
Root
Device Family Pack (DFP)
EVSYS
Peripheral Library
CMSIS Pack
System
Core
Harmony Core Service
RTOS
Core Service
wolfCrypt Library
SYS_TIME
Core Service
LIB_ZLIB
LIB_WOLFCRYPT
Cryptographic (Crypto) Library
LIB_CRYPTO
LIB_WOLFCRYPT
Bootloader
PIC32CX-BZ2 Bootloader
LIB_CRYPTO
LIB_WOLFCRYPT
MEMORY
UART
NVM
Peripheral Library
MEMORY
TIME
System Service
Core Service
SYS_TIME
TC3
Peripheral Library
TMR
TMR


Configuration Options
Bootloader
Enable DFU
DFU Mode
GPIO Trigger
GPIO Port
PORT_B
GPIO Pin
4
Enable Console
ECC Public Key
,0x5d,0x66,0xc5,0xbc,0x30,0x3c,0x84
Supported Authentication Methods
None
SHA256
ECDSA256

Right click on UART dependency(Red Diamond in Bootloader Component) and select SERCOM0 (for Curiosity board) to connect the dependency or select different SERCOM based on the board and connect the dependency



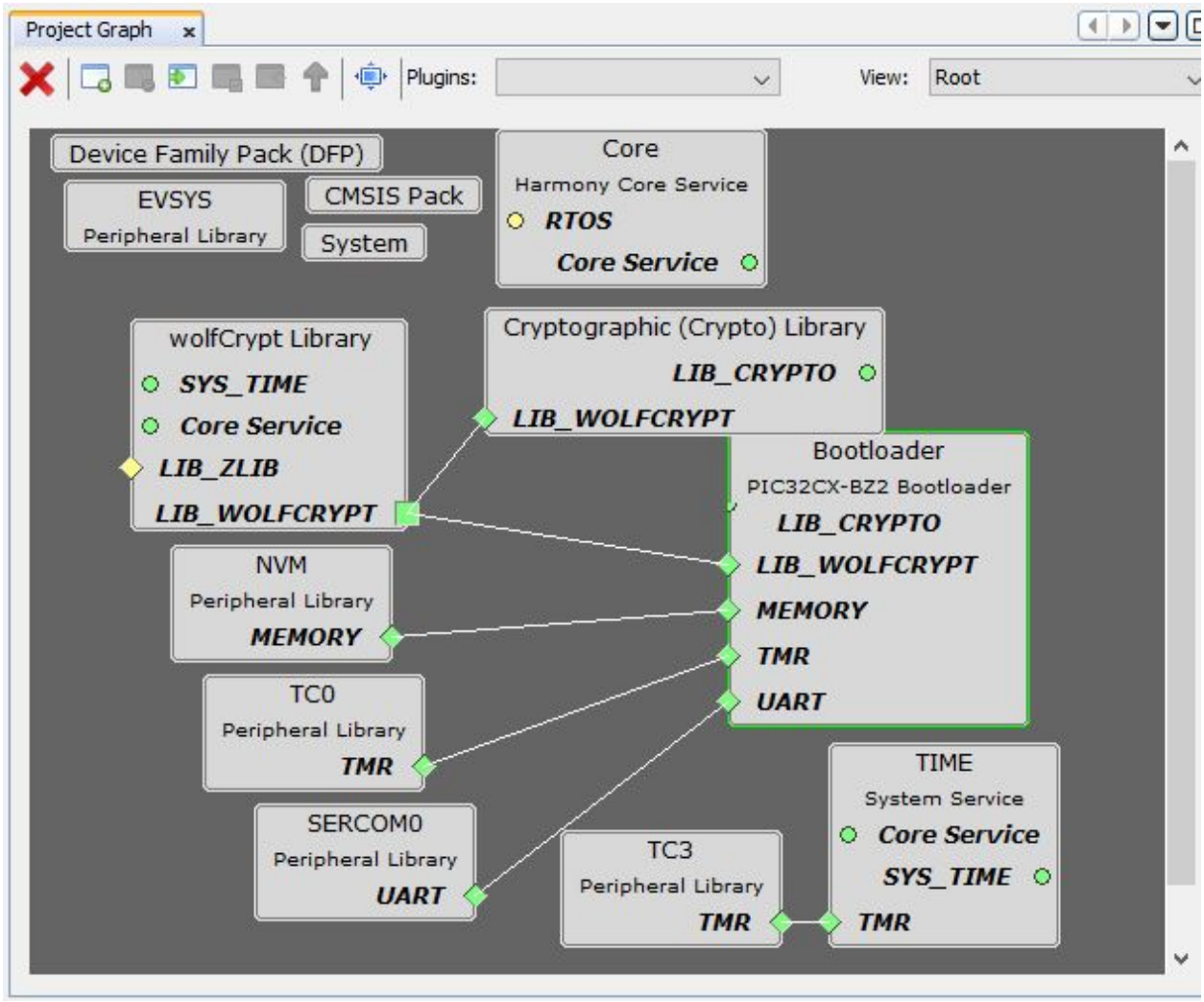


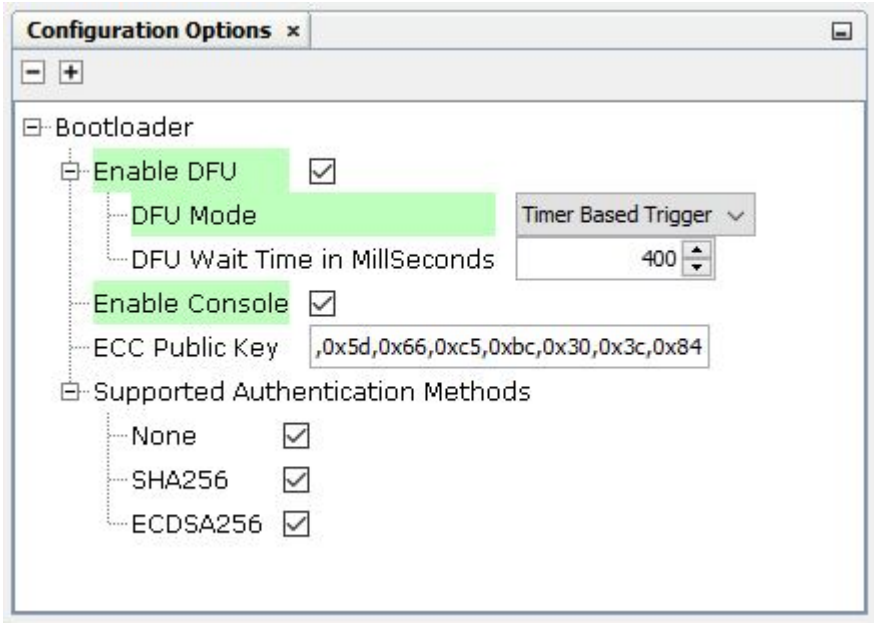
By default, GPIO Trigger DFU Mode is enabled where you need to hold the GPIO button and reset button to put the bootloader into the DFU mode. Use GPIO Port and Pin option to change the port and pin based on the board . Skip to Step 4 with the current configuration. Default Settings are based on the GPIO used in curiosity board

SKIP to Step 7 to generate code with the above configuration changes

Step 5: Configuration for using Timer based Trigger DFU Mode

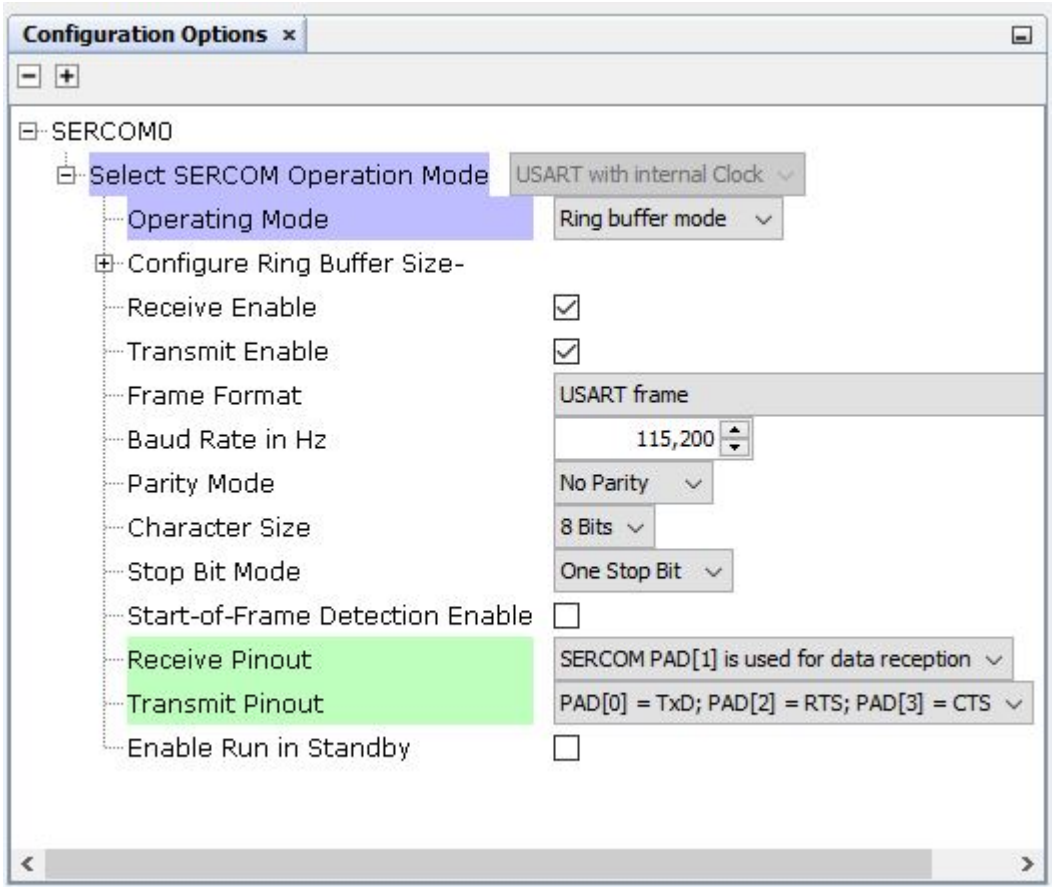
In order to use timer-based trigger DFU Mode where bootloader will be in DFU Mode for the defined amount of time before jumping to the application, select the DFU mode to Timer Based Trigger. Change the DFU wait time if needed. This is the time (in milliseconds) where the bootloader will be in DFU mode before jumping to the application. Accept the auto connection of Timer TC0 which is needed for timer operation. The project graph and configuration will look like this after Timer based Trigger DFU mode is selected


Project Graph
Plugins:
View:
Root
Device Family Pack (DFP)
EVSYS
Peripheral Library
CMSIS Pack
System
Core
Harmony Core Service
RTOS
Core Service
wolfCrypt Library
SYS_TIME
Core Service
LIB_ZLIB
LIB_WOLFCRYPT
Cryptographic (Crypto) Library
LIB_CRYPTO
LIB_WOLFCRYPT
Bootloader
PIC32CX-BZ2 Bootloader
LIB_CRYPTO
LIB_WOLFCRYPT
MEMORY
TMR
UART
NVM
Peripheral Library
MEMORY
TC0
Peripheral Library
TMR
SERCOM0
Peripheral Library
UART
TC3
Peripheral Library
TMR
TIME
System Service
Core Service
SYS_TIME
TMR


Configuration Options
Bootloader
Enable DFU
DFU Mode
Timer Based Trigger
DFU Wait Time in MillSeconds
400
Enable Console
ECC Public Key
,0x5d,0x66,0xc5,0xbc,0x30,0x3c,0x84
Supported Authentication Methods
None
SHA256
ECDSA256

Step 6: Make sure to modify Receive pinout and Transmit pinout in SERCOM Configuration based on the board

The below SERCOM0 configuration is for Curiosity board



Step 7: Generate code once the required configuration done

Go to Resource Management(MCC) Tab , then Press ‘Generate’ for generating the code with the selected configuration

Step 8: Temporary steps to do

The below temporary steps has to be done in order to bringup a proper working bootloader firmware since these are internal changes yet to be incorporated

- In the generated project , Find and replace the **crypt_ecc_pukcl.c and ecc.c** files with the files from wireless_pic32cxbz_wbz\utilities\pic32cx-bz\tempBtl
- In **plib_clk.c** file, comment the PMD Disable configuration

  //CFG_REGS->CFG_PMD2 = 0xf3000000;

Step 9: Build and download the image

Build the image and download to the target

4.2 DFU Functionality - Serial image bootloader

DFU, the device firmware upgrade in bootloader is used to load new image which is received from host over serial interface and writes into the flash(Slot 1). The host may be python script or pc tool.

There are two possible ways the device bootloader can be put into DFU mode

- **GPIO trigger DFU Mode**
  By holding GPIO pin/button and resetting the board helps to enter DFU mode
- **Timer based trigger DFU Mode**
  Upon boot, bootloader enters into DFU mode and will be in DFU mode for the defined time for example: 400ms, then upon timeout, control jumps to application. So the host application/script should be sending messages to load the new image very frequently so that when reset is triggered, it loads the new image since the bootloader will be in DFU mode

Using Python Scripts for serial bootloading(DFU) of image

The following steps provides information on how to load a new image into slot 1 by serial bootloading using python scripts

GPIO trigger DFU Mode

1. After downloading the bootloader(GPIO trigger DFU) image(Refer Step 4 4.1. Create and generate bootloader standalone project for bootloader image for GPIO trigger DFU Configuration), put the bootloader into DFU Mode by holding GPIO button and press reset button on the Curiosity board. Verify the console that 'DFU Now!' is shown for confirmation of DFU mode if Enable Console option is enabled
2. Copy **flash_load_2ndSlot.py** and **progctrl.py** python files (from wireless_pic32cxbz_wbz\utilities\pic32cx-bz\dfuPythonScripts) into a folder where image is available(For example: image.bin).
3. Open command prompt in the same folder and run the below command
   python flash_load_2ndSlot.py -i image.bin
4. The image will get loaded into the slot by showing output like this

```
Loading Image ...(Size: 6912)
Progress: |                                        | 100.0% Complete
```

Timer based trigger DFU Mode

1. After downloading the bootloader(Timer Based Trigger DFU) image(Refer Step 5 4.1. Create and generate bootloader standalone project for bootloader image for Timer Based Trigger DFU Configuration), you will be able to see the bootloader enters DFU Mode, then exits after the defined time (400ms default) and jumps to the application image if valid application image available
2. Copy **flash_load_2ndSlot_timer.py** and **progctrlOptimized.py** python files(from wireless_pic32cxbz_wbz\utilities\pic32cx-bz\dfuPythonScripts) into a folder where image is available(For example: image.bin).
3. Open command prompt in the same folder and run the below command
   python flash_load_2ndSlot_timer.py -i image.bin
4. Now reset the board so that bootloader enters into DFU mode as per Timer Based Trigger DFU mode
5. The image will get loaded into the slot by showing output like this

```
**** Reset the board to boot in DFU Mode at startup****
Error: No response
Error: No response
**** Reset the board to boot in DFU Mode at startup****
Error: No response
Error: No response
**** Reset the board to boot in DFU Mode at startup****
Error: No response
Loading Image ...(Size: 6912)
Progress: |                                        | 100.0% Complete
```

4.3 Generic Source Information

The following macros, typedef, functions defintions help for generic functions of bootloader

4.3.1 FW_IMAGE_BLOCK_SIZE Macro

4.3.1.1 C

```
#define FW_IMAGE_BLOCK_SIZE                 4096
```

4.3.1.2 Description

Firmware Image Block Size

4.3.2 FW_IMAGE_EXPECTED_CONT_IDX Macro

4.3.2.1 C

```
#define FW_IMAGE_EXPECTED_CONT_IDX          1
```

4.3.2.2 Description

Firmware Image Expected Continuous Index

4.3.3 FW_IMAGE_EXPECTED_PL_LEN Macro

4.3.3.1 C

```
#define FW_IMAGE_EXPECTED_PL_LEN            0xE0
```

4.3.3.2 Description

Firmware Image Expected Payload Length

4.3.4 HEADER_SIZE Macro

4.3.4.1 C

```
#define HEADER_SIZE                         512
```

4.3.4.2 Description

The size of the reserved header block

4.3.5 IMG_MEM_TOPOLOGY_COUNT Macro

4.3.5.1 C

```

/** Image Memory Topology Count - Number of Image memory slots */
#define IMG_MEM_TOPOLOGY_COUNT              1 //5
```

4.3.5.2 Description

Image Memory Topology Count - Number of Image memory slots

4.3.6 MAX_AUTH_KEY_LEN Macro

4.3.6.1 C

```
#define MAX_AUTH_KEY_LEN                    48
```

4.3.6.2 Description

Maximum Authentication Key Length

4.3.7 MAX_MEM_TOPOLOGIES Macro

4.3.7.1 C

```
/** Maximum memory topologies on the device (one SRAM and one Flash)) */
#define MAX_MEM_TOPOLOGIES                  2
```

4.3.7.2 Description

Maximum memory topologies on the device (one SRAM and one Flash))

4.3.8 MAX_SLOTS Macro

4.3.8.1 C

```
#define MAX_SLOTS                           3
```

4.3.8.2 Description

The maximum possible number of slots (one SRAM slot and 2 Flash slots))

4.3.9 METADATA_HEADER_SIZE Macro

4.3.9.1 C

```
#define METADATA_HEADER_SIZE                0x200
```

4.3.9.2 Description

Size of the MetaData header

4.3.10 SLOT_PARAMS Struct

4.3.10.1 C

```
typedef struct
{
    // Header Offset
    uint32_t        hdrOffset;
    // Slot Size
    uint32_t        slotSize[2];
    // Executable flag
    bool            executable;
}SLOT_PARAMS;
```

4.3.10.2 Description

Defines the slot information

4.3.11 SLOT0_BASE_ADDR Macro

4.3.11.1 C

```
#define SLOT0_BASE_ADDR                     0x01000000
```

4.3.11.2 Description

Base Address of Slot 0

4.3.12 SLOT0_FIRMWARE Macro

4.3.12.1 C

```
#define SLOT0_FIRMWARE                      (SLOT0_BASE_ADDR + METADATA_HEADER_SIZE)
```

4.3.12.2 Description

Slot 0 Firmware Start Address

4.3.13 SLOT0_HEADER Macro

4.3.13.1 C

```
#define SLOT0_HEADER                            SLOT0_BASE_ADDR
```

4.3.13.2 Description

Slot 0 Header Start Address

4.3.14 SLOT1_BASE_ADDR Macro

4.3.14.1 C

```
#define SLOT1_BASE_ADDR                         0x01080000
```

4.3.14.2 Description

Base Address of Slot 1

4.3.15 SLOT1_FIRMWARE Macro

4.3.15.1 C

```
#define SLOT1_FIRMWARE                          (SLOT1_BASE_ADDR + METADATA_HEADER_SIZE)
```

4.3.15.2 Description

Slot 1 Firmware Start Address

4.3.16 SLOT1_HEADER Macro

4.3.16.1 C

```
#define SLOT1_HEADER                            SLOT1_BASE_ADDR
```

4.3.16.2 Description

Slot 1 Header Start Address

4.3.17 UNAUTH_FW_SIZE Macro

4.3.17.1 C

```
#define UNAUTH_FW_SIZE                          0xFFFFFFFF
```

4.3.17.2 Description

Size of the UnAuthenticated Firmware Size - For Error purpose

4.3.18 AUTH_STATUS Enum

4.3.18.1 C

```
typedef enum
{
    AUTH_STATUS_BUSY = 0x530839fa,
    AUTH_STATUS_FAILED = 0xe97d40ce,
    AUTH_STATUS_SUCCESS = 0x0ac60ce4
} AUTH_STATUS;
```

4.3.18.2 Description

Identifies the possible authentication statuses

4.3.19 DEVICE_CONTEXT Struct

4.3.19.1 C

```
typedef struct
{
    // Valid Toplogies
    const IMG_MEM_TOPOLOGY *     validTops[MAX_MEM_TOPOLOGIES];
    // Valid slots information
    VALID_SLOT                   validSlots[MAX_SLOTS];
    // Number of Topologies
    uint8_t topologyCount;
    // Number of slots
    uint8_t slotCount;
} DEVICE_CONTEXT;
```

4.3.19.2 Description

Defines the device context information ie., valid tops and valid slots

4.3.20 IMG_MEM_INTERFACE Struct

4.3.20.1 C

```
typedef struct
{
    //Memory Initialize
    IMG_MEM_INITIALIZE          fInit;
    // Memory Write
    IMG_MEM_WRITE               fWrite;
    // Memory Read
    IMG_MEM_READ                fRead;
    // Memory Erase
    IMG_MEM_ERASE               fErase;
    // Memory Read JEDEC Identifier
    IMG_MEM_READ_JEDEC_ID       fReadId;
} IMG_MEM_INTERFACE;
```

4.3.20.2 Description

Defines the image memory interface functions for initialization, read, write, erase and read id

4.3.21 IMG_MEM_TOPOLOGY Struct

4.3.21.1 C

```
typedef struct
{
    uint16_t                    u16ErasePageSz;     // Erase page size
    uint16_t                    u16ProgRowSz;       // Programming row size
    uint32_t                    u32UmmAddrStart;    // Unified memory model address start
    uint32_t                    u32TotSize;         // Total flash size
    uint8_t                     u8SlotCount;        // Count of slots in memory
    const SLOT_PARAMS *         pSlots;             // Pointer to array of slot structs
    uint32_t                    u32CalIdx;          // Index of calibration data
    const IMG_MEM_INTERFACE *   ifFlash;            // Flash interface
    uint8_t                     u8DevIdCount;       // Count of valid device IDs
    const uint32_t *            pDevIds;            // Pointer to array of device IDs
    uint32_t                    u32DevIdMask;       // Device ID negative mask
    uint32_t                    u32AddrPosMask;     // UMM address positive mask
    uint32_t                    u32AddrNegMask;     // UMM addr negative mask
    uint32_t                    u32BlankCheck;     // UMM addr negative mask
}IMG_MEM_TOPOLOGY;
```

4.3.21.2 Description

Defines the image memory interface topology defintions

4.3.22 VALID_SLOT Struct

4.3.22.1 C

```
typedef struct
{
    // Firmware Image Header
    FW_IMG_HDR *                pHdr;
    // Slot Information
    const SLOT_PARAMS *         pSlot;
    // Topology Information
    const IMG_MEM_TOPOLOGY *    pTop;
} VALID_SLOT;
```

4.3.22.2 Description

Defines the valid slot information like header,slot number,etc

4.3.23 FW_IMG_HDR Struct

4.3.23.1 C

```
typedef struct
{
    // Meta Data Sequence Number.
    uint32_t        MD_SEQ_NUM;
    // Meta Data Revision.
    uint8_t         MD_REV;
    // Meta Data Container Index.
    uint8_t         MD_CONT_IDX;
    // Meta Data Payload Length.
    uint16_t        MD_PL_LEN;
    // Meta Data Coherence.
    uint32_t        MD_COHERENCE;
    // Meta Data Authentication Method.
    uint8_t         MD_AUTH_METHOD;
    // Meta Data Authentication Key.
    uint8_t         MD_AUTH_KEY;
    // Meta Data Decryption Method.
    uint8_t         MD_DEC_METHOD;
    // Meta Data Decryption Key.
    uint8_t         MD_DEC_KEY;

    // Firmware Image Revision.
    uint32_t        FW_IMG_REV;
    // Firmware Image Source Address.
    uint32_t        FW_IMG_SRC_ADDR;
    // Firmware Image Destionation Address.
    uint32_t        FW_IMG_DST_ADDR;
    // Firmware Image Image Length.
    uint32_t        FW_IMG_LEN;
    // Firmware Image Authentication Method.
    uint8_t         FW_IMG_AUTH_METHOD;
    // Firmware Image Authentication Key.
    uint8_t         FW_IMG_AUTH_KEY;
    // Firmware Image Decryption Method.
    uint8_t         FW_IMG_DEC_METHOD;
    // Firmware Image Decryption Key.
    uint8_t         FW_IMG_DEC_KEY;
    // Firmware Image Signature.
    uint8_t         FW_IMG_SIG[96];

    // PAT Image Source Address.
    uint32_t        PAT_IMG_SRC_ADDR;
    // PAT Image Destination Address.
    uint32_t        PAT_IMG_DST_ADDR;
    // PAT Image Length.
    uint32_t        PAT_IMG_LEN;
    // PAT Image Signature.
    uint8_t         PAT_IMG_SIG[96];

    // MetaData Signature.
    uint8_t         MD_SIG[96];
} FW_IMG_HDR;
```

4.3.23.2 Description

Defines the firmware image header information used in bootloader

4.3.24 IMG_MEM_AuthenticateHeaderStart Function

4.3.24.1 C

```
bool IMG_MEM_AuthenticateHeaderStart(uint8_t * digest, FW_IMG_HDR * hdr, uint8_t * x, uint8_t
* y);
```

4.3.24.2 Description

This function begin a header authentication operation. Note: This function must not be called while the Public Key engine is in use

4.3.24.3 Parameters

Param	Description
digest	48-byte buffer for the hash digest (maintain until authentication done)
ctx	A hash context (this must persist)
hdr	Pointer to a firmware image header
x	The x term of an ECDSA P-384 public key
y	The y term of an ECDSA P-384 public key

4.3.24.4 Returns

status of the authentication start

4.3.25 IMG_MEM_FindValidTopologies Function

4.3.25.1 C

```
void IMG_MEM_FindValidTopologies(DEVICE_CONTEXT * ctx);
```

4.3.25.2 Description

This function populate the DEVICE_CONTEXT structure with the valid topologies for the device and the count of valid topologies

4.3.25.3 Parameters

Param	Description
ctx	Device context

4.3.25.4 Returns

None

4.3.26 IMG_MEM_SlotSort Function

4.3.26.1 C

```
void IMG_MEM_SlotSort(DEVICE_CONTEXT * ctx);
```

4.3.26.2 Description

This function sorts valid slots by priority

4.3.26.3 Parameters

Param	Description
ctx	A device context structure with cached and validated header pointers in the header slots

4.3.26.4 Returns

None

4.3.27 IMG_MEM_ValidateHeader Function

4.3.27.1 C

```
bool IMG_MEM_ValidateHeader(DEVICE_CONTEXT * ctx, FW_IMG_HDR * fwHdr);
```

4.3.27.2 Description

This function validate a header Criteria - Metadata revision, Sequence number (not 0 or 0xFFFFFFFF), rollback counter, image len % 4096 != 0, firmware and header auth method is ECDSA on P-384, encryption methods are None, Firmware image header is 0xE0 bytes, metadata container index is 1

4.3.27.3 Parameters

Param	Description
ctx	A device context structure
fwHdr	Firmware image header pointer
top	Pointer to an image memory topology

4.3.27.4 Returns

FW_IMG_HDR Firmware image header pointer

4.3.27.5 C

```
IMG_MEM_TOPOLOGY* IMG_MEM_GetTopologyByAddress (DEVICE_CONTEXT * ctx,
```

4.3.27.6 Description

This function returns a topology based on a specific address Prerequisite: ctx initialized with FindValidTopologies

4.3.27.7 Parameters

Param	Description
ctx	A device context structure
address	address to get topology
top	Pointer to an image memory topology

4.3.27.8 Returns

IMG_MEM_TOPOLOGY topology pointer or NULL

4.3.28 IMG_MEM_AuthenticateHeaderStatusGet Function

4.3.28.1 C

```
AUTH_STATUS IMG_MEM_AuthenticateHeaderStatusGet(void);
```

4.3.28.2 Description

This function check the status of a header authentication operation Prerequisite: Header authentication started with IMG_MEM_AuthenticateHeaderStart

4.3.28.3 Parameters

None

4.3.28.4 Returns

AUTH_STATUS Success, failure, or busy (no result available)

4.3.29 IMG_MEM_CacheAndValidateHeaders Function

4.3.29.1 C

```
void IMG_MEM_CacheAndValidateHeaders(DEVICE_CONTEXT * ctx, uint8_t * buffer);
```

4.3.29.2 Description

This function cache, validate, and sort headers from all firmware image slots for valid topologies Prerequisite: ctx initialized with FindValidTopologies

4.3.29.3 Parameters

Param	Description
ctx	A device context structure
buffer	A buffer for cached headers (HEADER_SIZE MAX_SLOTS bytes)

4.3.29.4 Returns

None

4.3.30 IMG_MEM_CacheHeader Function

4.3.30.1 C

```
FW_IMG_HDR* IMG_MEM_CacheHeader(DEVICE_CONTEXT * ctx,
```

4.3.30.2 Description

This function cache a header from a firmware image slot

4.3.30.3 Parameters

Param	Description
ctx	A context structure
top	Pointer to an image memory topology
pSlot	The image header slot to cache the header from
buffer	Pointer to a buffer in which to cache the header
bufSlot	Offset in the buffer in HEADER_SIZE units. If a header is cached the contents of this pointer will be incremented by 1.

4.3.30.4 Returns

FW_IMG_HDR Firmware image header pointer

4.4 DFU Source Information

The following macros, typedef, functions defintions help to understand DFU functionality of bootloader

4.4.1 dfu Function

4.4.1.1 C

```
void dfu(const IMG_MEM_TOPOLOGY ** tops, uint8_t count);
```

4.4.1.2 Description

This routine will initiate the device firmware upgrade where it waits for commands and response for commands received from serial.

4.4.1.3 Parameters

Param	Description
IMG_MEM_TOPOLOGY tops	valid toplogies list
uint8_t count	number of valid topologies

4.4.1.4 Returns

None

4.4.2 program_exec_main Function

4.4.2.1 C

```
int32_t program_exec_main(const IMG_MEM_TOPOLOGY ** tops, uint8_t count);
```

4.4.2.2 Description

This routine is the entry point for the programming executive. It receives commands from the host, dispatches the command, and then sends response back to the host.

4.4.2.3 Parameters

Param	Description
IMG_MEM_TOPOLOGY tops	valid toplogies list
uint8_t count	number of valid topologies

4.4.2.4 Returns

None

4.4.3 Crc32Init Function

4.4.3.1 C

```
void Crc32Init(uint16_t seed);
```

4.4.3.2 Description

This routine initializes the CRC32 operation

4.4.3.3 Parameters

Param	Description
uint16_t seed	seed for CRC calculation

4.4.3.4 Returns

None

4.4.4 Crc32Add Function

4.4.4.1 C

```
void Crc32Add(uint8_t* pBuff, uint32_t bSize);
```

4.4.4.2 Description

This routine adds the given data into CRC32 caculation and calculates CRC

4.4.4.3 Parameters

Param	Description
uint8_t pBuff	pointer to the data to add
uint32_t bSize	size of the data to add

4.4.4.4 Returns

None

4.4.5 Crc32Result Function

4.4.5.1 C

```
uint32_t Crc32Result(void);
```

4.4.5.2 Description

This routine gets/returns the calculated CRC

4.4.5.3 Parameters

None

4.4.5.4 Returns

uint32_t - Calculated CRC

4.4.6 UART_ERROR Enum

4.4.6.1 C

```
typedef enum
{
    // UART - No Error
    UART_ERROR_NONE = 0,
    // UART - Overrun error
    UART_ERROR_OVERRUN = 0x02,
    // UART - Framing error
    UART_ERROR_FRAMING = 0x04,
    // UART - Parity error
    UART_ERROR_PARITY = 0x08

} UART_ERROR;
```

4.4.7 UART_Init Function

4.4.7.1 C

```
void UART_Init(void);
```

4.4.7.2 Description

This function initializes UART for read notifcation and thershold setting.

4.4.7.3 Parameters

None

4.4.7.4 Returns

None

4.4.8 UART_Read Function

4.4.8.1 C

```
uint32_t UART_Read(uint8_t *rb, const uint32_t len, const int32_t wait);
```

4.4.8.2 Description

This function reads the SERCOM UART for the given length.

4.4.8.3 Parameters

Param	Description
rb	read buffer
len	length to read
wait	wait timeout

4.4.8.4 Returns

None

4.4.9 UART_Write Function

4.4.9.1 C

```
void UART_Write(int8_t *wb, uint32_t len);
```

4.4.9.2 Description

This function writes the given buffer in SERCOM UART for the given length.

4.4.9.3 Parameters

Param	Description
wb	write buffer
len	length to write

4.4.9.4 Returns

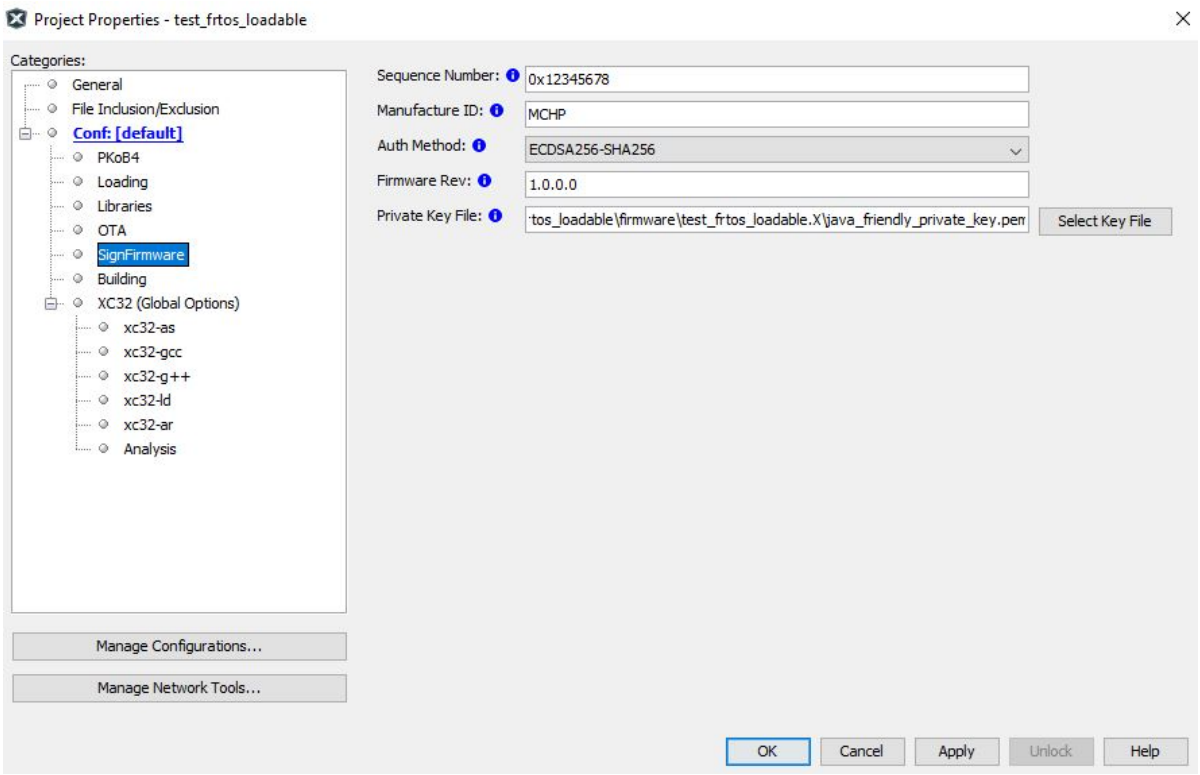
None

5. PIC32CX-BZ2 Bootloader Services Component Help

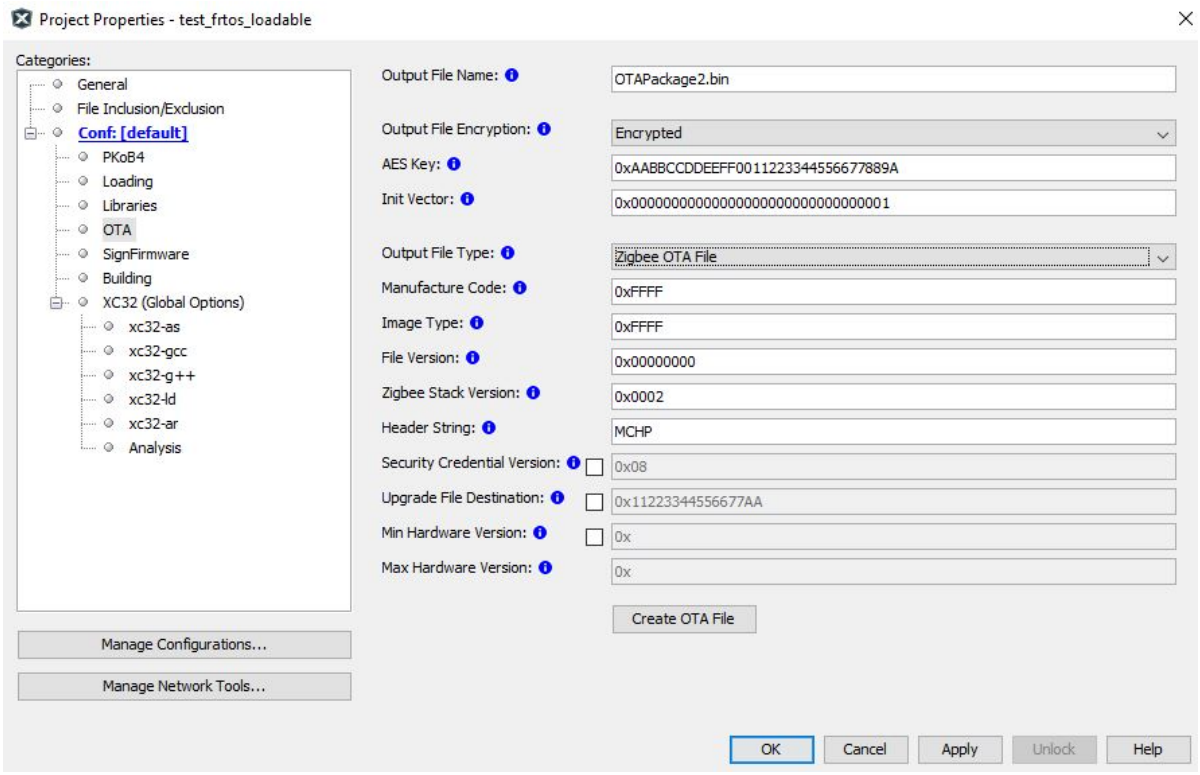
The PIC32CX-BZ2 Bootloader Services is a utility which helps in creating signed firmware image for OTA with the provided header and OTA header information. Please follow MCC Project Graph and how to add Bootloader services component which is available in Device Resources - Libraries → Harmony → Wireless → Bootloader Services

Bootloader services utility functional activities with Harmony 3 code generation

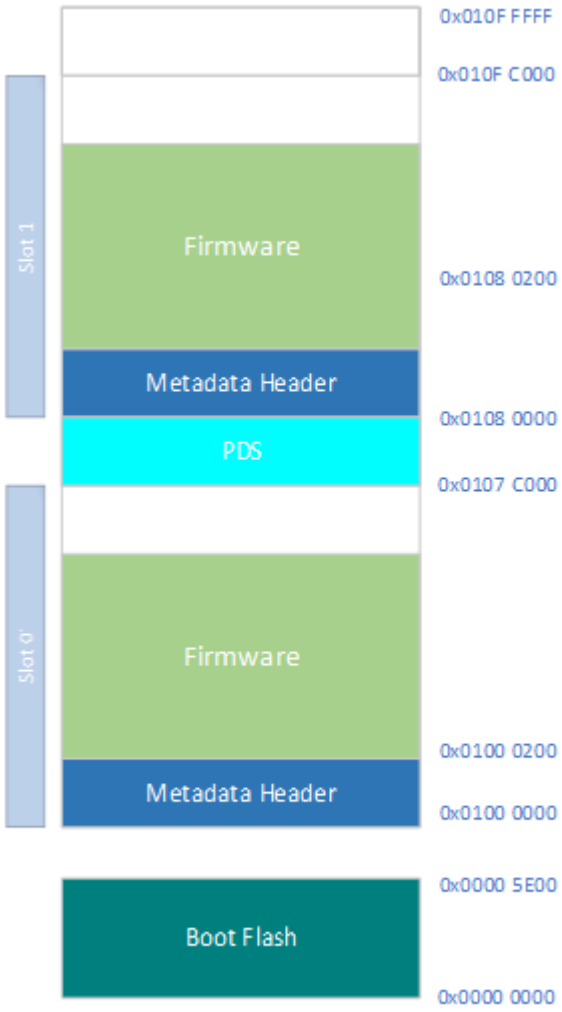
- Adds the autoload.py script which gets loaded in the project (See Screenshot) for getting the required information from user for creating signed Firmware



- Embedding OTA header information as part of the signed firmware binary image.

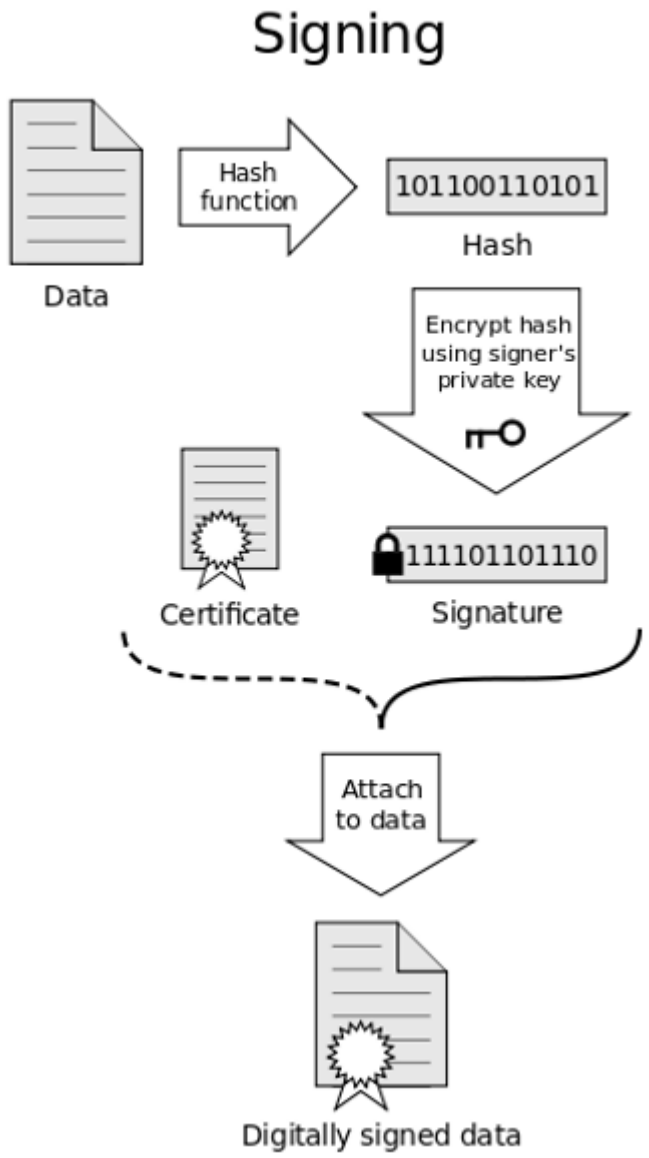


- Generates linker script with the below memory layout



Signing

The complete image including firmware and meta header is signed by the below process


Signing
Hash function
101100110101
Hash
Data
Encrypt hash using signer's private key
111101101110
Signature
Certificate
Attach to data
Digitally signed data