M2351 Mask ROM Library

Application Note for 32-bit NuMicro® Family

Document Information

Abstract	The M2351 Mask ROM library (MKROMLib) is a software library pre-written in the M2351 Mask ROM. This document is used to introduce the usage of MKROMLib and the benefits of MKROMLib for a software developer.
Apply to	NuMicro® M2351 series

Table of Contents

1 Overview

The “MKROMLib” is a software library which is resident in M2351 Mask ROM. It provides a set of built-in functions that can be utilized to enable Flash, Crypto and firmware download functions by application programs.

The content of MKROMLib is tamper-proof due to the Read-Only feature of Mask ROM. Besides, the memory attribute of Mask ROM is configured as Execute-Only to prevent a thief from reading the library or tracing the instruction execution. Chapter 2 will briefly introduce the functions and usage of the MKROMLib. Chapter 3, 4 and 5 will introduce the benefits and feature of each library in detail, then demonstrate how to use the libraries with the sample code and list the library functions.

2 Mask ROM Library

MKROMLib consists of three parts for different applications that including Non-secure Callable library, Crypto library and SecureISP library.

2.1 Functional Description

The whole functional view in MKROMLib is shown in Figure 2-1 and briefly described as follows.

- Non-secure callable library is used for non-secure code developer to access the non-secure Flash and necessary chip information and status for non-secure code
- Secure Crypto library is used for secure code developer to call the crypto related APIs for their crypto application
- SecureISP library is used for secure code developer to create a secure encrypted channel with SecureISP Tool

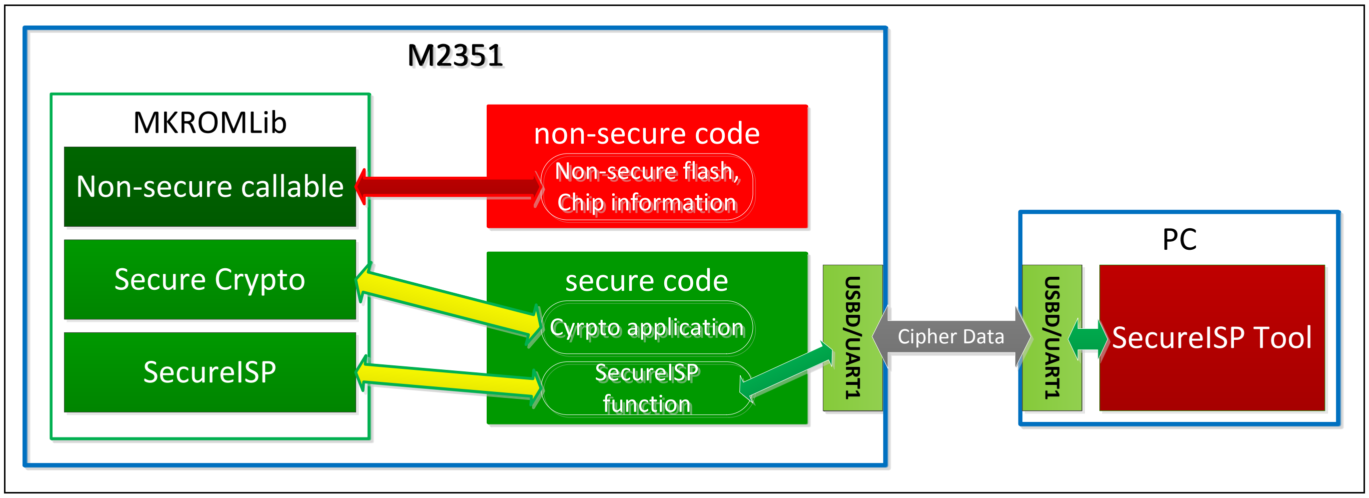


Figure 2-1 Functional View in MKROMLib

2.2 Usage

To call the APIs in MKROMLib, a software developer must add the MKROMLib library file to the project.

The MKROMLib library file could be found at “\Library\StdDriver\src” in M2351. The API header file of MKROMLib has been included in BSP header, NuMicro.h.

In different develop environments, the related MKROMLib file is necessary as listed below,

- In KEIL project, the library file name is MKROMLib_Keil.lib.
  For example:

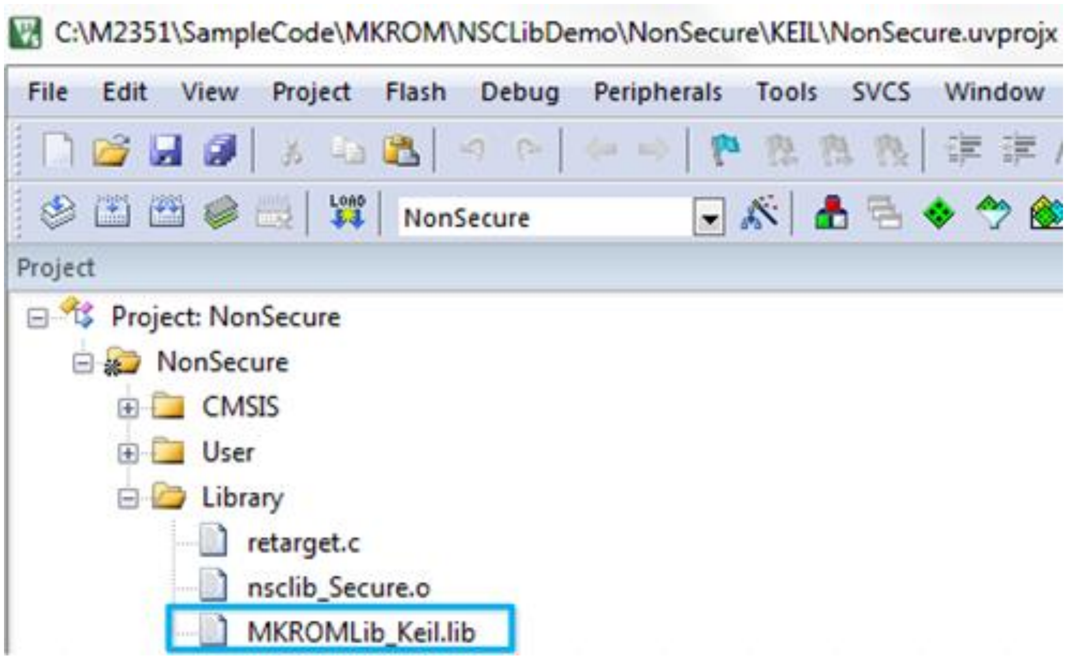

C:\M2351\SampleCode\MKROM\NSCLibDemo\NonSecure\KEIL\NonSecure.uvprojx
File Edit View Project Flash Debug Peripherals Tools SVCS Window
NonSecure
Project
Project: NonSecure
NonSecure
CMSIS
User
Library
retarget.c
nsclib_Secure.o
MKROMLib_Keil.lib

Figure 2-2 Add MKROMLib_Keil.lib in KEIL Project

- In IAR project, the library file name is MKROMLib_IAR.a.
  For example:

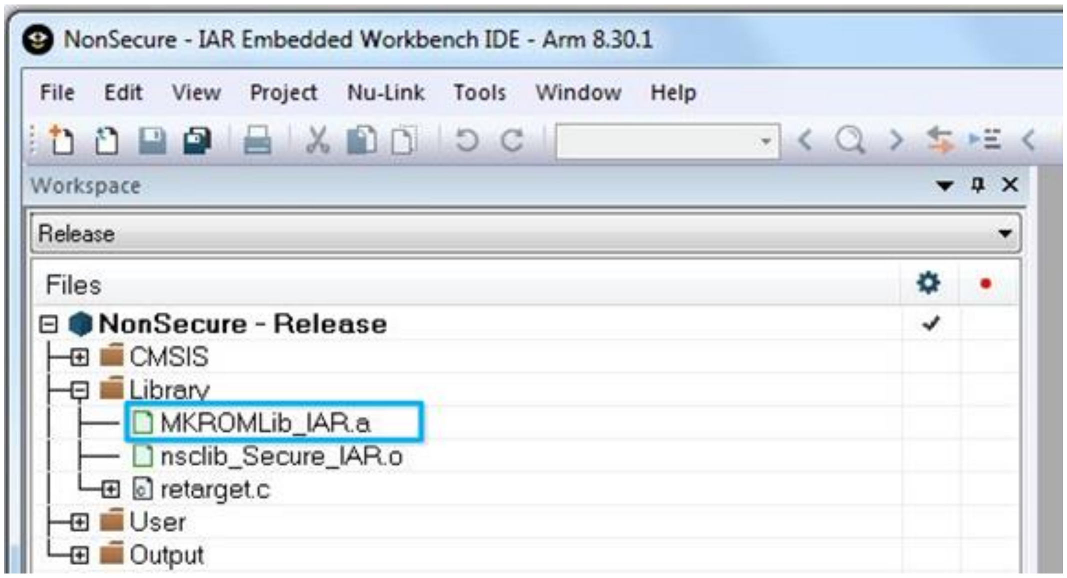


Figure 2-3 Add MKROMLib_IAR.a in IAR Project

- In GCC project, the library file name is libmkrom.a.
  For example:

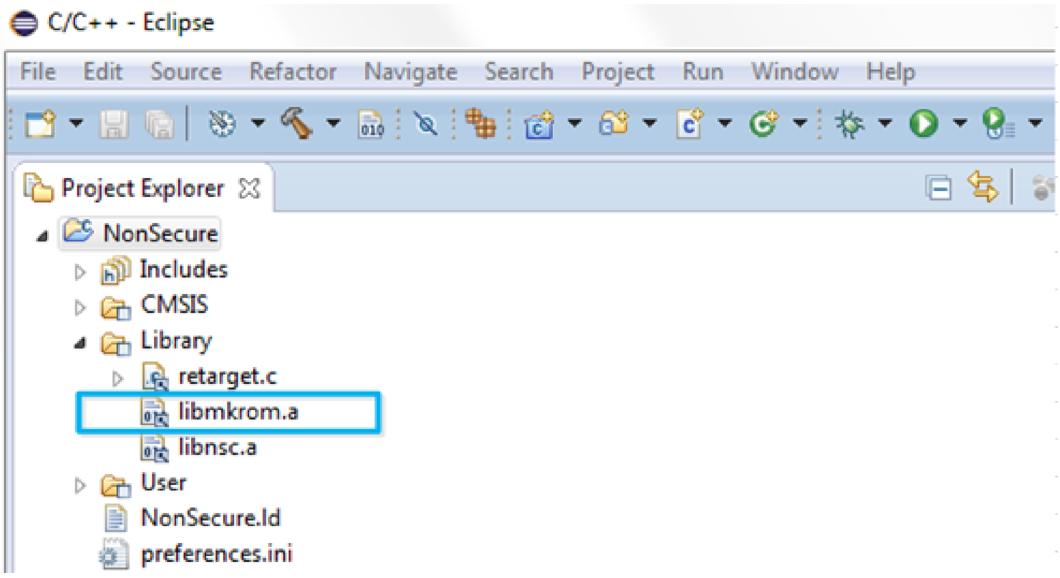


Figure 2-4 Add libmkrom.a in GCC Project

3 Non-secure Callable Library

The M2351 MKROM non-secure callable library is a suite of programming code, that is used to access the non-secure Flash, configure XOM setting for XOM region is in non-secure Flash region, perform KPROM related functions except enabling KPROM and get chip ID information.

Since the FMC controller in M2351 is a secure controller, the non-secure code cannot use the FMC controller to implement the FMC related applications. Therefore, the non-secure code developer can call MKROM non-secure callable API to achieve the FMC related applications supported in MKROMLib.

It is unnecessary for secure code developer to provide the additional functions (code) to achieve such functions for non-secure code developer and the secure code developer can also use the library to access non-secure Flash directly.

Figure 3-1 shows how the MKROM non-secure callable library works when the secure and non-secure code call non-secure API.

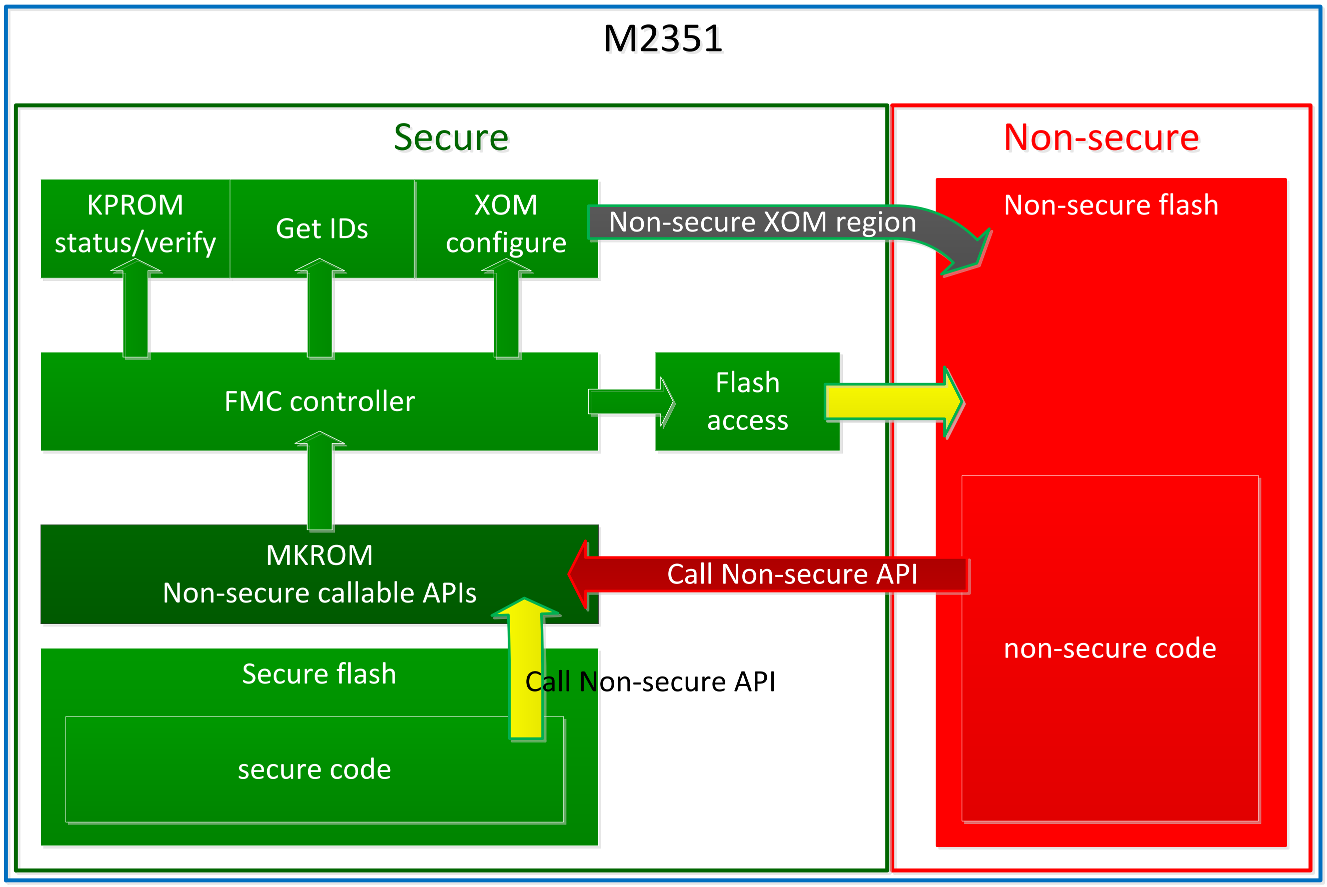


Figure 3-1 Access MKROM Non-Secure Library in Secure and Non-secure Code

3.1 Features

- Both secure code and non-secure code developer can use this library.
- Only non-secure Flash region can be accessed.
- Supports erase, write, read Flash and set the XOM region within non-secure Flash
- Supports get necessary information for non-secure region, including CID, DID, PID, UID, UCID, non-secure boundary address and KPROM status
- Supports for KEIL, IAR and GCC development tool

3.2 Basic Configuration

Secure and non-secure code developers must add the MKROMLib library file in the project to perform MKROM non-secure callable API

  - In KEIL project, the library file name is MKROMLib_Keil.lib
  - In IAR project, the library file name is MKROMLib_IAR.a.
  - In GCC project, the library file name is libmkrom.a.

3.3 NSCLibDemo Sample Code

This section describes the usage of MKROM non-secure callable API in secure and non-secure code by using KEIL MDK projects.

All the sample code could be found at “\SampleCode\MKROM\NSCLibDemo” in M2351 BSP, including “Secure” folder for secure code and “NonSecure” folder for non-secure code.

The security attributes (TrustZone® Configuration) for secure code and non-secure code must be configured for this demo. For the detailed usage of configuring TrustZone®, please refer to the related TrustZone® application note. The following section describes the basic system configuration only.

3.3.1 System Configuration

In secure code project, the partition_M2351.h file is used to configure the secure/non-secure Flash region and others secure/non-secure attributes for the further demo program.

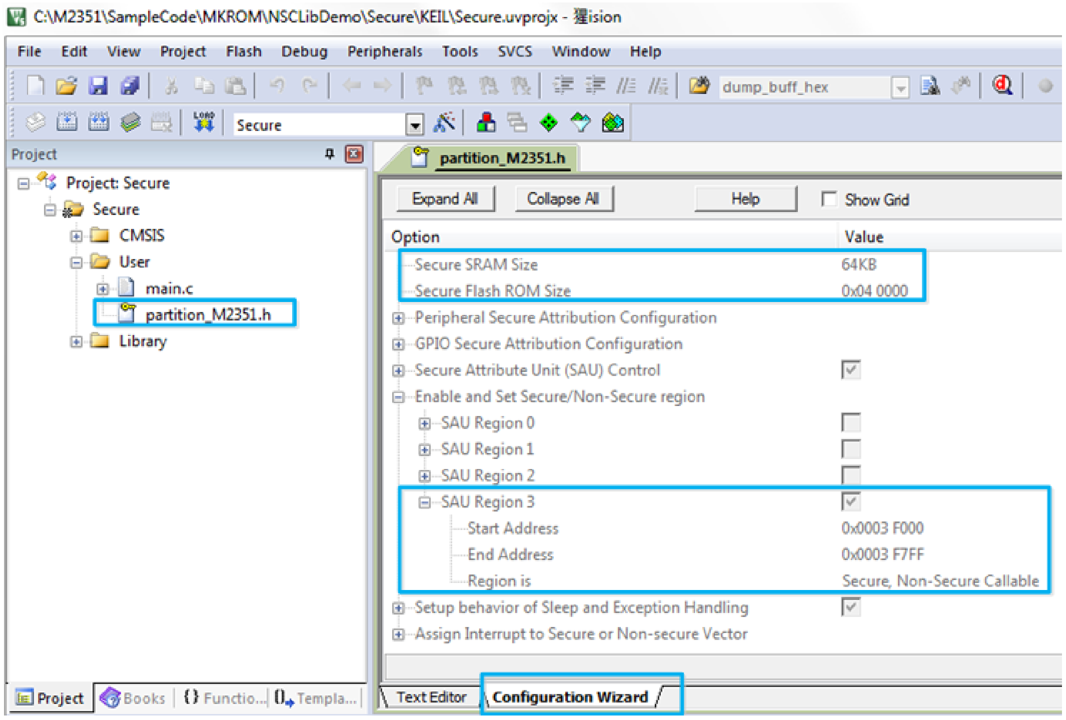
The detailed “Configuration Wizard” list in partition_M2351.h is as follows,

- Secure SRAM Size, select 64KB
- Secure Flash ROM Size, input 0x40000
- Peripheral Secure Attribute Configuration, select “UART” UART0 as Non-secure
- Enable and Set Secure/Non-secure Region, configure “SAU Region 3” as bellows
  - Start Address: 0x3F000
  - End Address: 0x3F7FF

  - Region is: Secure, Non-Secure Callable
- Configure MKROM Non-secure callable function region

The default MKROM non-secure callable function region and attribute are configured in "SAU Region 5" as bellows list, and developer is unnecessary to configure it again.

  - Start Address: 0x807E00
  - End Address: 0x807FFF
  - Region is: Secure, Non-Secure Callable
- "Configuration Wizard" example:


C:\M2351\SampleCode\MKROM\NSCLibDemo\Secure\KEIL\Secure.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
dump_buff_hex
Secure
Project
Project: Secure
Secure
CMSIS
User
main.c
partition_M2351.h
Library
Project Books Functio... Templa...
partition_M2351.h
Expand All
Collapse All
Help
Show Grid
Text Editor
Configuration Wizard

Option	Value
Secure SRAM Size	64KB
Secure Flash ROM Size	0x04 0000
Peripheral Secure Attribution Configuration	
GPIO Secure Attribution Configuration	
Secure Attribute Unit (SAU) Control	✓
Enable and Set Secure/Non-Secure region	
SAU Region 0	☐
SAU Region 1	☐
SAU Region 2	☐
SAU Region 3	✓
Start Address	0x0003 F000
End Address	0x0003 F7FF
Region is	Secure, Non-Secure Callable
Setup behavior of Sleep and Exception Handling	✓
Assign Interrupt to Secure or Non-secure Vector	

Figure 3-2 Example of partition_M2351.h

- Figure 3-3 shows the memory map of NSCLibDemo code.

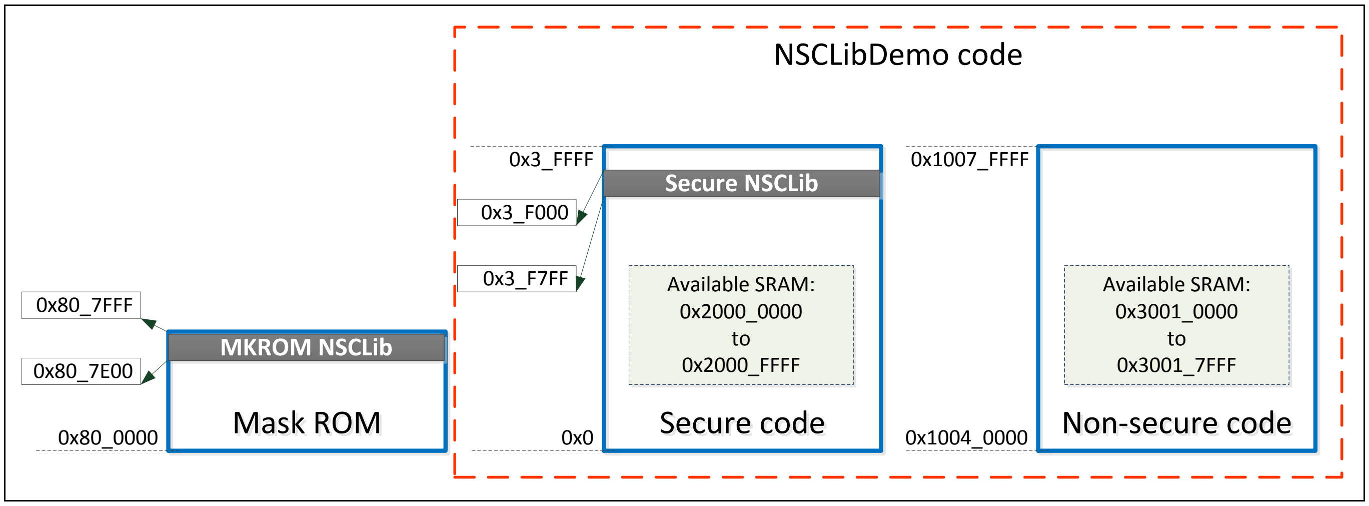


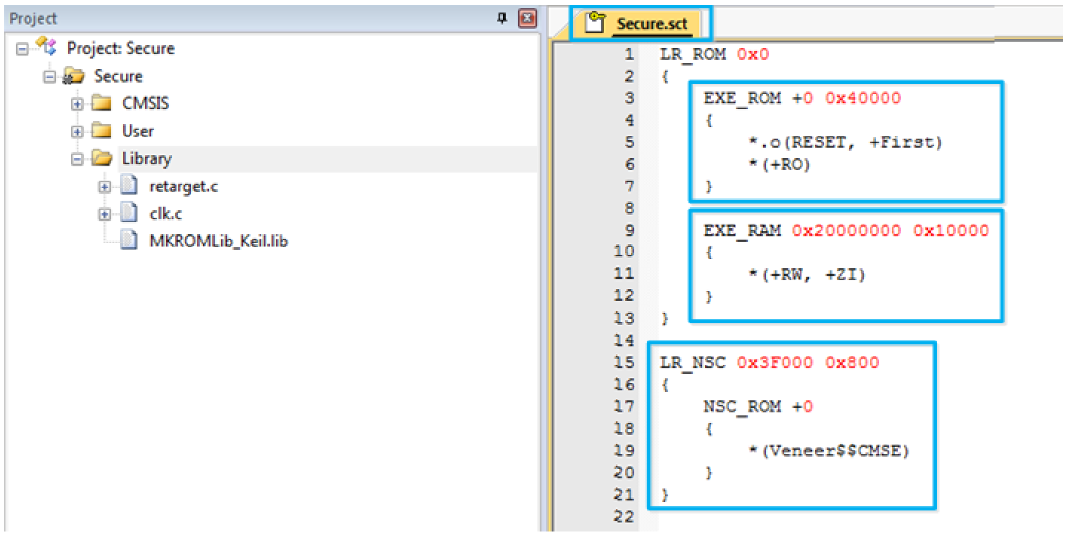
Figure 3-3 Memory Map of NSCLibDemo

3.3.2 Secure Code

This section describes the basic settings and execution result of the secure code.

- Configure executable Flash and SRAM region

  Open Secure.sct file at "\SampleCode\MKROM\NSCLibDemo\Secure\Keil" folder to configure the executable Flash region from 0x0 to 0x3FFFF, SRAM region from 0x20000000 to 0x2000FFFF and secure code non-secure callable function region from 0x3F000 to 0x3F7FF.

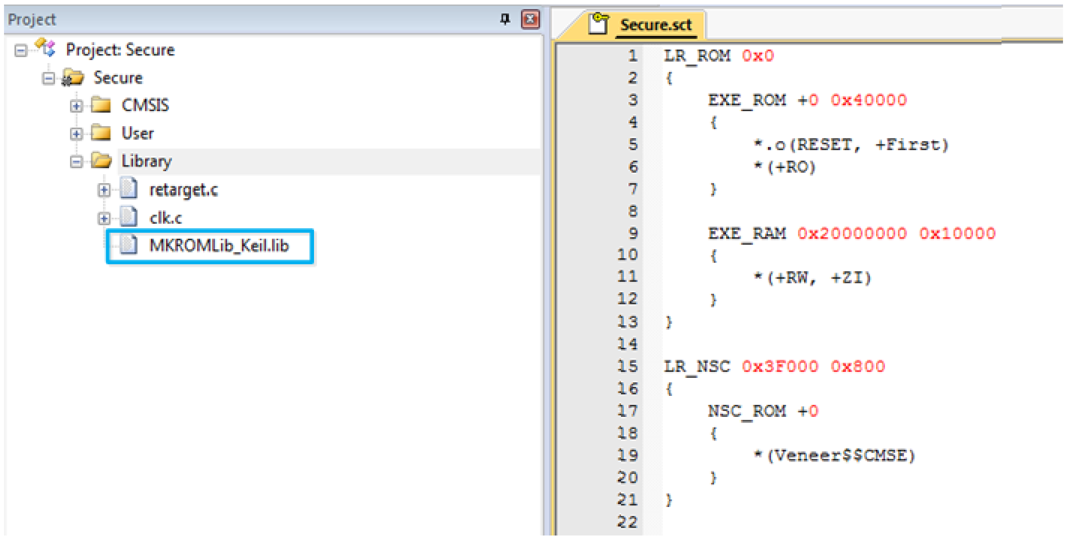


```
LR_ROM 0x0
{
    EXE_ROM +0 0x40000
    {
        *.o(RESET, +First)
        *(+RO)
    }

    EXE_RAM 0x20000000 0x10000
    {
        *(+RW, +ZI)
    }
}

LR_NSC 0x3F000 0x800
{
    NSC_ROM +0
    {
        *(Veneer$$CMSE)
    }
}
```

Figure 3-4 Configure Secure.sct in Secure Code

- Add MKROM library file

  Add MKROMKLib_Keil.lib to the secure code project for performing the MKROM non-secure callable API to get chip information and access non-secure Flash.

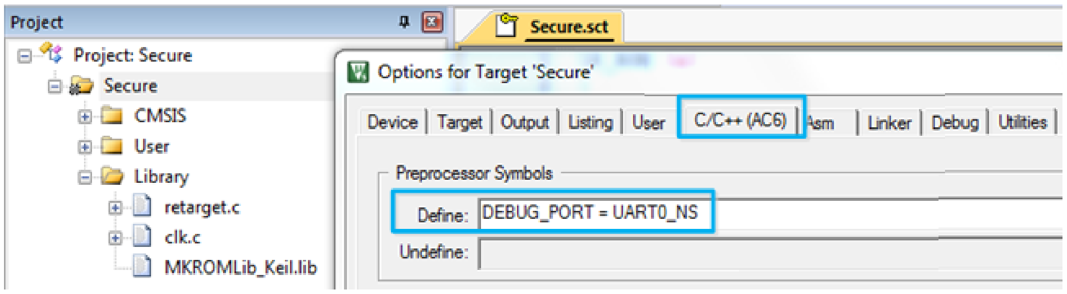


```
LR_ROM 0x0
{
    EXE_ROM +0 0x40000
    {
        *.o(RESET, +First)
        *(+RO)
    }

    EXE_RAM 0x20000000 0x10000
    {
        *(+RW, +ZI)
    }
}

LR_NSC 0x3F000 0x800
{
    NSC_ROM +0
    {
        *(Veneer$$CMSE)
    }
}
```

Figure 3-5 Add MKROMLib_Keil.lib in Secure Code

- Define UART debug port

  Since UART0 has been configured as non-secure peripheral in System Configuration. To display the debug message, the UART0 access pointer must be redefined as the non-secure access pointer UART0_NS in the project Options.

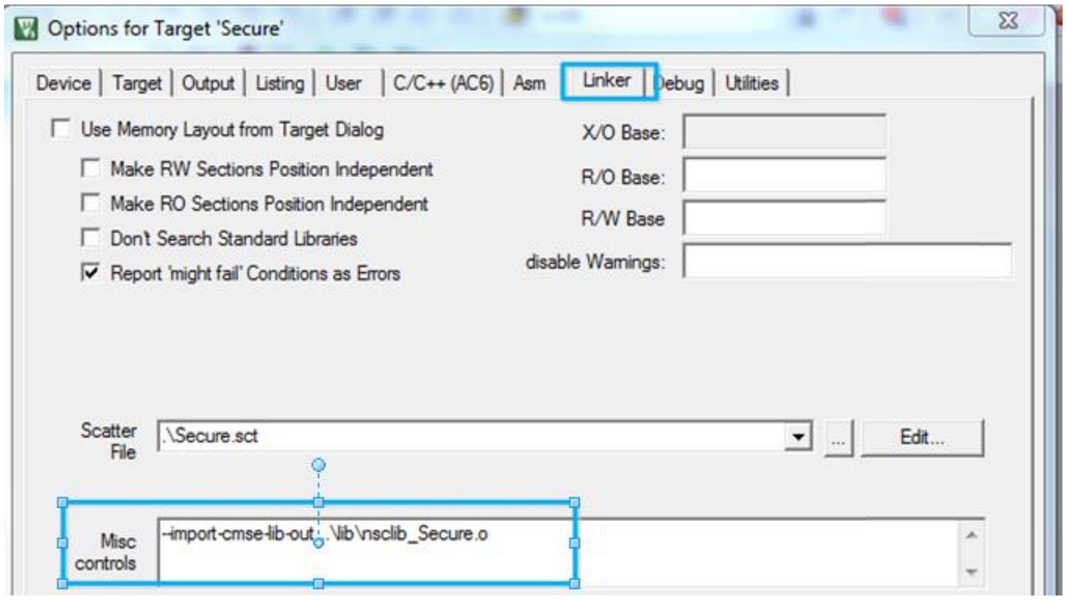


Define: DEBUG_PORT = UART0_NS

Figure 3-6 Define Non-secure UART0 Port

- Output nsclib_Secure.o

  Configure “Linker” setting to export the nsclib_Secure.o, and then non-secure code can use this object file to get the system core clock frequency.



- [ ] Use Memory Layout from Target Dialog
- [ ] Make RW Sections Position Independent
- [ ] Make RO Sections Position Independent
- [ ] Don't Search Standard Libraries
- [x] Report 'might fail' Conditions as Errors

X/O Base:

R/O Base:

R/W Base

disable Warnings:

Scatter File: .\Secure.sct

Misc controls: --import-cmse-lib-out .\lib\nsclib_Secure.o

Figure 3-7 Output the nsclib_Secure.o File

- Get chip information

  When the secure code is run, demo code will call MKROMLib API to read the non-secure Flash region setting and basic chip information.

```
/* The setting of Non-secure flash base address must be the same as SAU setting. */
printf("Maximum APROM flash size = %d bytes.\n", BL_EnableFMC());
printf("Non-secure flash base = %d (0x%08x).\n", BL_GetNSBoundary(), SCU->FNSADDR);
if(SCU->FNSADDR != FMC_SECURE_ROM_SIZE)
{
    printf("Set Non-secure base address fail!\n");
    while(1) {}
}
printf("\n");


/* Show basic chip information */
printf("Basic chip info:\n");
printf("PID:    0x%08x\n", BL_ReadPID());
printf("UID-0:  0x%08x\n", BL_ReadUID(0));
printf("UID-1:  0x%08x\n", BL_ReadUID(1));
printf("UID-2:  0x%08x\n", BL_ReadUID(2));
printf("UCID-0: 0x%08x\n", BL_ReadUCID(0));
printf("UCID-1: 0x%08x\n", BL_ReadUCID(1));
printf("UCID-2: 0x%08x\n", BL_ReadUCID(2));
printf("UCID-3: 0x%08x\n", BL_ReadUCID(3));
```

- Verify non-secure Flash

  Call MKROMLib API to erase, program and verify non-secure Flash.

```
    /* Prepare Non-secure flash data */
    u32Addr = 0x10070000UL;
    printf("Page erase [0x%08x] ... ", u32Addr);
    if(BL_FlashPageErase(u32Addr))
    {
        printf("Fail!\n");
        while(1) {}
    }
    else
    {
        printf("Ok.\nProgram and verify data on [0x%08x] ... ", u32Addr);
        BL_FlashWrite(u32Addr, 0x12345678UL);
        if(BL_FlashRead(u32Addr) == 0x12345678UL)
        {
            printf("Pass.\n");
        }
        else
        {
            printf("Fail!\n");
            while(1) {}
        }
    }
```

- Jump to non-secure code

  After verifying the non-secure Flash data, secure code can call Nonsecure_Init() to jump to non-secure code to demonstrate the MKROM non-secure callable API in non-secure code.

```
    /* Jump to perform MKROM Non-secure API in Non-secure region */
    printf("Hit any key, then jump to perform MKROM Non-secure API in Non-secure
region.\n\n");
    getchar();

    Nonsecure_Init(); /* Jump to Non-secure code */
```

The code below shows the Nonsecure_Init() function.

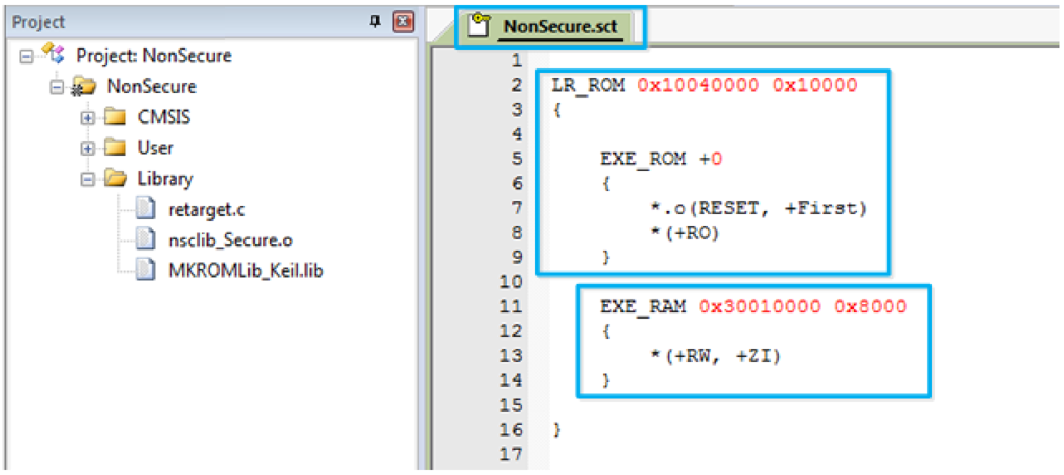
```
void Nonsecure_Init(void)
{
    NonSecure_funcptr fp;
    /* SCB_NS.VTOR points to the Non-secure vector table base address. */
    SCB_NS->VTOR = NEXT_BOOT_BASE;

    /* 1st Entry in the vector table is the Non-secure Main Stack Pointer. */
    __TZ_set_MSP_NS(*((uint32_t *)SCB_NS->VTOR));      /* Set up MSP in Non-secure code */

    /* 2nd entry contains the address of the Reset_Handler (CMSIS-CORE) function */
    fp = ((NonSecure_funcptr)(*(((uint32_t *)SCB_NS->VTOR) + 1)));

    /* Clear the LSB of the function address to indicate the function-call
       will cause a state switch from Secure to Non-secure */
    fp = cmse_nsfptr_create(fp);

    /* Check if the Reset_Handler address is in Non-secure space */
    if(cmse_is_nsfptr(fp) && (((uint32_t)fp & 0xf0000000) == 0x10000000))
    {
        printf("Execute Non-secure code ...\n");
        fp(0); /* Non-secure function call */
    }
    else
    {
        /* Something went wrong */
        printf("No code in Non-secure region!\n");
        printf("CPU will halted at Non-secure state\n");

        /* Set nonsecure MSP in nonsecure region */
        __TZ_set_MSP_NS(NON_SECURE_SRAM_BASE+512);

        /* Try to halted in Non-secure state (SRAM) */
        M32(NON_SECURE_SRAM_BASE) = JUMP_HERE;
        fp = (NonSecure_funcptr)(NON_SECURE_SRAM_BASE+1);
        fp(0);

        while(1) {}
    }
}
```

3.3.3 Non-secure Code

This section describes the basic settings and execution result of non-secure code.

- Configure executable Flash and SRAM region

  Open NonSecure.sct file at "\SampleCode\MKROM\NSCLibDemo\NonSecure\Keil" folder to configure the executable Flash region from 0x10040000 to 0x1004FFFF and SRAM region from 0x30010000 to 0x30017FFF.



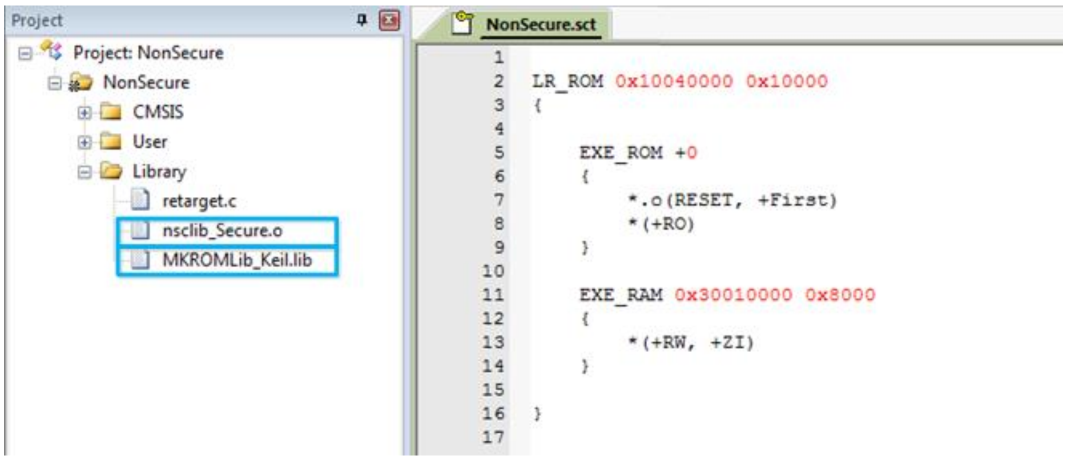
```

LR_ROM 0x10040000 0x10000
{

    EXE_ROM +0
    {
        *.o(RESET, +First)
        *(+RO)
    }

    EXE_RAM 0x30010000 0x8000
    {
        *(+RW, +ZI)
    }

}

```

Figure 3-8 Configure NonSecure.sct in Non-secure Code

- Add library files

  Add MKROMKLib_Keil.lib to the non-secure code project for performing the MKROM non-secure callable API.

  Add the nsclib_Secure.o that provided by the secure code to get the system core clock frequency by calling GetSystemCoreClock() API.

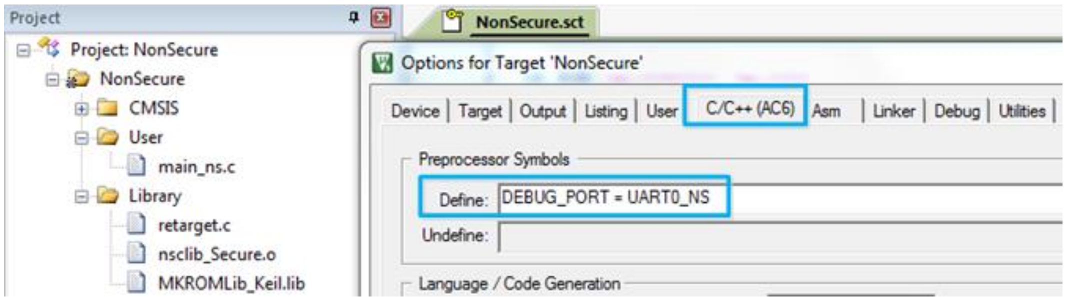


```

LR_ROM 0x10040000 0x10000
{

    EXE_ROM +0
    {
        *.o(RESET, +First)
        *(+RO)
    }

    EXE_RAM 0x30010000 0x8000
    {
        *(+RW, +ZI)
    }

}

```

Figure 3-9 Add nsclib_Secure.o and MKROMLib_Keil.lib in Non-secure Code

- Define UART debug port

  Since UART0 has been configured as non-secure peripheral in System Configuration. To display the debug message, the UART0 access pointer must be redefined as the non-secure access pointer UART0_NS in the project Options.



Define: DEBUG_PORT = UART0_NS
Undefine:

Figure 3-10 Define Non-secure UART0 port

- Get chip information

  When the CPU is run in non-secure region, demo code will perform MKROMLib API to read the basic chip information.

```
    /* Show basic chip information */
    printf("[Basic chip info]:\n");
    printf("Maximum APROM flash size = %d bytes.\n", BL_EnableFMC());
    printf("Non-secure flash base = %d (0x%08x).\n", BL_GetNSBoundary(),
BL_GetNSBoundary());
    printf("PID:    0x%08x\n", BL_ReadPID());
    printf("UID-0:  0x%08x\n", BL_ReadUID(0));
    printf("UID-1:  0x%08x\n", BL_ReadUID(1));
    printf("UID-2:  0x%08x\n", BL_ReadUID(2));
    printf("UCID-0: 0x%08x\n", BL_ReadUCID(0));
    printf("UCID-1: 0x%08x\n", BL_ReadUCID(1));
    printf("UCID-2: 0x%08x\n", BL_ReadUCID(2));
    printf("UCID-3: 0x%08x\n", BL_ReadUCID(3));
```

- Verify default non-secure Flash

  Call BL_FlashRead to obtain the non-secure Flash data at address 0x70000 and verify it.

```
    /* Check data on Non-secure 0x7_0000 */
    /* Only support Non-secure flash access */
    u32Addr = 0x10070000UL;



    /* BL_FlashRead API */
    u32Data = BL_FlashRead(u32Addr);
    printf("Check Non-secure 0x%08x data is 0x%x ... ", u32Addr, u32Data);
    if(u32Data == 0x12345678UL)
    {
        printf("Pass.\n");
    }
    else
    {
        printf("Fail!\n");
        while(1) {}
    }
```

- Program and verify non-secure Flash

  After verifying the default non-secure Flash data, demo code will use the MKROMLib APIs: BL_FlashPageErase, BL_CheckFlashAllOne, BL_FlashMultiWrite, BL_FlashWrite, BL_FlashMultiRead and BL_FlashRead to demonstrate accessing the non-secure Flash.

```
    /* Page erase then check page data should be all one
         by BL_FlashPageErase and BL_CheckFlashAllOne */
    printf("Page Erase [0x%08x] ... ", u32Addr);
    if(BL_FlashPageErase(u32Addr) == 0)
    {
        printf("Ok.\n");
    }
    else
    {
        printf("Fail!\n");
        while(1) {}
    }
    printf("Check [0x%08x] one page are all one ... ", u32Addr);
    u32Status = BL_CheckFlashAllOne(u32Addr, FMC_FLASH_PAGE_SIZE);
    if(u32Status == BL_FLASH_ALLONE)
    {
        printf("Pass.\n");
    }
    else
    {
        printf("Fail! (0x%08x)\n", u32Status);
        while(1) {}
    }


    /* Program/Read Non-secure flash by single read/write and multi-read/write API */
    printf("\n*** Perform flash Read/Write/Multi-Read/Multi-Write API ***\n");
    for(i = 0; i < (FMC_FLASH_PAGE_SIZE / 4); i++)
    {
        g_au32FlashData[i] = 0xA5A50000UL + i;
    }
    /* BL_FlashMultiWrite first 1024 bytes */
    printf("Multi-Write [0x%08x] 1024 bytes ... ", u32Addr);
    if(BL_FlashMultiWrite(u32Addr, (uint32_t *)g_au32FlashData, (FMC_FLASH_PAGE_SIZE / 2))
== 0)
    {
        printf("Ok.\n");
    }
    else
    {
        printf("Fail! (0x%08x)\n", u32Status);
        while(1) {}
    }
    /* BL_FlashWrite last 1024 bytes */
    printf("Single Write [0x%08x] 1024 bytes ... ", (uint32_t)(u32Addr +
(FMC_FLASH_PAGE_SIZE / 2)));
    for(i = 0; i < ((FMC_FLASH_PAGE_SIZE / 4) / 2); i++)
    {
        if(BL_FlashWrite(u32Addr + (FMC_FLASH_PAGE_SIZE / 2) + (i * 4),
g_au32FlashData[((FMC_FLASH_PAGE_SIZE / 4) / 2) + i]) != 0)
        {
            printf("Fail! (0x%0x: 0x%08x)\n\n", (uint32_t)(u32Addr + (FMC_FLASH_PAGE_SIZE
/ 2) + (i * 4)), (uint32_t)(g_au32FlashData[((FMC_FLASH_PAGE_SIZE / 4) / 2) + i]));
            while(1) {}
        }
    }
    printf("Ok.\n");


    /* BL_FlashMultiRead one page data */
    printf("Multi-Read [0x%08x] one page ... ", u32Addr);
    if(BL_FlashMultiRead(u32Addr, (uint32_t *)g_au32GetData, FMC_FLASH_PAGE_SIZE) == 0)
    {
        printf("Ok.\n");
    }
    else
    {
        printf("Fail! (0x%08x)\n", u32Status);
        while(1) {}
    }
    /* BL_FlashRead one page data and compare data */
    printf("Single read then compare all data ... ");
    for(i = 0; i < (FMC_FLASH_PAGE_SIZE / 4); i++)
    {
        uint32_t data_tmp;
        if(BL_FlashRead(u32Addr + (i * 4)) != (0xA5A50000UL + i))
        {
            printf("Fail! (0x%08x: 0x%08x)\n", (u32Addr + (i * 4)), BL_FlashRead(u32Addr + 
(i * 4)));
            while(1) {}
        }
        data_tmp = g_au32FlashData[i];
        if(data_tmp != g_au32GetData[i])
        {
            printf("Fail! (0x%08x. W:0x%08x, R:0x%08x)\n", (u32Addr + (i * 4)), data_tmp, 
g_au32GetData[i]);
            while(1) {}
        }
    }
    printf("Pass.\n");
```

3.4 Non-secure Callable API List

The following section describes the MKROM non-secure callable API in detail. For the MKROMLib header file, refer to “M2351BSP\Library\StdDriver\inc\mkromlib.h”.

3.4.1 BL_CheckFlashAllOne (For Non-secure Region)

- uint32_t BL_CheckFlashAllOne (uint32_t u32NSAddr, uint32_t u32ByteCount)
- Parameters:

- [in] u32NSAddr	Non-secure Flash region to be calculated. u32NSAddr must be a page size aligned address.
- [in] u32ByteCount	Byte counts of non-secure Flash area to be calculated. It must be multiple of 2048 bytes.

- Returns:
  - 0xF0F00000: u32NSAddr region isn't in non-secure area
  - 0xF0F00001: Wrong u32NSAddr or u32ByteCount parameter

- ■ 0xF0F00003**:** u32NSAddr isn't valid Flash address
- ■ -1**:** Execute Check Flash All One operation failed
- ■ 0xA11FFFFF**:** The contents of verified non-secure Flash area are 0xFFFFFFFF
- ■ 0xA1100000**:** Some contents of verified non-secure Flash area are not 0xFFFFFFFF

This API is used to check specified non-secure Flash area data are all 0xFFFFFFFF or not.

3.4.2 BL_DisableFMC

- void BL_DisableFMC (void)
- Parameter**:** None
- Return**:** None

This API will disable relative settings for disable FMC ISP function and lock register write-protect until last ISP operation is finished.

3.4.3 BL_EnableFMC

- uint32_t BL_EnableFMC (void)
- Parameter**:** None
- Return**:**
  - ■ Maximum APROM size

This API will unlock register write-protect, enable relative settings for access FMC ISP commands and return maximum APROM by chip package.

3.4.4 BL_EraseXOMRegion (For Non-secure Region)

- int32_t BL_EraseXOMRegion (uint32_t u32XOMBase)
- Parameter**:**

  - [in] u32XOMBase Specified XOM region to be erase
- Returns**:**
  - ■ 0xF0F00000**:** u32XOMBase or erase XOM region isn't in non-secure area
  - ■ 0xF0F00001**:** u32XOMBase isn't page size aligned
  - ■ 0xF0F00003**:** u32XOMBase isn't valid Flash address
  - ■ 0xF0F00008**:** Invalid u32XOMBase address
  - ■ -1**:** Erase XOM region failed
  - ■ 0**:** Erase XOM region success

This API will erase specified XOM region data and relative XOM setting.

3.4.5 BL_FlashChecksum (For Non-secure Region)

- uint32_t BL_FlashChecksum (uint32_t u32NSAddr, uint32_t u32ByteCount)
- Parameter**:**

- [in] u32NSAddr	Non-secure Flash region to be calculated. u32NSAddr must be a page size aligned address.
- [in] u32ByteCount	Byte counts of non-secure Flash area to be calculate, it must be multiple of 2048 bytes.

- Returns**:**
  - ■ 0xF0F00000**:** u32NSAddr region isn't in non-secure area
  - ■ 0xF0F00001**:** Wrong u32NSAddr or u32ByteCount parameter
  - ■ 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - ■ -1**:** Execute CRC32 operation failed
  - ■ Result of CRC32 checksum

This API will calculate the CRC32 checksum result of specified non-secure Flash area. The starting address and calculated size must be all 2048 bytes page size aligned.

3.4.6 BL_FlashMultiRead (For Non-secure Region)

- int32_t BL_FlashMultiRead (uint32_t u32NSAddr,

```
                             uint32_t *pu32NSRamBuf,
                             uint32_t u32Size)
```

- Parameter**:**

- [in] u32NSAddr	Starting non-secure Flash address
- [out] pu32NSRamBuf	Non-secure SRAM buffer address to store reading data
- [in] u32Size	Total read byte counts, it should be word aligned and maximum size is one page size

- Returns**:**
  - ■ 0xF0F00000**:** u32NSAddr or pu32NSRamBuf region isn't in non-secure area
  - ■ 0xF0F00001**:** Wrong u32NSAddr，pu32NSRamBuf or u32Size parameter
  - ■ 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - ■ 0xF0F00004**:** pu32NSRamBuf isn't valid SRAM address
  - ■ -1**:** Multi-words read failed
  - ■ 0**:** Read operation is success

This API is used to read multi-words data start from specified non-secure Flash address. The maximum read size is one page size, 2048 bytes.

3.4.7 BL_FlashMultiWrite (For Non-secure Region)

- int32_t BL_FlashMultiWrite (uint32_t u32NSAddr,
  uint32_t *pu32NSRamBuf,
  uint32_t u32Size)
- Parameter**:**

- [in] u32NSAddr	Starting non-secure Flash address
- [in] pu32NSRamBuf	Non-secure SRAM buffer address to store program data
- [in] u32Size	Total program byte counts, it should be word aligned and maximum size is one page size

- Returns**:**
  - 0xF0F00000**:** u32NSAddr or pu32NSRamBuf region isn't in non-secure area
  - 0xF0F00001**:** Wrong u32NSAddr，pu32NSRamBuf or u32Size parameter
  - 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - 0xF0F00004**:** pu32NSRamBuf isn't valid SRAM address
  - -1**:** Multi-words write failed
  - 0**:** Program operation is finished

This API is used to program multi-words data start from specified non-secure Flash address. The maximum program size is one page size, 2048 bytes.

3.4.8 BL_FlashRead (For Non-secure Region)

- uint32_t BL_FlashRead (uint32_t u32NSAddr)
- Parameter**:**

- [in] u32NSAddr	Non-secure Flash address

- Returns**:**
  - 0xF0F00000**:** u32NSAddr isn't in non-secure area
  - 0xF0F00001**:** u32NSAddr isn't word aligned
  - 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - -1**:** Flash read failed
  - The data of specified non-secure address

This API is used to read word data from specified non-Secure Flash address.

3.4.9 BL_FlashPageErase (For Non-secure Region)

- int32_t BL_FlashPageErase (uint32_t u32NSAddr)
- Parameter**:**

- [in] u32NSAddr	Non-secure Flash region to be erased and must be a page size aligned address

- Returns**:**
  - 0xF0F00000**:** u32NSAddr region isn't in non-secure area
  - 0xF0F00001**:** u32NSAddr isn't page size aligned
  - 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - -1**:** Page erase failed
  - 0**:** Page erase success

This API is used to perform page erase command on specified non-Secure Flash address. This address must be a page size aligned address.

3.4.10 BL_FlashWrite (For Non-secure Region)

- int32_t BL_FlashWrite (uint32_t u32NSAddr, uint32_t u32Data)
- Parameter**:**

- [in] u32NSAddr	Non-secure Flash address
- [in] u32Data	32-bit data to program

- Return status**:**
  - 0xF0F00000**:** u32NSAddr isn't in non-secure area
  - 0xF0F00001**:** u32NSAddr isn't word aligned
  - 0xF0F00003**:** u32NSAddr isn't valid Flash address
  - -1**:** Flash write failed
  - 0**:** Program command is finished

This API is used to program word data into specified non-Secure Flash address.

3.4.11 BL_GetISPStatus

- uint32_t BL_GetISPStatus (void)
- Parameter**:** None
- Returns**:**
  - 0**:** ISP operation is finished
  - 1**:** ISP operation is in processing

This API indicates ISP operation in in processing or finished.

3.4.12 BL_GetKPROMCounter

- uint32_t BL_GetKPROMCounter (void)
- Parameter**:** None
- Return**:**
  - KPKEYCNT register status

This API can read KPROM KPKEYCNT register status.

3.4.13 BL_GetKPROMPowerOnCounter

- uint32_t BL_GetKPROMPowerOnCounter (void)
- Parameter**:** None
- Return**:**
  - KPCNT register status

This API can read KPROM KPCNT register status.

3.4.14 BL_GetNSBoundary

- uint32_t BL_GetNSBoundary (void)
- Parameter**:** None
- Return**:**
  - Current non-secure boundary

This API can get current non-secure boundary address.

3.4.15 BL_GetKPROMStatus

- uint32_t BL_GetKPROMStatus (void)
- Parameter**:** None
- Return**:**
  - KPKEYSTS register status

This API can read KPROM KPKEYSTS register status.

3.4.16 BL_GetVersion

- uint32_t BL_GetVersion (void)
- Parameter**:** None

- Return**:**
  - Version number of MKROM

This API will return the MKROM version number.

3.4.17 BL_GetXOMActiveStatus

- uint32_t BL_GetXOMActiveStatus (uint32_t u32XOM)
- Parameter**:**
  - [in] u32XOM Specified XOM region，it must be between 0~3
- Returns**:**
  - 0xF0F00001**:** Invalid u32XOM number
  - 0**:** Current XOM region isn't active yet
  - 1**:** Current XOM region is active

This API will return specified XOM region active status.

3.4.18 BL_GetXOMEraseStatus

- int32_t BL_GetXOMEraseStatus (void)
- Parameter**:** None
- Returns**:**
  - -1**:** Erase XOM operation failed
  - 0**:** Erase XOM operation success

This API will return the XOM erase operation is success or not.

3.4.19 BL_ReadCID

- uint32_t BL_ReadCID (void)
- Parameter**:** None
- Return**:**
  - The company ID (32-bit)

The company ID of Nuvoton is fixed to be 0xDA.

3.4.20 BL_ReadDID

- uint32_t BL_ReadDID (void)
- Parameter**:** None

- Return**:**
  - The device ID (32-bit)

This function is used to read device ID.

3.4.21 BL_ReadPID

- uint32_t BL_ReadPID (void)
- Parameter**:** None
- Return**:**
  - The product ID (32-bit)

This function is used to read product ID.

3.4.22 BL_ReadUCID

- uint32_t BL_ReadUCID (uint32_t u32Index)
- Parameter**:**

  - [in] u32Index Index of the UCID to read and u32Index must be 0，1，2，or 3
- Return**:**
  - The UCID of specified index

This function is used to read unique chip ID (UCID).

3.4.23 BL_ReadUID

- uint32_t BL_ReadUID (uint32_t u32Index)
- Parameter**:**

  - [in] u32Index UID index. 0=UID[31:0], 1=UID[63:32], 2=UID[95:64]
- Return**:**
  - The 32-bit unique ID data of specified UID index

To read out specified 32-bit unique ID.

3.4.24 BL_ReadXOMRegion (For Non-secure Region)

- int32_t BL_ReadXOMRegion (uint32_t u32XOM,
  uint32_t *pu32Base,
  uint32_t *pu32PageCnt)
- Parameter**:**

- [in] u32XOM	Specified XOM region，it must be between 0~3
- [out] pu32Base	Return specified XOM base address
- [out] pu32PageCnt	Return specified XOM page count

- Returns**:**
  - 0xF0F00000**:** pu32Base, pu32PageCnt or XOM base address isn't in non-secure area
  - 0xF0F00001**:** Wrong u32XOM, pu32Base or pu32PageCnt parameter
  - 0xF0F00003**:** XOM base address isn't valid Flash address
  - 0xF0F00004**:** pu32Base or pu32PageCnt isn't valid SRAM address
  - 0xF0F00005**:** XOM region isn't configured
  - 0**:** Read specified XOM setting success

This API will read specified XOM relative settings.

3.4.25 BL_ResetChip

- void BL_ResetChip (void)
- Parameter**:** None
- Return**:** None

This API will perform reset CHIP command to reset chip.

3.4.26 BL_SetFlashAllLock

- int32_t BL_SetFlashAllLock (void)
- Parameter**:** None
- Returns**:**
  - -1**:** Set Flash all lock failed
  - 0**:** Set Flash all lock operation is success

This API will protect all Flash region read/write operate by ICE interface.

3.4.27 BL_SetXOMRegion (For Non-secure Region)

- int32_t BL_SetXOMRegion (uint32_t u32XOM,
  uint32_t u32Base,
  uint32_t u32PageCnt,
  uint32_t u32IsDebugMode)
- Parameter**:**

- [in] u32XOM	Specified XOM region, it must be between 0~3
- [in] u32Base	Base address of XOM region
- [in] u32PageCnt	Page count of XOM region
- [in] u32IsDebugMode	1: Enable XOM debug mode, Others: Disable XOM debug mode

- Returns**:**
  - 0xF0F00000**:** XOM region isn't in non-secure area
  - 0xF0F00001**:** Wrong u32XOM, u32Base or u32PageCnt parameter
  - 0xF0F00003**:** u32Base isn't valid Flash address
  - 0xF0F00006**:** XOM region has configured
  - 0xF0F00007**:** XOM region has active
  - -1**:** Set XOM failed
  - 0**:** Set specified XOM success

This API will set specified XOM active.

3.4.28 BL_TrgKPROMCompare

- int32_t BL_TrgKPROMCompare(uint32_t key0, uint32_t key1, uint32_t key2)
- Parameter**:**

- [in] key0	KPROM key0
- [in] key1	KPROM key1
- [in] key2	KPROM key2

- Returns**:**
  - 0xF0F00009**:** KPROM function isn't enabled
  - 0xF0F0000A**:** Trigger KPROM key comparison is FORBID
  - 0xF0F0000B**:** KPROM Key is mismatch
  - 0xF0F0000C**:** KPROM still locked
  - 0**:** KPROM Key are matched

With this API, user can unlock KPROM write-protection and then execute FMC program command well.

4 Crypto Library

The M2351 MKROM Crypto library supports cryptographic functions which are implemented with the M2351 hardware cryptographic accelerator. The supported functions include AES, TDES, SHA, ECC and TRNG and can only be called in the secure code.

If a software developer needs to create a crypto application, developer can add the MKROM Crypto library to achieve the crypto application directly and unnecessary to write complex crypto function code for it.

Figure 4-1 shows how the MKROM Crypto library works with H/W Crypto accelerator when the secure code calls Crypto API.

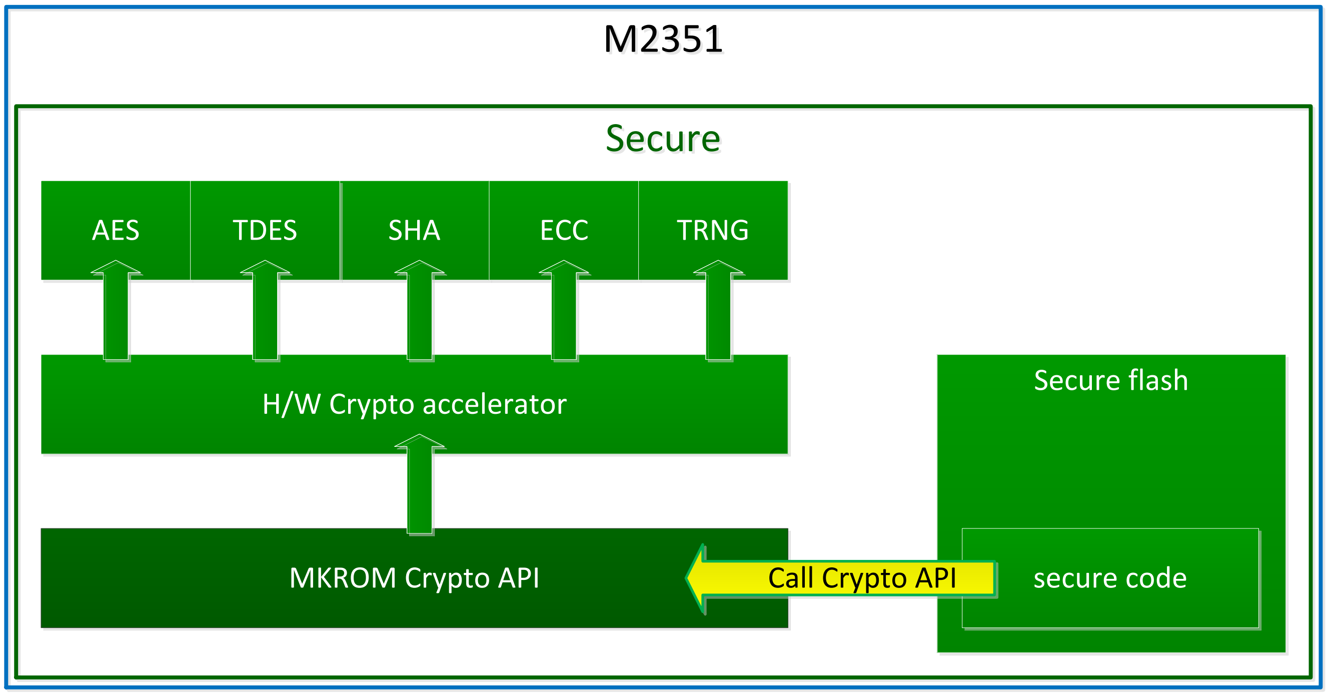


Figure 4-1 Access MKROM Crypto Library in Secure Code

4.1 Features

- AES supports:
  - 128, 192, and 256 bits key
  - Encryption and decryption
  - ECB, CBC, CFB, OFB, CTR, CBC-CS1, CBC-CS2, and CBC-CS3 mode
- TDES supports:
  - Two keys or three keys mode
  - Encryption and decryption
  - ECB, CBC, CFB, OFB, and CTR mode

- SHA supports:
  - SHA-160, SHA-224, SHA-256, SHA-384, and SHA-512
- ECC supports:
  - NIST P-256
  - ECC private and public key pair generation
  - ECDSA signature generation and verification
  - ECDH shared key generation
- TRNG supports:
  - 256 bits random number generation
- Supports for KEIL, IAR and GCC development tool

4.2 Basic Configuration

1. Firstly, a secure code developer must add the MKROMLib library file in the project to perform MKROM Crypto API
   - In KEIL project, the library file name is MKROMLib_Keil.lib
   - In IAR project, the library file name is MKROMLib_IAR.a.
   - In GCC project, the library file name is libmkrom.a.
2. Allocate a global XCRPT_T variable

   A global variable g_xcrpt for MKROM Crypto library is default declared in the library.

```
    /* Initialize a global XCRPT_T data for performing MKROM Crypto library. */
    XCRPT_T g_xcrpt = {CRPT, 0};
```

   And a constant definition XCRPT as the access address of g_xcrpt in mkromlib.h, then software developer can use it for access MKROM Crypto library directly.

```
/**
  * @details    XCRPT_T is structure for accessing MKROM Crypto library
  */
typedef struct
{
    CRPT_T      *crpt;          /*!< The pointer of the CRC module */
    ECC_CURVE   *pCurve;        /*!< Internal use for ECC */
    ECC_CURVE   Curve_Copy;     /*!< Internal use for ECC */
    uint32_t    AES_CTL[4];     /*!< AES channel selection */
    uint32_t    TDES_CTL[4];    /*!< TDES channel selection */
} XCRPT_T;

/* Define a global constant XCRPT as the access address of g_xcrpta for using MKROM Crypto
library. */
```

```
extern XCRPT_T g_xcrpt;
#define XCRPT (&g_xcrpt)
```

3. Enable Crypto module clock first

   Secure code developer must enable the Crypto module clock before accessing the MKROM Crypto library.

```
    /* Enable CRYPTO module clock */
    CLK_EnableModuleClock(CRPT_MODULE);
```

4.3 CRYPTOLibDemo Sample Code

All the sample codes for demonstrating Crypto library are located in the "M2351BSP\SampleCode\MKROM\CRYPTOLibDemo" folder.

The description of each sample code is shown in Table 4-1.

Sample Code	Description
AES_Demo	Use MKROM Crypto library to show AES-128 ECB mode encrypt/decrypt function.
ECC_Demo	Use MKROM Crypto library to demonstrate the ECC P-256 signature generation and verification.
ECC_ECDH	Use MKROM Crypto library to demonstrate how to calculate share key by A private key and B private key.
ECC_KeyGeneration	Use MKROM Crypto library to show ECC P-256 key generation function.
ECC_SignatureGeneration	Use MKROM Crypto library to show ECC P-256 ECDSA signature generation function.
ECC_SignatureVerification	Use MKROM Crypto library to show ECC P-256 ECDSA signature verification function.
SHA_Demo	Use MKROM Crypto library to generate SHA-256 message digest.
TDES_Demo	Use MKROM Crypto library to show Triple DES CBC mode encrypt/decrypt function.
TRNG_Demo	Use MKROM Crypto library to generate random numbers.

Table 4-1 Crypto Sample Code

The following sections describe the usage of Crypto library sample code in detail.

4.3.1 AES_Demo Code

This sample code shows the AES-128 ECB mode encrypt/decrypt function.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable CRYPTO IRQ and AES interrupt

  Prepare CRYPTO IRQ function for AES.

```
void CRPT_IRQHandler()
{
    if(AES_GET_INT_FLAG(CRPT))
    {
        g_u32IsAES_done = 1;
        AES_CLR_INT_FLAG(CRPT);
    }
}
```

  Enable NVIC_EnableIRQ(CRPT_IRQn) and AES_ENABLE_INT(CRPT);

```
    /* Enable CRYPTO IRQ and AES interrupt. */
    NVIC_EnableIRQ(CRPT_IRQn);
    AES_ENABLE_INT(CRPT);
```

- Perform AES encryption

  Execute AES-128 ECB encrypt.

```
    /*---------------------------------------
     *  AES-128 ECB mode encrypt
     *---------------------------------------*/
    XAES_Open(XCRPT, 0, 1, AES_MODE_ECB, AES_KEY_SIZE_128, AES_IN_OUT_SWAP);
    XAES_SetKey(XCRPT, 0, g_au32MyAESKey, AES_KEY_SIZE_128);
    XAES_SetInitVect(XCRPT, 0, g_au32MyAESIV);
    XAES_SetDMATransfer(XCRPT, 0, (uint32_t)g_au8InputData, (uint32_t)g_au8OutputData,
sizeof(g_au8InputData));

    g_u32IsAES_done = 0;
    XAES_Start(XCRPT, 0, CRYPTO_DMA_ONE_SHOT);
    while(g_u32IsAES_done == 0) {}
    AES_ENABLE_INT(CRPT);
```

- Perform AES decryption

Execute AES-128 ECB decrypt.

```
        /*---------------------------------------
         *  AES-128 ECB mode decrypt
         *---------------------------------------*/
        XAES_Open(XCRPT, 0, 0, AES_MODE_ECB, AES_KEY_SIZE_128, AES_IN_OUT_SWAP);
        XAES_SetKey(XCRPT, 0, g_au32MyAESKey, AES_KEY_SIZE_128);
        XAES_SetInitVect(XCRPT, 0, g_au32MyAESIV);
        XAES_SetDMATransfer(XCRPT, 0, (uint32_t)g_au8OutputData, (uint32_t)g_au8InputData,
sizeof(g_au8InputData));

        g_u32IsAES_done = 0;
        XAES_Start(XCRPT, 0, CRYPTO_DMA_ONE_SHOT);
        while(g_u32IsAES_done == 0) {}
```

4.3.2 ECC_Demo Code

This sample code demonstrates how to create an ECC private/public key pair, then generate an ECDSA signature and perform the ECDSA signature verification.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable ECC interrupt

  To enable ECC interrupt, and then the MKROM Crypto library can polling the ECC operation status.

```
    /* Enable ECC interrupt. */
    ECC_ENABLE_INT(CRPT);
```

- Create ECC private/public key pair

  Use TRNG to create an ECC private/public key pair.

```
    /* Initial TRNG */
    XTRNG_RandomInit(&rng, TRNG_PRNG | TRNG_LIRC32K);

    do
    {
        /* Generate random number for private key */
        XTRNG_Random(&rng, au8r, i32NBits / 8);

        for(i = 0, j = 0; i < i32NBits / 8; i++)
        {
            d[j++] = Byte2Char(au8r[i] & 0xf);
            d[j++] = Byte2Char(au8r[i] >> 4);
        }
        d[j] = 0; // NULL end

        printf("Private key = %s\n", d);

        /* Check if the private key valid */
        if(XECC_IsPrivateKeyValid(XCRPT, ECC_CURVE_TYPE, d))
        {
            //printf("Private key check ok\n");
            break;
        }
        else
        {
            /* Invalid key */
            printf("Current private key is not valid. Need a new one.\n");
        }
    }
    while(1);

    /* Generate public */
    if(XECC_GeneratePublicKey(XCRPT, ECC_CURVE_TYPE, d, Qx, Qy) < 0)
    {
        printf("ECC key generation failed!!\n");
        while(1) {}
    }
```

- Generate ECDSA signature and verify ECDSA signature

  Generate a random number again for XECC_GenerateSignature API to generate ECDSA signature, and then calling XECC_VerifySignature API to verify the ECDSA signature.

```
    /* Generate random number k */
    XTRNG_Random(&rng, au8r, i32NBits / 8);

    for(i = 0, j = 0; i < i32NBits / 8; i++)
    {
         k[j++] = Byte2Char(au8r[i] & 0xf);
         k[j++] = Byte2Char(au8r[i] >> 4);
    }
    k[j] = 0; // NULL end
    printf("  k  = %s\n", k);

    if(XECC_IsPrivateKeyValid(XCRPT, ECC_CURVE_TYPE, k))
    {
         //printf("Private key check ok\n");
    }
    else
    {
        /* Invalid key */
        printf("Current k is not valid\n");
        while(1) {}
    }

    /* Generate signature */
    if(XECC_GenerateSignature(XCRPT, ECC_CURVE_TYPE, msg, d, k, R, S) < 0)
    {
        printf("ECC signature generation failed!!\n");
        while(1) {}
    }

    /* Verify signature */
    i32Err = XECC_VerifySignature(XCRPT, ECC_CURVE_TYPE, msg, Qx, Qy, R, S);
    if(i32Err < 0)
    {
        printf("ECC signature verification failed!!\n");
        while(1) {}
    }
    else
    {
        printf("ECC digital signature verification OK.\n");
    }
```

4.3.3 ECC_ECDH Code

This sample code demonstrates how to create ECC private/public key pair-A and key pair-B, and then calculate the ECDH share key by private key-A and public key-B or private key-B and public key-A.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable ECC interrupt

  To enable ECC interrupt, and then the MKROM Crypto library can polling the ECC operation status.

```
    /* Enable ECC interrupt. */
    ECC_ENABLE_INT(CRPT);
```

- Create ECC private/public key pair

  Use TRNG to create ECC private/public key pair-A and key pair-B.

```
    /* Initial TRNG */
    XTRNG_RandomInit(&rng, TRNG_PRNG | TRNG_LIRC32K);

//-----------------------------------------------------------------------
    // Generate a private key A
    GenPrivateKey(&rng, d, ECC_KEY_SIZE);
    printf("Private key A = %s\n", d);

    // Generate public Key A
    if(XECC_GeneratePublicKey(XCRPT, ECC_CURVE_TYPE, d, Qx, Qy) < 0)
    {
        printf("ECC key generation failed!!\n");
        while(1) {}
    }

//-----------------------------------------------------------------------
    // Generate a private key B
    GenPrivateKey(&rng, d2, ECC_KEY_SIZE);
    printf("Private key  B = %s\n", d2);

    // Generate public Key B
    if(XECC_GeneratePublicKey(XCRPT, ECC_CURVE_TYPE, d2, Qx2, Qy2) < 0)
    {
        printf("ECC key generation failed!!\n");
        while(1) {}
    }
```

- Calculate ECDH share key

  Call XECC_GenerateSecretZ API to generate ECDH share key.

```
//-----------------------------------------------------------------------
```

```
    // Calcualte Share Key by private key A and publick key B
    if(XECC_GenerateSecretZ(XCRPT, ECC_CURVE_TYPE, d, Qx2, Qy2, k) < 0)
    {
        printf("ECC ECDH share key calculation fail!!\n");
        while(1) {}
    }
    printf("Share key calculated by A = %s\n", k);

    // Calcualte Share Key by private key B and publick key A
    if(XECC_GenerateSecretZ(XCRPT, ECC_CURVE_TYPE, d2, Qx, Qy, k2) < 0)
    {
        printf("ECC ECDH share key calculation fail!!\n");
        while(1) {}
    }
```

4.3.4 ECC_KeyGeneration Code

This sample code demonstrates how to generate an ECC P-256 public key.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable ECC interrupt and generate ECC public key

  To enable ECC interrupt, and then the MKROM Crypto library can polling the ECC operation status.

  After that, demo code will execute XECC_GeneratePublicKey API to generate ECC public key with input private key-d.

```
    /* Enable ECC interrupt. */
    ECC_ENABLE_INT(CRPT);

    if(XECC_GeneratePublicKey(XCRPT, ECC_CURVE_TYPE, d, key1, key2) < 0)
    {
        printf("ECC key generation failed!!\n");
        while(1) {}
    }
```

4.3.5 ECC_SignatureGeneration Code

This sample code demonstrates how to generate an ECC P-256 ECDSA signature.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable ECC interrupt and generate ECDSA signature

  To enable ECC interrupt, and then the MKROM Crypto library can polling the ECC operation status.

  After that, demo code will execute XECC_GenerateSignature API to generate ECDSA signature.

```
    /* Enable ECC interrupt. */
    ECC_ENABLE_INT(CRPT);

    if(XECC_GenerateSignature(XCRPT, ECC_CURVE_TYPE, sha_msg, d, k, sig_R, sig_S) < 0)
    {
        printf("ECC signature generation failed!!\n");
        while(1);
    }
```

4.3.6 ECC_SignatureVerification Code

This sample code demonstrates how to verify the ECC P-256 ECDSA signature.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable ECC interrupt and perform ECDSA signature verification

  To enable ECC interrupt, and then the MKROM Crypto library can polling the ECC operation status.

  After that, demo code will execute XECC_VerifySignature API to verify ECDSA signature.

```
    /* Enable ECC interrupt. */
    ECC_ENABLE_INT(CRPT);

    if(XECC_VerifySignature(XCRPT, ECC_CURVE_TYPE, sha_msg, Qx, Qy, R, S) < 0)
    {
        printf("ECC signature verification failed!!\n");
        while(1) {}
    }
```

4.3.7 SHA_Demo Code

This sample code shows how to use the SHA-256 function to generate a message digest.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable CRYPTO IRQ and SHA interrupt

  Prepare CRYPTO IRQ function for SHA.

```
void CRPT_IRQHandler()
{
    if(SHA_GET_INT_FLAG(CRPT))
    {
        g_u32IsSHA_done = 1;
        SHA_CLR_INT_FLAG(CRPT);
    }
}
```

Enable NVIC_EnableIRQ(CRPT_IRQn) and SHA_ENABLE_INT(CRPT);

```
    /* Enable CRYPTO IRQ and SHA interrupt. */
    NVIC_EnableIRQ(CRPT_IRQn);
    SHA_ENABLE_INT(CRPT);
```

- Start SHA operation

  Perform SHA algorithm by calling related APIs to start calculating the digest.

```
    /*---------------------------------------
     *  SHA-256
     *---------------------------------------*/
    XSHA_Open(XCRPT, SHA_MODE_SHA256, SHA_IN_OUT_SWAP, 0);

    printf("Input string data is %s.\n\n", g_acInputString);
    XSHA_SetDMATransfer(XCRPT, (uint32_t)&g_acInputString[0],  sizeof(g_acInputString)-1);

    g_u32IsSHA_done = 0;
    XSHA_Start(XCRPT, CRYPTO_DMA_ONE_SHOT);
    while(g_u32IsSHA_done == 0) {}
```

- Read SHA digest

  Execute XSHA_Read() to get the digest after complete the SHA operation.

```
    /* Read the SHA digest. */
    XSHA_Read(XCRPT, au32Output)
```

4.3.8 TDES_Demo Code

This sample code shows the TDES CBC mode encrypt/decrypt function.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
    /* Enable CRYPTO module clock. */
    CLK_EnableModuleClock(CRPT_MODULE);
```

- Enable CRYPTO IRQ and TDES interrupt

  Prepare CRYPTO IRQ function for TDES.

```
void CRPT_IRQHandler()
{
    if(TDES_GET_INT_FLAG(CRPT))
    {
        g_u32IsTDES_done = 1;
        TDES_CLR_INT_FLAG(CRPT);
    }
}
```

  Enable NVIC_EnableIRQ(CRPT_IRQn) and TDES_ENABLE_INT(CRPT);

```
    /* Enable CRYPTO IRQ and TDES interrupt. */
    NVIC_EnableIRQ(CRPT_IRQn);
    TDES_ENABLE_INT(CRPT);
```

- Perform TDES encryption

  Execute TDES CBC encrypt.

```
    /*---------------------------------------
     *  TDES CBC mode encrypt
     *---------------------------------------*/
    XTDES_Open(XCRPT, 0, 1, 1, 1, TDES_MODE_CBC, TDES_IN_OUT_WHL_SWAP);
    XTDES_SetKey(XCRPT, 0, g_au8MyTDESKey);
    XTDES_SetInitVect(XCRPT, 0, g_au32MyTDESIV[0], g_au32MyTDESIV[1]);
    XTDES_SetDMATransfer(XCRPT, 0, (uint32_t)g_au8InputData, (uint32_t)g_au8OutputData,
sizeof(g_au8InputData));


    g_u32IsTDES_done = 0;
    XTDES_Start(XCRPT, 0, CRYPTO_DMA_ONE_SHOT);
    while(g_u32IsTDES_done == 0) {}
```

- Perform TDES decryption

  Execute TDES CBC decrypt.

```
        /*---------------------------------------
         *  TDES CBC mode decrypt
         *---------------------------------------*/
        XTDES_Open(XCRPT, 0, 0, 1, 1, TDES_MODE_CBC, TDES_IN_OUT_WHL_SWAP);
        XTDES_SetKey(XCRPT, 0, g_au8MyTDESKey);
        XTDES_SetInitVect(XCRPT, 0, g_au32MyTDESIV[0], g_au32MyTDESIV[1]);
        XTDES_SetDMATransfer(XCRPT, 0, (uint32_t)g_au8OutputData, (uint32_t)g_au8InputData,
sizeof(g_au8InputData));

        g_u32IsTDES_done = 0;
        XTDES_Start(XCRPT, 0, CRYPTO_DMA_ONE_SHOT);
        while(g_u32IsTDES_done == 0) {}
```

4.3.9 TRNG_Demo Code

This sample code demonstrates how to use Crypto library to generate random numbers.

- Enable module clock

  Enable CRTPYO module clock in the SYS_Init().

```
        /* Enable CRYPTO module clock. */
        CLK_EnableModuleClock(CRPT_MODULE);
```

- Initialize a TRNG generator, and then generate random number

  Execute XTRNG_RandomInit API first to initial the TRNG generator, and then execute XTRNG_Random API to get the random number 1 and random number 2.

```
        /* Initial TRNG */
        XTRNG_RandomInit(&rng, TRNG_PRNG | TRNG_LIRC32K);

        pu8Buf = (uint8_t *)au32RandVal;

        /* Generate random number 1 */
        XTRNG_Random(&rng, pu8Buf, BYTE_COUNT);

        printf("Random numbers 1:\n");
        for(i=0; i<(BYTE_COUNT/4); i++)
        {
            printf("  0x%08x", au32RandVal[i]);
        }
        /* Generate random number 2 */
        XTRNG_Random(&rng, pu8Buf, BYTE_COUNT);

        printf("Random numbers 2:\n");
        for(i=0; i<(BYTE_COUNT/4); i++)
        {
            printf("  0x%08x", au32RandVal[i]);
        }
        printf("\n");
```

4.4 Crypto API List

The following section details the MKROM Crypto API. For the MKROM header file, refer to "M2351BSP\Library\StdDriver\inc\mkromlib.h".

4.4.1 XAES_Open

- void XAES_Open (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t u32EncDec, uint32_t u32OpMode, uint32_t u32KeySize, uint32_t u32SwapType)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	AES channel. Must be 0~3.
- [in] u32EncDec	1: AES encode; 0: AES decode
- [in] u32OpMode	AES operation mode, including: AES_MODE_ECB, AES_MODE_CBC, AES_MODE_CFB, AES_MODE_OFB, AES_MODE_CTR, AES_MODE_CBC_CS1, AES_MODE_CBC_CS2, AES_MODE_CBC_CS3
- [in] u32KeySize	u32KeySize is AES key size, including: AES_KEY_SIZE_128, AES_KEY_SIZE_192, AES_KEY_SIZE_256
- [in] u32SwapType	u32SwapType is AES input/output data swap control, including: AES_NO_SWAP, AES_OUT_SWAP, AES_IN_SWAP, AES_IN_OUT_SWAP

- Return**:** None

This API is used to enable AES encrypt/decrypt function.

4.4.2 XAES_Start

- void XAES_Start (XCRPT_T *xcrpt, int32_t u32Channel, uint32_t u32DMAMode)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	AES channel. Must be 0~3.
- [in] u32DMAMode	AES DMA control, including: CRYPTO_DMA_ONE_SHOT: One shot AES encrypt/decrypt. CRYPTO_DMA_CONTINUE: Continuous AES encrypt/decrypt. CRYPTO_DMA_LAST: Last AES encrypt/decrypt of a series of XAES_Start.

- Return**:** None

This API is used to start AES encrypt/decrypt.

4.4.3 XAES_SetKey

- void XAES_SetKey (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t au32Keys[], uint32_t u32KeySize)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	AES channel. Must be 0~3.
- [in] au32Keys	A word array contains AES keys.
- [in] u32KeySize	u32KeySize is AES key size, including: AES_KEY_SIZE_128, AES_KEY_SIZE_192, AES_KEY_SIZE_256

- Return**:** None

This API is used to set AES keys.

4.4.4 XAES_SetInitVect

- void XAES_SetInitVect (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t au32IV[])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	AES channel. Must be 0~3.
- [in] au32IV	A four entry word array contains AES initial vectors.

- Return**:** None

This API is used to set AES initial vectors.

4.4.5 XAES_SetDMATransfer

- void XAES_Set DMATransfer (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t u32SrcAddr, uint32_t u32DstAddr, uint32_t u32TransCnt)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	AES channel. Must be 0~3.
- [in] u32SrcAddr	AES DMA source address.
- [in] u32DstAddr	AES DMA destination address.
- [in] u32TransCnt	AES DMA transfer byte count.

- Return**:** None

This API is used to configure the AES DMA transfer.

4.4.6 XTDES_Open

- void XTDES_Open (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t u32EncDec, int32_t Is3DES, int32_t Is3Key, uint32_t u32OpMode, uint32_t u32SwapType)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	TDES channel. Must be 0~3.
- [in] u32EncDec	1: TDES encode; 0: TDES decode.
- [in] Is3DES	1: TDES; 0: DES.
- [in] Is3Key	1: TDES 3 key mode; 0: TDES 2 key mode.
- [in] u32OpMode	TDES operation mode, including: TDES_MODE_ECB, TDES_MODE_CBC, TDES_MODE_CFB, TDES_MODE_OFB, TDES_MODE_CTR
- [in] u32SwapType	u32SwapType is TDES input/output data swap control and word swap control, including: TDES_NO_SWAP, TDES_WHL_SWAP, TDES_OUT_SWAP, TDES_OUT_WHL_SWAP, TDES_IN_SWAP, DES_IN_WHL_SWAP, TDES_IN_OUT_SWAP, TDES_IN_OUT_WHL_SWAP

- Return**:** None

This API is used to enable TDES encrypt/decrypt function.

4.4.7 XTDES_Start

- void XTDES_Start (XCRPT_T *xcrpt, int32_t u32Channel, uint32_t u32DMAMode)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	TDES channel. Must be 0~3.
- [in] u32DMAMode	TDES DMA control, including:
	CRYPTO_DMA_ONE_SHOT:
	One shot TDES encrypt/decrypt.
	CRYPTO_DMA_CONTINUE
	Continuous TDES encrypt/decrypt.
	CRYPTO_DMA_LAST
	Last TDES encrypt/decrypt of a series of XTDES_Start.

- Return**:** None

This API is used to start TDES encrypt/decrypt.

4.4.8 XTDES_SetKey

- void XTDES_SetKey (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t au32Keys[3][2])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	TDES channel. Must be 0~3.
- [in] au32Keys[3][2]	The TDES keys. au32Keys[0][0] is Key0 high word and au32Keys[0][1] is key0 low word.

- Return**:** None

This API is used to set TDES keys.

4.4.9 XTDES_SetInitVect

- void XTDES_SetInitVect (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t u32IVH, uint32_t u32IVL)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	TDES channel. Must be 0~3.
- [in] u32IVH	TDES initial vector high word.

- [in] u32IVL	TDES initial vector low word.

- Return**:** None

This API is used to set TDES initial vectors.

4.4.10 XTDES_SetDMATransfer

- void XTDES_SetDMATransfer (XCRPT_T *xcrpt, uint32_t u32Channel, uint32_t u32SrcAddr, uint32_t u32DstAddr, uint32_t u32TransCnt)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32Channel	TDES channel. Must be 0~3.
- [in] u32SrcAddr	TDES DMA source address.
- [in] u32DstAddr	TDES DMA destination address.
- [in] u32TransCnt	TDES DMA transfer byte count.

- Return**:** None

This API is used to configure the TDES DMA transfer.

4.4.11 XSHA_Open

- void XSHA_Open (XCRPT_T *xcrpt, uint32_t u32OpMode, uint32_t u32SwapType, uint32_t hmac_key_len)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32OpMode	SHA operation mode, including: SHA_MODE_SHA1, SHA_MODE_SHA224, SHA_MODE_SHA256
- [in] u32SwapTyperess	u32SwapType is the SHA input/output data swap control, including: SHA_NO_SWAP, SHA_OUT_SWAP, SHA_IN_SWAP, SHA_IN_OUT_SWAP
- [in] hmac_key_len	HMAC key byte count.

- Return**:** None

This API is used to enable SHA encrypt function.

4.4.12 XSHA_Start

- void XSHA_Start (XCRPT_T *xcrpt, uint32_t u32DMAMode)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32DMAMode	SHA DMA control, including: CRYPTO_DMA_ONE_SHOT: One shot SHA encrypt. CRYPTO_DMA_CONTINUE: Continuous SHA encrypt. CRYPTO_DMA_LAST: Last SHA encrypt of a series of XSHA_Start.

- Return**:** None

This API is used to start SHA encrypt.

4.4.13 XSHA_SetDMATransfer

- void XSHA_SetDMATransfer (XCRPT_T *xcrpt, uint32_t u32SrcAddr, uint32_t u32TransCnt)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] u32SrcAddr	SHA DMA source address.
- [in] u32TransCnt	SHA DMA transfer byte count.

- Return**:** None

This API is used to configure the SHA DMA transfer.

4.4.14 XSHA_Read

- void XSHA_Read (XCRPT_T *xcrpt, uint32_t u32Digest[])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [out] u32Digest	The SHA encrypt output digest.

- Return**:** None

This API is used to read the SHA digest.

4.4.15 XECC_IsPrivateKeyValid

- Int32_t XECC_IsPrivateKeyValid (XCRPT_T *xcrpt, E_ECC_CURVE ecc_curve, char private_k[])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] ecc_curve	The pre-defined ECC curve.
- [in] private_k	The input private key.

- Returns**:**
  - 1**:** Is valid.
  - 0**:** Is not valid.
  - -1**:** Invalid curve.

This API is used to check if the private key is placed in valid range of curve.

4.4.16 XECC_GeneratePublicKey

- Int32_t XECC_GeneratePublicKey (XCRPT_T *xcrpt, E_ECC_CURVE ecc_curve, char *private_k, char public_k1[], char public_k2[])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] ecc_curve	The pre-defined ECC curve.
- [in] private_k	The input private key.
- [out] public_k1	The output public key 1.
- [out] public_k2	The output public key 2.

- Returns**:**
  - 0**:** Success.
  - -1**:** "ecc_curve" value is invalid.

This API is used to generate a public key pair by a specified ECC private key and ECC curve.

4.4.17 XECC_GenerateSignature

- Int32_t XECC_GenerateSignature (XCRPT_T *xcrpt, E_ECC_CURVE ecc_curve, char *message, char *d, char *k, char *R, char *S)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] ecc_curve	The pre-defined ECC curve.
- [in] message	The hash value of source context.

- [in] d	The private key.
- [in] k	The selected random integer.
- [out] R	R of the (R,S) pair digital signature.
- [out] S	S of the (R,S) pair digital signature.

- Returns**:**
  - 0**:** Success.
  - -1**:** "ecc_curve" value is invalid.

This API is used to generate an ECDSA digital signature.

4.4.18 XECC_VerifySignature

- Int32_t XECC_VerifySignature (XCRPT_T *xcrpt, E_ECC_CURVE ecc_curve, char *message, char *public_k1, char *public_k2, char *R, char *S)
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] ecc_curve	The pre-defined ECC curve.
- [in] message	The hash value of source context.
- [in] public_k1	The public key 1.
- [in] public_k2	The public key 2.
- [in] R	R of the (R,S) pair digital signature.
- [in] S	S of the (R,S) pair digital signature.

- Returns**:**
  - 0**:** Success.
  - -1**:** "ecc_curve" value is invalid.
  - -2**:** Verification failed.

This API is used to perform the ECDSA digital signature verification.

4.4.19 XECC_GenerateSecretZ

- Int32_t XECC_GenerateSecretZ (XCRPT_T *xcrpt, E_ECC_CURVE ecc_curve, char *private_k, char public_k1[], char public_k2[], char secret_z[])
- Parameters**:**

- [in] xcrpt	The pointer of the global XCRPT data.
- [in] ecc_curve	The pre-defined ECC curve.
- [in] private_k	One's own private key
- [in] public_k1	The other party's public key 1.

- [in] public_k2	The other party's public key 2.
- [out] secret_z	The ECC CDH secret Z.

- Returns**:**
  - 0**:** Success.
  - -1**:** "ecc_curve" value is invalid.

This API is used to generate an ECDH shared key.

4.4.20 XECC_Reg2Hex

- void XECC_Reg2Hex (int32_t count, uint32_t volatile reg[], char output[])
- Parameters**:**

- [in] count	Byte counts for convert.
- [in] reg	The input data buffer.
- [out] output	The output data buffer .

- Return**:** None

This API is used to convert the data to hex format.

4.4.21 XECC_Hex2Reg

- void XECC_Hex2Reg (char input[], uint32_t volatile reg[])
- Parameters**:**

- [in] input	The input data buffer.
- [in] reg	The output data buffer.

- Return**:** None

This API is used to convert the data in a register data format.

4.4.22 XECC_GetIDECCSignature

- Int32_t XECC_GetIDECCSignature (uint32_t *R, uint32_t *S)
- Parameters**:**

- [out] R	R of the (R,S) pair digital signature.
- [out] S	S of the (R,S) pair digital signature.

- Returns**:**
  - -1**:** Get R, S digital signature fail.
  - 0**:** Success.

This API will return the ECDSA R, S digital signature of chip ID, include PDID, UID0~2 and

UCID0~3.

4.4.23 XTRNG_RandomInit

- Int32_t XTRNG_RandomInit (XTRNG_T *rng, uint32_t opt)
- Parameters**:**

- [in] rng	The structure of random number generator.
- [in] opt	Operation modes. Possible options are, (XTRNG_PRNG | XTRNG_LIRC32K), (XTRNG_PRNG | XTRNG_LXT), (XTRNG_SWRNG | XTRNG_LIRC32K), (XTRNG_SWRNG | XTRNG_LXT)

- Returns**:**
  - -1**:** Fail.
  - 0**:** Success.

This API is used to initial random number generator. After initial this API success, user can call XTRNG_Random API to generate the random number.

4.4.24 XTRNG_Random

- Int32_t XTRNG_Random (XTRNG_T *rng, uint8_t *p, uint32_t size)
- Parameters**:**

- [in] rng	The structure of random number generator.
- [out] p	Starting buffer address to store random number.
- [in] size	Total byte counts of random number.

- Returns**:**
  - -1**:** Fail.
  - 0**:** Success.

This API is used to generate random number.

5 SecureISP Library

Before introducing the SecureISP library, the SecureISP function and how it works will be described first..

The M2351 SecureISP function is a command communication mode between M2351 chip and external device (likes PC or a remote device).

The communication interface supports USBD and UART peripherals. User can send commands from external device to M2351 target chip through USBD/UART interface for achieving specified command operation on M2351 target chip.

For the security and to protect the sensitive data, a secure encrypted channel is necessary to be generated between the M2351 chip and external device.

Figure 5-1 shows how the SecureISP function generates a secured encrypted channel between the M2351 and external device with bi-directional authentication and a random AES key.

- Bi-directional authentication
  - Perform the 1st ECDH key exchange in advance to generate an AES key for server and client authentication
- A random session key
  - Generate a random session key through 2nd ECDH key exchange for command encryption/decryption

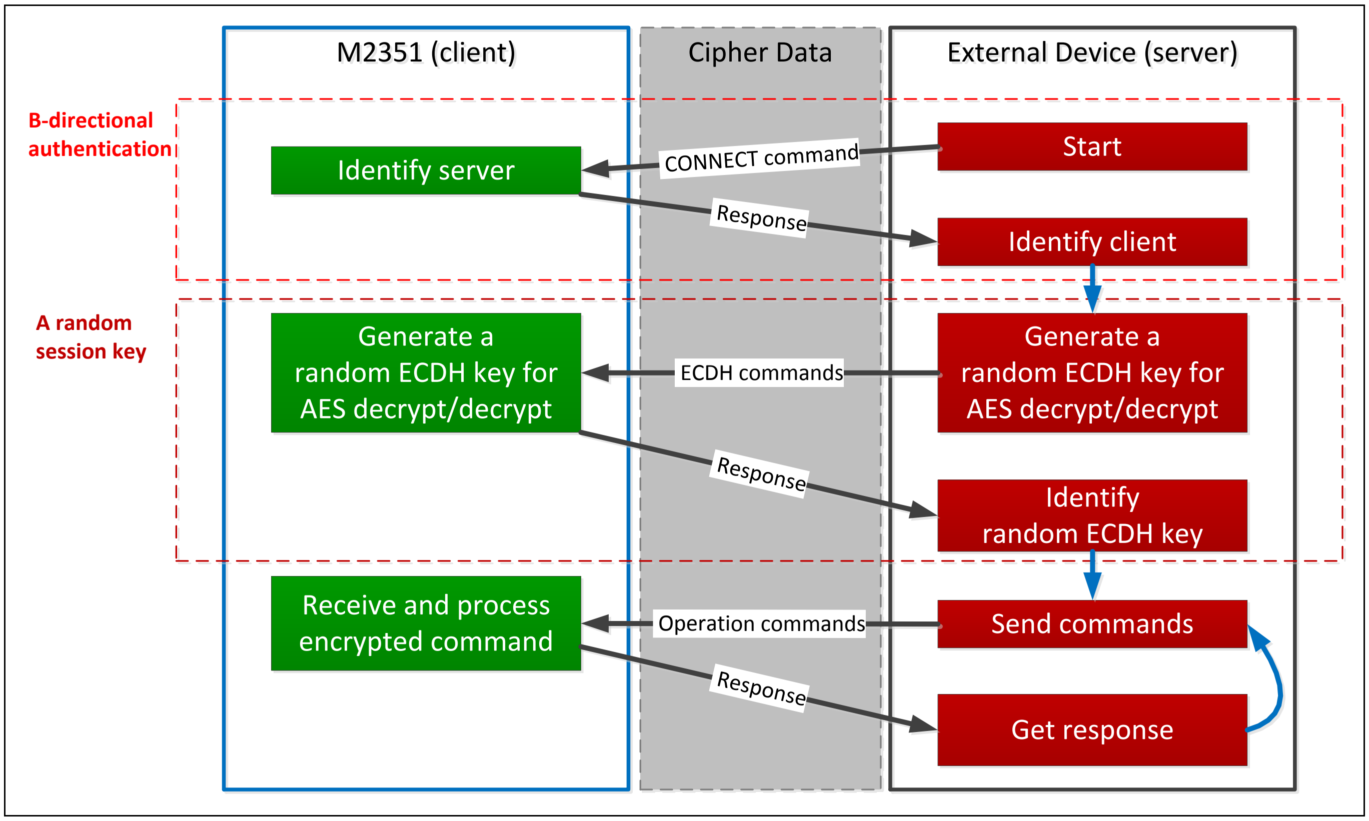


Figure 5-1 Secure Encrypted Channel Generation Flow

The MKROM SecureISP library is used for secure code developer only to implement the SecureISP function on M2351 chip.

And user can use the Nuvoton NuMicro® SecureISP Tool as a server to send commands after connecting with M2351 USBD/UART1 interfaces.

The following figure shows the SecureISP operation flow in the MKROM library. The detailed process flow is,

- Step 1: Secure code calls the BL_SecureISPInit API to enable the SecureISP function
- Step 2: SecureISP Tool sends command to M2351 via USDB/UART1 interface
- Step 3: Secure code receives the USDB/UART1 data input interrupt event
- Step 4: Secure code calls the process USBD/UART1 interrupt API in MKROM
- Step 5: MKROM executes the related process interrupt function to get command
- Step 6: MKROM processes the command, and then responds the data via M2351 USBD/UART1 interface
- Step 7: SecureISP Tool receives the response command via USBD/UART1 interface

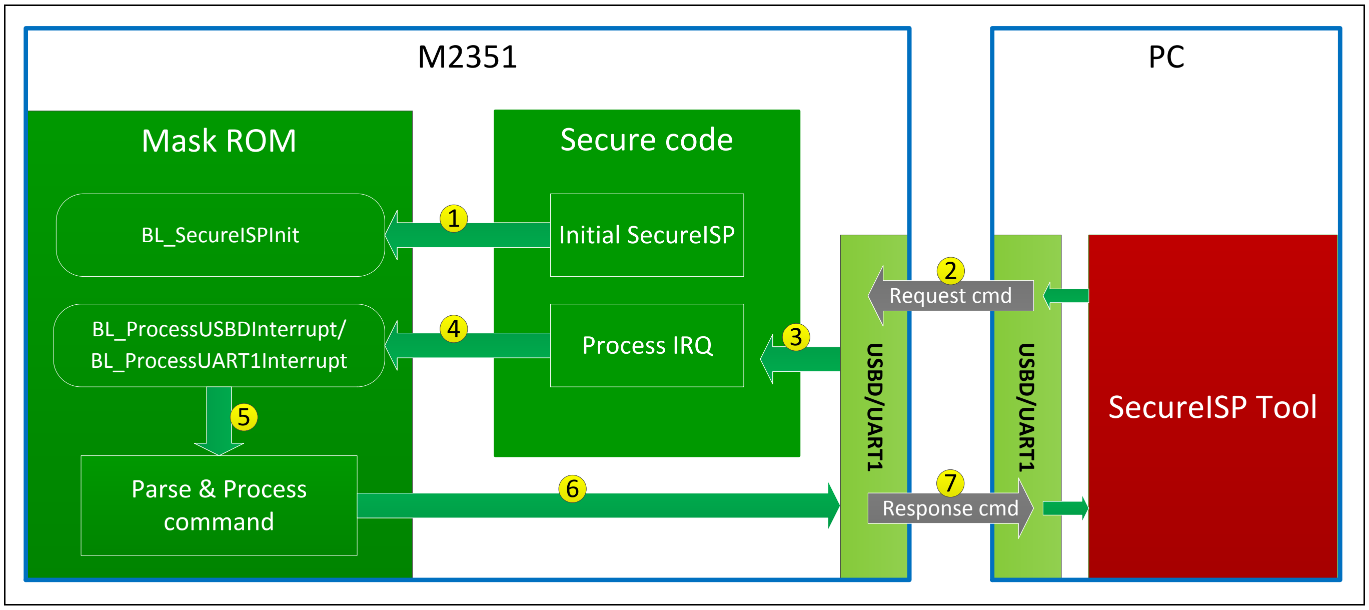


Figure 5-2 SecureISP Operation Flow

5.1 Features

- Supports bi-directional authentication on M2351 (client) and external device (server)
- Supports a random session key for command encryption/decryption with AES-256 CFB mode

- Supports Flash erase, write and mass erase commands
- Supports related KPROM commands, includes enable/disable KPROM and key comparison
- Supports related XOM commands, includes set and erase XOM region
- Supports All Region Lock and Secure Region Lock commands
- Supports OTP read and write commands
- Supports update User Configuration command
- Supports read chip ID and status command
- Provides vendor command to customize a SecureISP command
- Supports mask command to disable the specified SecureISP commands
- Configurable UART1 pins and USB peripheral clock source
- Configurable connection time-out time
- Supports for KEIL, IAR and GCC development tool

5.2 Basic Configuration

1. Firstly, a secure code developer must add the MKROMLib library file in the project to perform MKROM SecureISP API
   - In KEIL project, the library file name is MKROMLib_Keil.lib
   - In IAR project, the library file name is MKROMLib_IAR.a.
   - In GCC project, the library file name is libmkrom.a.
2. Declare ISP_INFO_T and BL_USBD_INFO_T variables

   There are two global variables must be declared in secure code for the SecureISP application.

   - g_ISPInfo, which is used for setting the initial values for USBD/UART1 SecureISP function.
   - g_USBDInfo, which is internal use while SecureISP function is running in USB mode.

```
/* Global variables for initializing SecureISP function */
ISP_INFO_T      g_ISPInfo = {0};
BL_USBD_INFO_T  g_USBDInfo = {0};
```

3. Configure USBD settings
   Before initializing SecureISP USBD function, the related USBD peripheral settings must be configured correctly.
   The following example shows how to configure USBD, including USBD module clock frequency and peripheral pins settings.

```
static int32_t ConfigureUSBDISP(void)
{
    printf("[Configure USBD]\n");

    /* Enable Internal RC 48MHz clock */
    CLK_EnableXtalRC(CLK_PWRCTL_HIRC48EN_Msk);

    /* Waiting for Internal RC clock ready */
    CLK_WaitClockReady(CLK_STATUS_HIRC48STB_Msk);

    /* Use HIRC48 as USB clock source */
    CLK_SetModuleClock(USBD_MODULE, CLK_CLKSEL0_USBSEL_HIRC48, CLK_CLKDIV0_USB(1));

    /* Select USBD */
    SYS->USBPHY = (SYS->USBPHY & ~SYS_USBPHY_USBROLE_Msk) | SYS_USBPHY_OTGPHYEN_Msk |
SYS_USBPHY_SBO_Msk;

    /* Enable IP clock */
    CLK_EnableModuleClock(USBD_MODULE);

    /* USBD multi-function pins for VBUS, D+, D-, and ID pins */
    SYS->GPA_MFPH &= ~(SYS_GPA_MFPH_PA12MFP_Msk | SYS_GPA_MFPH_PA13MFP_Msk |
SYS_GPA_MFPH_PA14MFP_Msk | SYS_GPA_MFPH_PA15MFP_Msk);
    SYS->GPA_MFPH |= (SYS_GPA_MFPH_PA12MFP_USB_VBUS | SYS_GPA_MFPH_PA13MFP_USB_D_N |
SYS_GPA_MFPH_PA14MFP_USB_D_P | SYS_GPA_MFPH_PA15MFP_USB_OTG_ID);

    printf("\n");

    return 0;
}
```

4. Configure UART1 settings

   Before initializing SecureISP UART1 function, the related UART1 peripheral settings must be configured correctly.

   The following example shows how to configure UART1, including select UART1 TXD/RXD pins and module clock frequency.

   The UART1 working baud-rate in SecureISP function is always configured as 115200 bps according to the module clock frequency.

```
static int32_t ConfigureUART1ISP(uint32_t mode)
{
    printf("[Configure UART1]\n");
    switch(mode)
    {
        case 0:
             printf("UART1: RX = PB.6, TX = PB.7\n");
             /* UART1: TX = PB.7, RX = PB.6 */
             SYS->GPB_MFPL &= ~(SYS_GPB_MFPL_PB7MFP_Msk | SYS_GPB_MFPL_PB6MFP_Msk);
             SYS->GPB_MFPL |= (UART1_TXD_PB7 | UART1_RXD_PB6);
             break;

        case 1:
               printf("UART1: TX = PA.9, RX = PA.8\n");
               /* UART1: TX = PA.9, RX = PA.8 */
               SYS->GPA_MFPH &= ~(SYS_GPA_MFPH_PA9MFP_Msk | SYS_GPA_MFPH_PA8MFP_Msk);
               SYS->GPA_MFPH |= (UART1_TXD_PA9 | UART1_RXD_PA8);
               break;

        case 2:
               printf("UART1: TX = PF.0, RX = PF.1\n");
               /* UART1: TX = PF.0, RX = PF.1 */
               SYS->GPF_MFPL &= ~(SYS_GPF_MFPL_PF0MFP_Msk | SYS_GPF_MFPL_PF1MFP_Msk);
               SYS->GPF_MFPL |= (UART1_TXD_PF0 | UART1_RXD_PF1);
               break;

        case 3:
               printf("UART1: TX = PB.3, RX = PB.2\n");
               /* UART1: TX = PB.3, RX = PB.2 */
               SYS->GPB_MFPL &= ~(SYS_GPB_MFPL_PB3MFP_Msk | SYS_GPB_MFPL_PB2MFP_Msk);
               SYS->GPB_MFPL |= (UART1_TXD_PB3 | UART1_RXD_PB2);
               break;

        /* Others */
        default:
               printf("UART1: TX = PA.3, RX = PA.2\n");
               /* UART1: TX = PA.3, RX = PA.2 */
               SYS->GPA_MFPL &= ~(SYS_GPA_MFPL_PA3MFP_Msk | SYS_GPA_MFPL_PA2MFP_Msk);
               SYS->GPA_MFPL |= (UART1_TXD_PA3 | UART1_RXD_PA2);
               break;
    }

    /* Enable UART1 module clock */
    CLK_EnableModuleClock(UART1_MODULE);
    CLK_SetModuleClock(UART1_MODULE, CLK_CLKSEL1_UART1SEL_HIRC, CLK_CLKDIV0_UART1(1));

    printf("\n");

    return __HIRC;
}
```

5. Install IRQ function

   In SecureISP application, it is necessary to enable USBD/UART1 interrupt and use related IRQ function to receive command. After that, the parse and process command function will be execution in the MKROM.

   The following example shows how to call the related SecureISP API in USBD/UART1 IRQ function to receive command from SecureISP Tool.

```
void USBD_IRQHandler(void)
{
   /* Process USBD data */
   BL_ProcessUSBDInterrupt((uint32_t *)g_ISPInfo.pfnUSBDEP, (uint32_t *)&g_ISPInfo,
(uint32_t *)&g_USBDInfo);
}

void UART1_IRQHandler(void)
{
   /* Process UART1 data */
   BL_ProcessUART1Interrupt((uint32_t *)&g_ISPInfo);
}
```

6. Initialize and enable SecureISP function

   After related USBD/UART1 peripheral settings and initial values are configured correctly, secure code can call BL_SecureISPInit API to enable SecureISP function.

   The following partial code explains how to call the related functions to enable SecureISP function in secure code.

```
int main(void)
{
    ……

    /* Clear global variables */
    memset((void *)&g_ISPInfo, 0x0, sizeof(ISP_INFO_T));
    memset((void *)&g_USBDInfo, 0x0, sizeof(BL_USBD_INFO_T));

    /* Configure USBD ISP */
    ConfigureUSBDISP();

    /* Configure UART1 ISP */
    g_ISPInfo.UARTClockFreq = ConfigureUART1ISP(0x0);

    /* Configure time-out time for checking the SecureISP Tool connection */
    g_ISPInfo.timeout = SystemCoreClock;

    /* Initialize and start USBD and UART1 SecureISP function */
    BL_SecureISPInit(&g_ISPInfo, &g_USBDInfo, USB_UART_MODE);

    printf("\nExit SecureISP.\n");

    while(1) {}
}
```

5.3 Vendor Command

Secure code developer can develop their own vendor command to support customize command in SecureISP function.

A vendor function is required for supporting vendor command in secure code and must contain two arguments for processing vendor command via BL_GetVendorData and BL_ReturnVendorData API.

The bellowing code shows an example of vendor function that returns chip ID when server sends the 1st data 0x1000 in vendor command.

```
void Exec_VendorFunction(uint32_t *pu32Buf, uint32_t u32Len)
{
    uint32_t i, au32Data[12];

    memset((void *)au32Data, 0x0, sizeof(au32Data));
    BL_GetVendorData((uint32_t *)&g_ISPInfo, au32Data, pu32Buf);

    printf("Received data are :\n");
    for(i=0; i<(u32Len/4); i++)
        printf("0x%08x, ", au32Data[i]);
    printf("\n\n");

    if(au32Data[0] == 0x1000) // return IDs
    {
        u32Len = 4 * 8;
        au32Data[0] = SYS->PDID;
        au32Data[1] = BL_ReadUID(0);
        au32Data[2] = BL_ReadUID(1);
        au32Data[3] = BL_ReadUID(2);
        au32Data[4] = BL_ReadUCID(0);
        au32Data[5] = BL_ReadUCID(1);
        au32Data[6] = BL_ReadUCID(2);
        au32Data[7] = BL_ReadUCID(3);
        BL_ReturnVendorData(au32Data, u32Len, pu32Buf);
    }
}
```

This following example shows how to configure vendor function in SecureISP before SecureISP function runs.

```
    ……
    /* Configure UART1 ISP */
    g_ISPInfo.UARTClockFreq = u32UartClkFreq;

    /* Configure user's vendor function */
    g_ISPInfo.pfnVendorFunc = Exec_VendorFunction;

    /* Configure time-out time for checking the SecureISP Tool connection */
    g_ISPInfo.timeout = SystemCoreClock;

     /* Initialize and start USBD and UART1 SecureISP function */
     BL_SecureISPInit(&g_ISPInfo, &g_USBDInfo, u32ISPmode);
```

5.4 SecureISP Tool

While SecureISP function is running, user can use Nuvoton NuMicro® SecureISP Tool to communicate with M2351 chip via USBD/UART1 interface.

An example of communication operation using SecureISP USBD mode to program Flash data and configure the chip settings are as follows,

- Open Nuvoton NuMicro® SecureISP Tool

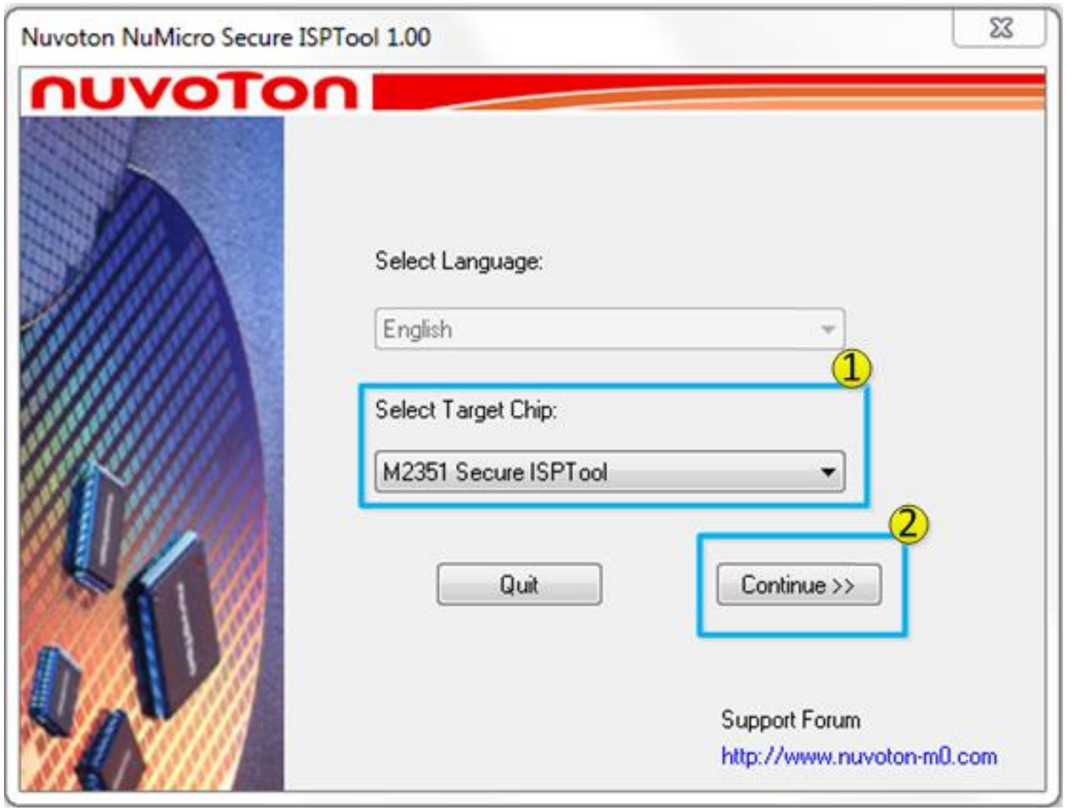


Figure 5-3 Select M2351 SecureISP Tool

- Select “Connection Interface” and start “Connect”

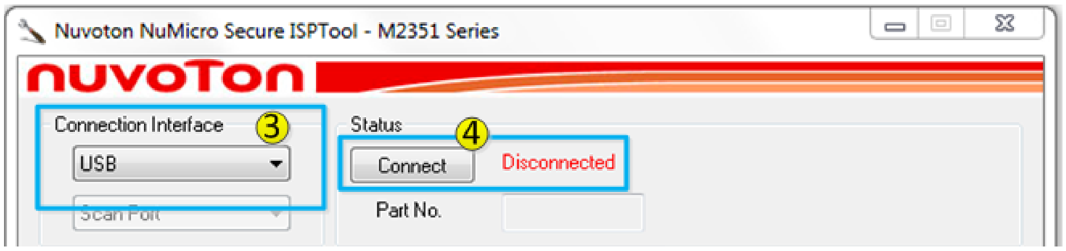


Figure 5-4 Connect with M2351 USB

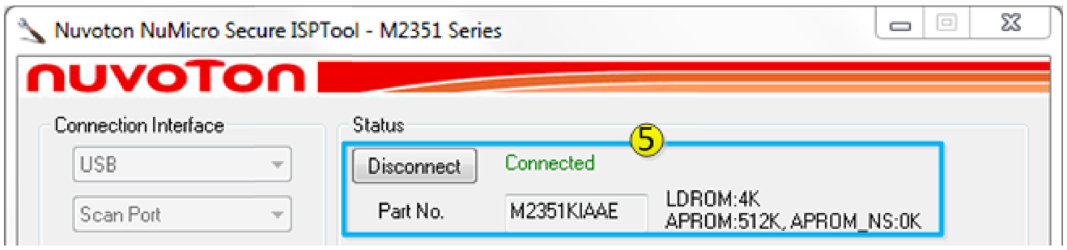


Figure 5-5 Connection and Chip Status

After the connection is successful, SecureISP Tool shows the “Connected” and chip information.

- Load file for programming in “LDROM”, “APROM”, “APROM_NS” and “OTP”

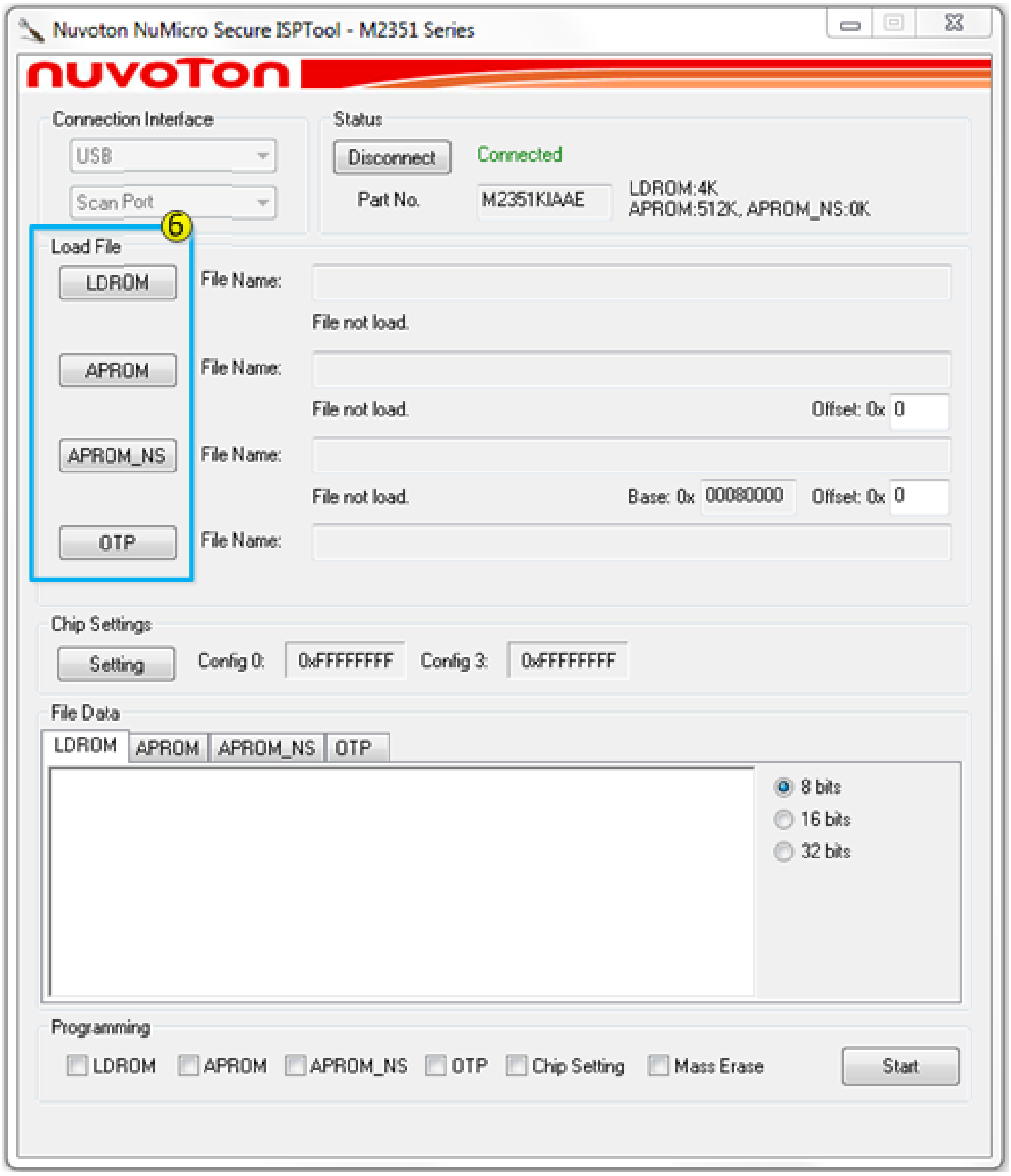
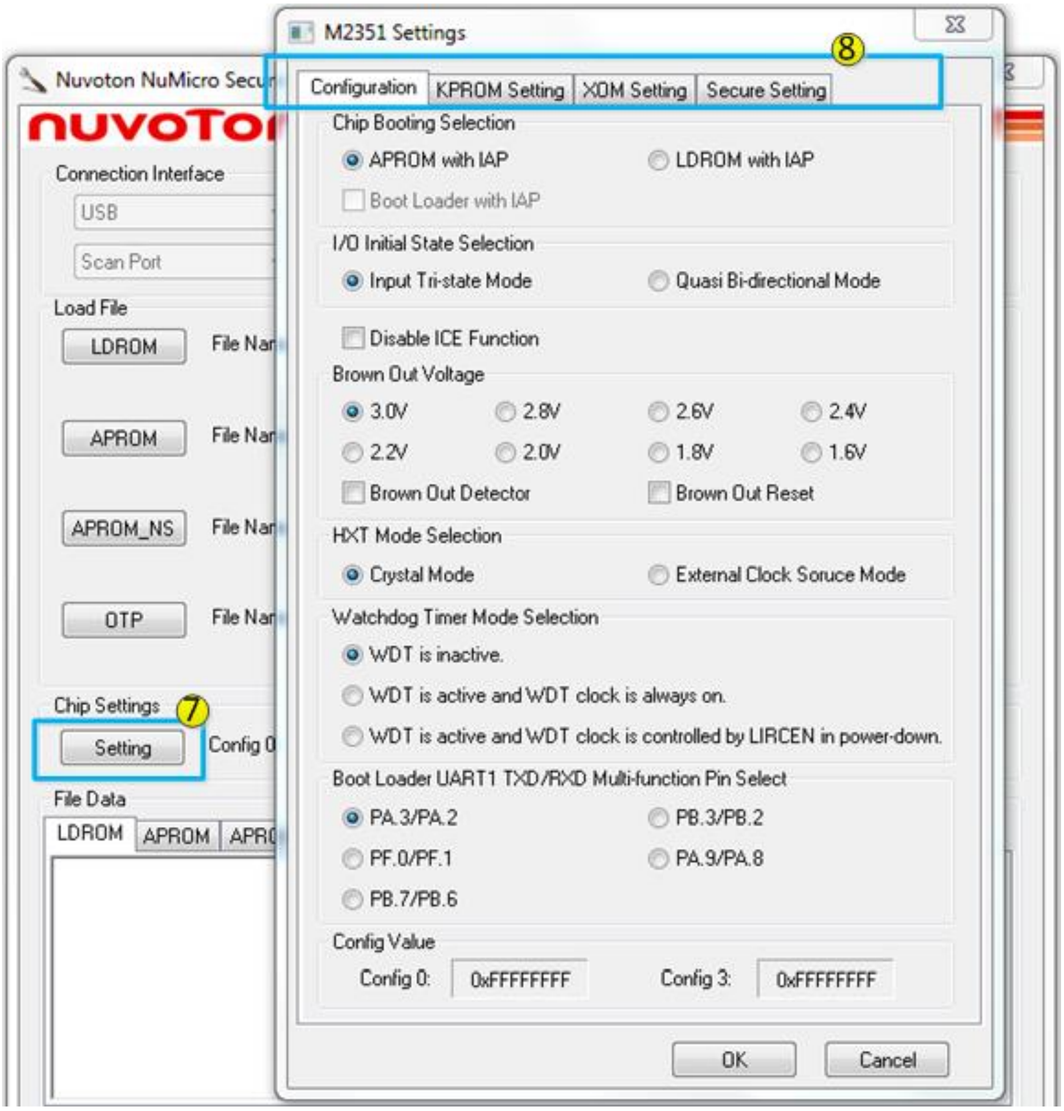


Figure 5-6 Programmable Regions

- Configure Chip Settings in “Setting”



M2351 Settings

Configuration | KPROM Setting | XOM Setting | Secure Setting

Chip Booting Selection
[✓ APROM with IAP] LDROM with IAP
Boot Loader with IAP

I/O Initial State Selection
[✓ Input Tri-state Mode] Quasi Bi-directional Mode

Disable ICE Function

Brown Out Voltage
[✓ 3.0V] 2.8V 2.6V 2.4V
2.2V 2.0V 1.8V 1.6V
Brown Out Detector Brown Out Reset

HXT Mode Selection
[✓ Crystal Mode] External Clock Soruce Mode

Watchdog Timer Mode Selection
[✓ WDT is inactive.]
WDT is active and WDT clock is always on.
WDT is active and WDT clock is controlled by LIRCEN in power-down.

Boot Loader UART1 TXD/RXD Multi-function Pin Select
[✓ PA.3/PA.2] PB.3/PB.2
PF.0/PF.1 PA.9/PA.8
PB.7/PB.6

Config Value
Config 0: 0xFFFFFFFF Config 3: 0xFFFFFFFF

Figure 5-7 Configuration Settings

5.5 SecureISPDemo Sample code

This section describes the SecureISPDemo sample code in detail by function. For all the sample code, refer to “M2351BSP\SampleCode\MKROM\SecureISPDemo”.

- Global g_ISPInfo and g_USBDInfo variables

```
/* Global variables for initializing SecureISP function */
ISP_INFO_T      g_ISPInfo = {0};
BL_USBD_INFO_T  g_USBDInfo = {0};
```

- USBD and UART1 IRQ function

```
void USBD_IRQHandler(void)
```

```
{
    /* Process USBD data */
    BL_ProcessUSBDInterrupt((uint32_t *)g_ISPInfo.pfnUSBDEP, (uint32_t *)&g_ISPInfo,
(uint32_t *)&g_USBDInfo);
}

void UART1_IRQHandler(void)
{
    /* Process UART1 data */
    BL_ProcessUART1Interrupt((uint32_t *)&g_ISPInfo);
}
```

- Configure USBD and UART1 peripheral function
  - USBD configuration:

```
static int32_t ConfigureUSBDISP(void)
{
    printf("[Configure USBD]\n");

    /* Enable Internal RC 48MHz clock */
    CLK_EnableXtalRC(CLK_PWRCTL_HIRC48EN_Msk);

    /* Waiting for Internal RC clock ready */
    CLK_WaitClockReady(CLK_STATUS_HIRC48STB_Msk);

    /* Use HIRC48 as USB clock source */
    CLK_SetModuleClock(USBD_MODULE, CLK_CLKSEL0_USBSEL_HIRC48, CLK_CLKDIV0_USB(1));

    /* Select USBD */
    SYS->USBPHY = (SYS->USBPHY & ~SYS_USBPHY_USBROLE_Msk) | SYS_USBPHY_OTGPHYEN_Msk |
SYS_USBPHY_SBO_Msk;

    /* Enable IP clock */
    CLK_EnableModuleClock(USBD_MODULE);

    /* USBD multi-function pins for VBUS, D+, D-, and ID pins */
    SYS->GPA_MFPH &= ~(SYS_GPA_MFPH_PA12MFP_Msk | SYS_GPA_MFPH_PA13MFP_Msk |
SYS_GPA_MFPH_PA14MFP_Msk | SYS_GPA_MFPH_PA15MFP_Msk);
    SYS->GPA_MFPH |= (SYS_GPA_MFPH_PA12MFP_USB_VBUS | SYS_GPA_MFPH_PA13MFP_USB_D_N |
SYS_GPA_MFPH_PA14MFP_USB_D_P | SYS_GPA_MFPH_PA15MFP_USB_OTG_ID);

    printf("\n");
    return 0;
}
```

- UART1 configuration:

```
static int32_t ConfigureUART1ISP(uint32_t mode)
{
    printf("[Configure UART1]\n");

    switch(mode)
    {
        case 0:
            printf("UART1: RX = PB.6, TX = PB.7\n");
            /* UART1: TX = PB.7, RX = PB.6 */
            SYS->GPB_MFPL &= ~(SYS_GPB_MFPL_PB7MFP_Msk | SYS_GPB_MFPL_PB6MFP_Msk);
            SYS->GPB_MFPL |= (UART1_TXD_PB7 | UART1_RXD_PB6);
            break;

        case 1:
            printf("UART1: TX = PA.9, RX = PA.8\n");
            /* UART1: TX = PA.9, RX = PA.8 */
            SYS->GPA_MFPH &= ~(SYS_GPA_MFPH_PA9MFP_Msk | SYS_GPA_MFPH_PA8MFP_Msk);
            SYS->GPA_MFPH |= (UART1_TXD_PA9 | UART1_RXD_PA8);
            break;

        case 2:
            printf("UART1: TX = PF.0, RX = PF.1\n");
            /* UART1: TX = PF.0, RX = PF.1 */
            SYS->GPF_MFPL &= ~(SYS_GPF_MFPL_PF0MFP_Msk | SYS_GPF_MFPL_PF1MFP_Msk);
            SYS->GPF_MFPL |= (UART1_TXD_PF0 | UART1_RXD_PF1);
            break;

        case 3:
            printf("UART1: TX = PB.3, RX = PB.2\n");
            /* UART1: TX = PB.3, RX = PB.2 */
            SYS->GPB_MFPL &= ~(SYS_GPB_MFPL_PB3MFP_Msk | SYS_GPB_MFPL_PB2MFP_Msk);
            SYS->GPB_MFPL |= (UART1_TXD_PB3 | UART1_RXD_PB2);
            break;

        /* Others */
        default:
            printf("UART1: TX = PA.3, RX = PA.2\n");
            /* UART1: TX = PA.3, RX = PA.2 */
            SYS->GPA_MFPL &= ~(SYS_GPA_MFPL_PA3MFP_Msk | SYS_GPA_MFPL_PA2MFP_Msk);
            SYS->GPA_MFPL |= (UART1_TXD_PA3 | UART1_RXD_PA2);
            break;
    }

    /* Enable UART1 module clock */
    CLK_EnableModuleClock(UART1_MODULE);
    CLK_SetModuleClock(UART1_MODULE, CLK_CLKSEL1_UART1SEL_HIRC, CLK_CLKDIV0_UART1(1));

    printf("\n");

    return __HIRC;
}
```

- Vendor command function

```
void Exec_VendorFunction(uint32_t *pu32Buf, uint32_t u32Len)
{
    uint32_t i, au32Data[12];
    uint32_t au32Sign[8];

    memset((void *)au32Data, 0x0, sizeof(au32Data));
    BL_GetVendorData((uint32_t *)&g_ISPInfo, au32Data, pu32Buf);

    printf("Received data are :\n");
    for(i=0; i<(u32Len/4); i++)
        printf("0x%08x, ", au32Data[i]);
    printf("\n\n");

    if(au32Data[0] == 0x1000) // return IDs
    {
        u32Len = 4 * 8;
        au32Data[0] = SYS->PDID;
        au32Data[1] = BL_ReadUID(0);
        au32Data[2] = BL_ReadUID(1);
        au32Data[3] = BL_ReadUID(2);
        au32Data[4] = BL_ReadUCID(0);
        au32Data[5] = BL_ReadUCID(1);
        au32Data[6] = BL_ReadUCID(2);
        au32Data[7] = BL_ReadUCID(3);
        BL_ReturnVendorData(au32Data, u32Len, pu32Buf);
    }
}
```

● Configure customize vendor function and then enable USBD/UART1 SecureISP

```
int main(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    /* Init System, peripheral clock and multi-function I/O */
    SYS_Init();

    /* Init UART for printf */
    UART_Init();

    printf("\n\nCPU @ %d Hz\n", SystemCoreClock);
    printf("+------------------------------------------------+\n");
    printf("|    Initial USBD/UART1 SecureISP Sample Code    |\n");
    printf("+------------------------------------------------+\n\n");

    /* Clear global variables */
    memset((void *)&g_ISPInfo, 0x0, sizeof(ISP_INFO_T));
    memset((void *)&g_USBDInfo, 0x0, sizeof(BL_USBD_INFO_T));

    /* Configure USBD ISP */
    ConfigureUSBDISP();

    /* Configure UART1 ISP */
    g_ISPInfo.UARTClockFreq = ConfigureUART1ISP(0x0);

    /* Configure user's vendor function */
    g_ISPInfo.pfnVendorFunc = Exec_VendorFunction;

    /* Configure time-out time for checking the SecureISP Tool connection */
    g_ISPInfo.timeout = SystemCoreClock;

    printf("\n[Hit any key to enter SecureISP function]\n\n");
    getchar();
    printf("Enter SecureISP......\n");

    /* Initialize and start USBD and UART1 SecureISP function */
    BL_SecureISPInit(&g_ISPInfo, &g_USBDInfo, USB_UART_MODE);

    printf("\nExit SecureISP.\n");

    while(1) {}
}
```

5.6 SecureISP API List

The following section describes the MKROM SecureISP API in detail. For the MKROM header file, refer to “M2351BSP\Library\StdDriver\inc\mkromlib.h”.

5.6.1 BL_SecureISPInit

- int32_t BL_SecureISPInit (ISP_INFO_T *pISPInfo, BL_USBD_INFO_T *pUSBDInfo, E_ISP_MODE mode)
- Parameters**:**

- [in] pISPInfo	The ISP information data buffer address.
- [in] pUSBDInfo	USB data buffer for SecureISP USB mode.
- [in] mode	Operation mode. Possible options are: USB_MODE, UART_MODE, USB_UART_MODE

- Return**:** Return process status and exit SecureISP.

Executing this API will initialize USB or UART1 SecureISP function. User can use SecureISP Tool to communicate with target chip.

5.6.2 BL_ProcessUSBDInterrupt

- int32_t BL_ProcessUSBDInterrupt (uint32_t *pfnEPTable, uint32_t *pInfo, uint32_t *pUSBDInfo)
- Parameters**:**

- [in] pfnEPTable	Starting address to store EP callback function.
- [in] pInfo	The ISP information data buffer address.
- [in] pUSBDInfo	USB data buffer for SecureISP USB mode.

- Returns**:**
  - -1**:** Execute API in non-secure code.
  - 0**:** Process USBD interrupt event success.

This API is used to process USBD command and should be called in USBD_IRQHandler().

5.6.3 BL_ProcessUART1Interrupt

- int32_t BL_ProcessUART1Interrupt (uint32_t *pInfo)
- Parameters**:**

- [in] pInfo	The ISP information data buffer address.

- Returns**:**
  - -1**:** Execute API in non-secure code.
  - 0**:** Process UART1 interrupt event success.

This API is used to process UART1 command and should be called in UART1_IRQHandler().

5.6.4 BL_GetVendorData

- int32_t BL_GetVendorData (uint32_t *pInfo, uint32_t *pu32Data, uint32_t *pu32Buf)
- Parameters**:**

- [in] pInfo	The ISP information data buffer address.
- [out] pu32Data	Data buffer to store vendor data. Maximum buffer size is 44 bytes.
- [in] pu32Buf	Internal used data buffer address.

- Returns**:**
  - 0**:** Success.
  - -1**:** Invalid command packet.
  - -2**:** Not in vendor function.

This API is used to get the vendor data and should be called in the vendor function.

5.6.5 BL_ReturnVendorData

- int32_t BL_ReturnVendorData (uint32_t *pu32Data, uint32_t u32Len, uint32_t *pu32Buf)
- Parameters**:**

- [in] pu32Data	Data buffer to store response data.
- [in] u32Len	Data buffer length, maximum size is 40 bytes.
- [in] pu32Buf	Internal used data buffer address.

- Returns**:**
  - 0**:** Success.
  - -1**:** Invalid command packet.
  - -2**:** Not in vendor function.

This API is used to return vendor data to server and should be called in the vendor function.

6 Conclusion

As Flash Memory Controller (FMC) is always secure, the non-secure code cannot directly program the FMC to perform ISP functions to access Flash memory. Therefore, the non-secure code needs to call the MKROM non-secure callable library to perform necessary ISP functions to access non-secure flash region.

The Crypto function is also provided in the MKROM Library. The secure code can directly call the Crypto function to execute cryptography operations to reduce the code size.

The SecureISP function in MKROMLib provides an easy way to create secure encrypted channel between a client and a server via USB or UART interface. It also provides rich features for in-system-programming usage. The vendor command of SecureISP could maximize the flexibility for customizing user’s ISP firmware.

Revision History

Date	Revision	Description
2018.08.09	1.00	1. Initially issued.