Software Attack Protection on PIC32CM LS00 MCUs

AN5880

Product Page Links

Introduction

The modern embedded systems are increasingly susceptible to software attacks, which are malicious activities aimed at exploiting software vulnerabilities to gain unauthorized access, steal data, disrupt services, or inflict other forms of damage. Concurrently, protecting intellectual property remains critically important.

This document provides guidelines on safeguarding the PIC32CM LS00 MCU against software attacks using the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit. By leveraging the Boot ROM's Secure Hash Algorithm 2 (SHA-256) Authentication, the PIC32CM LS00 can identify unauthorized code fragments in the non-secure memory and replace them with an authentic copy of the same from the secure memory region.

Table of Contents

1. Hardware and Software Requirements

1.1. PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit

The PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit is ideal for evaluating and prototyping with the secure and ultra-low power PIC32CM LS00 Arm® Cortex®-M23 microcontrollers. The MCU integrates Arm TrustZone® technology and enhanced Peripheral Touch Controller (PTC) and smart analog, such as Op Amps, ADC, DAC, and Analog Comparators.

The kit includes an on-board Nano Embedded Debugger (nEDBG), eliminating the need for external tools to program or debug. The following are key features of the PIC32CM LS00 MCU:

- 48 MHz Arm Cortex-M23 Core
- 512 KB Flash and 64 KB SRAM
- Immutable Secure boot, Crypto accelerator, Anti-tamper detection

The PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit is available at Microchip Direct.

1.2. MPLAB® X Integrated Development Environment (IDE) and MPLAB XC Compilers

The MPLAB X IDE is an expandable, highly configurable software program that incorporates powerful tools to discover, configure, develop, debug, and qualify embedded designs for most of the Microchip's microcontrollers.

- The MPLAB X IDE is available on the Microchip Website. This document describes the MPLAB X IDE version 6.20.
- The MPLAB XC Compilers are available on the Microchip Website. This document describes MPLAB XC32 version 4.45.

1.3. MPLAB Harmony v3

MPLAB Harmony v3 is a fully integrated software development framework that provides flexible and interoperable software modules that enable dedicating resources to create applications for 32-bit PIC® and SAM devices, rather than dealing with device details, complex protocols, and library integration challenges.

It includes the MPLAB Code Configurator (MCC), an easy-to-use development tool with a graphical user interface (GUI) that simplifies device setup, library selection, configuration, and application development. The MCC is available as a plug-in that integrates with the MPLAB X IDE and has a separate Java executable for stand-alone use with other development environments.

The application discussed in this document uses the following MPLAB Harmony v3 repositories. These repositories can be downloaded from GitHub:

- csp v3.20.0 (MPLAB Harmony v3 Chip Support Package)
- bootloader v3.7.0
  or
- Use the MCC Content Manager to download the above-mentioned repository

2. Software Attack Protection Using PIC32CM LS00 MCUs

2.1. Boot ROM Features

The PIC32CM LS00/LS60 series incorporate a hardware or software cryptographic accelerator (CRYA) that facilitates Advanced Encryption Standard (AES) encryption and decryption, Secure Hash Algorithm 2 (SHA-256) authentication, and Galois Counter Mode (GCM) encryption and authentication through a suite of APIs.

The CRYA cryptographic accelerator is configured as a client on the IOBUS port and is controlled by the CPU through assembly code stored in the Boot ROM.

Advanced Encryption Standard (AES) adheres to the American Federal Information Processing Standard (FIPS) Publication 197 specification. AES processes data in 128-bit blocks. The key size for an AES cipher determines the number of transformation rounds required to convert the input plaintext into the final output, known as ciphertext. AES utilizes a symmetric-key algorithm, meaning the same key is employed for both encryption and decryption.

SHA-256 is a cryptographic hash function that generates a 256-bit hash from a data block, which is processed in 512-bit chunks.

Galois/Counter Mode (GCM) is an operational mode for AES that integrates the Counter (CTR) mode with an authentication hash function.

2.2. Secure Hash Algorithm 2 (SHA-256) Authentication

The main purpose of a hash function is to create a distinct digital identifier for a specific set of data, similar to a fingerprint. Unlike error detection codes, each data set must be linked to a unique identifier.

In practical terms, a hash function takes input of varying lengths and produces an output of a fixed size known as a message digest. It has several important attributes, including excellent diffusion, which guarantees a significantly different output with even a small change in input.

Although the fixed output size theoretically limits the ability to generate a unique digest for every possible piece of data, hash functions are designed to make it extremely difficult to find two messages that produce the same digest, effectively creating the appearance of uniqueness for practical purposes.

2.3. Usage of SHA-256 APIs from Boot ROM

The cryptographic accelerator (CRYA) APIs are located in a dedicated Boot ROM area. This area is execute-only, meaning the CPU cannot do any loads but can call the APIs. The Boot ROM memory space is a secure area, only the secure application can directly call these APIs.

Table 2-1. CRYA APIs Addresses

CRYA API	Address
AES Encryption	0x02006804
AES Decryption	0x02006808
SHA256 Init	0x02006810
SHA256 Update	0x02006814
SHA256 Final	0x02006818
SHA256 Process (legacy API)	0x02006800
GCM Process	0x0200680C

The API is composed of the following functions which must be called in a specific order:

1. SHA-256 Init to initiate a SHA256_CTX structure.

2. SHA-256 Update to add a message to be computed in the digest.
3. SHA-256 Final to compute the digest.

Note: SHA-256 Update can be called several times in the case several messages are to be included in the digest computation.

The SHA-256 structure to define is called SHA56_CTX:

```
typedef struct
{
    /* Digest result of SHA256 */
    uint32_t digest[8];
    /* Length of the message */
    uint64_t length;
    /* Holds the size of the remaining part of data */
    uint32_t remain_size;
    /* Buffer of remaining part of data (512 bits data block) */
    uint8_t remain_ram[64];
    /* RAM buffer of 256 bytes used by crya_sha_process */
    uint32_t process_buf[64];

} SHA256_CTX;
```

The SHA-256 Init function entry point is located at the Boot ROM address 0x02006810:

```
typedef void (*crya_sha256_init_t) (SHA256_CTX *context);

#define crya_sha256_init ((crya_sha256_init_t) (0x02006810 | 0x1))
```

The SHA-256 Update function entry point is located at the Boot ROM address 0x02006814:

```
typedef void (*crya_sha256_update_t) (SHA256_CTX *context, const unsigned char *data, size_t
length);

#define crya_sha256_update ((crya_sha256_update_t) (0x02006814 | 0x1))
```

The SHA-256 Final function entry point is located at the Boot ROM address 0x02006818:

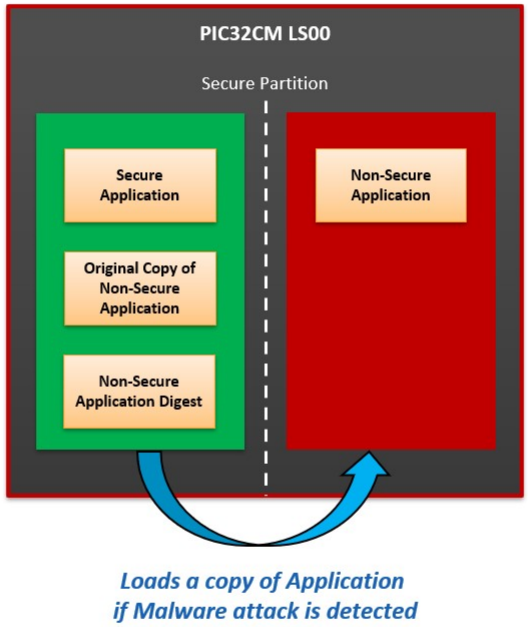
```
typedef void (*crya_sha256_final_t) (SHA256_CTX *context, unsigned char output[32]);

#define crya_sha256_final ((crya_sha256_final_t) (0x02006818 | 0x1))
```

2.4. Prevention of Non-Secure Region against Software Attacks

During device startup, the secure application calculates a unique identifier (digest) for the Non-Secure firmware and stores it in secure memory (Secure Data Flash). A periodic verification is necessary to ensure the integrity of the Non-Secure firmware. A timer will check the firmware authenticity at specific time intervals.

If malware or unauthorized code is injected into the Non-Secure application, the calculated digest of the updated firmware will not match the expected digest of the genuine Non-Secure application. As a result, the secure application will restore the original copy from the secured memory to the Non-Secure Flash region, preventing system downtime.

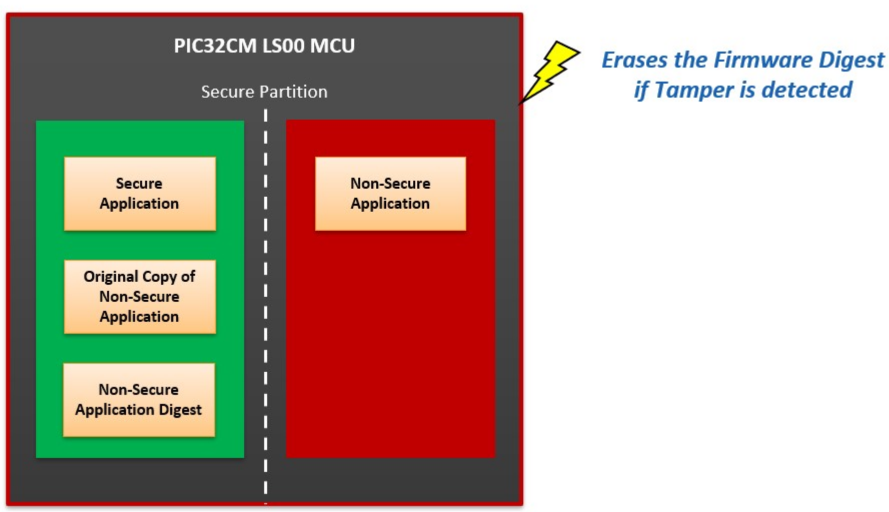
Figure 2-1. Software Attack Protection of Non-Secure Firmware



2.4.1. Simulation of Software Attack

The PIC32CM LS00/LS60 family of devices features tamper detection with a tamper erase security function within the Real Time Clock (RTC) peripheral. To simulate a software attack, the tamper erase option is employed to erase data stored in the secure memory region. Upon detecting any tampering, the RTC peripheral within the secure application triggers a tamper-erase operation to delete the contents (firmware digest) in the Secure Data Flash region.

Figure 2-2. Software Attack Simulation using Tamper Detection

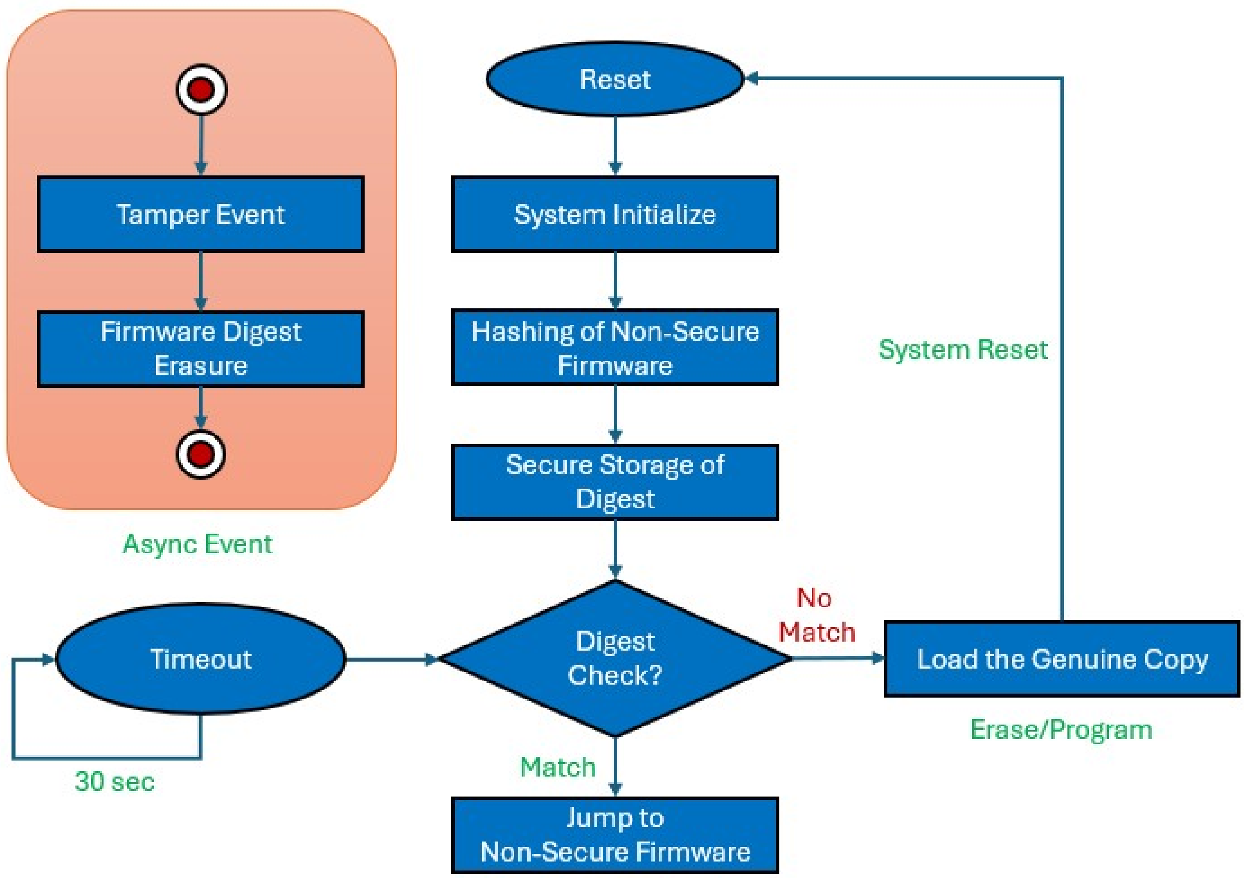


2.4.2. Execution Flow

Secure Firmware Execution Flow

The following figure illustrates the system-level execution flow of Secure firmware in the Software Attack Protection application.

Figure 2-3. Secure Application Execution Flow



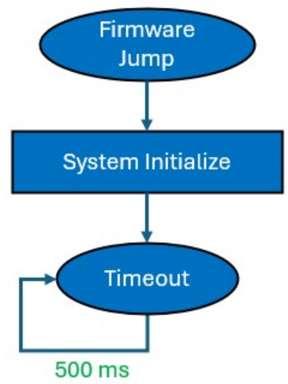
The Secure application executes in the following sequence:

1. After a system reset, the application initiates the Non-Secure firmware hashing process to generate a firmware digest.
2. The calculated digest is stored in the data Flash memory within the Secure region.
3. The firmware digest of the Non-Secure application is verified against the genuine copy in the Secure Flash memory.
4. Upon successful verification, the execution is jumped to the Non-Secure application.
5. If verification fails, the Non-Secure application is erased, and the genuine copy is loaded into the Non-Secure Flash region.
6. Every 30 seconds, the firmware digest is regenerated and cross-verified with the genuine copy to ensure authenticity.

Non-Secure Firmware Execution Flow

The following figure illustrates the system-level execution flow of Non-Secure firmware in the Software Attack Protection application.

Figure 2-4. Non-Secure Application Execution Flow



The Non-Secure application executes in the following sequence:

1. After a firmware jump from the Secure application, the Non-Secure firmware initializes the Non-Secure peripherals.
2. Toggles the LED1 for every 500 millisecond on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit.

Software Attack Simulation Execution Flow

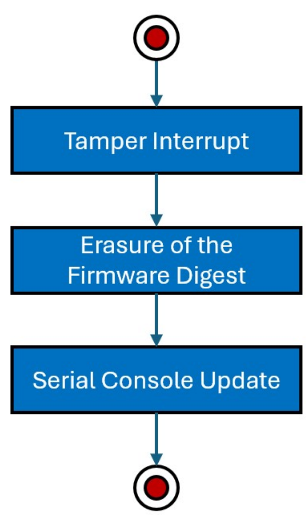
The simulation of a software attack is conducted in the following sequence:

1. Pressing the SW1 button on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit simulates a software attack by triggering a tamper event.
2. Following the tamper event, the RTC initiates the data Flash content erasure process.
3. Within the RTC tamper handler, a message indicating the initiation of the software attack is sent to the serial console.

Note: This execution happens inside the RTC interrupt handler of the Secure firmware.

The following figure illustrates the system-level execution flow of Software Attack in the Secure firmware.

Figure 2-5. Software Attack Execution Flow

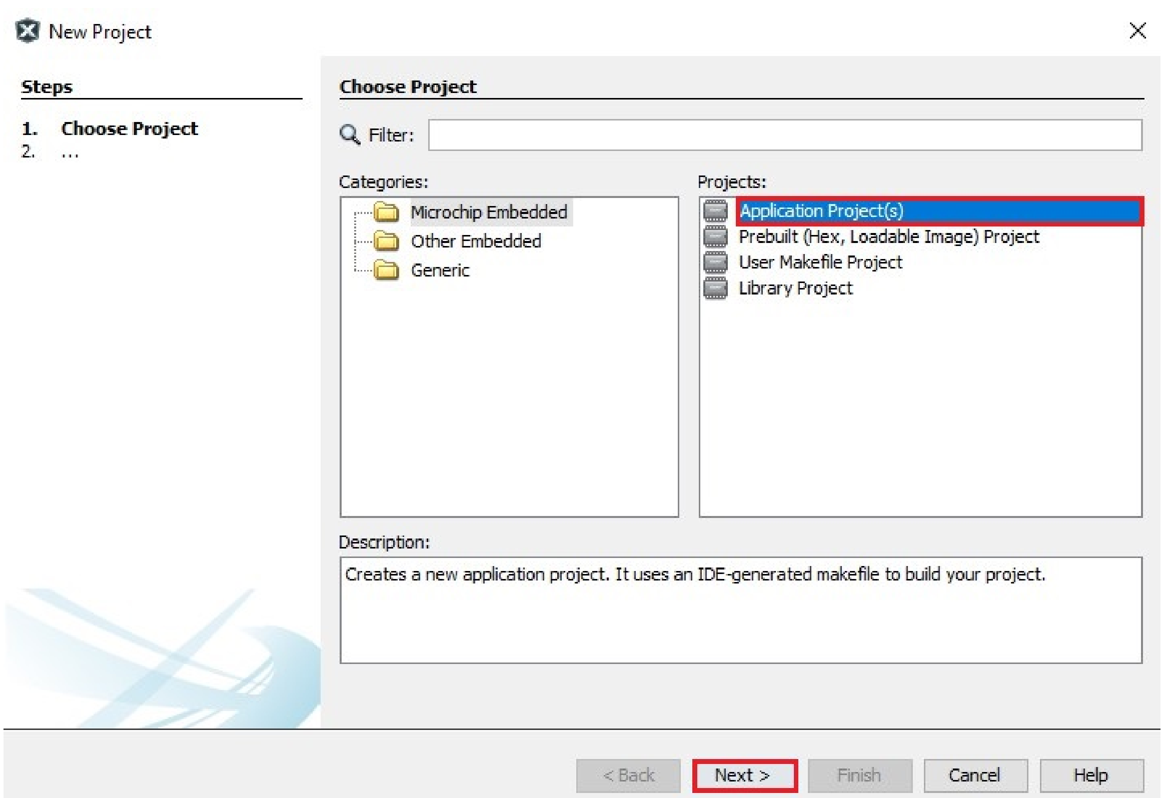


3. Implementing Software Attack Protection on The PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit using MPLAB Harmony v3 and MCC

To create an MPLAB Harmony v3-based project, follow these steps or download pre-developed demo project here.

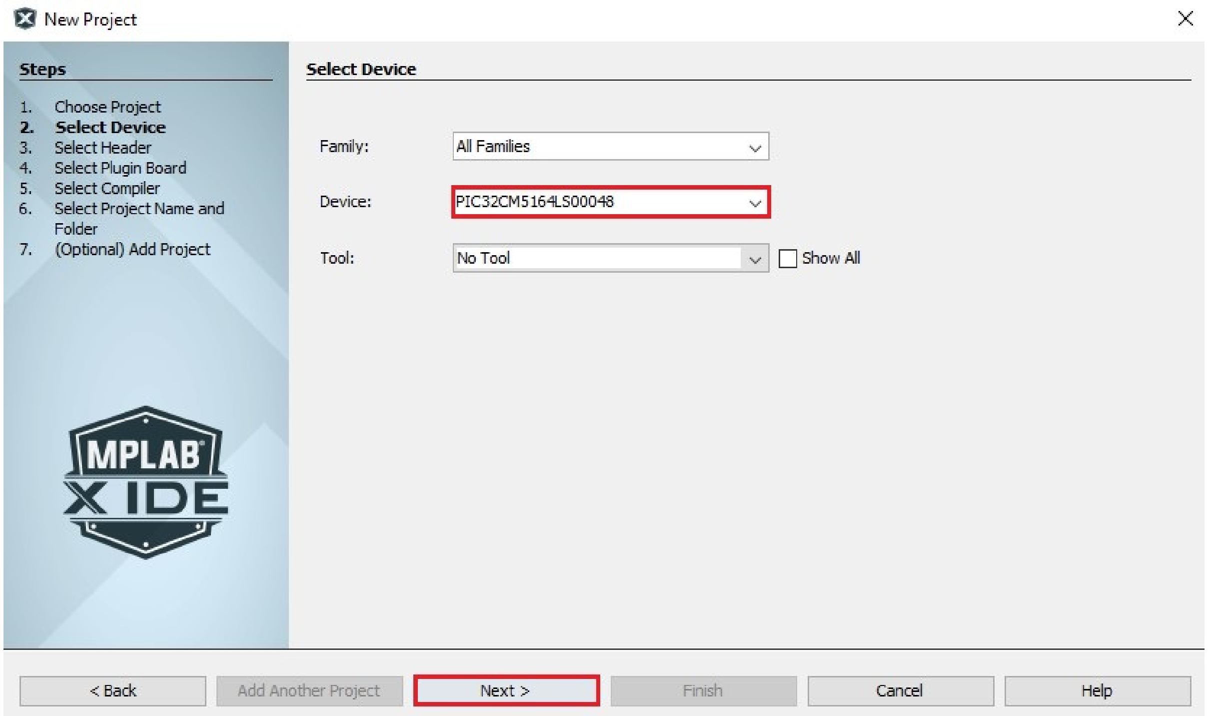
1. From the **Start** Menu, launch **MPLAB X IDE**.
2. Once MPLAB X IDE is open, from the File Menu, click **New Project** or click on the *New Project* icon.
3. In the **New Project** window, from the left Navigation pane, under Steps select **Choose Project.**
4. In the right Choose Project Properties Page:
   a. For Categories, select **Microchip Embedded.**
   b. For Projects, select **Application Project**.

Figure 3-1. New Project Creation



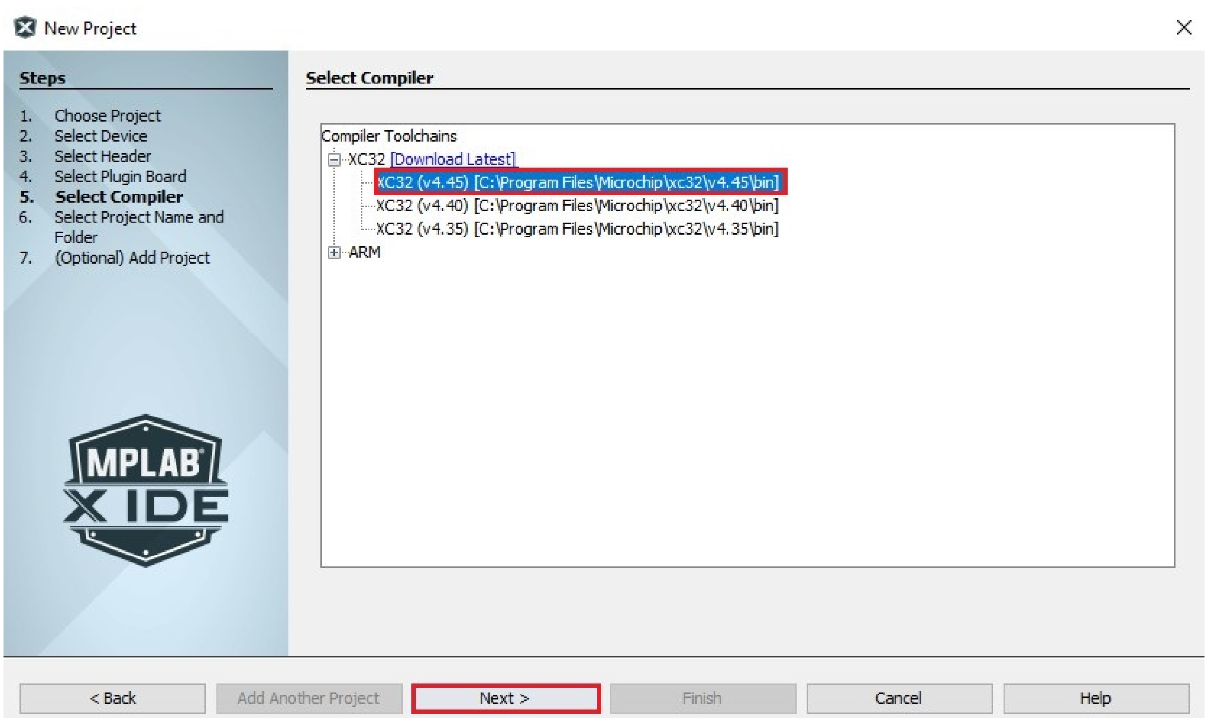
5. Click **Next**.
6. Under Steps select **Select Device** , and from the right Select Device Properties Page, for Device select **PIC32CM5164LS00048** to create the project on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit (The device entry will be reflected under the device).

Figure 3-2. Device Selection



7. Click **Next.**
8. Select **Select Compiler**, and from the right Select Compiler Properties Page click and expand XC32 and then select **XC32 Compiler.**

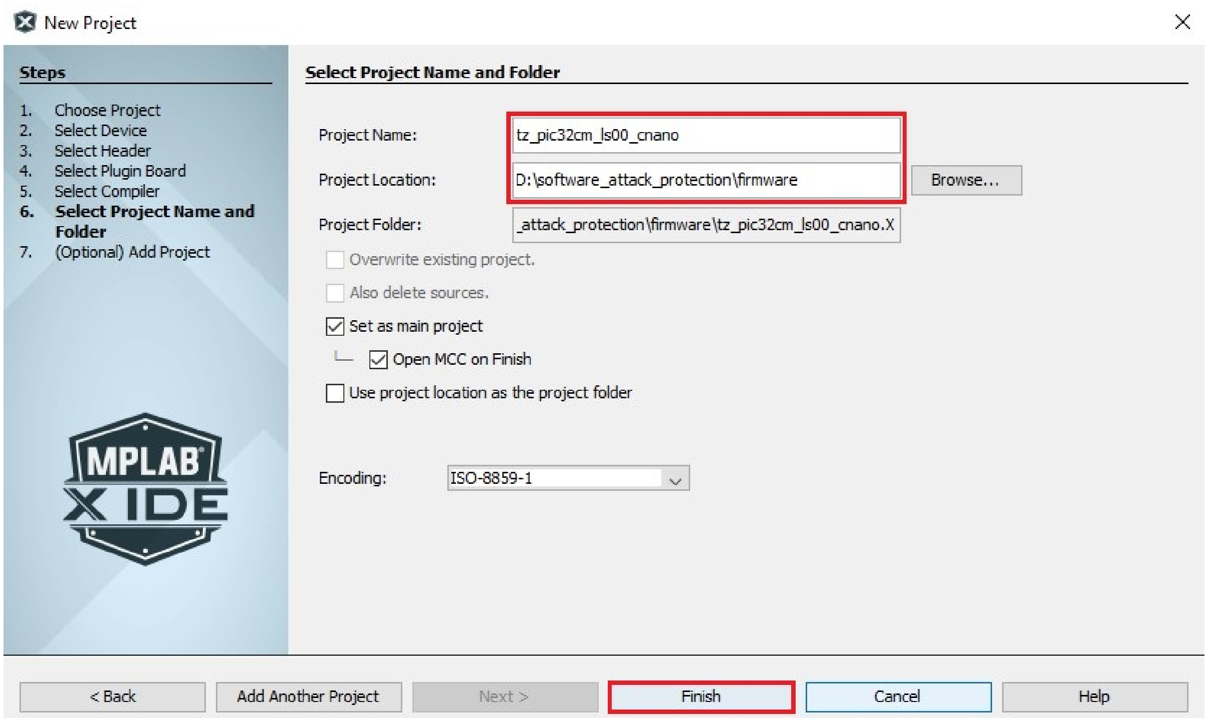
Figure 3-3. XC32 Compiler Selection



9. Click **Next**.

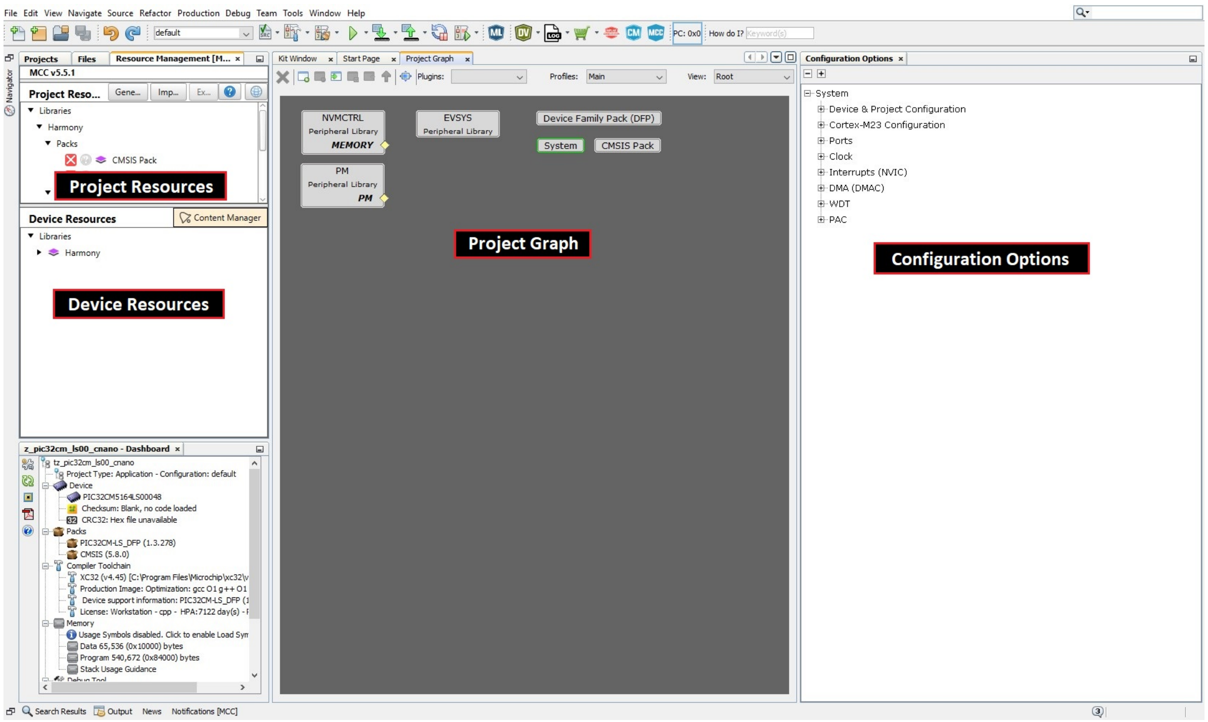
10. Select **Select Project Name and Folder** and from the right Select Project Name and Folder Properties Page enter these details:
    – Project Name: Enter *tz_pic32cm_ls00_cnano* (Indicates the project name that will be shown in MPLAB X IDE to set the project's name).
    – Location Project: Enter *D:\software_attack_protection\firmware* (Indicates the path to the root folder of the new project. All project files will be placed in this folder. The project location can be any valid path).
    – Project Folder: Read-only content (Automatically updates when users change the above entries).

Figure 3-4. Project Name and Folder Settings



11. Click **Finish** to launch the MCC.
12. The MCC plug-in will open in a new window, as shown in the following figure:

Figure 3-5. MPLAB Code Configurator Window

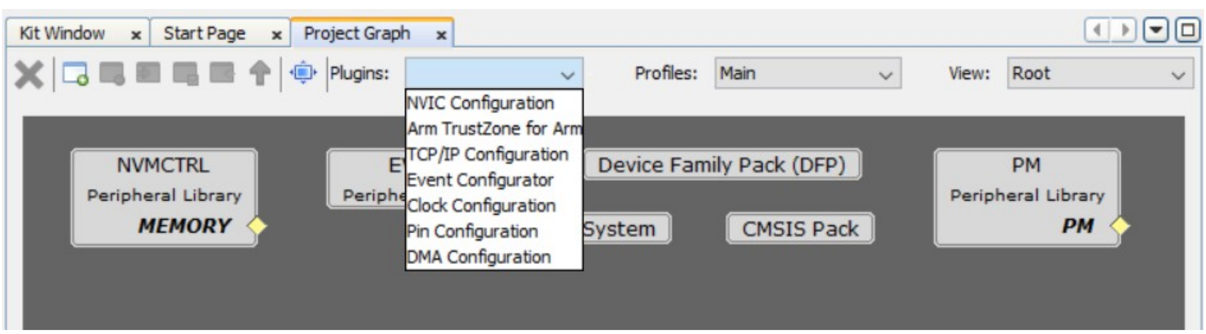


3.1. Adding and Configuring MPLAB Harmony Components

To add and configure MPLAB Harmony components using the MCC, follow these steps:

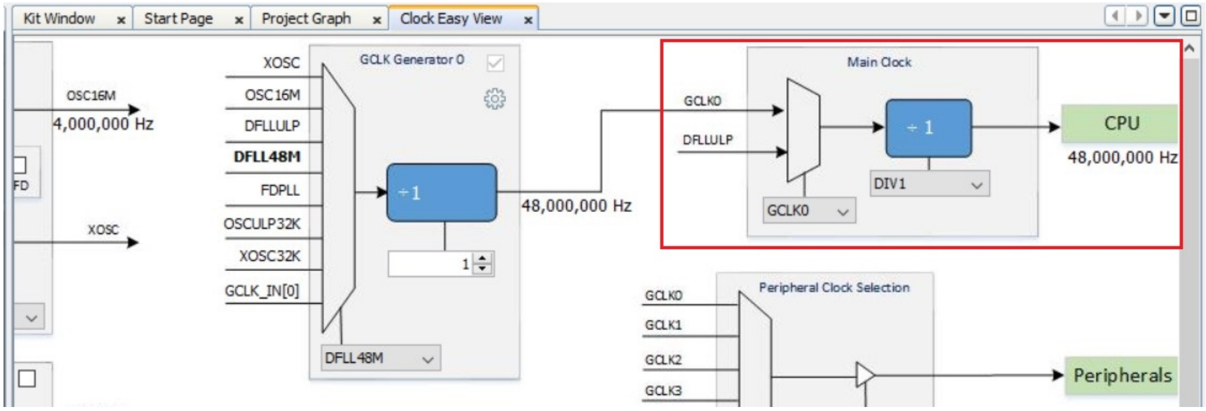
1. In the MCC window, from the Plugins drop-down list, select the required Configuration Window.

Figure 3-6. MPLAB Code Configurator – Plugins List



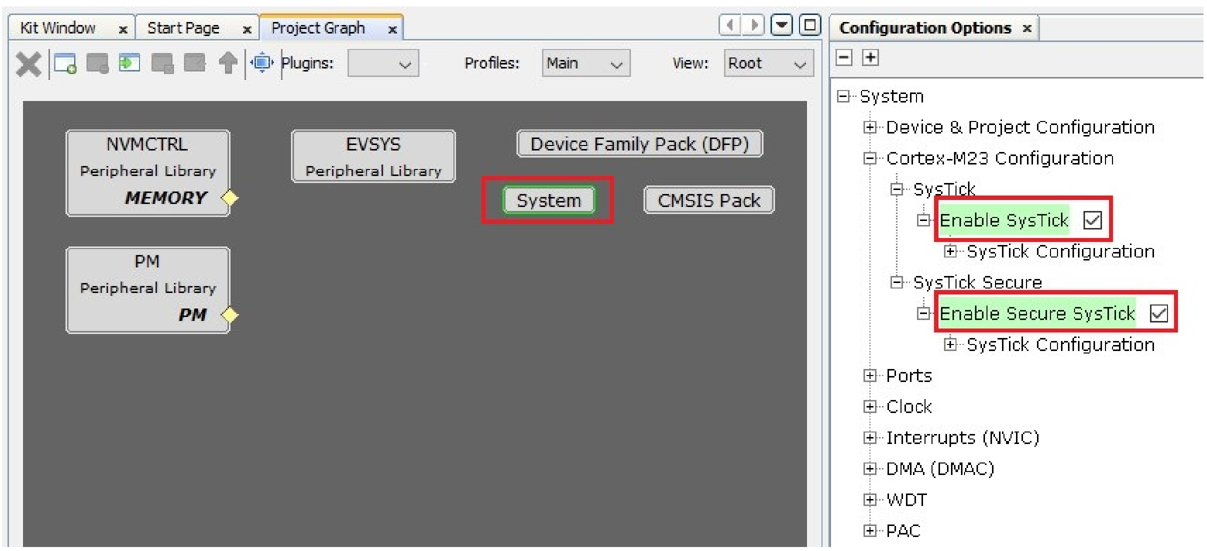
2. Select **Clock Configuration** to open the Clock Easy View window and verify that the Main Clock is set to 48 MHz.

Figure 3-7. MPLAB Code Configurator - GCLK Generator 0



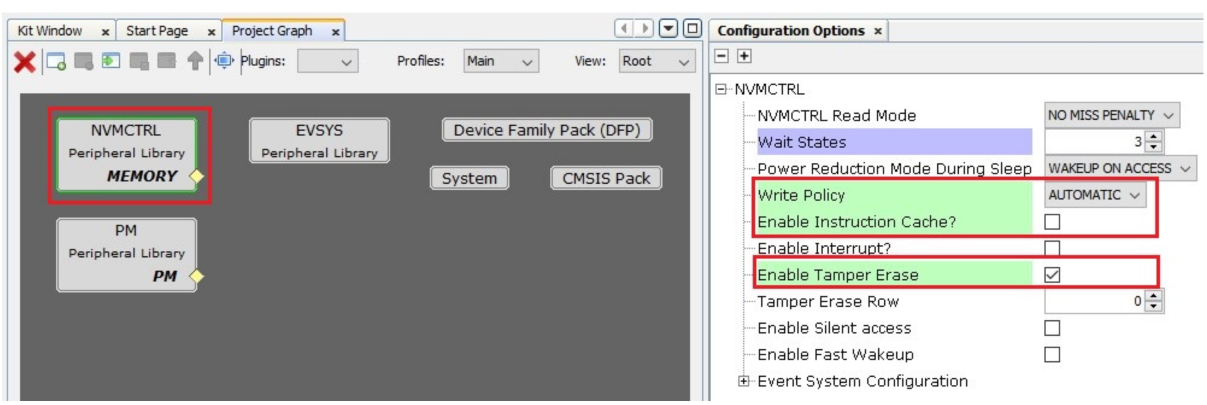
3. Click **Project Graph** and then select the **System** module. In the Configuration Options Properties Page, configure it as follows to enable the SysTick timer for the Secure and Non-Secure time delay.

Figure 3-8. MPLAB Code Configurator – SysTick Configuration



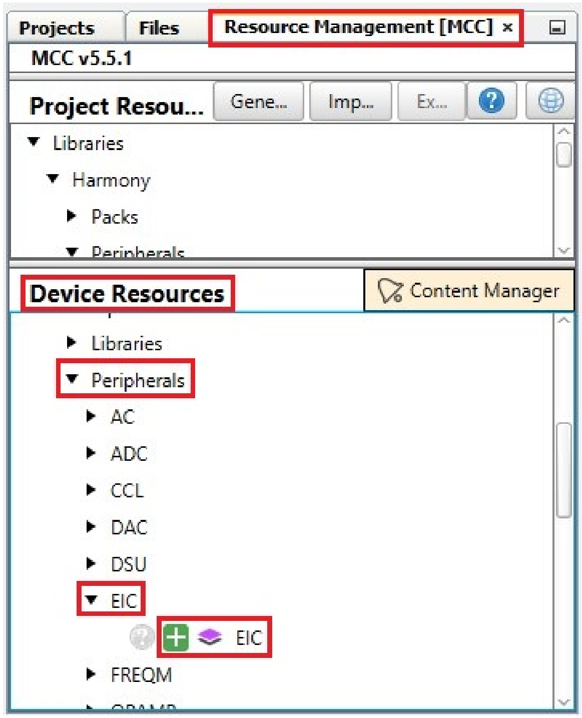
4. Select **NVMCTRL Peripheral Library MEMORY** and in the Configuration Options Properties Page, configure it as follows to enable the Tamper Erase feature.

Figure 3-9. MPLAB Code Configurator – NVMCTRL Configuration



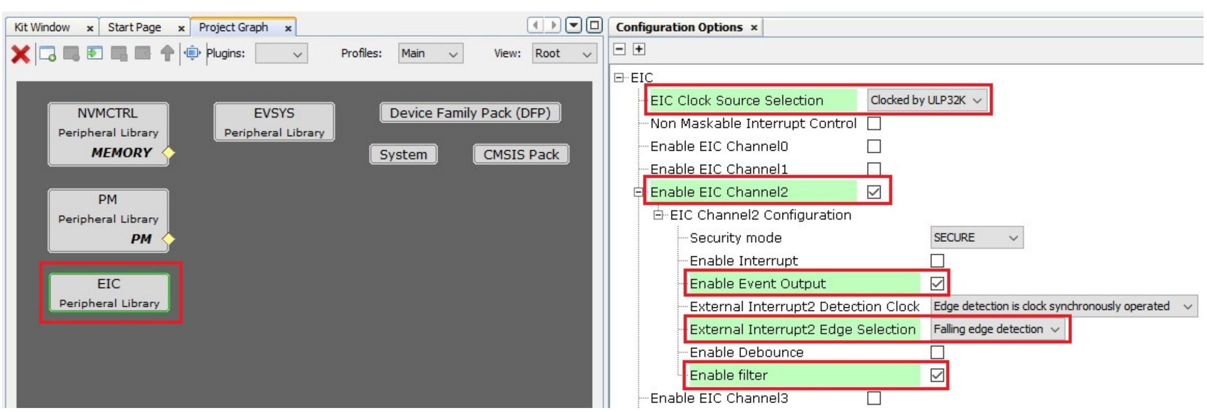
5. Click **Resource Management (MCC)** and under Device Resources, click and expand *Harmony* > *Peripherals* > *EIC*. Click **EIC** and observe that the EIC Peripheral Library block is added in the Project Graph Window.

Figure 3-10. MPLAB Code Configurator - Selection of EIC Peripheral



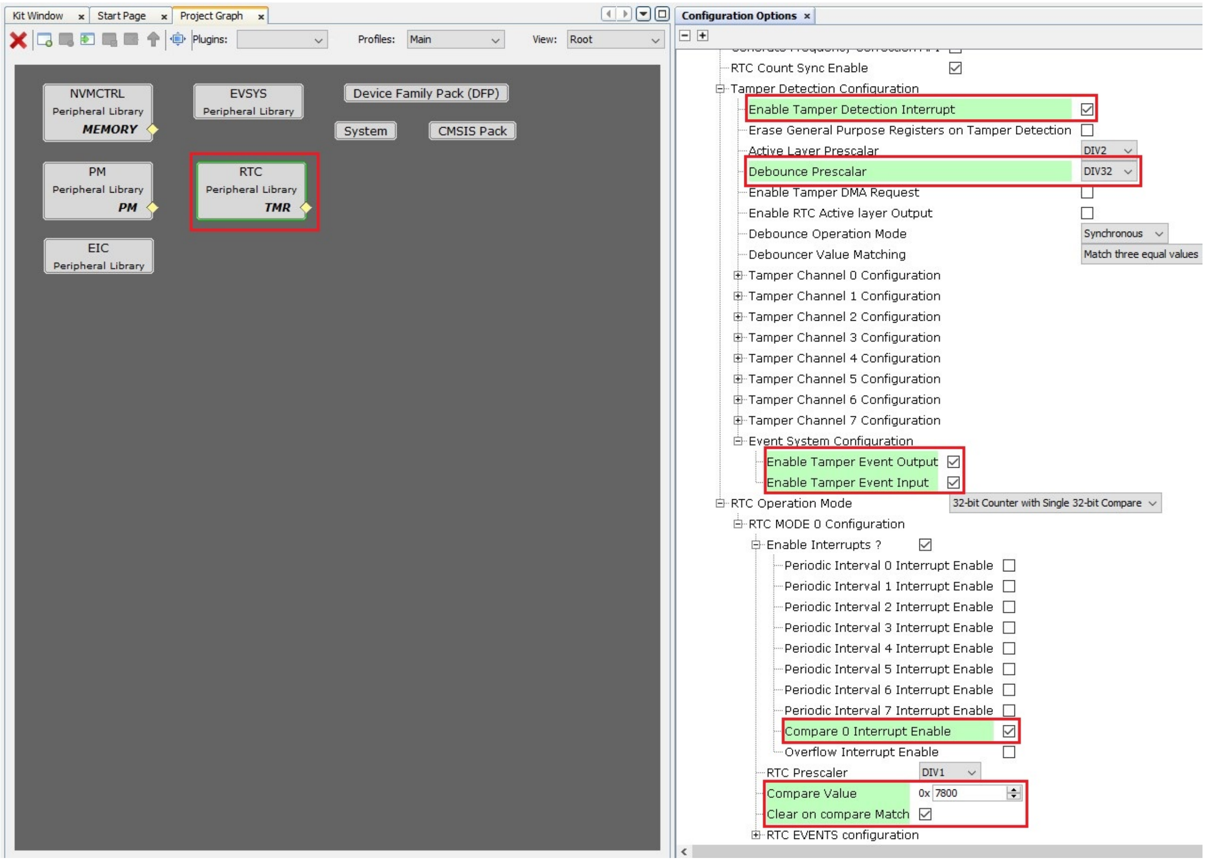
6. Select **EIC Peripheral Library** and in the right Configuration Options Property Page configure it as follows to use the EIC channel 2 (SW1) as tamper input.

Figure 3-11. MPLAB Code Configurator - EIC Configuration



7. Under Device Resources, click and expand *Harmony* > *Peripherals* > *RTC*. Click **RTC** and observe that the RTC Peripheral Library block is added in the Project Graph Window.
8. Select **RTC Peripheral Library** and in the Configurations Options Property page configure it as follows to generate a compare interrupt every 30 seconds and enable the tamper interrupt and events.

Figure 3-12. MPLAB Code Configurator - RTC Configuration



Notes: The Compare Value is set as 0x7800. This value generates an RTC compare interrupt every 30 seconds.

– RTC clock = 1024 Hz
– RTC Prescaler = 1
– Required Interrupt rate = 30s

Therefore, Compare Value = 30 x 1024 = 30,720 (i.e., 0x7800).

9. Under Device Resources:
   a. Click and expand *Harmony* > *Peripherals* > *SERCOM.* Click **SERCOM3** and observe that the SERCOM3 block is added in the Project Graph Window.
   b. Click and expand *Harmony* > *Peripherals* > *Tools.* Click **Secure STDIO** and observe that the Secure STDIO block is added in the Project Graph Window.

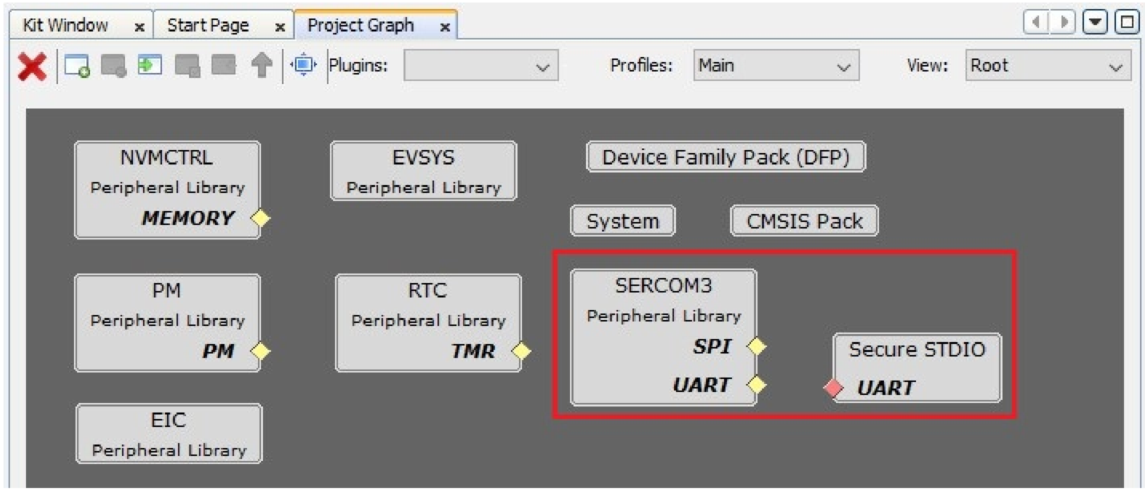


Figure 3-13. MPLAB Code Configurator – SERCOM and Secure STDIO Selection

10. Connect the SERCOM3 and Secure STDIO block by dragging the UART Yellow Diamond to the Red Diamond in the Secure STDIO block.

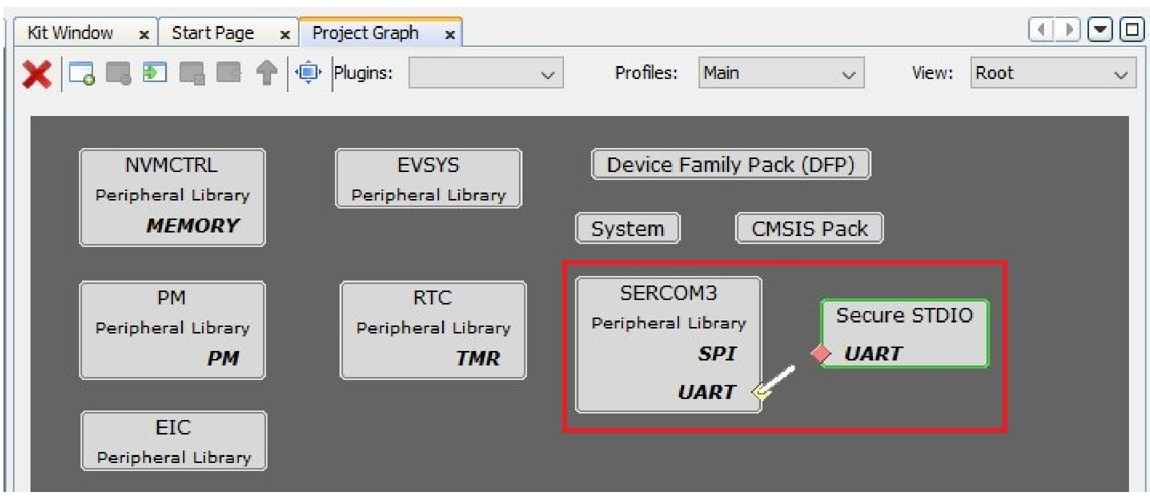


Figure 3-14. MPLAB Code Configurator – SERCOM and Secure STDIO Selection

11. In the left pane, select **SERCOM3 Peripheral Library**. In the Configuration Options property page, configure it as follows to print the data on the Serial Console at 115200 baud rate.

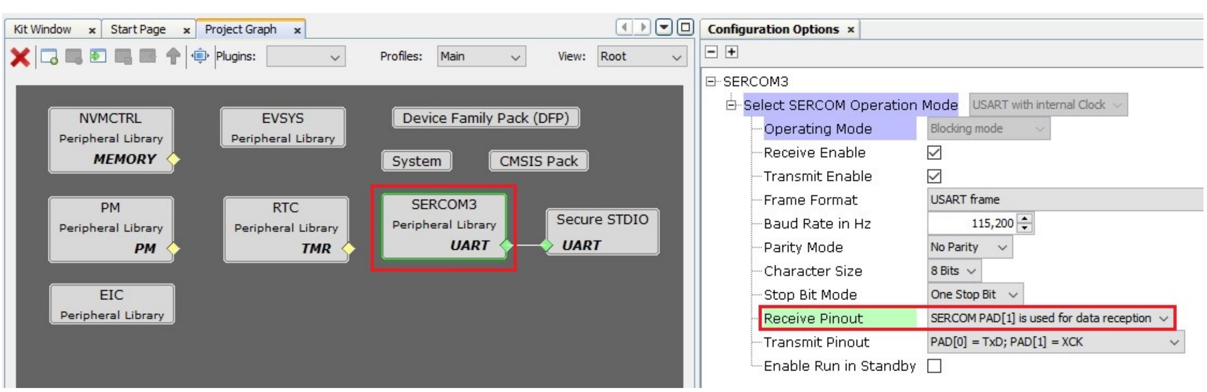
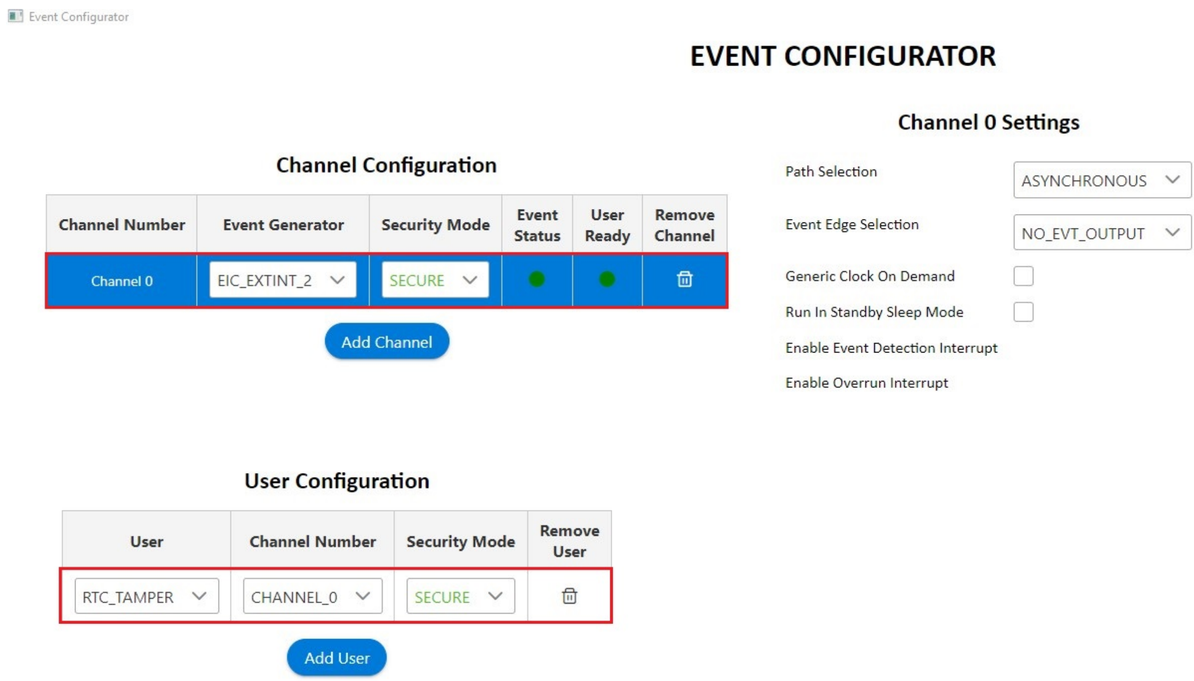


Figure 3-15. MPLAB Code Configurator – SERCOM3 Configuration

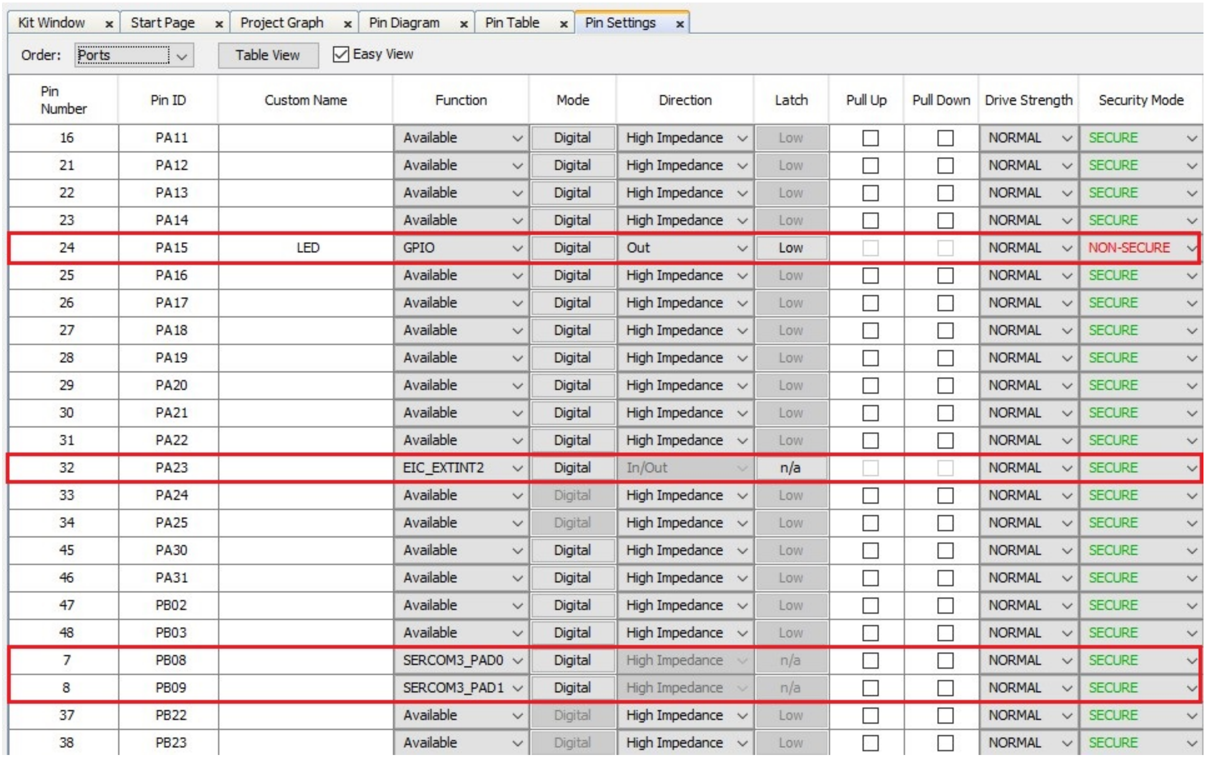
12. From the **Plugins** drop-down list, select *Event Configurator*. Add the Event Generator and Event User for tamper input as shown in the following figure.

Figure 3-16. MPLAB Code Configurator – Event Configuration



13. From the **Plugins** drop-down list select **Pin Configuration** and then click **Pin Settings** tab. Change the order to *Ports*. Make the pin configurations according to the application as indicated below.

Figure 3-17. Pin Settings Window - Pin Configuration



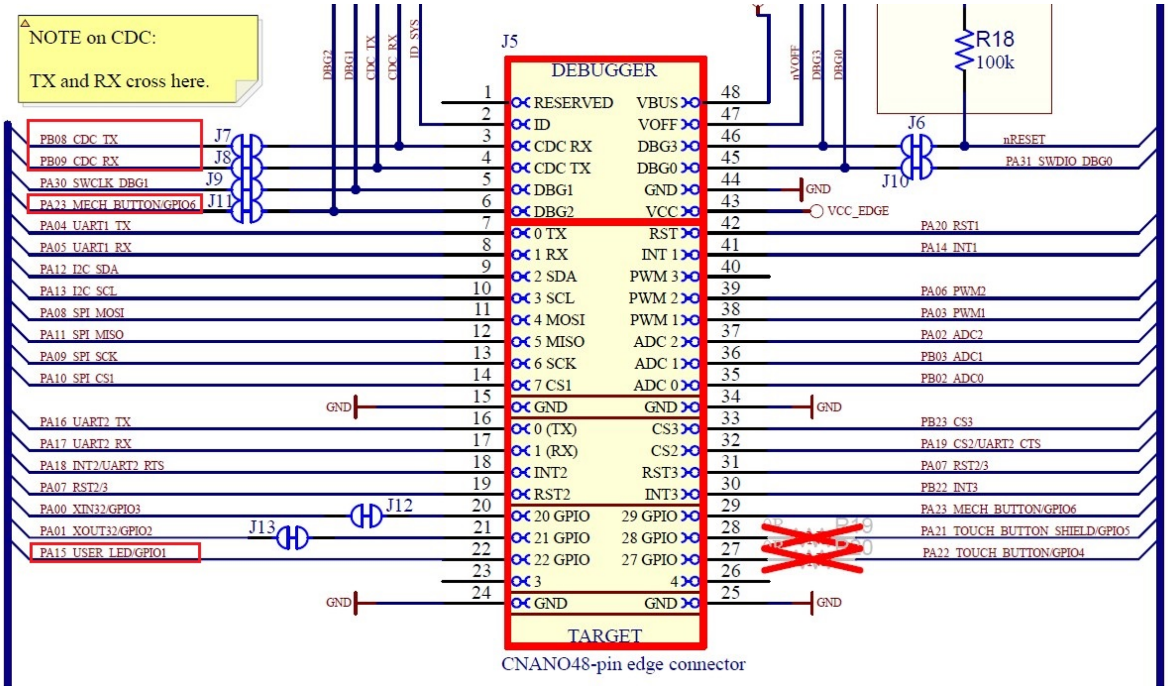
Pin Number	Pin ID	Custom Name	Function	Mode	Direction	Latch	Pull Up	Pull Down	Drive Strength	Security Mode
16	PA11		Available	Digital	High Impedance	Low			NORMAL	SECURE
21	PA12		Available	Digital	High Impedance	Low			NORMAL	SECURE
22	PA13		Available	Digital	High Impedance	Low			NORMAL	SECURE
23	PA14		Available	Digital	High Impedance	Low			NORMAL	SECURE
24	PA15	LED	GPIO	Digital	Out	Low			NORMAL	NON-SECURE
25	PA16		Available	Digital	High Impedance	Low			NORMAL	SECURE
26	PA17		Available	Digital	High Impedance	Low			NORMAL	SECURE
27	PA18		Available	Digital	High Impedance	Low			NORMAL	SECURE
28	PA19		Available	Digital	High Impedance	Low			NORMAL	SECURE
29	PA20		Available	Digital	High Impedance	Low			NORMAL	SECURE
30	PA21		Available	Digital	High Impedance	Low			NORMAL	SECURE
31	PA22		Available	Digital	High Impedance	Low			NORMAL	SECURE
32	PA23		EIC_EXTINT2	Digital	In/Out	n/a			NORMAL	SECURE
33	PA24		Available	Digital	High Impedance	Low			NORMAL	SECURE
34	PA25		Available	Digital	High Impedance	Low			NORMAL	SECURE
45	PA30		Available	Digital	High Impedance	Low			NORMAL	SECURE
46	PA31		Available	Digital	High Impedance	Low			NORMAL	SECURE
47	PB02		Available	Digital	High Impedance	Low			NORMAL	SECURE
48	PB03		Available	Digital	High Impedance	Low			NORMAL	SECURE
7	PB08		SERCOM3_PAD0	Digital	High Impedance	n/a			NORMAL	SECURE
8	PB09		SERCOM3_PAD1	Digital	High Impedance	n/a			NORMAL	SECURE
37	PB22		Available	Digital	High Impedance	Low			NORMAL	SECURE
38	PB23		Available	Digital	High Impedance	Low			NORMAL	SECURE

Notes:

- PB08, PB09: SERCOM3 TX and RX pins
- PA15: LED
- PA23: SWITCH

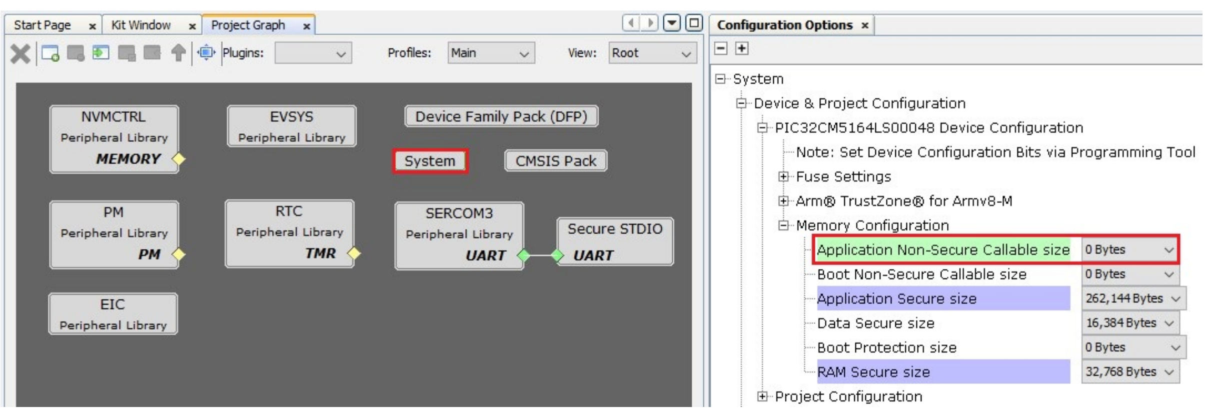
For additional information, refer to the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit User Guide (*DS70005567*).

Figure 3-18. PIC32CM LS00 Curiosity Nano+ Touch Board Pinout



14. Select **System** in the Project graph. In the Configuration Options property page, configure it as follows to set the Memory Configuration for the Non-Secure Callable Size to zero.

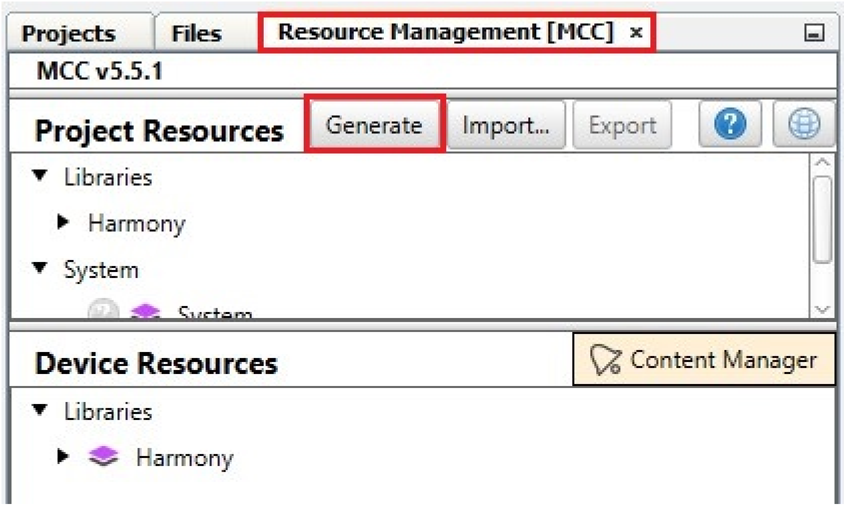
Figure 3-19. MPLAB Code Configurator – Memory Configuration



3.2. Generate Code

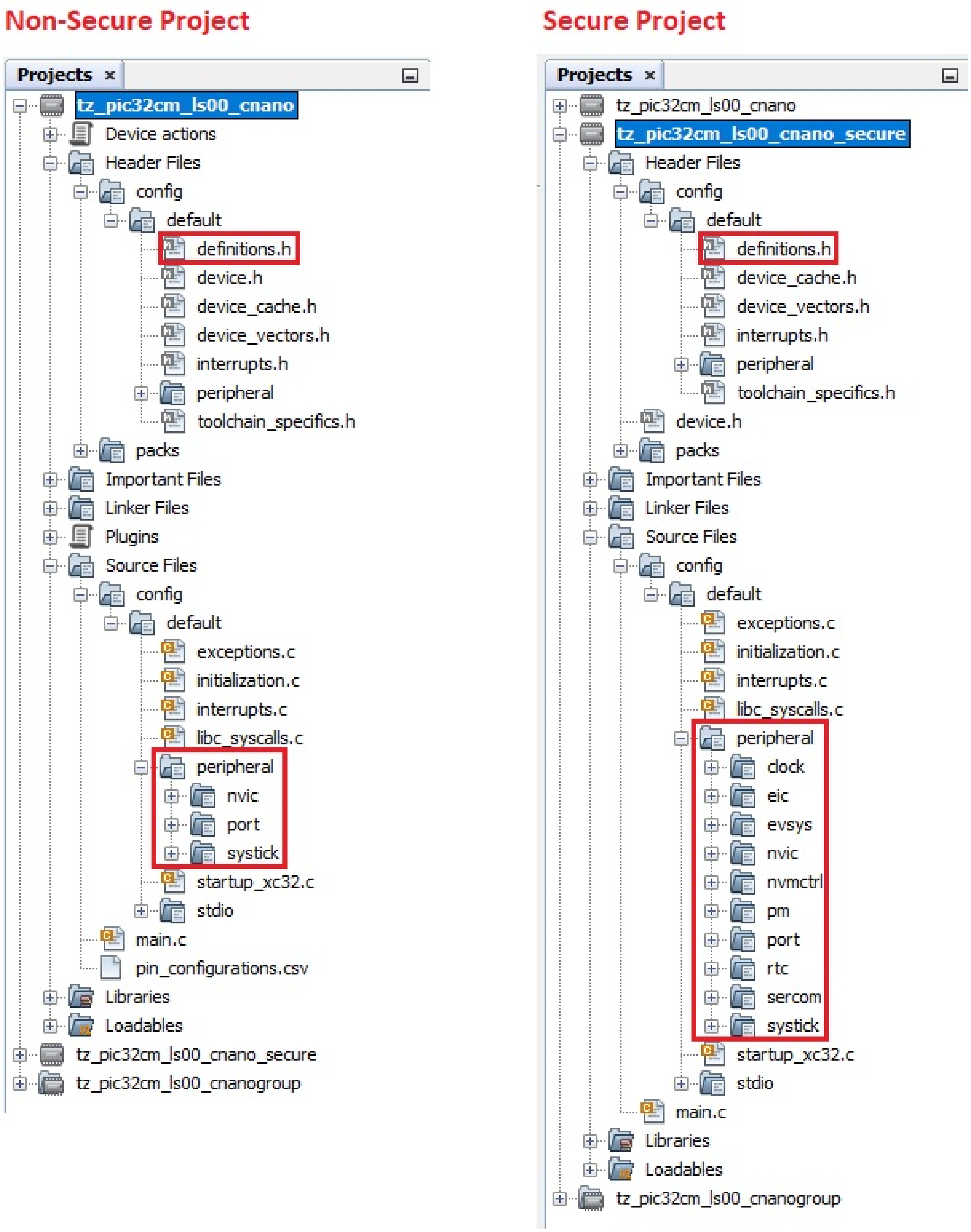
1. After configuring the peripherals, click *Resource Management [MCC]* and then click on the *Generate* tab.

Figure 3-20. Generation of Code



2. The generated code will add files and folders to the 32-bit MCC Harmony v3 project. In the generated code, notice the Peripheral Library files generated for SysTick, SERCOM, EIC, NVMCTRL, RTC, Event System, and PORT peripherals.

Figure 3-21. Generated Code on Non-Secure and Secure Projects



Notes:

- MCC generates the separate `main.c` file in Secure and Non-Secure Projects.
- MCC provides an option to change the generated file name, and if this option is not used, by default, the file name `main.c` is generated.

4. Adding Application Logic to the Non-Secure and Secure Projects

4.1. Adding the Non-Secure Application Logic

To develop and run the application, follow these steps:

1. Open the `main.c` file of the Non-Secure project (tz_pic32cm_ls00_cnano.X) and add the following code after the SYS_Initialize():

```
SYSTICK_TimerStart();
```

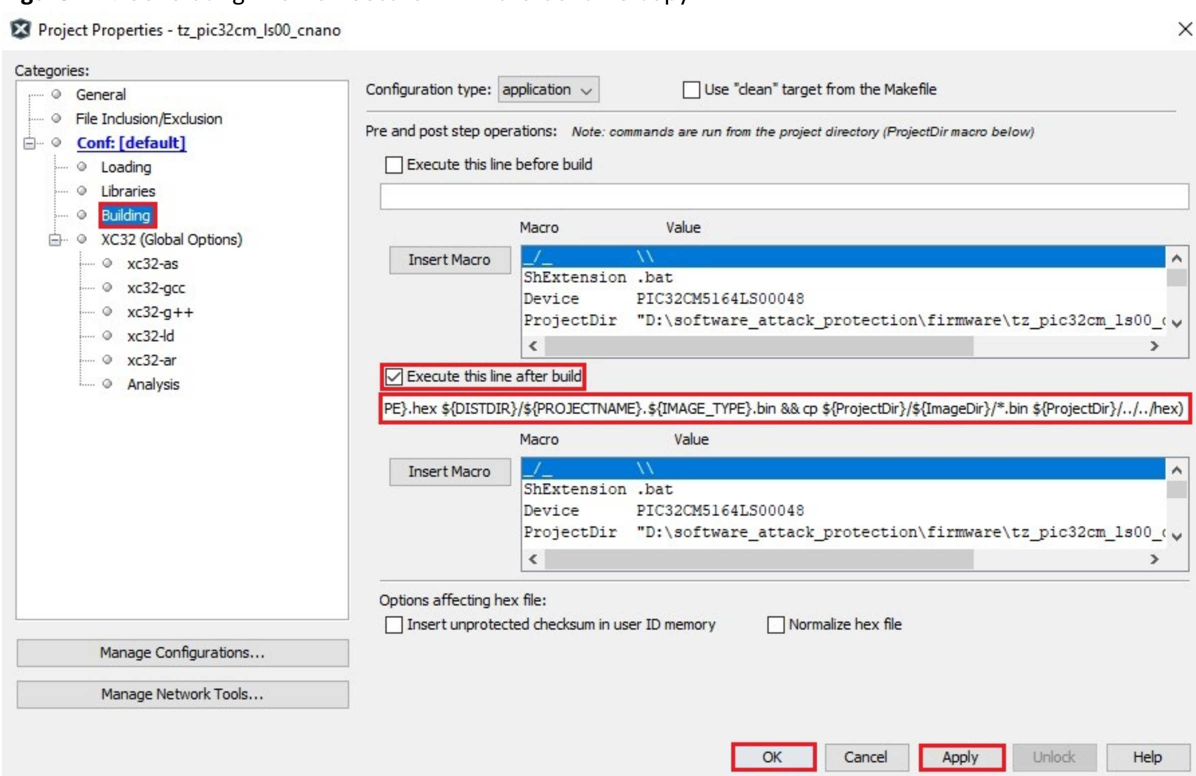
2. Inside the while loop, add the following code to toggle the LED at a default rate of 500 ms:

```
LED_Toggle();
SYSTICK_DelayMs(500);
```

3. Go to the Non-Secure Project Properties, and enter the post-build command for generating a Non-Secure firmware genuine copy:
   a. In the MPLAB X IDE Project Properties window perform these actions.
   b. Under the left Categories section, select **Building** and in the right Configuration properties page, select the **Execute this line after build** check box.
   c. Enter the following post command below the check box.

```
rm -rf ${ProjectDir}/../../hex && mkdir ${ProjectDir}/../../hex&& cp
${ProjectDir}/${ImageDir}/*.hex ${ProjectDir}/../../hex &&
${MP_CC_DIR}"/xc32-objcopy" -I ihex -O binary
${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.hex
${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bin && cp
${ProjectDir}/${ImageDir}/*.bin ${ProjectDir}/../../hex
```

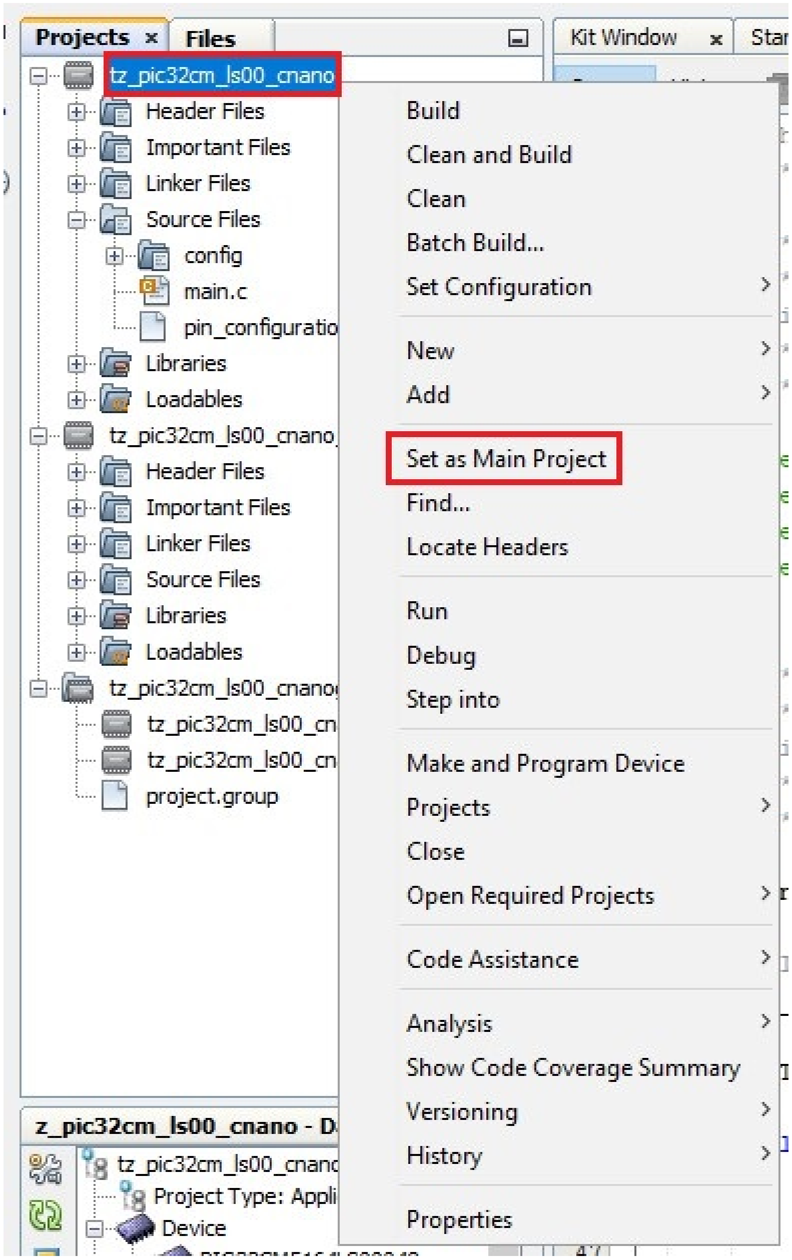
Figure 4-1. Generating The Non-Secure Firmware Genuine Copy



4. Click **Apply**, and then click **OK**.

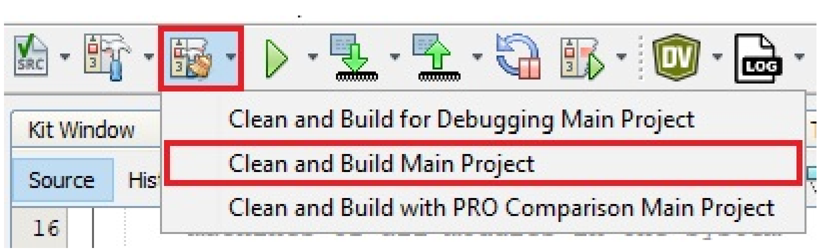
5. Under Projects, right-click on the **tz_pic32cm_ls00_cnano** and then select **Set as Main Project**.

Figure 4-2. Make the Non-Secure Project as Main Project



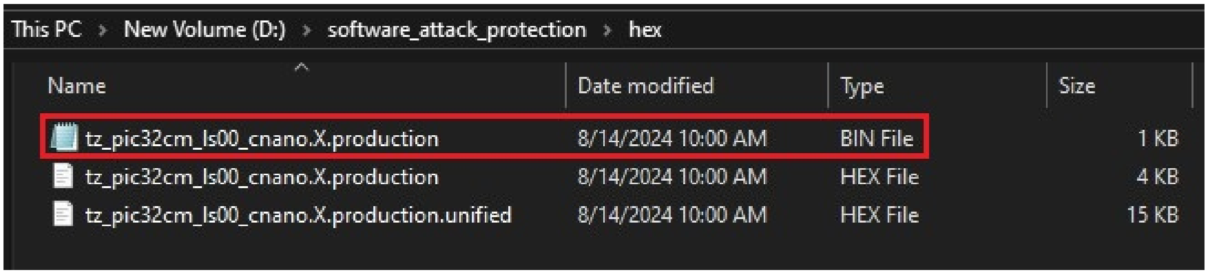
6. Build the project by clicking the *Clean and Build* icon or by selecting **Clean and Build Main Project** from the drop-down list and verify that the project builds successfully.

Figure 4-3. Clean and Build



7. Check if the binary file of the Non-Secure project is available in the hex folder location (path: *D:/software_attack_protection/hex*).

Figure 4-4. Location of Generated Binary File



8. Open the command prompt and navigate to the following location.
   Path: *<Harmony folder path>/ bootloader/tools*

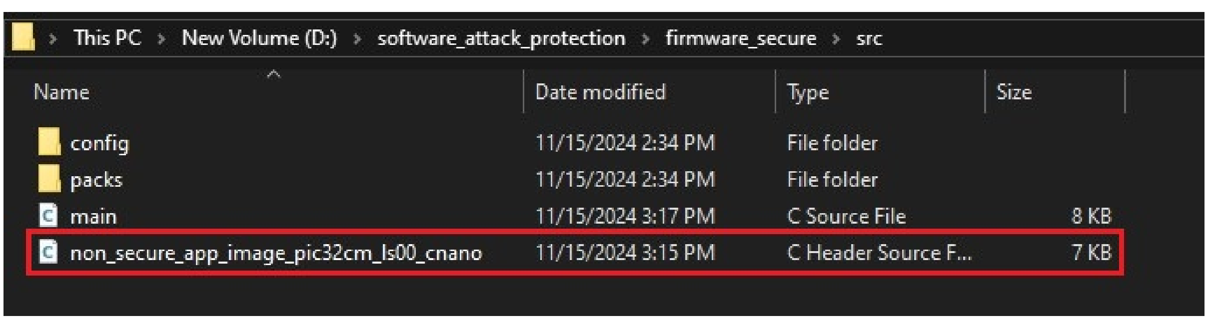
   Note: If the bootloader folder is not found inside the Harmony folder, download the bootloader package (v3.7.0 or above) using the MPLAB Content Manager.
9. Run the python script `btl_bin_to_c_array.py` to convert the Non-Secure application binary file to a C-style array containing Hex output.

```
python btl_bin_to_c_array.py -b
D:\software_attack_protection\hex\tz_pic32cm_ls00_cnano.X.production.bin -o
D:\software_attack_protection\firmware_secure\src\non_secure_app_image_pic32cm_ls00_cnano.h
 -d PIC32CM
```

Figure 4-5. Running the Python Script

10. Once the script is successfully executed, a header file of the Non-Secure Application Image (Genuine Copy) is found in the source folder of the Secure Project.

Figure 4-6. Genuine Copy of the Non-Secure Application Image



4.2. Adding the Secure Application Logic

To develop and run the application, follow these steps:

1. Declare the following variables and macros used by the secure application in the `main.c` file.

```
#include <string.h>
#include "non_secure_app_image_pic32cm_ls00_cnano.h"

#define APP_IMAGE_SIZE          sizeof(image_pattern)
```

```
#define APP_IMAGE_END_ADDR       (APP_IMAGE_START_ADDR + APP_IMAGE_SIZE)
#define NON_SECURE_APP_ADDR      (TZ_START_NS)

static uint8_t *appStart = (uint8_t *)NON_SECURE_APP_ADDR;
static uint8_t *dataStart = (uint8_t *)NVMCTRL_DATAFLASH_START_ADDRESS;

uint8_t firmware_digest_0[64];
uint8_t firmware_digest_1[32];
```

Figure 4-7. Declaration of Variables and Macros

```

#include <stddef.h>                       // Defines NULL
#include <stdbool.h>                      // Defines true
#include <stdlib.h>                       // Defines EXIT_FAILURE
#include "definitions.h"                  // SYS function prototypes

/* typedef for non-secure callback functions */
typedef void (*funcptr_void) (void) __attribute__((cmse_nonsecure_call));

#include <string.h>
#include "non_secure_app_image_pic32cm_ls00_cnano.h"

#define APP_IMAGE_SIZE           sizeof(image_pattern)
#define APP_IMAGE_END_ADDR       (APP_IMAGE_START_ADDR + APP_IMAGE_SIZE)
#define NON_SECURE_APP_ADDR      (TZ_START_NS)

static uint8_t *appStart = (uint8_t *)NON_SECURE_APP_ADDR;
static uint8_t *dataStart = (uint8_t *)NVMCTRL_DATAFLASH_START_ADDRESS;

uint8_t firmware_digest_0[64];
uint8_t firmware_digest_1[32];
```

2. Add the Boot ROM APIs in the `main.c` file to access them as follows.

```
typedef struct
{
    /* Digest result of SHA256 */
    uint32_t digest[8];
    /* Length of the message */
    uint64_t length;
    /* Holds the size of the remaining part of data */
    uint32_t remain_size;
    /* Buffer of remaining part of data (512 bits data block) */
    uint8_t remain_ram[64];
    /* RAM buffer of 256 bytes used by crya_sha_process */
    uint32_t process_buf[64];

} SHA256_CTX;

SHA256_CTX sha256_ctx;

typedef void (*crya_sha256_init_t) (SHA256_CTX *context);
typedef void (*crya_sha256_update_t) (SHA256_CTX *context, const unsigned char *data,
size_t length);
typedef void (*crya_sha256_final_t) (SHA256_CTX *context, unsigned char output[32]);

#define crya_sha256_init ((crya_sha256_init_t) (0x02006810 | 0x1))
#define crya_sha256_update ((crya_sha256_update_t) (0x02006814 | 0x1))
#define crya_sha256_final ((crya_sha256_final_t) (0x02006818 | 0x1))
```

3. Include the flash_write API in the `main.c` file to program the Non-Secure firmware in the Non-Secure Flash region and write the firmware digest in the Secure Data Flash.

```
static void flash_write(uint32_t addr, uint8_t *buf, uint32_t size)
{
    uint32_t end_addr = addr + size;
```

```
    if((addr & NVMCTRL_DATAFLASH_START_ADDRESS) == NVMCTRL_DATAFLASH_START_ADDRESS)
    {
        /* Unlock the Secure Data Flash region */
        NVMCTRL_RegionUnlock(NVMCTRL_SECURE_MEMORY_REGION_DATA);
        while(NVMCTRL_IsBusy());
    }

    else
    {
        /* Unlock the Non-Secure Flash region */
        NVMCTRL_RegionUnlock(NVMCTRL_MEMORY_REGION_APPLICATION);
        while(NVMCTRL_IsBusy());
    }

    do
    {
        if(addr % NVMCTRL_FLASH_ROWSIZE == 0)
        {
            /* Erase the row */
            NVMCTRL_RowErase(addr);
            while(NVMCTRL_IsBusy());
        }

        /* Program 64 byte page */
        NVMCTRL_PageWrite((uint32_t *)(buf), addr);
        while(NVMCTRL_IsBusy());

        addr += NVMCTRL_FLASH_PAGESIZE;
        buf += NVMCTRL_FLASH_PAGESIZE;

    }while (addr < end_addr);
}
```

4. Add the SHA-256 Hash and Non-Secure firmware verification API to the `main.c` file for calculating the Non-Secure firmware digest.

```
static void sha256_hash(SHA256_CTX *ctx, const uint8_t *message, uint32_t length,
        unsigned char digest[32])
{
    uint8_t dataBuf[64];

    uint32_t bufIdx = 0;

    crya_sha256_init(ctx);

    do
    {
        memcpy(dataBuf, &message[bufIdx], 64);

        crya_sha256_update(ctx, dataBuf, sizeof(dataBuf));
        bufIdx += 64;

    }while (bufIdx < APP_IMAGE_SIZE);

    crya_sha256_final(ctx, digest);
}
```

```
static bool non_secure_app_verify(void)
{
    sha256_hash(&sha256_ctx, appStart, APP_IMAGE_SIZE, firmware_digest_1);

    if(memcmp(dataStart, firmware_digest_1, 32) != 0)
    {
        printf("Firmware is Corrupted....!");
        printf("\n\r\n\r");

        printf("Firmware Digest after tamper detection:");
        printf("\n\r\n\r");

        for(int i=0; i<32; i++)
        {
            printf("0x%X ", dataStart[i]);

            if((i%8 == 0) && (i != 0))
            {
                printf("\n\r");
```

```
            }
        }
        flash_write(TZ_START_NS, (uint8_t *)&image_pattern, sizeof(image_pattern));

        sha256_hash(&sha256_ctx, image_pattern, APP_IMAGE_SIZE, firmware_digest_1);

        printf("\n\r\n\r");
        printf("Restored Firmware Digest:");
        printf("\n\r\n\r");

        for(int i=0; i<32; i++)
        {
            printf("0x%X ", firmware_digest_1[i]);

            if((i%8 == 0) && (i != 0))
            {
                printf("\n\r");
            }
        }
        printf("\n\r\n\r");
        printf("Genuine Firmware is restored");

    }
    else
    {
        return false;
    }

    return true;
}
```

5. Include the RTC Callback in the `main.c` for the tamper interrupt and 30-second timeout.

```
void timeout_handler(RTC_TIMER32_INT_MASK intCause, uintptr_t context)
{
    if(RTC_TIMER32_INT_MASK_CMP0  == (RTC_TIMER32_INT_MASK_CMP0 & intCause ))
    {
        if(non_secure_app_verify() == true)
        {
            SYSTICK_DelayMs(2000);

            NVIC_SystemReset();
        }
    }

    if (RTC_TIMER32_INT_MASK_TAMPER == (intCause & RTC_TIMER32_INT_MASK_TAMPER))
    {
        RTC_REGS->MODE2.RTC_TAMPID = RTC_TAMPID_Msk;

        printf("Software Attack Detected");
        printf("\n\r\n\r");
    }
}
```

6. Add the following code snippets after the SYS_Initialize API in the `main.c`.

```
sha256_hash(&sha256_ctx, image_pattern, APP_IMAGE_SIZE, firmware_digest_0);

flash_write(NVMCTRL_DATAFLASH_START_ADDRESS, firmware_digest_0, sizeof(firmware_digest_0));
```

Notes:

- `sha256_hash`: Calculates the digest of the Non-Secure Firmware.
- `flash_write`: Stores the firmware digest in the Secure Data Flash region.

```
SYSTICK_TimerStart();

    RTC_Timer32CallbackRegister(timeout_handler,0);
    RTC_Timer32Start();
    printf("\n\r----------------------------------------------------------");
    printf("\n\r           Software Attack Protection Demo                ");
    printf("\n\r----------------------------------------------------------\n\r");

    if(non_secure_app_verify() != true)
    {
```

```
        printf("\n\rFirmware is Genuine");
        printf("\n\r\n\r");
    }
```

Note:

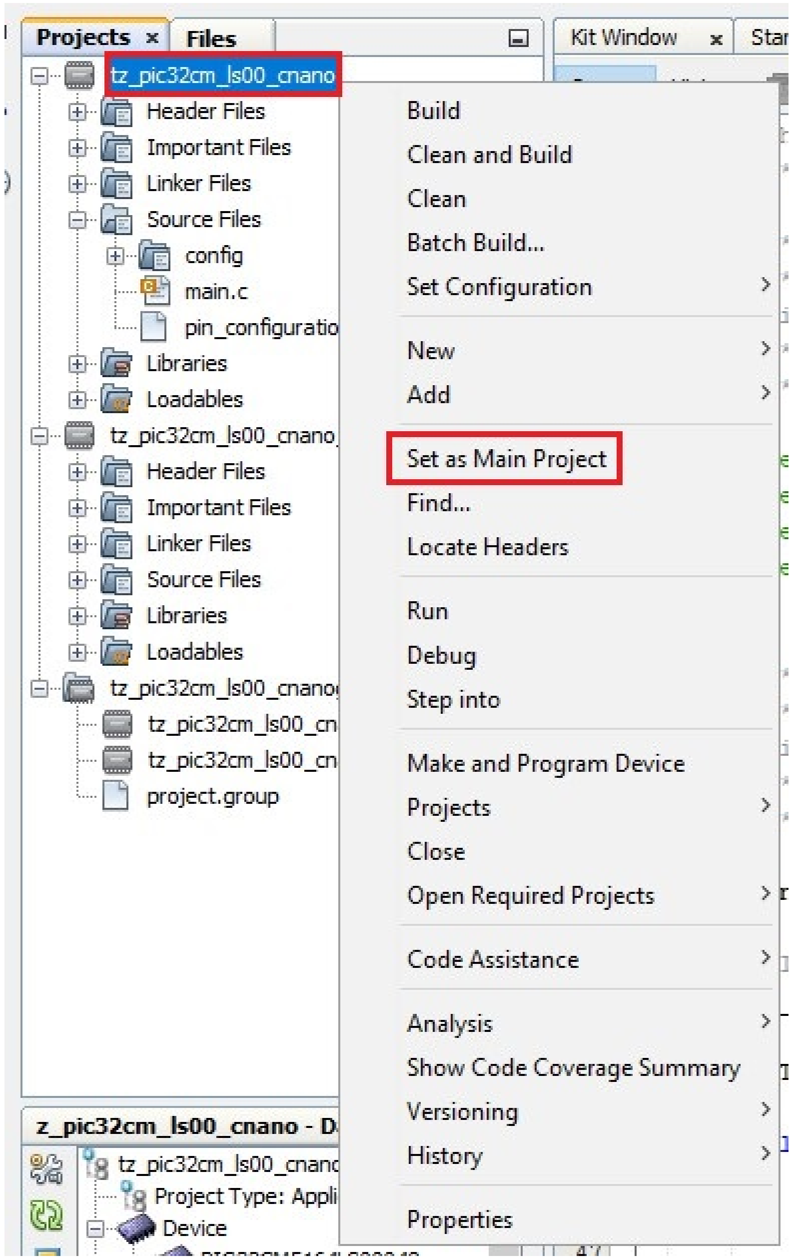
- non_secure_app_verify: Verify the Non-Secure firmware and program the genuine copy in the Non-Secure Flash region if the verification fails.

5. Building and Running the Application

Follow these steps to program the Software Attack Protection Application on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit.

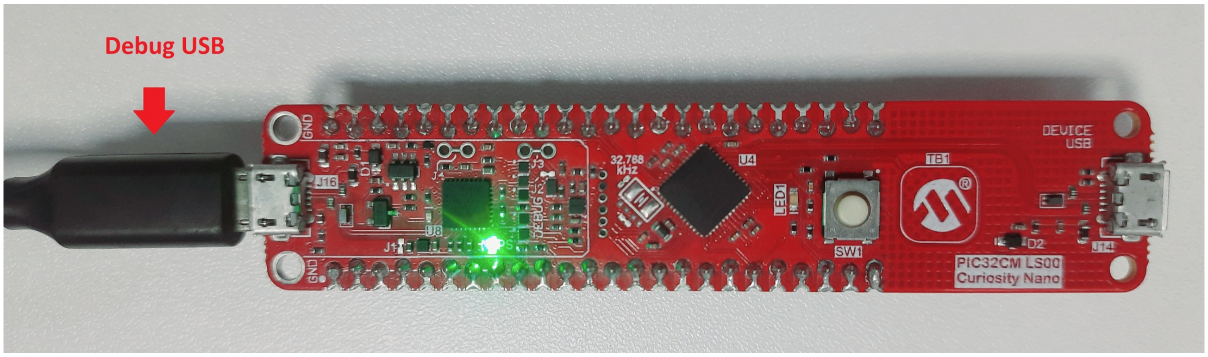
1. Set the *tz_pic32cm_ls00_cnano* project as the main project by right-clicking the project and selecting **Set as Main Project.**

   Figure 5-1. Make the Non-Secure Project as Main Project



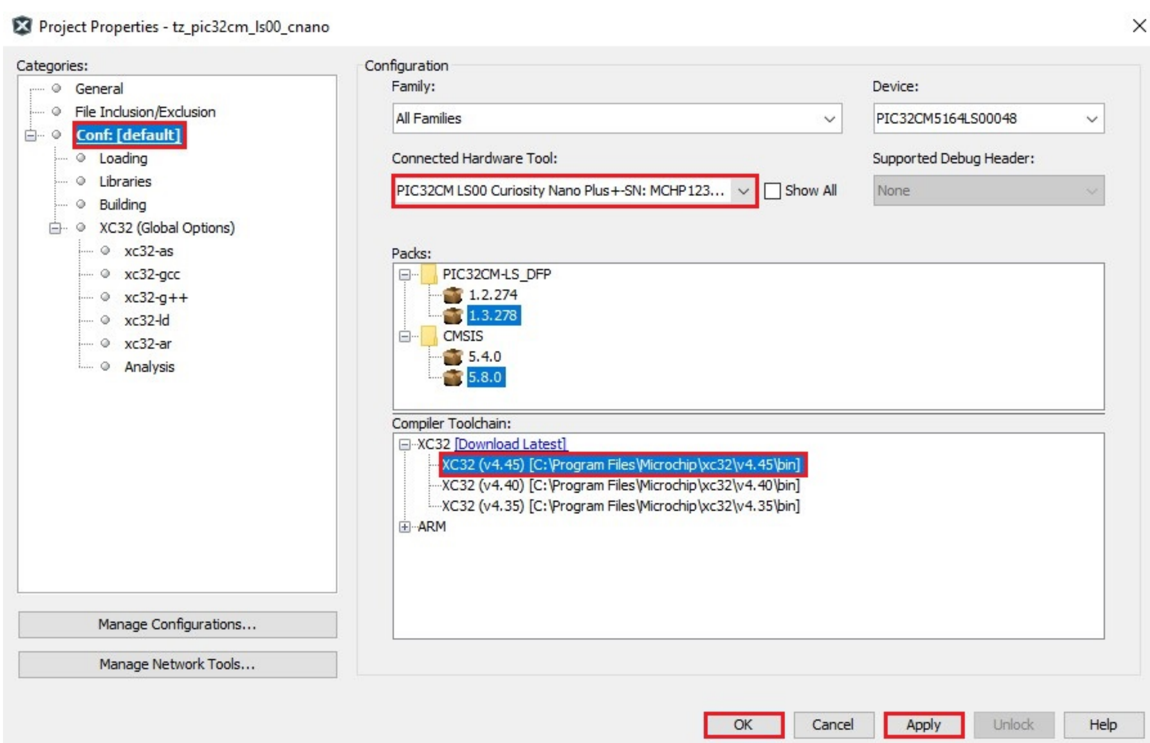
2. The PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit supports debugging using a Nano Embedded Debugger (nEDBG). Connect the *Type-A male to micro-B* USB cable to the *micro-B* USB port on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit to power and debug the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit.

Figure 5-2. PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit Hardware Setup



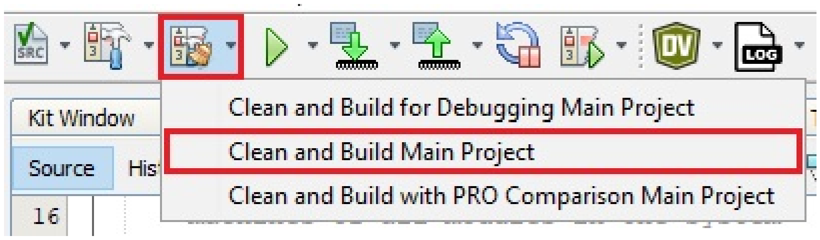
3. Go to the Project Properties, and select the Hardware Tool and Compiler:
   a. In the MPLAB X IDE Project Properties window perform these actions.
   b. Under the left Categories section, select *Conf: [default]*, and in the right Configuration properties sheet, select the Connected Hardware Tool and Compiler Toolchain.

Figure 5-3. Project Properties - PIC32CM LS00 Curiosity Nano+Touch Evaluation Kit



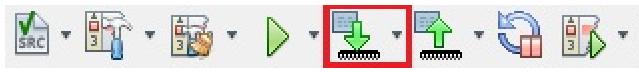
4. Click **Apply,** and then click **OK.**
5. Build the project by clicking on the *Clean and Build* icon or selecting **Clean and Build Main Project** from the drop-down item list and verify that the project builds successfully.

Figure 5-4. Clean and Build



6. Program the application by clicking the highlighted icon below.

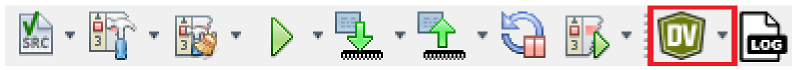
Figure 5-5. Program the Device



6. Observe the Output on the MPLAB Data Visualizer

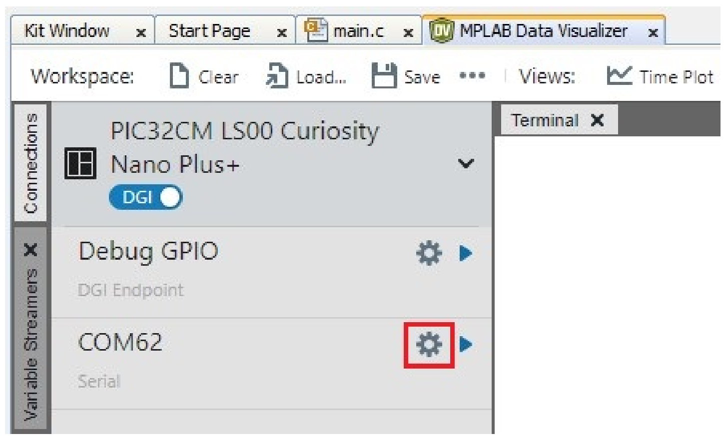
1. After building the application and completing the programming, open the MPLAB Data Visualizer by clicking the highlighted icon below.

Figure 6-1. Launch the MPLAB Data Visualizer



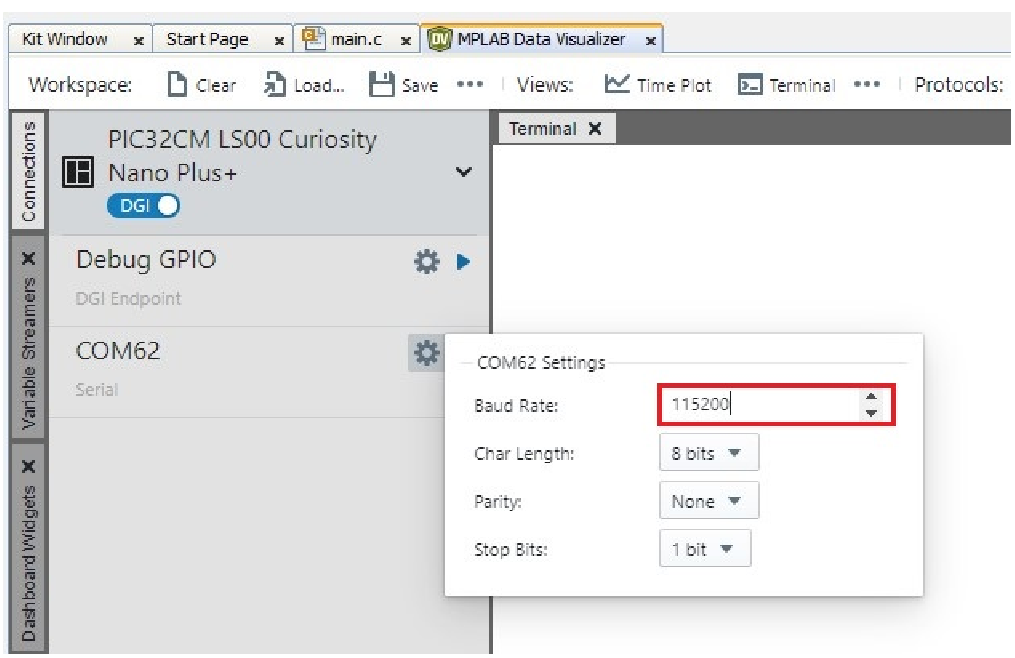
2. Configure the serial port setup of the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit by clicking the *Gear* icon shown below.

Figure 6-2. Serial Port Setup



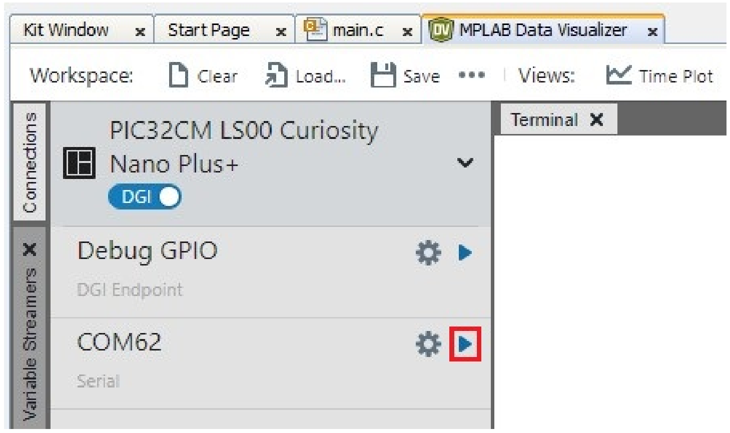
3. Set the 115200 as baud rate in the COM setting.

Figure 6-3. Baud Rate Configuration



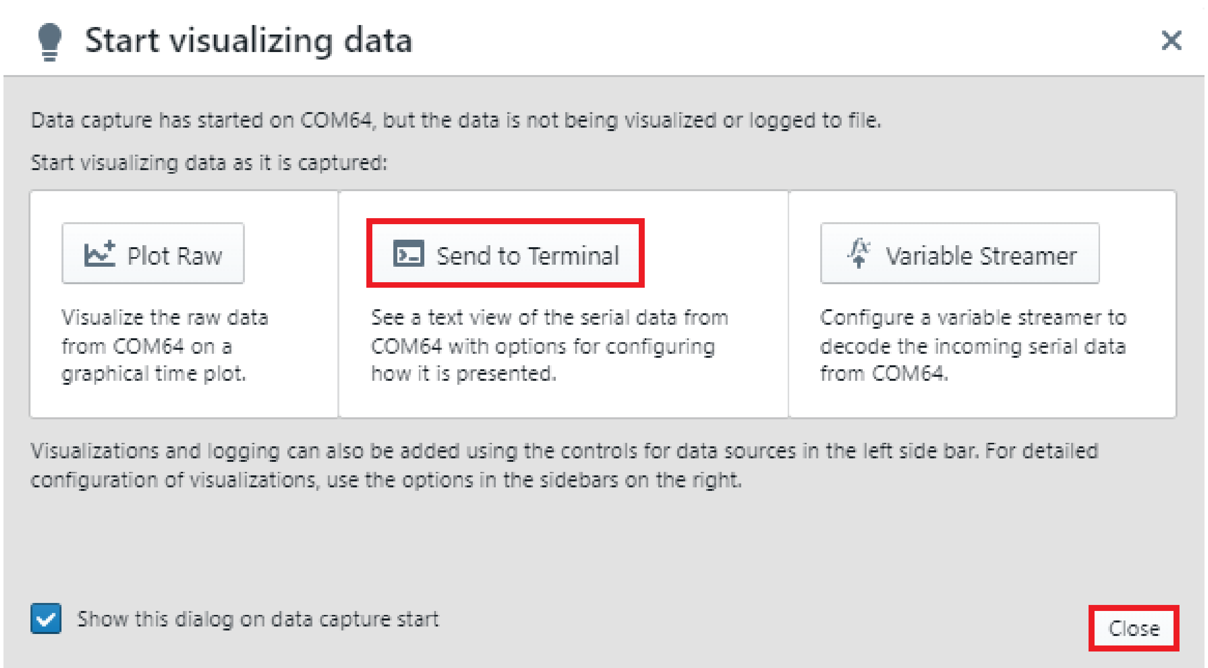
4. Open the Serial Port of the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit by clicking the *Play* icon as shown below.

Figure 6-4. Opening the Serial COM Port



5. Click **Send to Terminal** to view the serial console message, and then click **Close.**

Figure 6-5. Select the Terminal Option



6. Open the command prompt and navigate to the following location: *C:\Program Files\Microchip\MPLABX\v6.20\mplab_platform\mplab_ipe*.
   Note: The PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit has no Reset button to reset the MCU. To reboot the board, the `reset` command is sent to the nEDBG to reset the MCU with the help of MPLAB IPECMD.
7. Run the following command to reset the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit.

```
ipecmd.exe -P32CM5164LS00048 -TPNEDBG -OK
```

Figure 6-6. Resetting the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit

8. Observe the startup console message on the MPLAB Data Visualizer and the LED1 will toggle on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit.

Figure 6-7. Startup Console Message

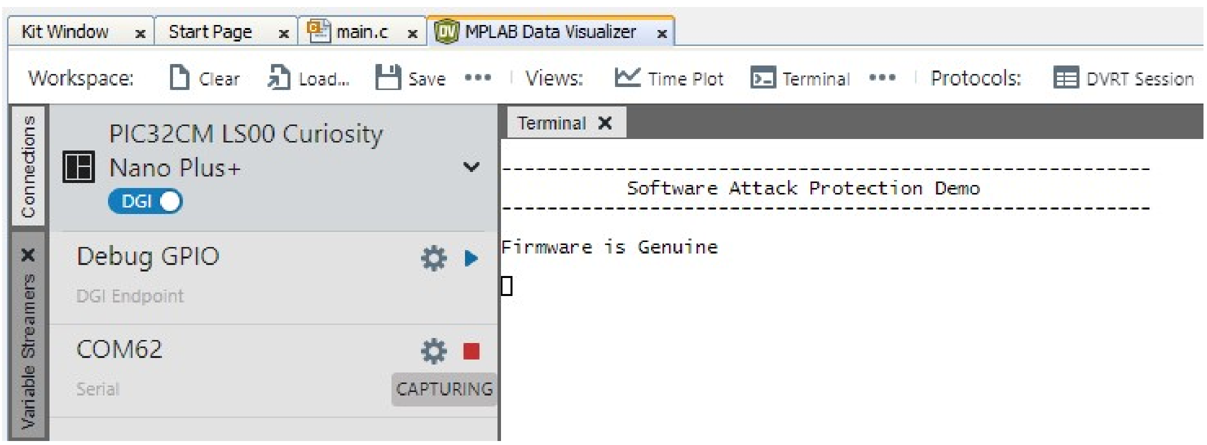
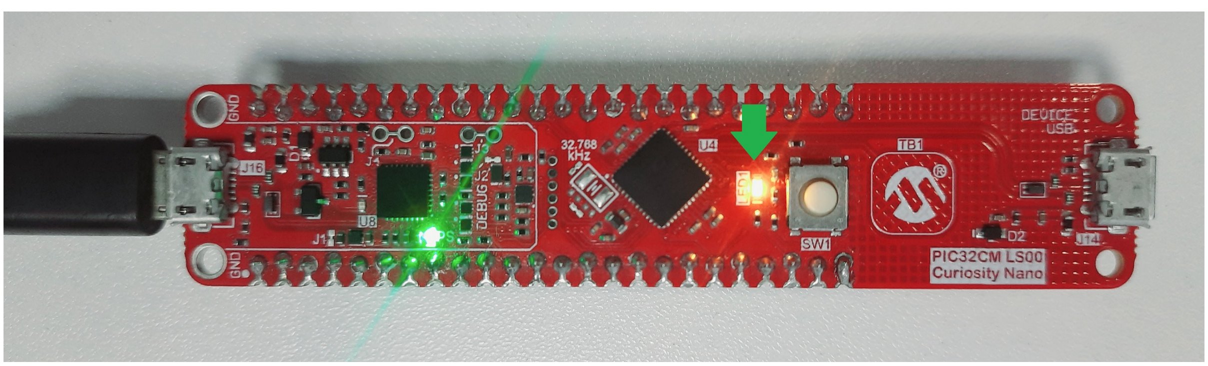
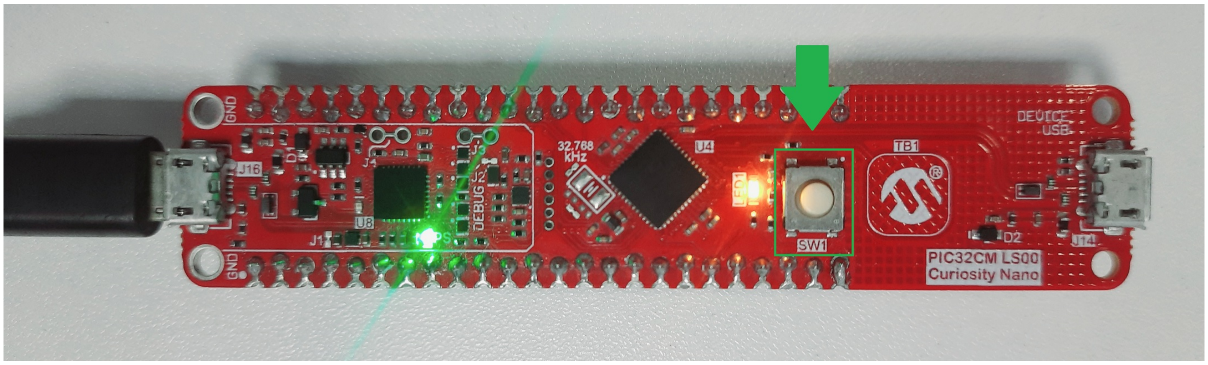


Figure 6-8. LED1 Toggling on PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit



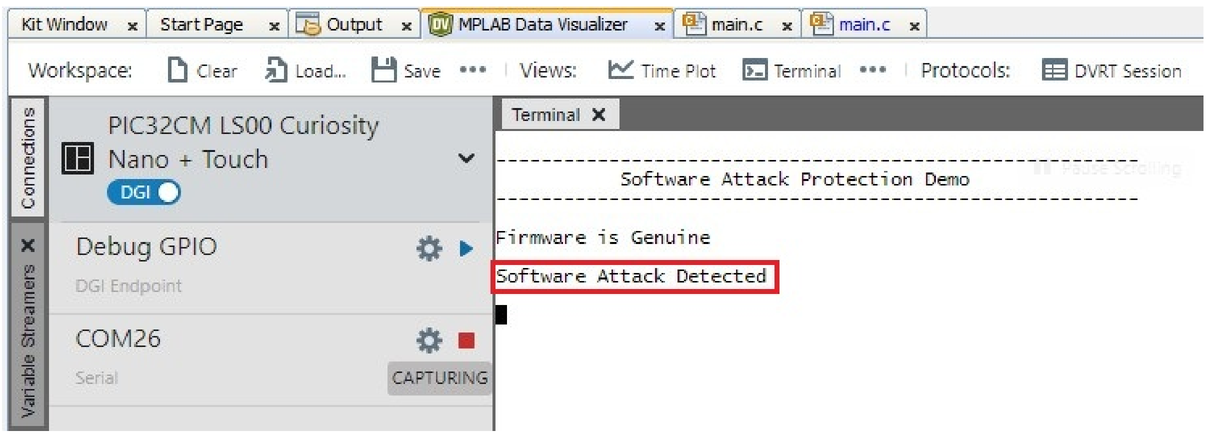
9. Press the SW1 button on the PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit to simulate a software attack.

Figure 6-9. Simulation of Software Attack



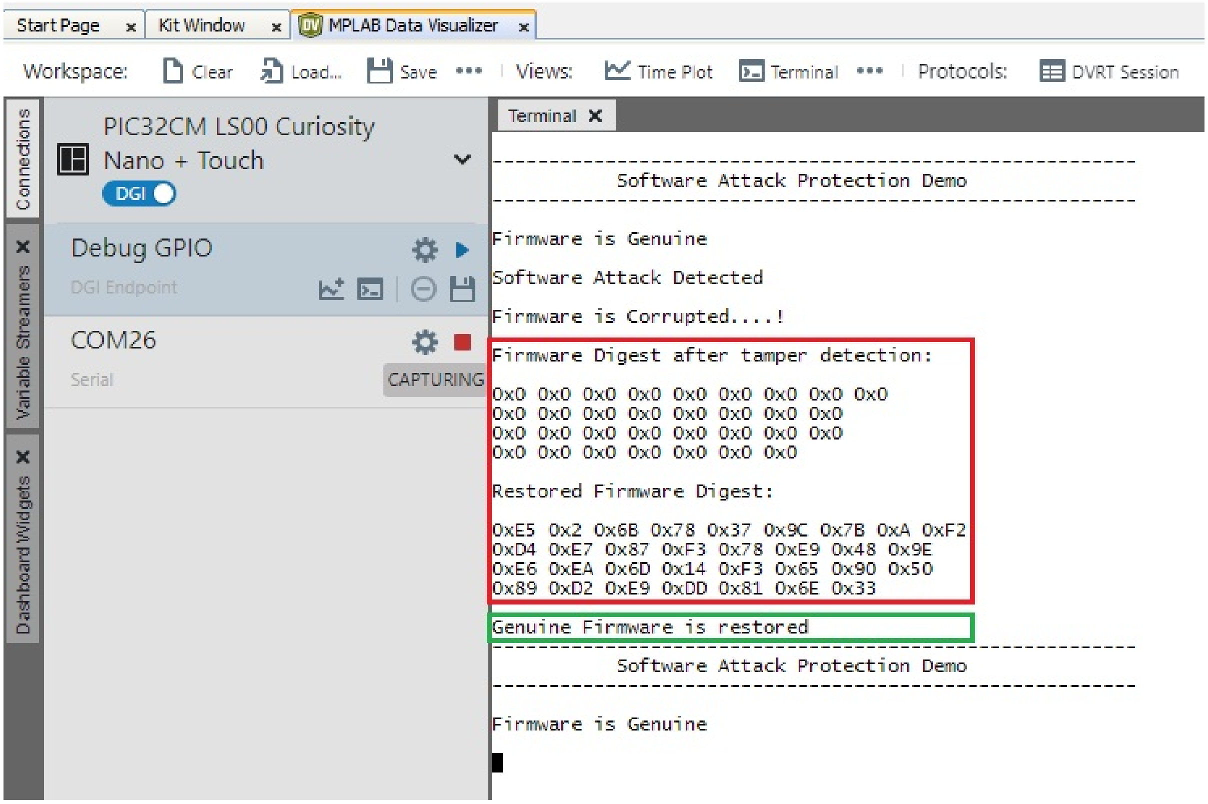
10. Observe the console message of software attack initiation on the MPLAB Data Visualizer.

Figure 6-10. Software Attack Initiation



11. Once the RTC timeout is reached, Secure Application starts the verification of Non-Secure firmware. If the verification fails, the genuine copy of the Non-Secure firmware will be programmed on the Non-Secure Flash region.

Figure 6-11. Non-Secure Firmware Verification



Note: After successfully programming a genuine copy, the secure application initiates a software reset.

7. Resources

- The following documents are available for download from the Microchip web site (www.microchip.com):
  - PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit User Guide (*DS70005567*)
  - PIC32CM LS00/LS60 Security Reference Guide (*DS00003992*)
- PIC32CM LS00 Curiosity Nano+ Touch Evaluation Kit
- Secure Boot on PIC32CM LS60 Curiosity Pro Evaluation Kit Using MPLAB® Harmony v3 Software Framework
- For additional information on MPLAB® Harmony v3, refer to the Microchip web site: developerhelp.microchip.com/xwiki/bin/view/software-tools/harmony/
- For more information on various applications, refer to: https://github.com/Microchip-MPLAB-Harmony/reference_apps
- For additional info about 32-bit Microcontroller Collaterals and Solutions, refer to 32-bit Microcontroller Collateral and Solutions Reference Guide (*DS70005534*)

8. Revision History

Revision A - April 2025

This is the initial release of this document.

Microchip Information

Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legal-information/microchip-trademarks.

ISBN: 979-8-3371-1010-3

Legal Notice

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip products are strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.

Product Page Links

PIC32CM2532LS00048, PIC32CM2532LS00064, PIC32CM2532LS00100, PIC32CM2532LS60048, PIC32CM2532LS60064, PIC32CM2532LS60100, PIC32CM5164LS00048, PIC32CM5164LS00064, PIC32CM5164LS00100, PIC32CM5164LS60048, PIC32CM5164LS60064, PIC32CM5164LS60100