Table of Contents

Prerequisites

If this lab is done in sequence with the other labs, then there are no prerequisites. All of the pre requisites should have been done in the "Getting Started" steps.
If you have not gone through these steps please go back to the Getting Started lab guide and completed the download and setup of the IDE and SDK.

1. Download the latest MCUXpresso IDE for your platform https://mcuxpresso.nxp.com. It is required that MCUXpresso IDE 11.1.1 or newer is installed.
2. Download the latest SDK for your platform https://mcuxpresso.nxp.com

Objectives

In this lab, you will learn:

- What is the secure Trusted Execution Environment on the RT685.
- How to use the CM33 Trust Zone.
- Non Secure ↔ Secure environment transitions.
- Identify the main error sources that trigger secure hard faults.
- Use a Secure GPIO to mask reading an IO-pin status with a non-secure environment.

Hardware

- NXP i.MX RT600 Evaluation Kit - MIMXRT685-EVK
- Micro-USB cable

Trusted Execution Environment TEE high level description

In this lab you will gain a basic understanding of the benefits of using the secure functions in the RT685 MCU. Specifically, the use of the Trusted Execution Environment TEE.

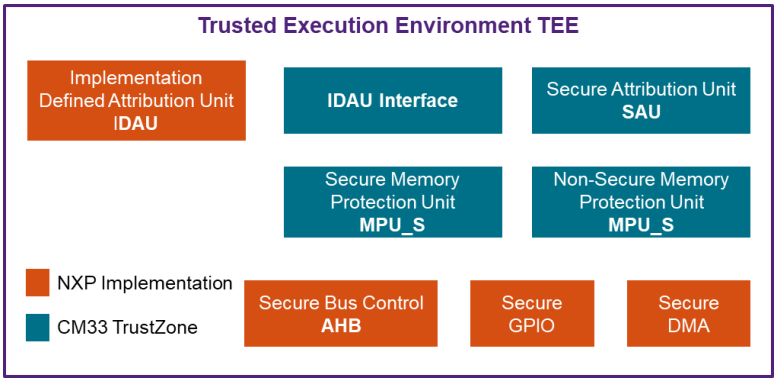


Figure 1. *TEE Block Diagram*

TrustZone®

TrustZone is an optional CM33 security extension available in RT685. This feature was designed to provide a foundation for improved system security. The MCU has Secure (**S**) and Non-Secure (**NS**) states.

Secure (**S**)

Memory and peripherals that are only accessible by S-software or S-masters. Illegitimate accesses that are made by NS software to S memory are blocked and raise an exception. A CPU in S-state only executes S-code, S-data is only accessible through S-code.

Non-Secure Callable (**NSC**)

Particular type of S location that is permitted to hold a Secure Gateway (**SG**) instruction to enable software to transition from NS to S state.

A portion of S-memory can be marked as Non-Secure Callable (**NSC**)memory which the main functionality is to provide cross-domain calls. NSC memory regions contain tables of small branch veneers or entry points.

Having veneer tables like entry points inside the NSC memory means that the S binary is never exposing to the NS world.

Non-Secure (**NS**)

Addresses used for memory and peripherals accessible by all software running on the device.

Secure Attribution Unit (**SAU**)

Controls the security state of a memory region, the RT685 provides 8 configurable SAU regions. The memory partition into Secure and Non-Secure regions is independent of the security state that the processor executes in.

NXP implementation

Implementation Defined Attribution Unit (**IDAU**)

The software can define up to eight SAU regions. If these regions are not enough, the application can use a device-specific controller logic, such as IDAU.

NXP IDAU implementation of ARM® TrustZone for core0 involves using address bit 28 to divide the address space into potential S and NS regions. Address bit 28 is not decoded in memory access hardware. Each physical has two potential addresses a S-address with bit 28 set and a NS-address with b28 clear. For example, Shared RAM start address is 0x3000_0000 in the S-world (b28=1) or 0x2000_0000 in the NS world (b28=0).

Figure 2 shows how the RT685 IDAU divides the memory into S / NS using the address b28. There are two alias locations for every memory or peripheral address.

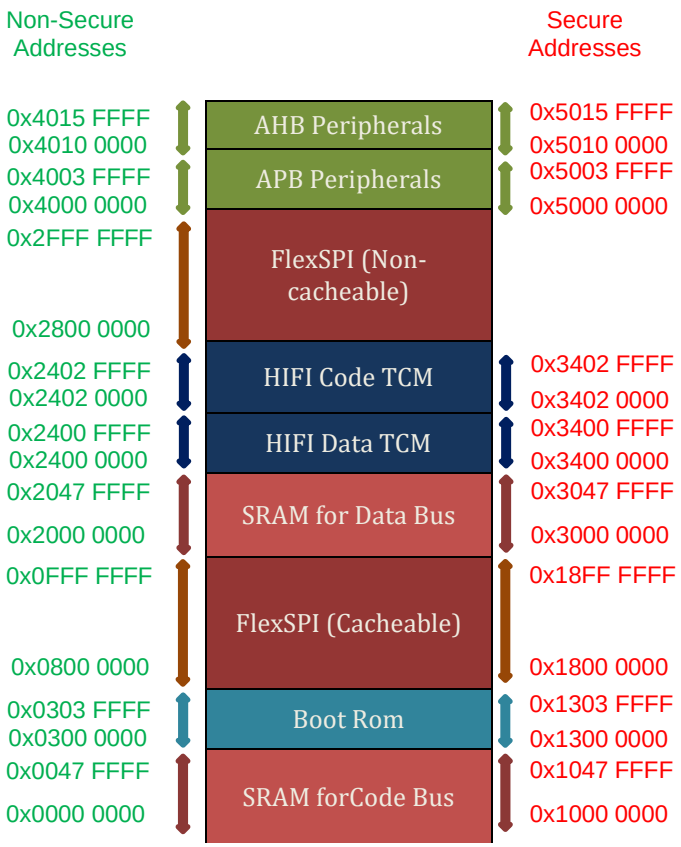


Figure 2. *RT685 Memory Map*

The memory partition into Secure and Non-Secure regions is independent of the security state that the processor executes in.

Secure bus controller

The RT685 use a matrix of secure bus controller modules to manage the data flow in the MCU. A combination of a Peripheral Protection Checker (**PPC)**, the Memory Protection Checker (**MPC**) and the Master Security Wrapper (**MSW**).

The **secure AHB** controller is a module on RT685 that allows programming security attributes for all PPCs, MPCs, or MSWs. Secure AHB controller provides a second protection layer for safe, trusted execution at system-level. With secure AHB each peripheral is capable of configuring individual access rules.

Secure DMA

For TrustZone, a DMA can be configured in S-mode for secure thread.

Secure GPIO

Secure GPIO has the same functions as normal GPIO. However, the access rules to this Secure GPIO for different security levels are configured through the Secure AHB controller which can only be accessed in Secure state.

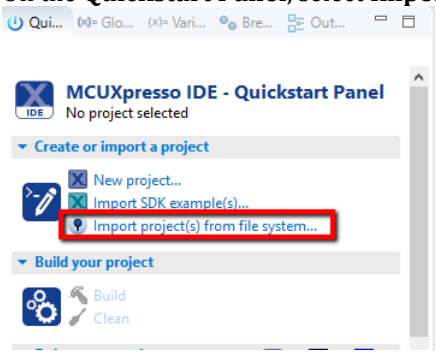
Running the lab

Import the MCUXpresso secure lab projects from file system

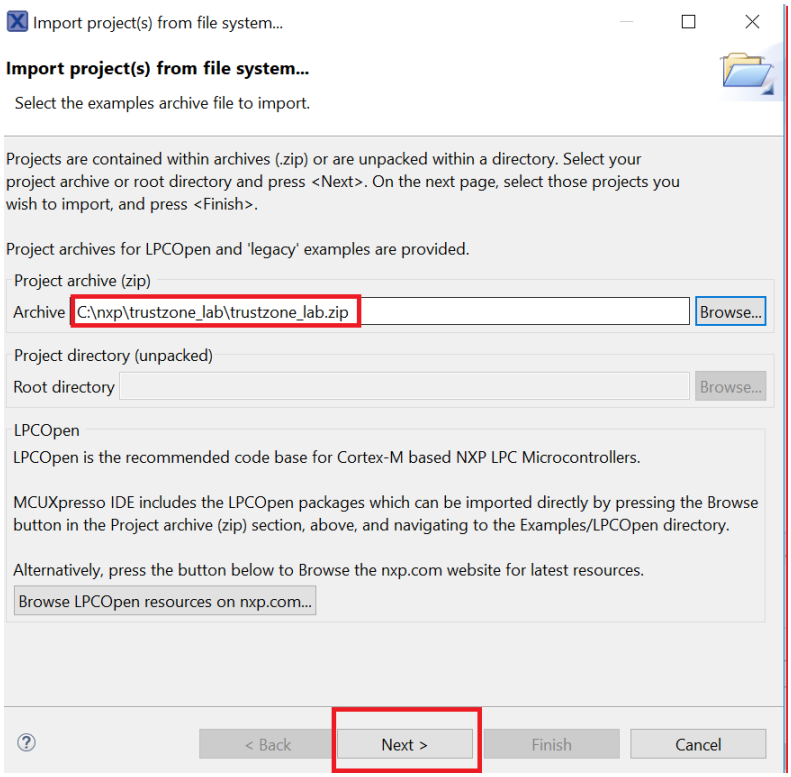
The trustzone lab zip file uses two projects: secure_ns and secure_s. Both projects use the RT685 CM33, **secure_ns** is the non-secure application, and **secure_s** contains all the TEE secure software and configuration.

1. Download the trustzone_lab.zip folder from the Download Day community from the same folder as this document.
2. Open MCUXpresso IDE 11.1.1 or newer.

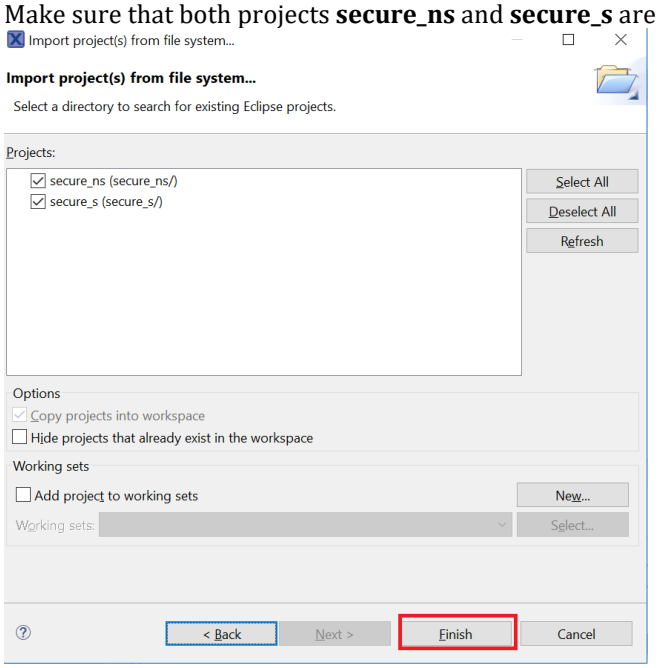
3. On the **Quickstart Panel**, select **Import project(s) from file system…**



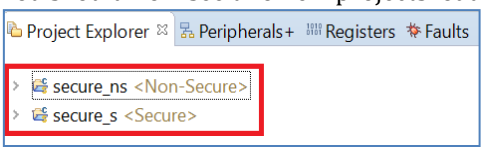
4. At the **Project archive (zip)** Browse… for the previously downloaded **trustzone_lab.zip** folder, then click **Next.**



5. Make sure that both projects **secure_ns** and **secure_s** are selected, then click **Finish.**



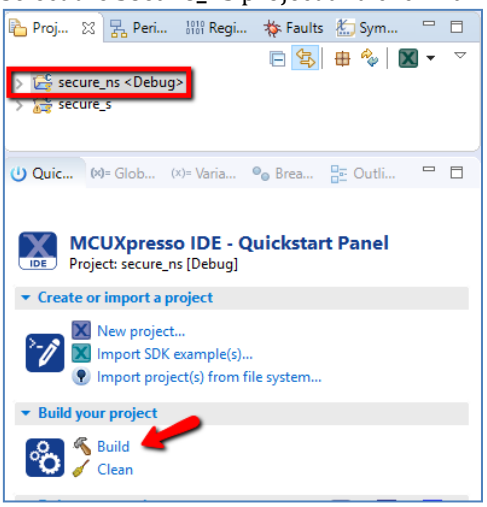
6. You should now see two new projects loaded into the workspace as shown below.



Build

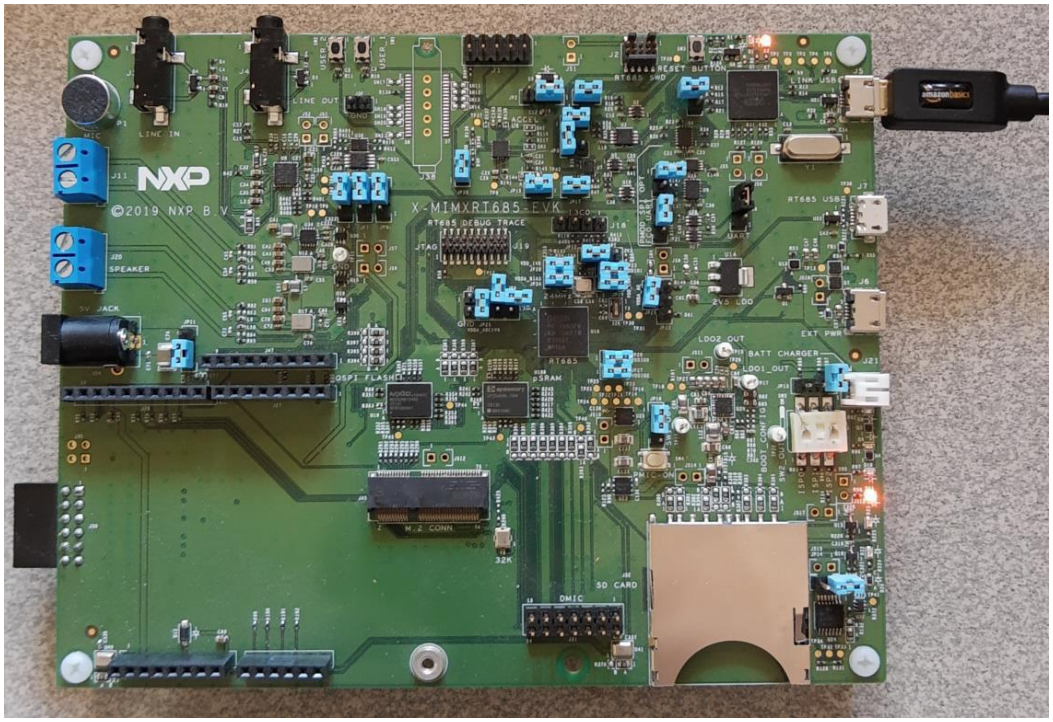
7. Now it's time to compile.
Select the **secure_s** project and click **Build.**

Wait until secure_s finish building without any errors.
Select the **secure_ns** project and click **Build.**

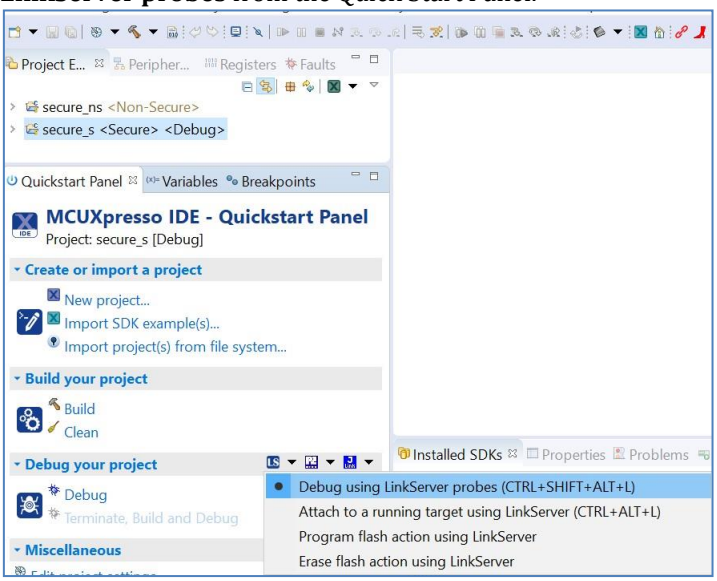


Debug

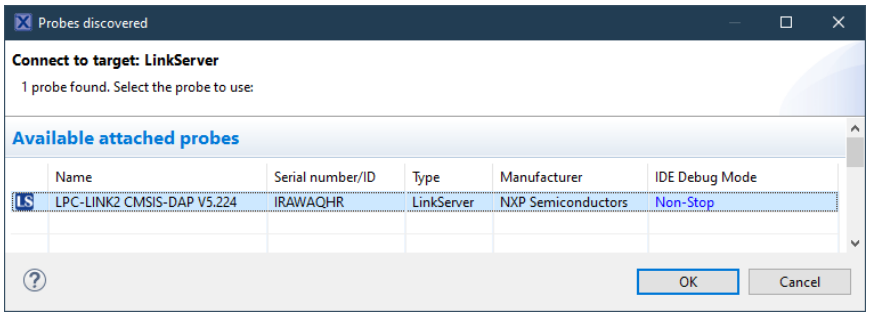
8. Connect the Debug Link port from your RT600 EVK to your PC using a micro-USB cable.



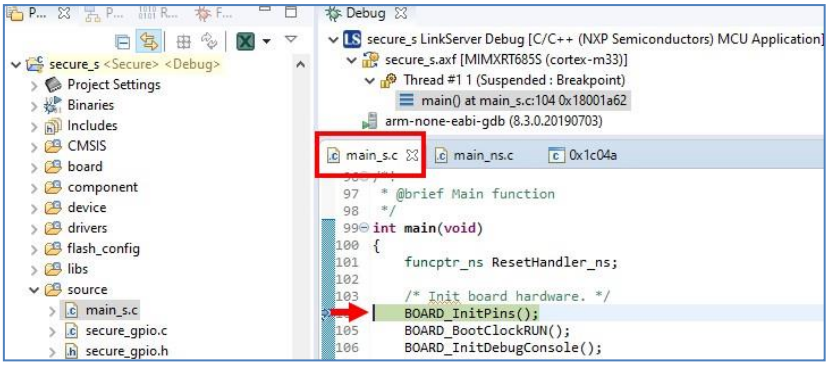
9. To program the secure project. Select the **secure_s** project, click on **Debug using LinkServer probes** from the Quick Start Panel.



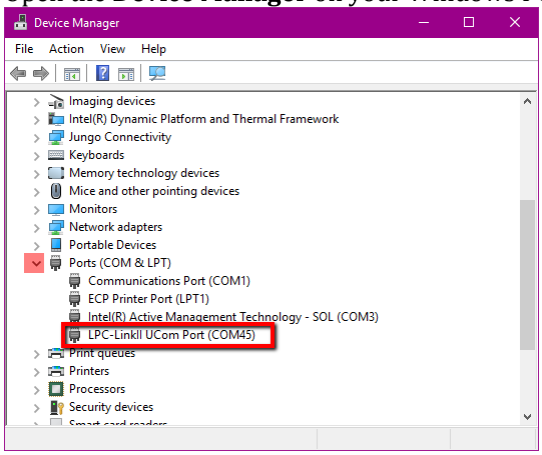
10. Select your probe and click **OK** to start programming your board.



11. Wait until the program is successfully downloaded.



12. Open the **Device Manager** on your Windows PC and identify the COM port of your board.



13. Open a Terminal emulator software (TeraTerm, PuTTY) and connect the COM port using the following settings.
    `115200 baudrate, 8 data bitsm, No parity, One stop bit, No flow control`

Lab instructions

This lab divides into:

TrustZone configuration

S ⟶ NS ⟶ S transition
Force secure fault exceptions… the wrong way of doing things
Secure GPIO

TrustZone configuration

To create a TrustZone application is necessary to divide the software into two projects one for Secure called secure_s and a second one for Non-Secure called secure_ns. One of the main differences between these two projects resides in the memory configuration.

Figure 3 shows how both projects use their corresponding IDAU address without overlapping the physical locations. The veneer table needs to use a SAU region to be defined as a NSC.

secure_ns :

Type	Name	Alias	Location	Size	Driver
Flash	QSPI_FLASH	Flash	0x8100000	0x100000	MIMXRT600_FlexSPI_B_MXIC_OPI.cfx
RAM	SRAM	RAM	0x20180000	0x80000	
RAM	USB_RAM	RAM2	0x40140000	0x4000	

secure_s:

Type	Name	Alias	Location	Size	Driver
Flash	QSPI_FLASH	Flash	0x18000000	0x100000	MIMXRT600_FlexSPI_B_MXIC_OPI_S.cfx
RAM	SRAM	RAM	0x30140000	0x40000	
RAM	USB_RAM	RAM2	0x50140000	0x4000	

Figure 3. *secure_ns & secure_s memory configuration*

The secure project configures TrustZone after every RESET. The file trustzone❼tzm_config.c contains one function **`BOARD_InitTrustZone`**, which configures complete TrustZone environment. It includes SAU, MPU's, AHB secure controller and some TrustZone related registers from System Control Block. This function is called from **`SystemInitHook`** function, it means during system initialization and prior jumping into main.

SAU configuration

SAU Region	Name	Type	Start	End
0	CODE_SRAM_NS	NS	0x000C 0000	0x0001 3FFF
1	CODE_FLASH_NS	NS	0x8100 0000	0x0820 0000
2	CODE_SRAM_NSC	NSC	0x100B FE00	0x100B FFFF
3	CODE_FLASH_NSC	NSC	0x180F FE00	0x180F FFFF
4	DATA_SRAM_NS	NS	0x2018 0000	0x201F FFFF
5	PERIPH_NS	NS	0x40000000	0x401FFFFF

AHB MPC

Secure Flash – 0x08000000 to 0x08100000 – 1 MB
Secure SRAM for Code – 0x20080000 to 0x200BFFFF – 255 kB
Secure SRAM for data – 0x20140000 to 0x2017FFFF – 255 kB

AHB PPC

Secure and privileged user access only: SYSCON, IOPCTL, FLEXCOMM0, SECURE GPIO

S → NS → S transition

S → NS

The S project sets NS main stack, vector table and then ends jumping to the NS project.

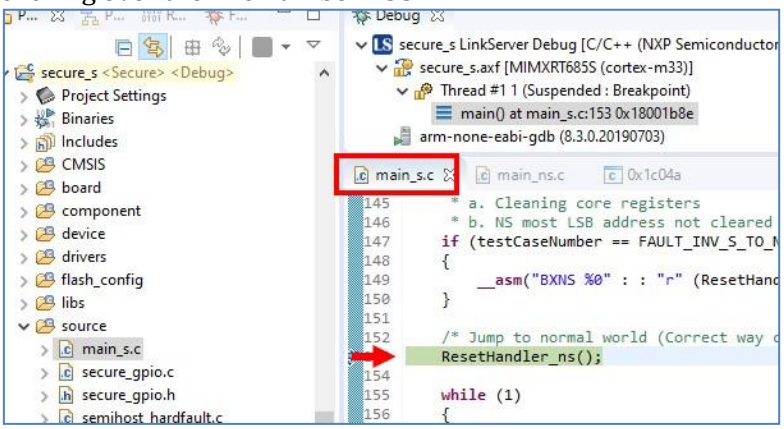
```
/* typedef for non-secure callback functions */
typedef void (*funcptr_ns) (void) __attribute__((cmse_nonsecure_call));
```

```
/* Set non-secure main stack (MSP_NS) */
__TZ_set_MSP_NS(*((uint32_t *)(NON_SECURE_START)));

/* Set non-secure vector table */
SCB_NS->VTOR = NON_SECURE_START;

/* Get non-secure reset handler */
ResetHandler_ns = (funcptr_ns)(*((uint32_t *)((NON_SECURE_START) + 4U)));
```

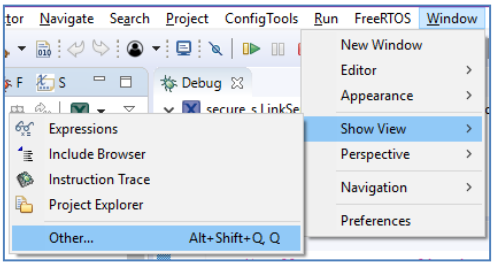
```
/* Jump to normal world (Correct way compare to Test 1) */
ResetHandler_ns();

while (1)
{
    /* This point should never be reached */
}
```

S → NS running instructions

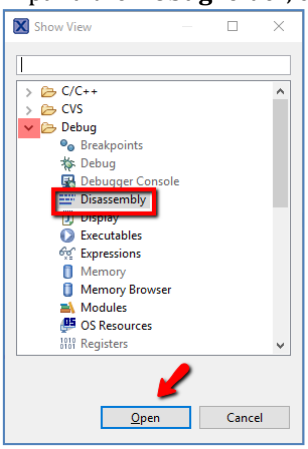
14. Go back to the MCUXpresso IDE.
15. Open **secure_s → source → main_s.c** look for line **153** and set a breakpoint by double clicking over the line number 153.



16. At the menu tool bar select **Window → Show View → Other ...**

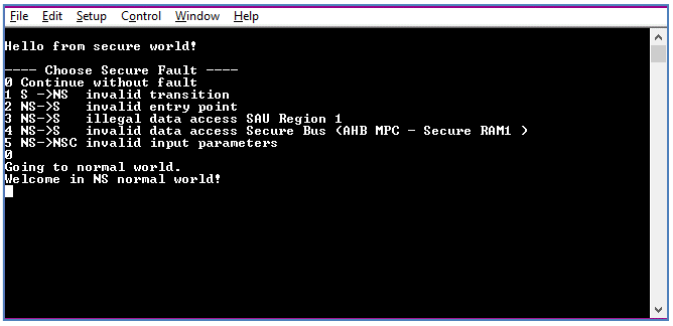


17. Expand the **Debug** folder, choose **Disassembly** then click **Open.**

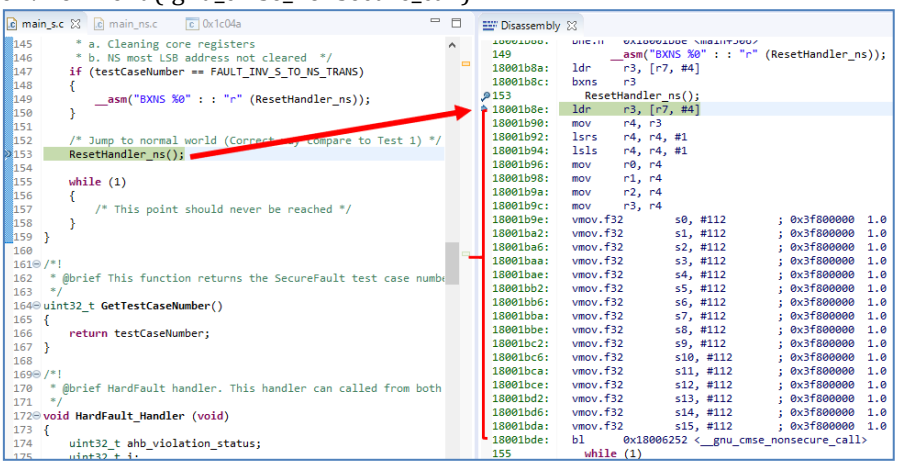


18. Click Resume or press F8.

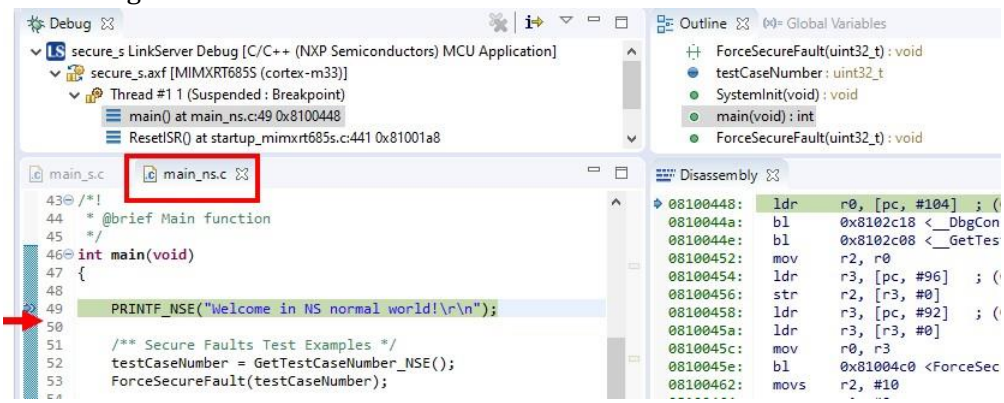
19. Go to the terminal then write a **0** to not force secure faults exceptions. Code execution stops in the **secure_s** project breakpoint.



20. Analyze all the assembly instruction prior jumping to the NS Reset Handler. The assembly code pushes all registers, zero all registers, finally ends switching to non-secure environment (gnu_cmse_nonsecure_call).



21. Remove the breakpoint by double clicking over it at **main_s.**
22. Open **secure_ns ❼ source ❼ main_ns.c** look for line **49** and set a breakpoint by double clicking over the line number.



23. Click Resume or press **F8.**
24. Code execution stops in the **secure_ns** project breakpoint.
25. Remove the breakpoint by double clicking over the line number **49.**

Case 0: NS ⟶ S

After RESET the TrustZone AHB PPC configuration sets FLEXCOMM0 at secure_s ⟶ trustzone → tzm_config.c with secured access only. FLEXCOMM0 is the UART handling PRINTFs. Therefore, just secure software is capable of printing things at the terminal.

```
/* Set FLEXCOMM0 as secure */
AHB_SECURE_CTRL->AHB_PERIPH0_SLAVE_RULE0 = AHB_SECURE_CTRL_AHB_PERIPH0_SLAVE_RULE0_FLEXCOMM0_RULE(0x3U);
```

If you try to PRINTF from the NS project, a hard fault event will occur because you are trying to access to a peripheral marked as S from an NS environment. To prevent the hard fault event, the application needs a safe way to call S code, this gets done through the NSC Secure Gateway SG API.

Figure 4 shows how to do a NS ⟶ S transition to PRINTF. veneer_table.c defines all the S entry functions exported to the NS world. Both projects S and NS require the veneer_table.h header file. Besides sharing the veneer_table.h, the secure_ns creates a connection to the secure_s by adding the secure_s_CMSE_lib.o library as a secure_ns linker input.

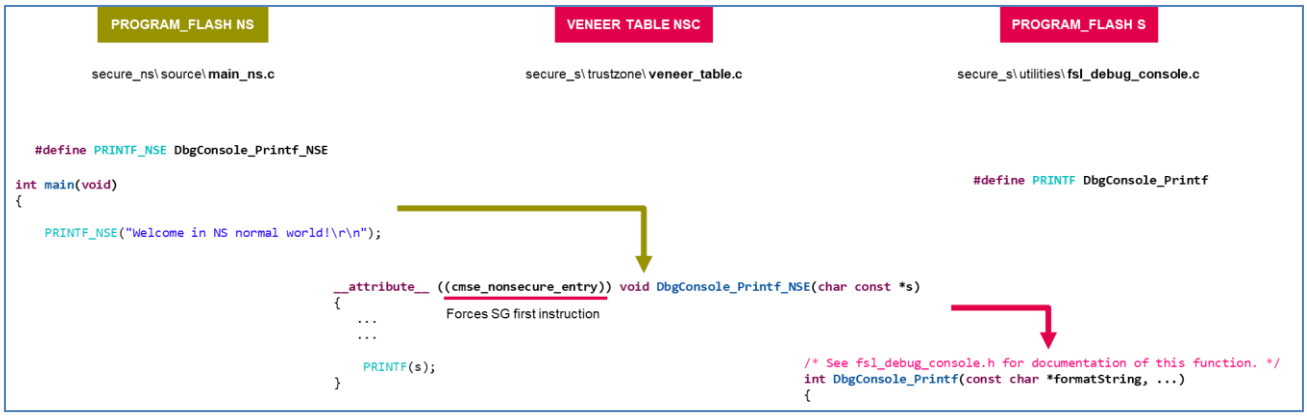


Figure 4. *PRINTF from NS*

Force secure hard fault events

Case 1. S ⟶ NS invalid transition

In this example, direct access to NS RESET is used to jump into the normal world secure_s ⟶ source ⟶ main_s.c.

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

```
/* Get non-secure reset handler */
ResetHandler_ns = (funcptr_ns)(*((uint32_t *)((NON_SECURE_START) + 4U)));
```

```
/* Test 1 S->NS invalid transition
 * Direct jump to NS ResetHandler without
 * a. Cleaning core registers
 * b. NS most LSB address not cleared  */
if (testCaseNumber == FAULT_INV_S_TO_NS_TRANS)
{
    __asm("BXNS %0" : : "r" (ResetHandler_ns));
}

/* Jump to normal world (Correct way compare to Test 1) */
ResetHandler_ns();
```

Issues

a. Secure core registers are not clear previous the jump causing a potential data leak.
b. The most LSB of NS address is NOT cleared triggering a secure fault.

Workaround

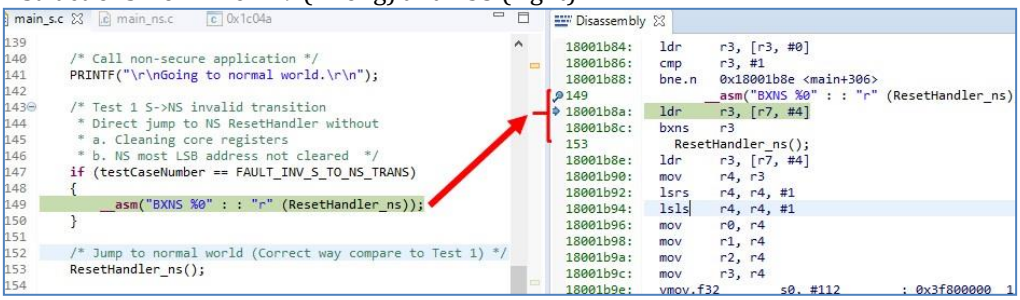
To solve both issues use the `__attribute__((cmse_nonsecure_call))`. When this attribute is added to your code, the compiler will:

a. Clear all used secure core registers to avoid a data leak.
b. Clear LSB address bit.
c. Jump to the NS address using the BXNS instruction.

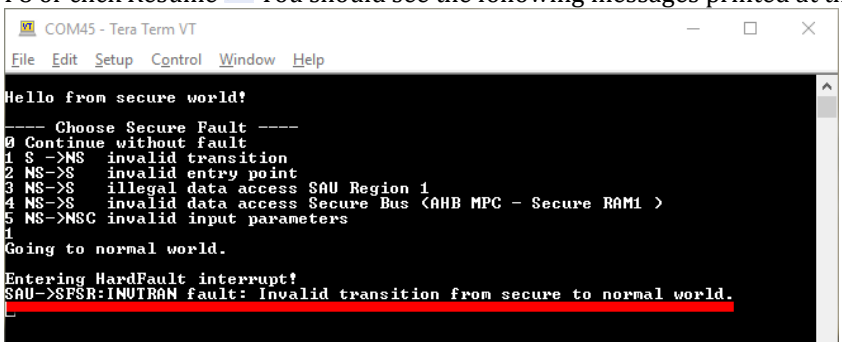
Running S ⟶ Invalid transition

26. Start the Debug session @ **secure_s ⟶ main_s.c.**
27. Set a new breakpoint @ **secure_s ⟶ main_s.c** line **149.**
28. F8 or click Resume.

29. Go to the terminal then type **1**. The software will stop in line 149. Compare disassembly instructions from line 149 (wrong) and 153 (right).



30. F8 or click Resume.
31. Software will stop in hard fault handler.
32. F8 or click Resume You should see the following messages printed at the terminal.

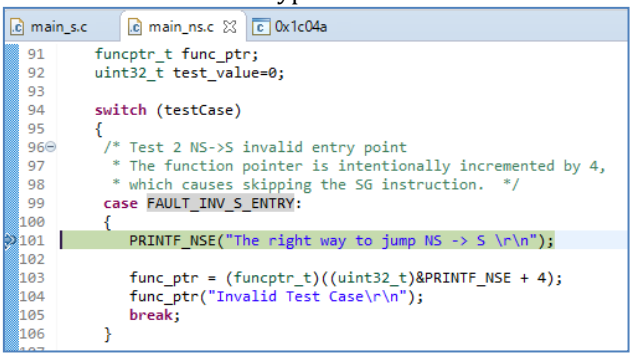


Case 2. NS ⟶ S invalid entry point

The PRINTF_NS entry point is intentionally increased by 4. Therefore, the Secure Gateway SG instruction is skipped causing secure fault event due to an illegal entry point to S world.

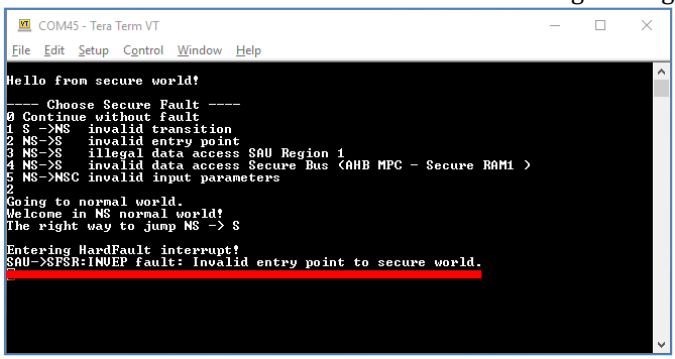
Running S ⟶ invalid transition

33. Set a new breakpoint @ **secure_ns** ⟶ **main_ns.c** on case FAULT_INV_S_ENTRY (line **101**).
34. **F8** or click Resume.
35. Go to the terminal then type **2**.



36. **F8** or click Resume.

37. Software will stop in hard fault handler.
38. **F8** or click Resume You should see the following messages printed at the terminal.

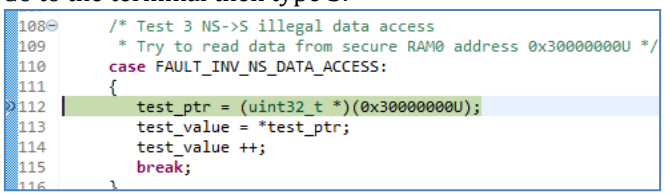


Case 3. NS ⟶ S illegal data access

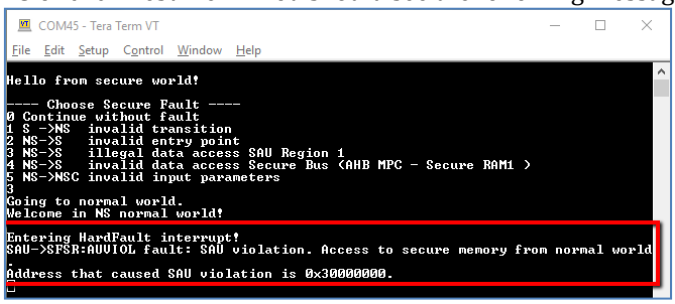
Invalid data access from normal world. In this example the pointer is set to address 0x30000000. This address has secure attribute (see SAU settings). If data is read from this address, the secure fault is generated. In NS world, the application doesn't have access to secure memory.

Running NS ⟶ S illegal data access

39. Set a new breakpoint @ **secure_ns ⟶ main_ns.c** line **112.**
40. **F8** or click Resume.
41. Go to the terminal then type **3**.



42. **F8** or click Resume.
43. Software will stop in hard fault handler.
44. **F8** or click Resume You should see the following messages printed at the terminal.



Case 4. NS ⟶ S invalid data access

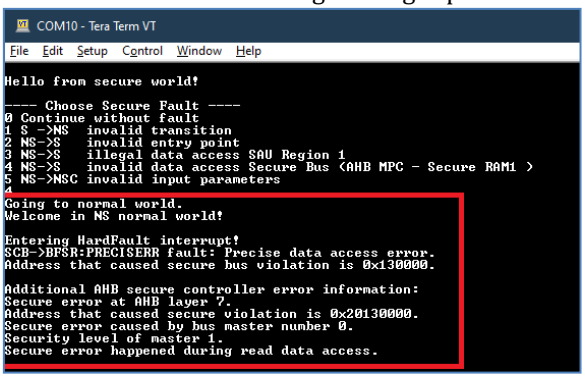
The pointer is set to address 0x00130000. This address has non-secure attribute in SAU but it has secure attribute in AHB secure controller. If data is read from this address, the data bus error is generated. Compare to test #3, this error is caught by the AHB secure controller, not by SAU, because in SAU this address is non-secure so the access from normal world is correct from SAU perspective.

Running NS ⟶ S invalid data access

45. Set a new breakpoint @ **secure_ns ⟶main_ns.c** line **124.**
46. **F8** or click Resume.
47. Go to the terminal then type **4.**

```
118     /* Test 4 NS->S invalid data access
119      * Try to read data from RAM1 address 0x00130000U
120      * SAU attribute = NS
121      * AHB Secure Controller = S */
122     case FAULT_INV_NS_DATA2_ACCESS:
123     {
124         test_ptr = (uint32_t *)(0x00130000U);
125         test_value = *test_ptr;
126     }
```

48. **F8** or click Resume.
49. Software will stop in hard fault handler.
50. **F8** or click Resume You should see the following messages printed at the terminal.
51. You should see the following messages printed at the terminal.



Case 5. NS ⟶ NSC invalid input parameter

The input parameter is set to address 0x30000000. This address has secure attribute (see SAU settings) This secure violation is not detected by secure fault, since the input parameter is used by secure function in secure mode.

So this function has access to whole memory. However every entry function should check source of all input data in order to avoid potential data leak from secure memory. The correctness of input data cannot be checked automatically. This has to be check by software using Test Target TT instructions.

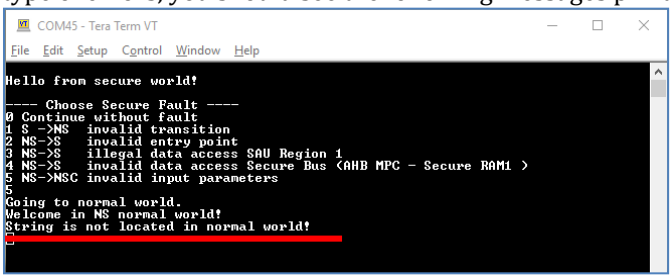
Running NS ⟶ NSC invalid input parameter

52. Set a new breakpoint @ **secure_ns ⟶ main_ns.c** line **134.**
53. **F8** or click Resume.
54. Go to the terminal then type **5.**

```
128     /*Test 5 NS->NSC invalid input parameter
129      * This function call uses secure RAM0 address 0x30000000U,
130      * but will not generate a secure fault, since the input is used by a S function
131      * NSC uses TT instruction to detect this type of error */
132     case FAULT_INV_INPUT_PARAMS:
133     {
134         PRINTF_NSE((char *)(0x30000000U));
135         break;
```

55. **F8** or click Resume.

56. Software will NOT stop in hard fault handler, but the PRINTF_NSE function checks for this type of errors, you should see the following messages printed at the terminal.



57. Click **Clean Up Debug.**

Secure GPIO

Due to the architecture of normal GPIO, all digital IO pins states are readable through the NS GPIO module from the GPIO read path, independent of the IOCON setting. As a result, there is a possibility of leaking information from S resources in the NS world.

For example, when an IO during IOCON initialization has configured the pin as Secure & PINT for GPIO is made secure, which means that this pin interrupt event is only visible to the S-world. However, in this case, the IO pin states can still be monitored by the NS world through normal GPIO read path. Making it very easy for the NS application to know when an S interrupt event occurs.

Using Secure GPIOs can prevent this type of information leakage, each Secure GPIO comes with a SEC_GPIO_MASK that allows or prevents GPIO readings.

Secure GPIO Flow Diagram

Figure 5 shows how the S & NS software runs.

1. S configures SW1 (SW2 in EVK) as a Secure GPIO with Secure PINT
2. NS can read SW1 (SW2 in EVK) and SW2 (SW1 in EVK) as normal GPIOs
3.

a) NS world polls SW1 and SW2 using normal GPIO registers
if SW1 == 1 (Not pressed) --> Blue LED ON
if SW1 == 0 (Pressed) --> Blue LED OFF

b)NS world monitor status of SecureGPIOMaskGPIO0_10 (SW1) through NSC:
if SecureGPIOMaskGPIOSW1 == 0 --> {Green LED ON
NS World **can't** read SW1 GPIO (Switch 2 in EVK) pin}
if SecureGPIOMaskGPIOSW1 == 1 --> {Green LED OFF
NS World **can** read SW1 GPIO (Switch 2 in EVK) pin}

c) S world monitor status of SecureGPIO SW2 (SW1 in EVK)
if SW2 == 1 (Not pressed) --> SecureGPIOMaskGPIOSW1 = 1: NS world can read SW1 port
if SW2 == 0 (Pressed) --> SecureGPIOMaskGPIOSW1 = 0: NS world can't read SW1 port

d) S world detects when SecureSW1 == 0 (Pressed), goes into the PINT event and prints something in the terminal.

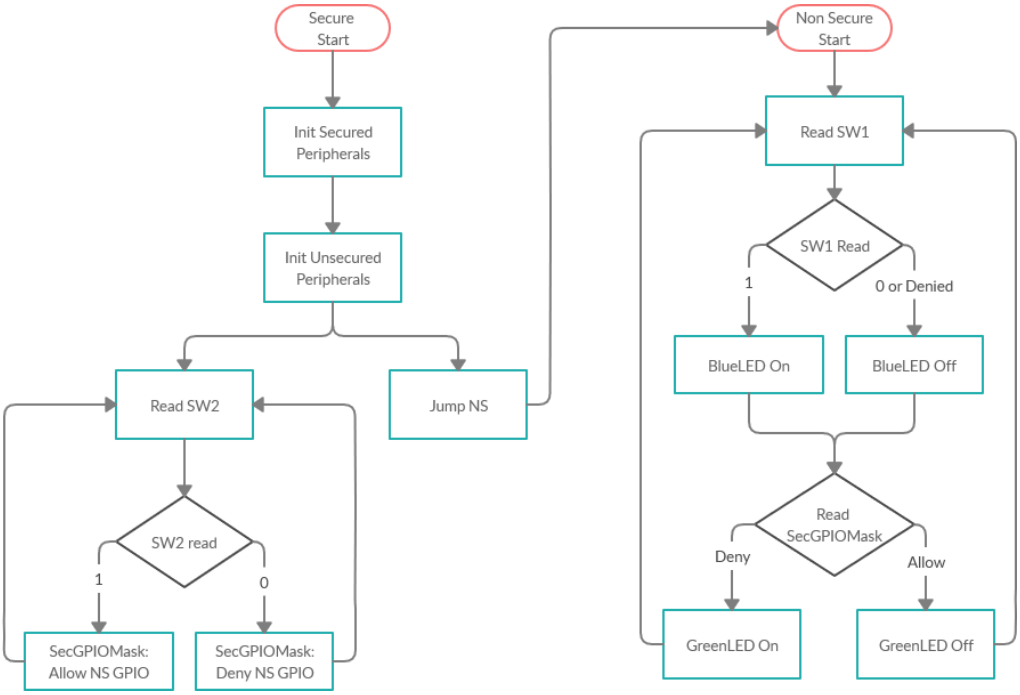
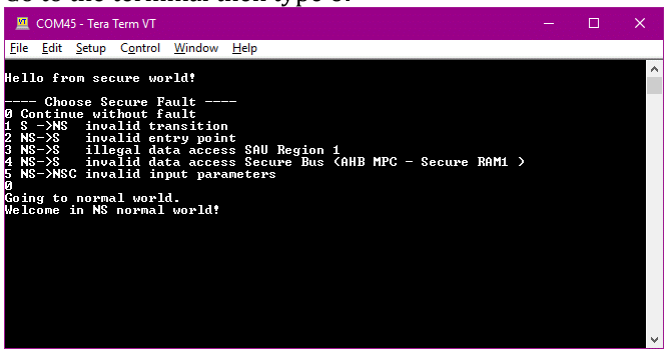


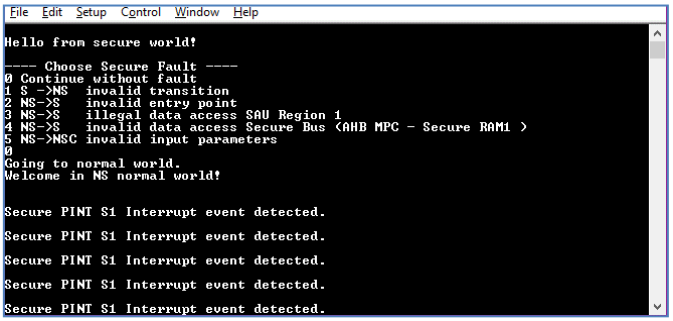
Figure 5. *Secure GPIO Flow Diagram*

Running Secure GPIO

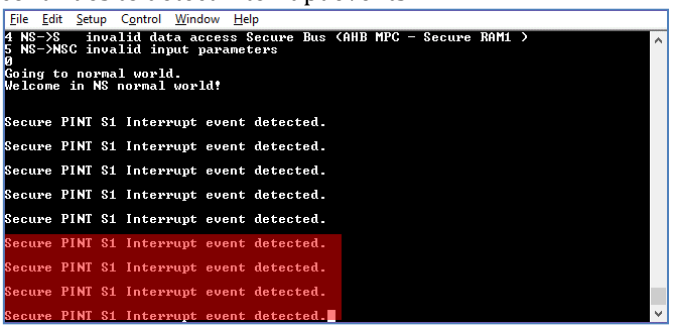
58. Press the **RESET** push button.
59. Go to the terminal then type **0.**



60. Blue LED is ON because NS normal GPIO was able to read the pin input value. Push several times SW1, you'll see LED OFF while pressing SW1. The green LED is OFF indicating that secure GPIO masking is disabled and SW1 read is allowed by secure as well non-secure world. The S application prints *"Secure PINT SW1 Interrupt event detected."* every time SW1 gets pressed.



61. Press SW2 once, Green LED turns ON indicating only secure world can read SW1 and Blue LED goes off because the secure GPIO masking is enabled. But if you push SW1 the S world continues to detect interrupt events.



62. Press SW2 again, Blue LED goes ON and Green LED turns off because the secure GPIO masking is disabled.

Additional resources

RT6xx User's Manual https://www.nxp.com/docs/en/user-guide/UM11147.pdf
TrustZone for cortex M-Arm https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m