AN5365

SAM L11 Security Reference Guide

Introduction

This document explains the different security features available on the Microchip SAM L11 microcontroller that fulfill the following security requirements of the most embedded systems:

- Software security, based on isolation of, and restricted access to certain data, resources, and code
- Physical security with anti-tampering and encrypted memory interfaces
- Communication security based on encryption, decryption, authentication algorithm, and strong key management storage and provisioning

The following sections provide programming examples that illustrates the SAM L11 key features to fulfill these requirements.

Table of Contents

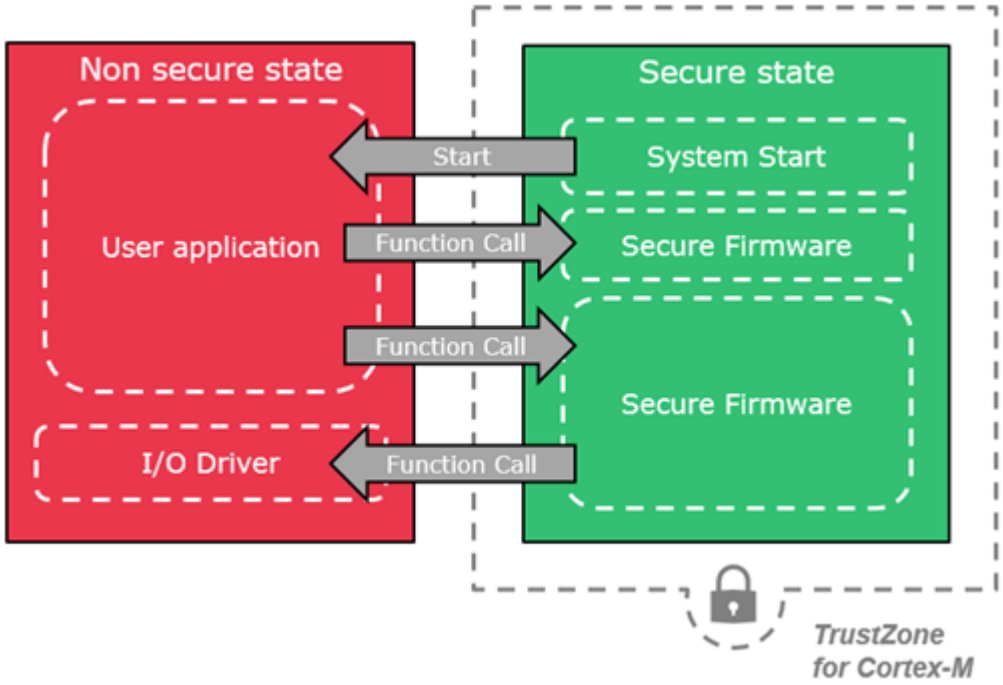
1. TrustZone for ARMv8-M Implementation in SAM L11

The central security element for the Microchip SAM L11 microcontroller is the implementation of the ARM® TrustZone® for an ARMv8-M device. The TrustZone technology is a System-on-Chip (SoC) and MCU system-wide approach to security that enables Secure and Non-Secure code to run on a single MCU.

TrustZone for an ARMv8-M device is based on a specific hardware that is implemented in the Cortex®-M23 core, which is combined with a dedicated Secure instructions set. It allows the creation of multiple software security domains that restricts access to selected memory, peripherals, and I/O to trusted software without compromising the system performances.

The main goal of the TrustZone for a ARMv8-M device is to simplify security assessment of a deeply embedded device. The principle behind the ARM® TrustZone® for a ARMv8-M embedded software application is illustrated in the following figure.

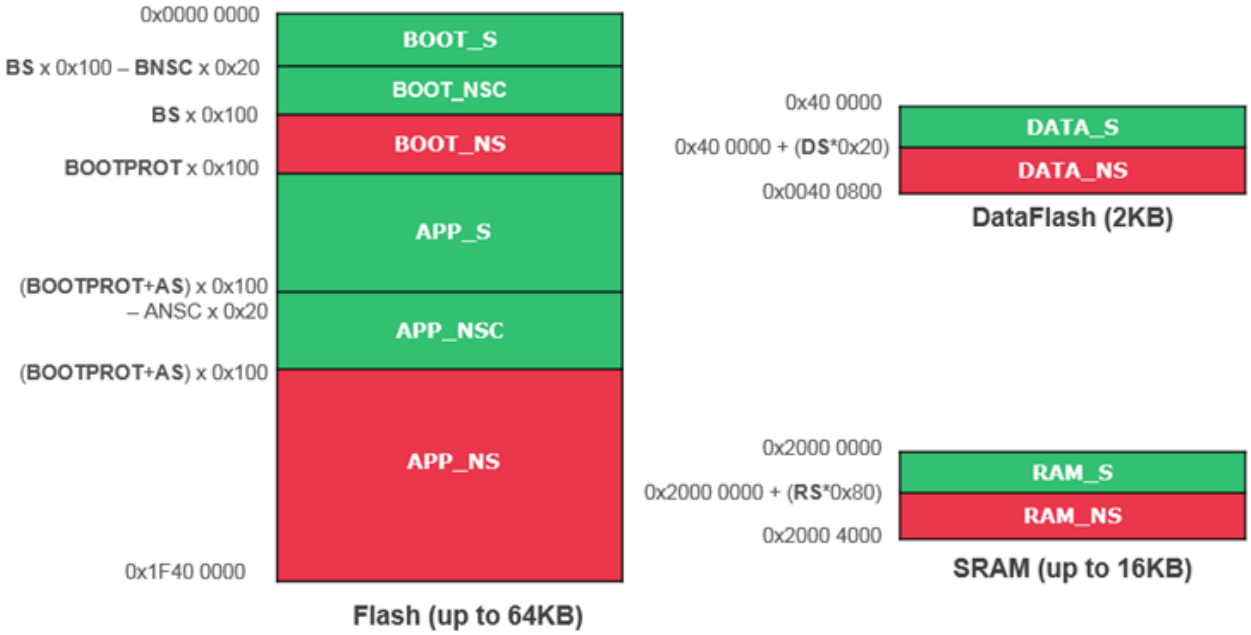
Figure 1-1. Standard Interactions Between Secure and Non-Secure States



1.1 Memory and Peripheral Security Attribution

To differentiate and isolate Secure code from Non-Secure code, the SAM L11 memory is partitioned into ten different memory regions as represented in the following figure. Each region size is configurable using dedicated NVM fuses, such as BS, BNSC, BOOTPROT, AS, ANSC, DS and RS.

Figure 1-2. SAM L11 Memory Partitioning



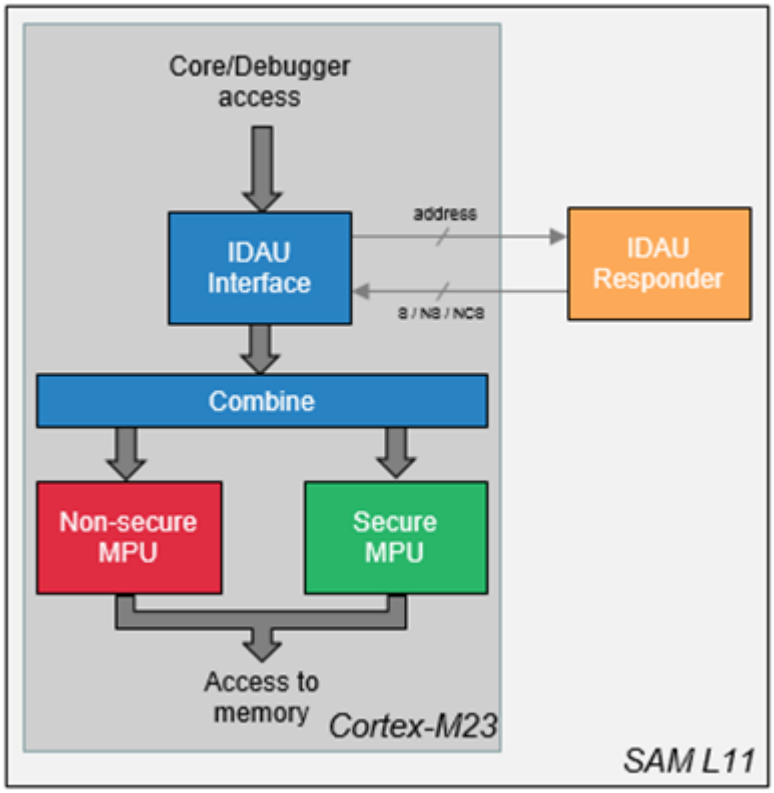
- Non-Secure (NS): Non-Secure addresses are used for memory and peripherals, which are accessible by all software that is running on the device.
- Secure (S): Secure addresses are used for memory and peripherals, which are accessible only by secure software.
- Non-Secure Callable (NSC): NSC is a special type of Secure memory location. It allows software to transition from a Non-secure to a Secure state.

The security attribute of each region will define the security state of the code stored in this region.

1.2 Secure and Non-Secure Code Execution

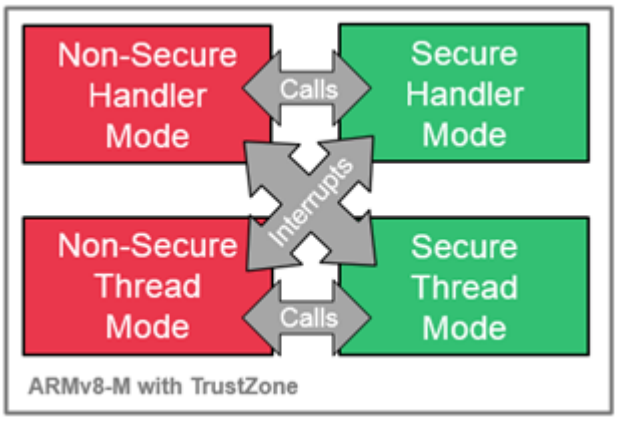
In the Cortex-M23 Core the security management is done with the IDAU interface. The IDAU interface controls access to executing specific instructions based on the current core security state and the address of the instruction.

Figure 1-3. IDAU Interface and Memory Accesses



Thanks to this implementation, a simple function call or an interrupt processing results in to be a branch to a specific security state as illustrated in the following figure. This allows for efficient calling by avoiding any code and execution overhead.

Figure 1-4. ARMv8-M With TrustZone States Transition



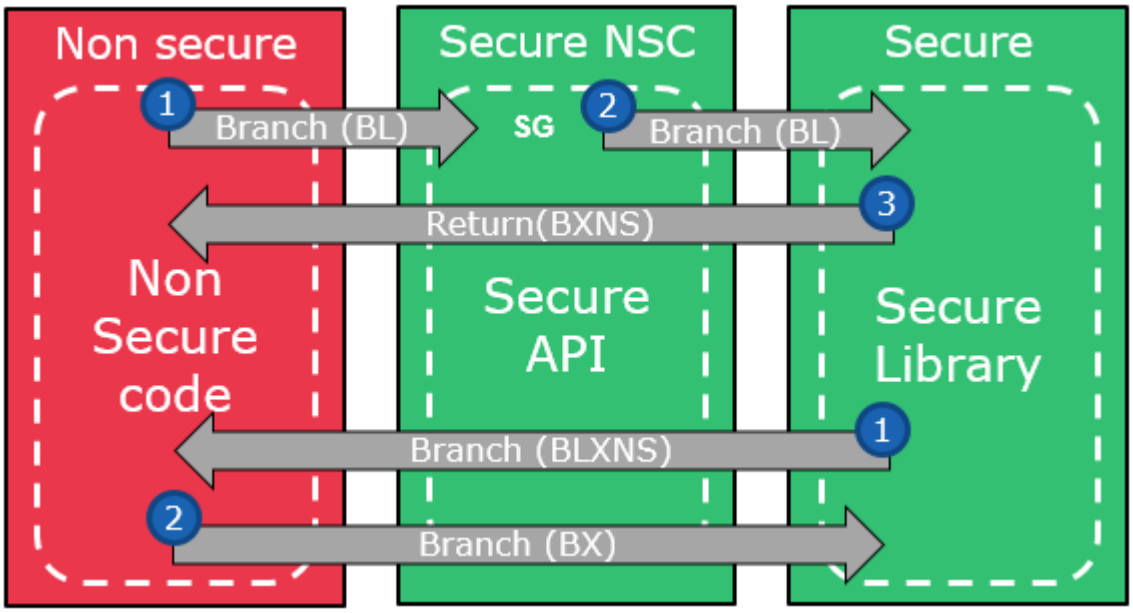
1.2.1 Secure and Non-Secure Functions Call

To prevent Secure code and data from being accessed from a Non-Secure state, Secure code must meet several requirements. The responsibility for meeting these requirements is shared between the MCU architecture, software architecture and the toolchain configuration. A set of Secure instructions are available to preserve and protect secure register values during the state transition handling. The Compiler Security Extension (CMSE) provided by ARM allows the user to manage the use of these new ARMv8-M Secure instruction sets on the Secure software side. Secure and Non-Secure function call mechanisms

are shown in the following figure. The following are key Secure instructions to handle for Secure or Non-Secure function calls.

- Secure Gateway (SG): Used for switching from a Non-Secure to a Secure state at the first instruction of a Secure entry point.
- Branch with exchange to Non-Secure state (BXNS): Used by the Secure software to branch, or return to the Non-secure program.
- Branch with link and exchange to Non-Secure state (BLXNS): Used by the Secure software to call the Non-Secure functions.

Figure 1-5. ARMv8-M Secure/Non-Secure Function Calls

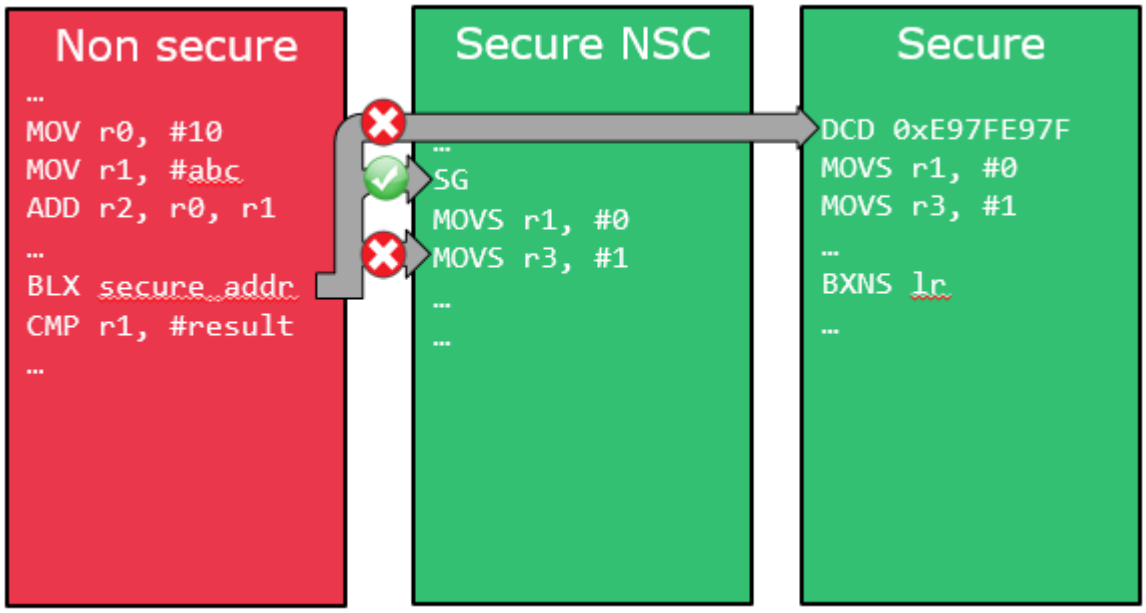


A direct API function call from the Non-Secure to the Secure software entry points is allowed only if the first instruction of the entry point is a SG, and is in a Non-Secure callable memory location, as shown in the following figure. The use of the special instructions (BXNS and BLXNS) are also required to branch to Non-Secure code.

The Secure gateway decouples the addresses of the Secure gateways (in NSC regions) from the rest of the Secure code. All the project Secure gateways are expected to be placed in the NSC memory, where all other code from the secure executable is expected to be placed in the secure memory regions. This limits the amount of code that can potentially be accessed by the non-secure state. This placement is under the control of the secure developer.

Any attempts to access secure regions from the non-secure code, or a mismatch between the code that is executed and the security state of the system results in a HardFault exception. See the following figure.

Figure 1-6. Security State and Call Mismatch



1.2.2 Secure and Non-Secure Interrupts Handling

The Cortex-M23 (ARMv8-M architecture) uses the same exception stacking mechanism as the ARMv7-M architecture, where a subset of the core registers is stored automatically into the stack (hardware context saving). This permits immediate execution of the interrupt handler without the need to perform a context save in the software. ARMV8-M extends this mechanism to provide enhanced security based on two different stack pointers (a Secure stack pointer and a Non-Secure stack pointer).

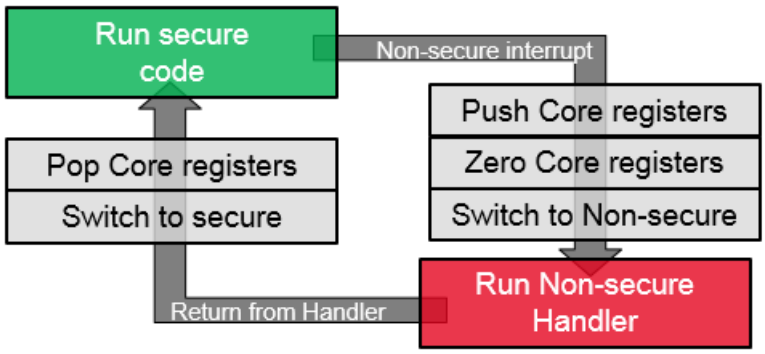
According to the priority settings configured in the Nested Vector Interrupt Controller (NVIC), Secure code execution can interrupt Non-Secure code execution, and Non-Secure code can interrupt Secure code execution. The NVIC registers at the core level are duplicated. This allows two vector table definitions, one for Secure and another for Non-Secure.

At product start-up, all interrupts are mapped by default to the Secure world (Secure vector table). Specific CMSIS functions accessible in the Secure world, allocate each interrupt vector to a non-secure handler (declared in Non-Secure vector table).

If the Secure code is running when a higher priority Non-Secure interrupt arrives, the core pushes all the register content into a dedicated secure stack. Registers are then zeroed automatically to prevent any information being leaved, and the core executes the non-secure exception handler.

When the Non-Secure handler execution is finished, the hardware recovers all the registers from the secure stack automatically. This mechanism is managed in hardware and does not require any software intervention. This allows a Secure handover from running Secure code to a Non-Secure interrupt handler, and returning to running Secure code.

Figure 1-7. Cortex-M 23 Interrupt Mechanism



2. Application Deployment Considerations

The SAM L11 system architecture combined with TrustZone for the ARMv8-M is set to three different access levels to the chip resources. Those levels depend on the Debug Access Level setting (DAL) of the target SAM L11 device.

2.1 Debug Access Level (DAL) and Chip Erase

The SAM L11 has three configurable debug access levels (DAL), which restrict programming and debug access to Secure and Non-Secure resources in the system.

- DAL2: Debug access with no restrictions in terms of memory and peripheral accesses
- DAL1: Access is limited to the Non-Secure memory regions. Secure memory region accesses are forbidden.
- DAL0: No Access is authorized except with a debugger using the Boot ROM Interactive mode

Note: Refer to the "Boot ROM" chapter of the "SAM L11 Data Sheet" for more details on Boot Interactive Mode.

DAL is combined with three key protected ChipErase commands that provide three levels of NVM erase granularity. The ChipErase command is used to increase the DAL level without compromising code security (that is, erase of the code before changing to higher DAL level).

Figure 2-1. ChipErase Commands

Important: The ChipErase commands (CMD_CE0, CMD_CE1, CMD_CE2 and CMD_CHIPERASE) are only issued using the Boot ROM Interactive mode.

Figure 2-2. SAM L11 Configurable ChipErase Key Fuses

Offset	Bit Pos.	Name			
0x00	7:0	Reserved			
0x01	15:8	BS			
0x02	23:16	Reserved	BNSC		
0x03	31:24	BOOTOPT			
0x04	39:32	BOOTPROT			
0x05	47:40	Reserved			
0x06	55:48	Reserved		BCREN	BCWEN
0x07	63:56	Reserved			
0x08-0x0B	95:64	BOCORCRC			
0x0C-0x0F	127:96	ROMVERSION			
0x10-0x1F	255:128	CEKEY0			
0x20-0x2F	383:256	CEKEY1			
0x30-0x3F	511:384	CEKEY2			
0x40-0x4F	639:512	CRCKEY			
0x50-0x6F	895:640	BOOTKEY			
0x70-0xDF	1791:896	Reserved			
0xE0-0xFF	2047:1792	BOCORHASH			

The DAL, ChipErase commands, and key fuses can be programmed to a SAM L11 target device using the Atmel Studio 7 (AS7) Device Programming Utility, as shown in image below.

Figure 2-3. ChipErase Commands Under AS7 Device Programming

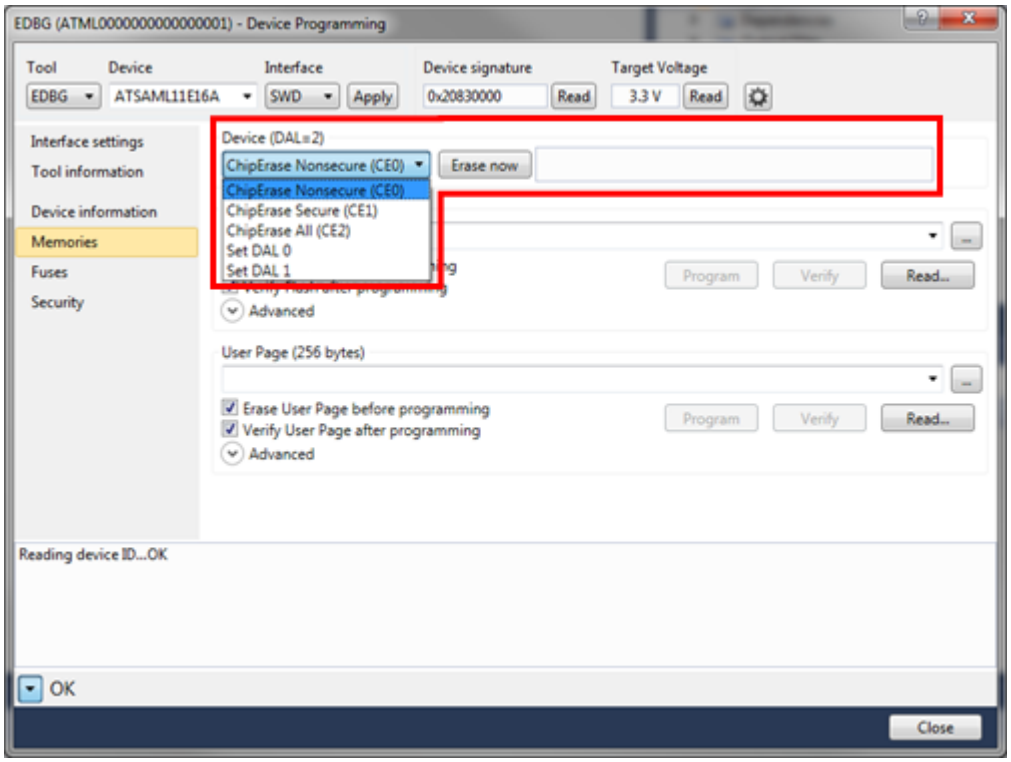
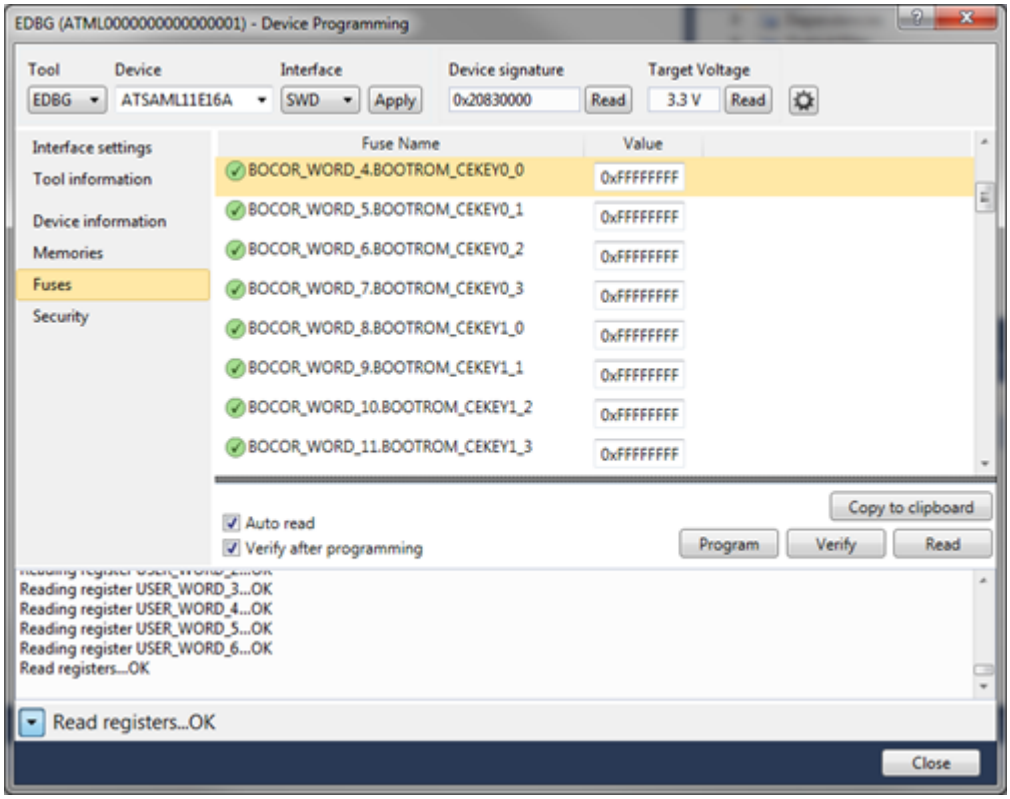
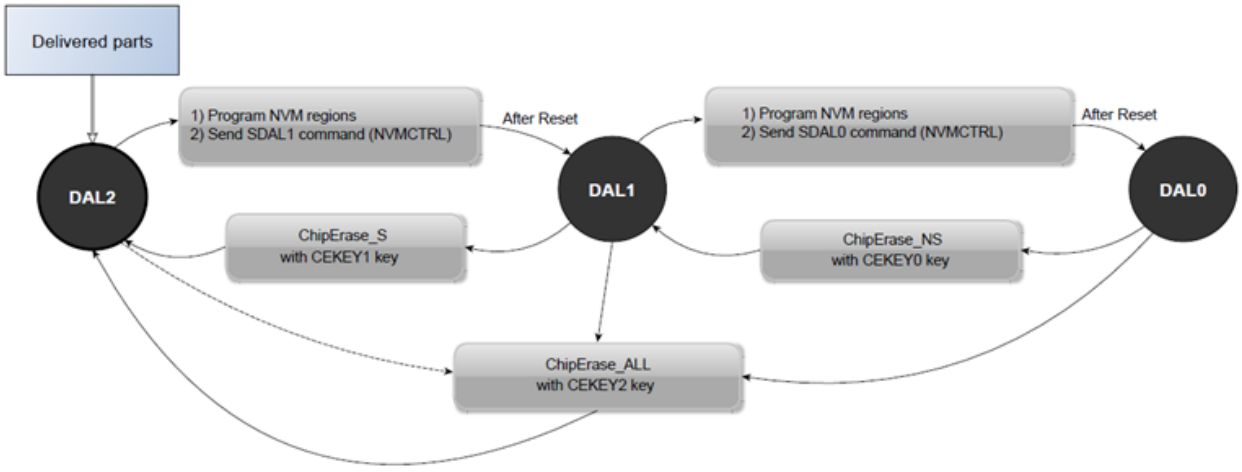


Figure 2-4. ChipErase Key Fuses Setting Under AS7 Device Programming



The following figure illustrates the use of Set DAL and ChipErase commands during the SAM L11 project deployment.

Figure 2-5. SAM L11 DAL and ChipErase Mechanism



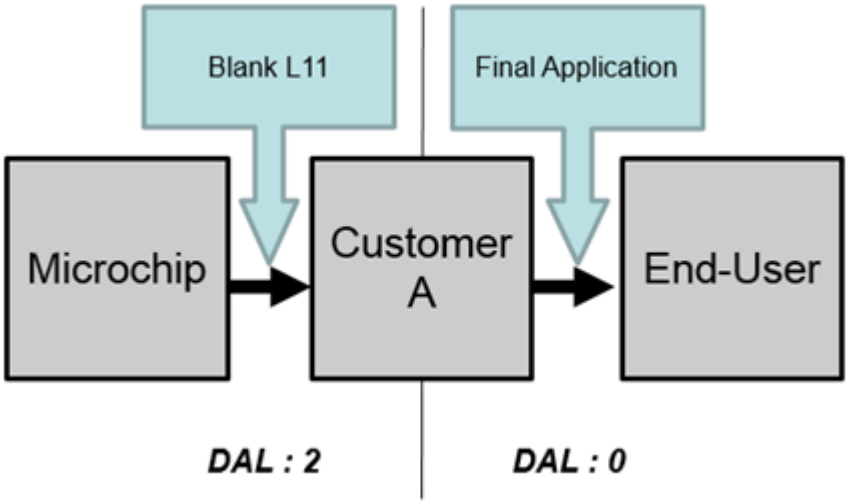
2.2 Customer A and Customer B

The combination of the system Set DAL and ChipErase with TrustZone for Cortex-M architecture allows two deployment approaches: A single-developer approach (Customer A) and a dual-developer approach (Customer A + Customer B).

2.2.1 Single-Developer Approach

In single developer approach, the developer (Customer A) is in charge of developing and deploying Secure and Non-Secure code. The application of Customer A can be protected by using DAL0.

Figure 2-6. Single Developer Approach

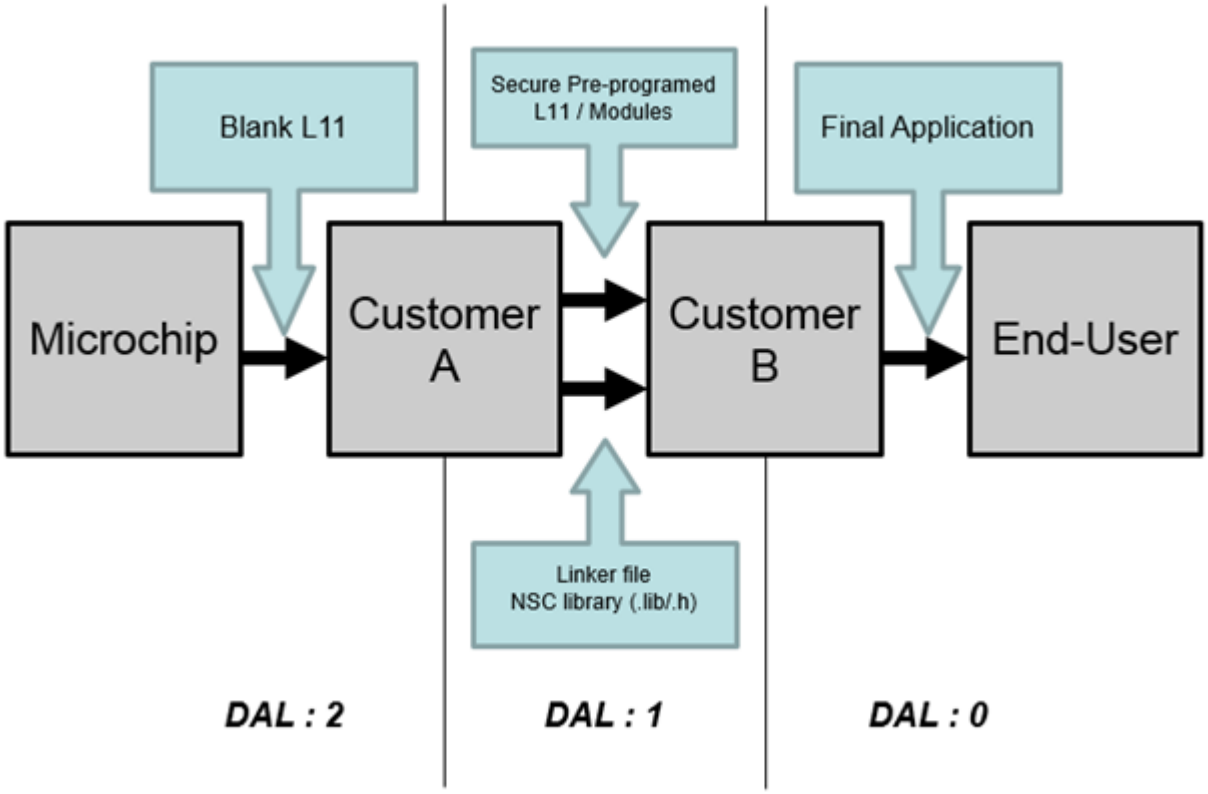


2.2.2 Dual-Developer Approach

In this approach, the first developer (Customer A) is in charge of developing the Secure application and its associated Non-Secure callable library (`.lib/.h`), and providing a predefined linker file to the second developer (Customer B). This Secure application is then loaded in the SAM L11 NVM and protected using the set DAL1 command to prevent further access to the Secure memory region of the device.

A second developer (Customer B), will then start his development on a preprogrammed SAM L11 with limited access to secure resources (call to Non-Secure API only). To do so, Customer B will use a linker file and the NSC library provided by customer A.

Figure 2-7. Dual Developer Approach



The following sections of this document describe the application development and deployment process to be implemented for both Customer A and Customer B sides.

3. How to Develop a SAM L11 Application Under Atmel Studio 7

When starting development on the SAM L11, Customer A and Custoemr B should follow two different approaches as the SAM L11 system architecture, combined with TrustZone for ARMv8-M, sets two different access levels to the chip resources, such as debug, memories and peripheral.

Atmel Studio 7 integrated development platform provides a full set of advanced features to accelerate the development of a SAM L11 application. The following sections illustrate the approaches to be followed by Customer A and Customer B to create, customize, and debug their application.

3.1 Create and Configure a Secure Project (Customer A)

To help Customer A (regardless of single or dual developer approaches) start with the SAM L11, Atmel Studio 7 provides a predefined Secure Solution template that illustrates basic Secure and Non-Secure application execution. This template can be used to evaluate and understand the TrustZone for ARMv8-M implementation in the device, or as a start-up point for custom solution development.

This section describes the following aspects of a secure solution development:

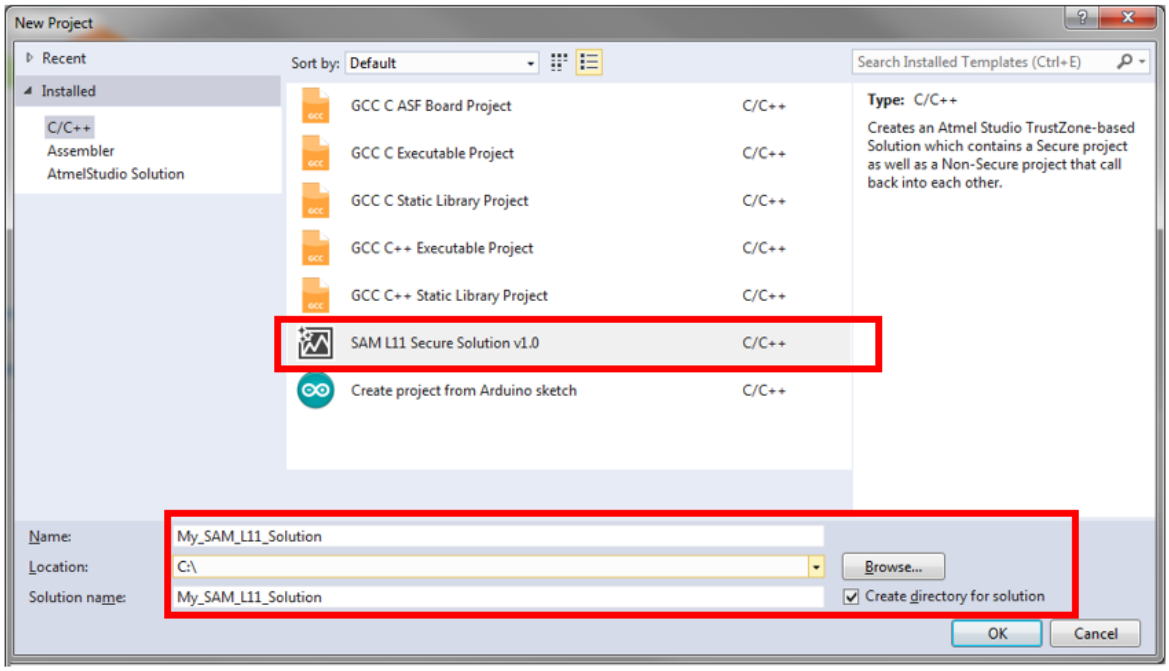
- How to create, build, and debug a new solution for the SAM L11 under Atmel Studio 7
- Secure solution template architecture overview

3.1.1 Create and Build the Solution

To create and build Secure Solution template follow these steps:

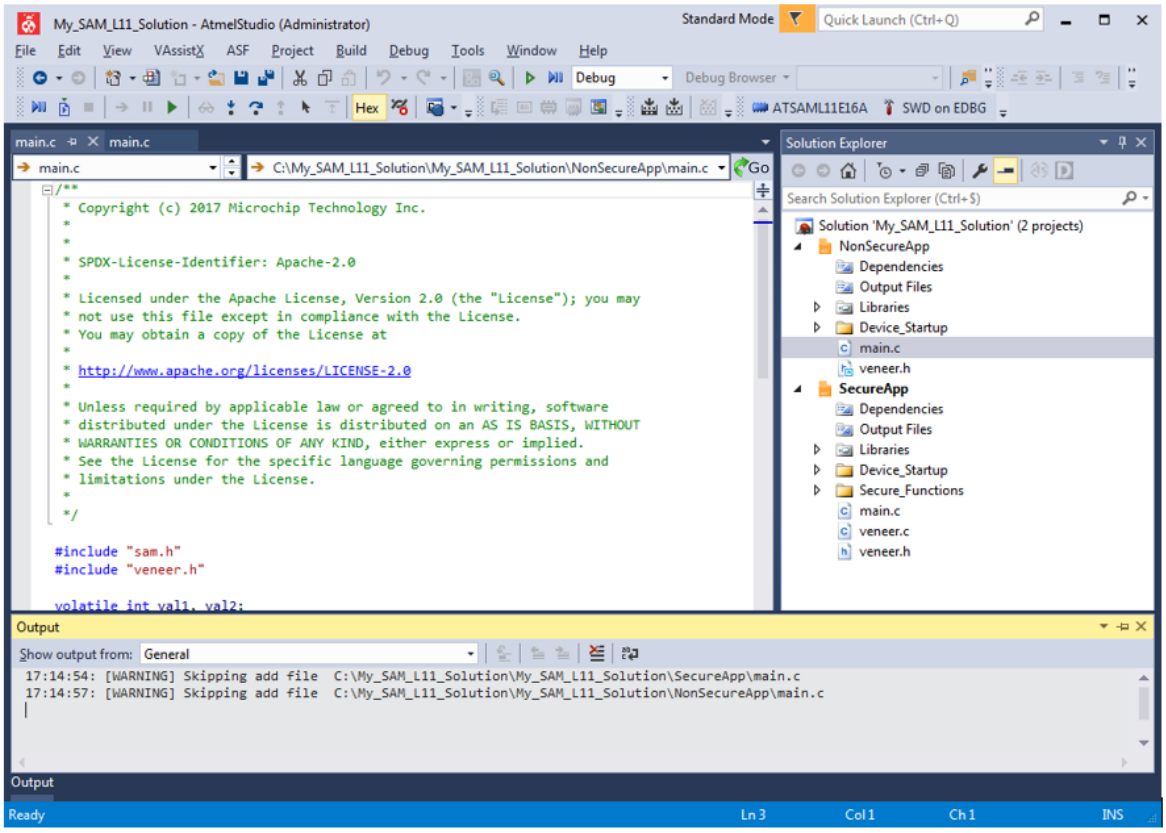
1. Open Atmel Studio 7.
2. Select *File* > *New* > *Project*….
3. Configure the new solution in the *New Project* window:
   3.1. Clcik the C/C++ tab.
   3.2. Select *SAM L11 Secure Solution*.
   3.3. Enter Name, Location, and Solution Name, and then click **OK**.
4. Create a new SAM L11 solution.

Figure 3-1. Create a New SAM L11 Solution Under AS7



5. The SAM L11 Secure Solution should appear as shown in figure below.

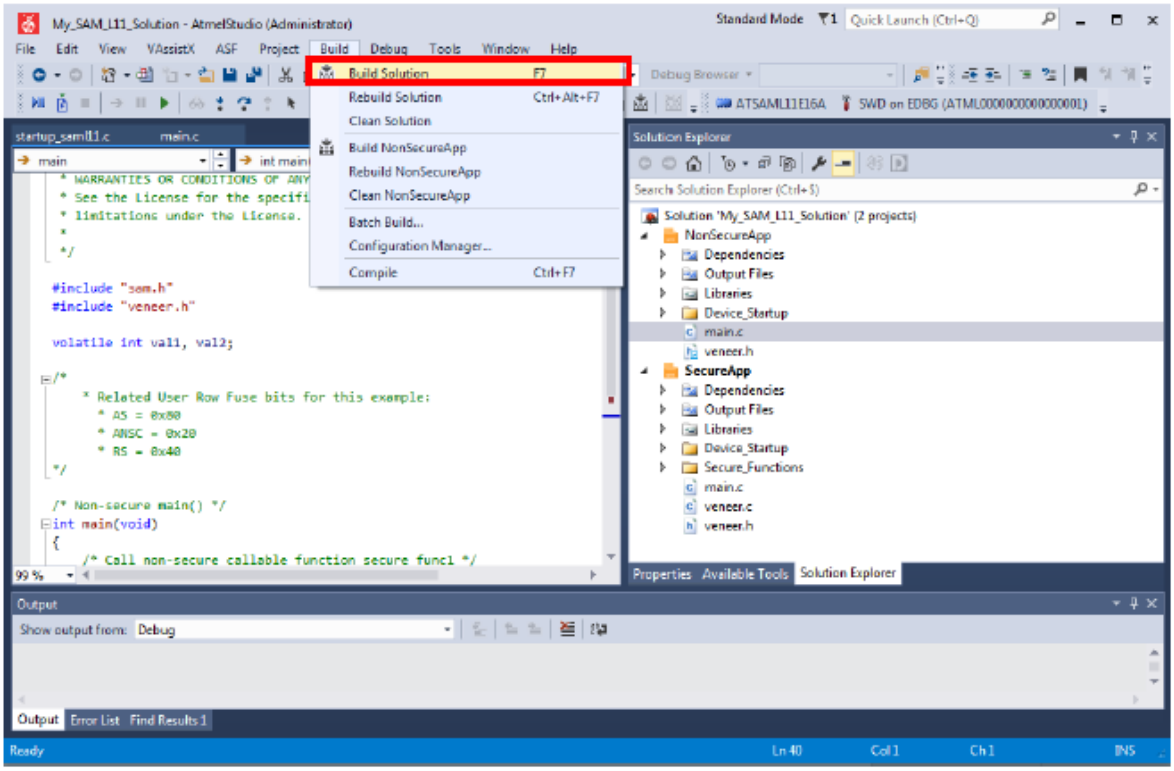
Figure 3-2. SAM L11 Secure Solution Under AS7



Note: The secure Solution Template processing will generate two warnings dues to override of the main files from both Secure and Non-Secure. The user should not consider these warnings.

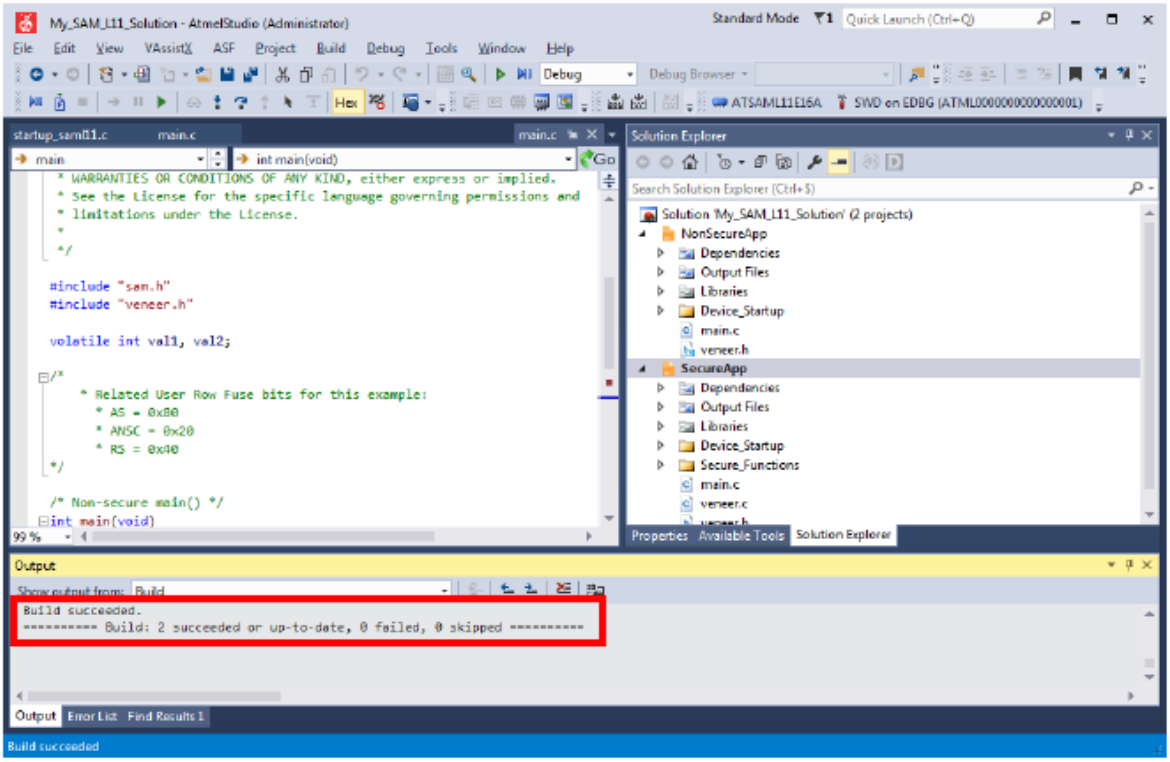
6. From the Build menu, select Build Solution (F7) to build the full solution.

Figure 3-3. Build Solution Under AS7



7. Ensure that no errors are reported in the output window.

Figure 3-4. Build Output Succeeded



This solution is the starting point for any bare-metal development on the SAM L11 device.

3.1.2 SAM L11 Secure Solution Architecture

The SAM L11 Secure Solution Template is composed of preconfigured Non-Secure and Secure projects. The project configuration aspects related to TrustZone for ARMv8-M implementation are already implemented to facilitate the development process.

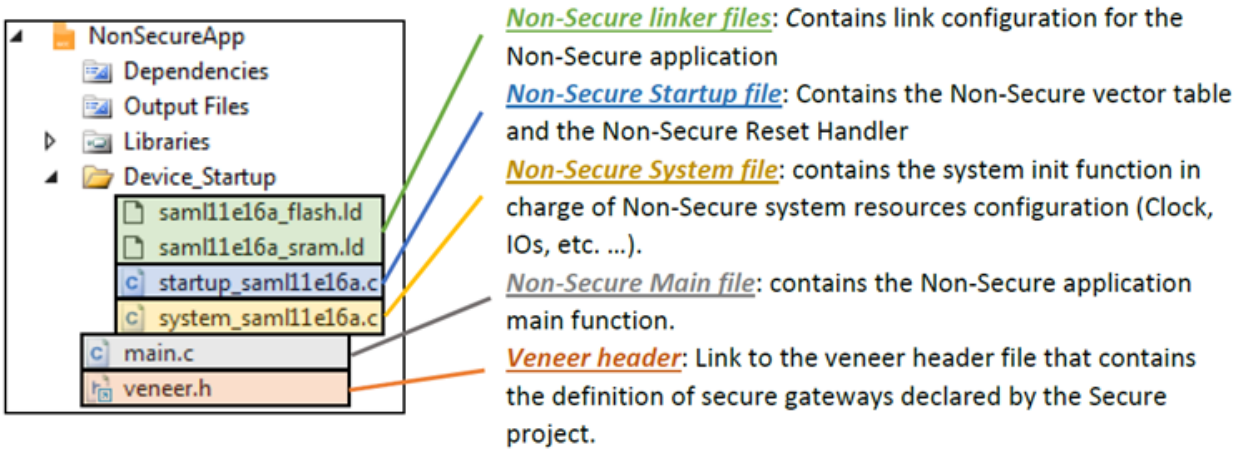
3.1.2.1 Non-Secure Project

The Non-Secure project is a standard application that runs in Non-Secure world.

This application can make use of all the system resources allocated to the Non-Secure world. It can also call predefined Non-Secure Callable (NSC) functions defined in the `veneer.h` file, which are provided by the Secure application.

The Non-Secure project architecture is shown below.

Figure 3-5. Non-Secure Project Architecture

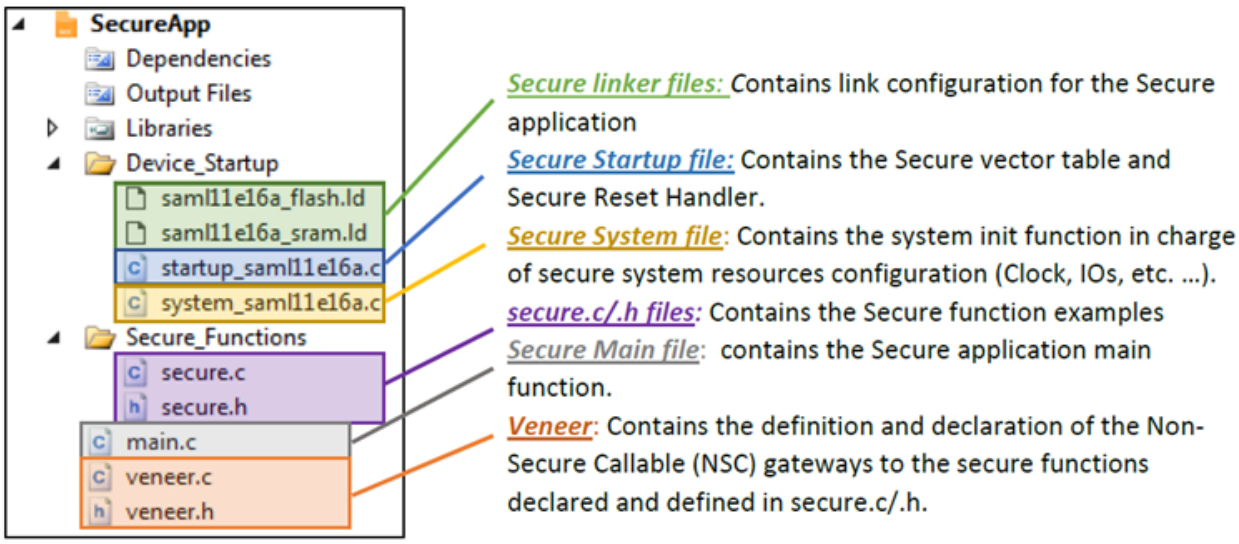


3.1.2.2 Secure Project

The Secure application project is in charge of the following applicative aspects:

- Initialization of the system security and resources attribution (memories as peripherals)
- Execution of the Secure functions or drivers
- Call to the Non-Secure application (main function)

Figure 3-6. Secure Project Architecture

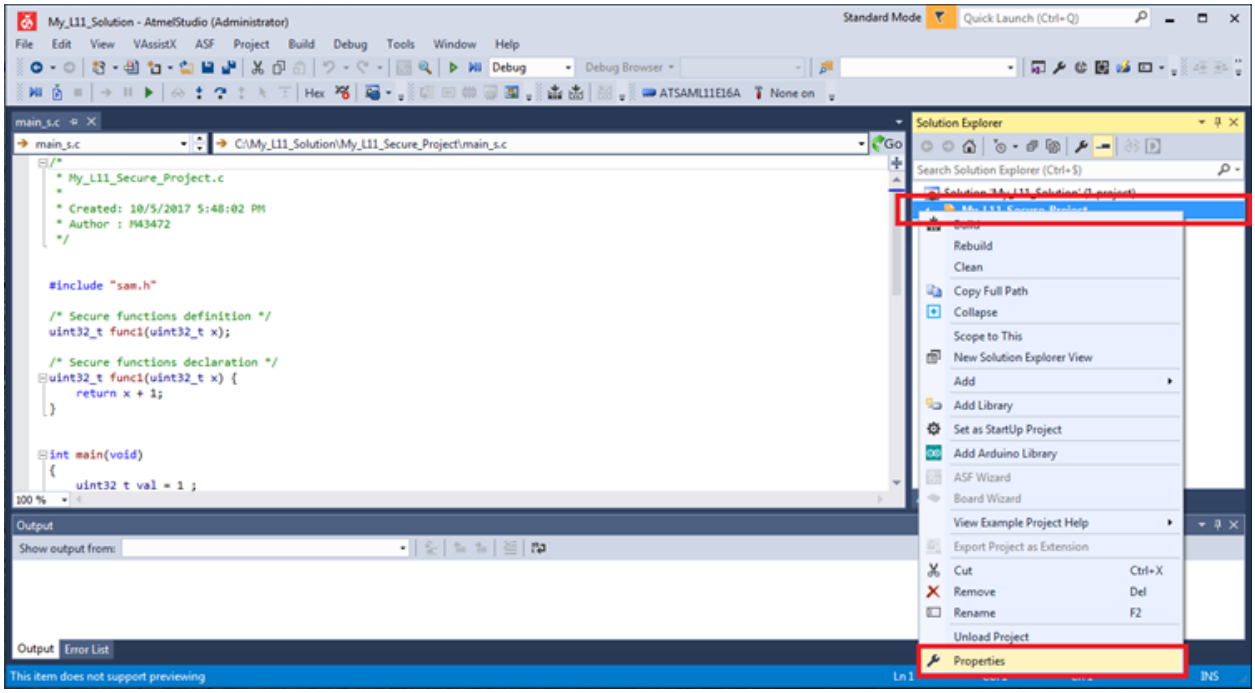


3.1.2.3 Project Properties

To access the project properties in the solution explore, on the Secure and Non-Secure projects, from the short-cut menu select Properties.

Note: Ensure that all on-going debug session should be stopped before accessing the project properties.

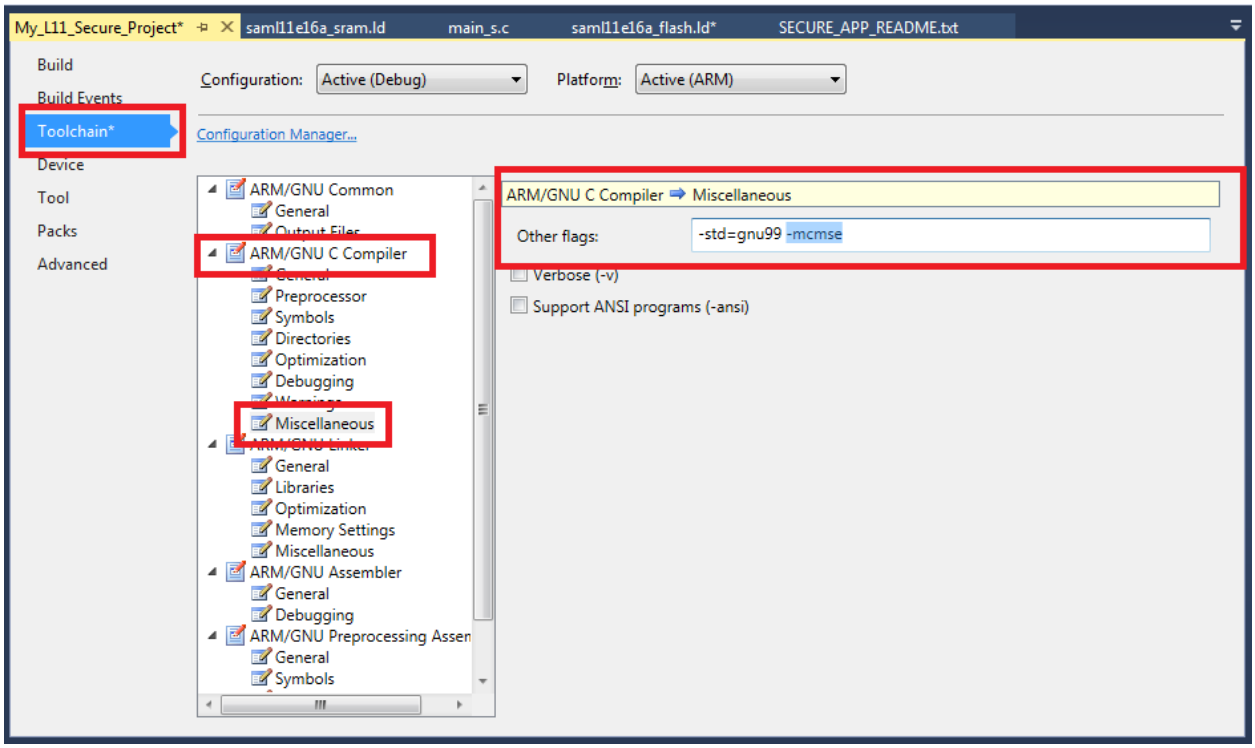
Figure 3-7. Access to Project Properties



In Secure project properties, the support of TrustZone for ARMv8-M instructions and compiler attributes is set using the *-mcmse* compiler flag.

This setting is accessible in the project properties window under *Toolchain>ARM/GNU C Compiler>Miscellaneous>Other flags*.

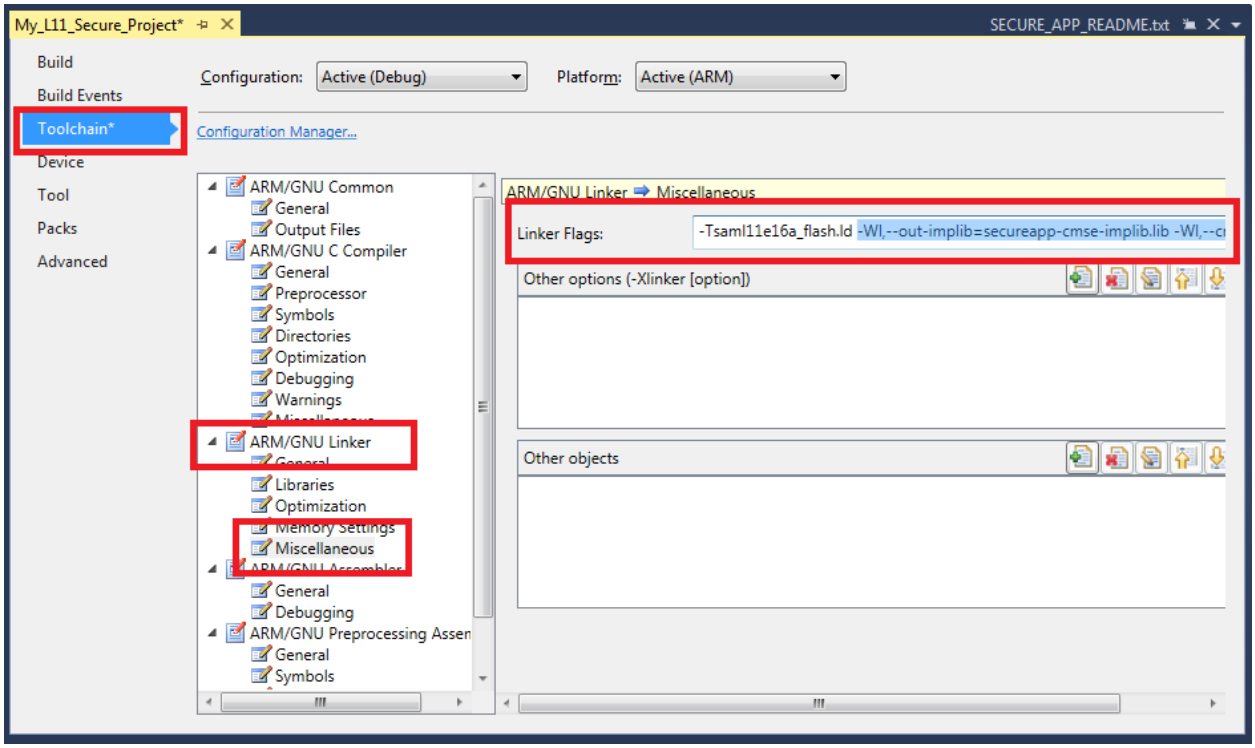
Figure 3-8. SAM L11 Secure Project CMSE Compiler Option



The bridge between the Secure world and Non-Secure worlds is done through the specific Secure Gateways (SG), which are also called Veneers, and are generated by the Secure project and placed during the Secure project link into the Non-Secure Callable (NSC) memory region.

To perform this action, *--cmse-implib* and *--out-implib* linker options should be defined in the Secure project properties.

Figure 3-9. SAM L11 Secure Project Linker --cmse implib and --out-implib Options



- *--out-implib* linker option: specify the Secure gateway imported library name to be generated by the linker
- *--cmse-implib* linker option: requests that the library specified by the *--out-implib* is a secure gateway import library, suitable for linking a Non-Secure executable against the Secure code as per the ARMv8-M Security Extension.

The Secure Solution template sets the following options: *-Wl,--out-implib=secureapp-cmse-implib.lib -Wl,--cmse-implib*. The secure project developer can customize the library name by changing the *--out-implib* option value.

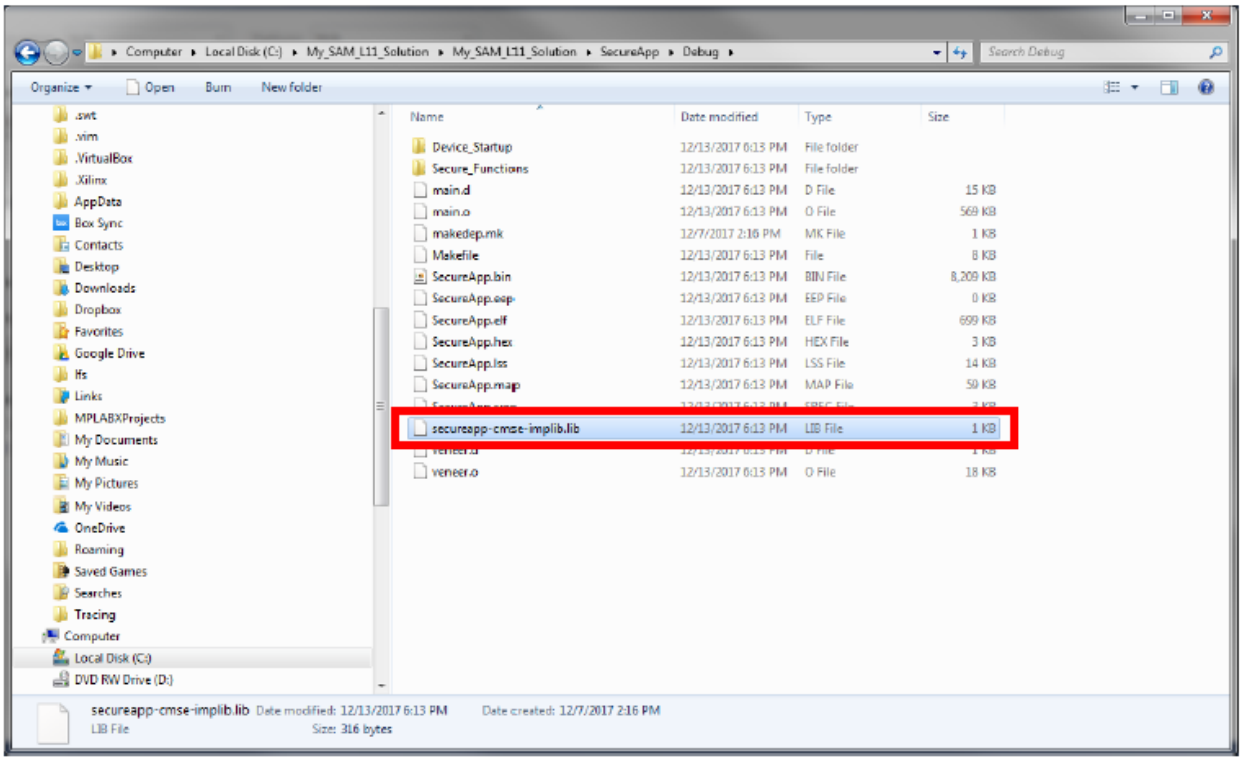
At the C code level, all functions, which are defined with the *cmse_nonsecure_entry* compiler attribute from the ARM CMSE extension, will be placed in the Generated Secure Gateway Import Library.

```
/* Non-secure callable (entry) function */
int __attribute__((cmse_nonsecure_entry)) secure_func1(int x)
{
    return func1(x);
}
```

`Secure/veener.c`

The Generated Secure Gateway import library (`secureapp-cmse-implib.lib`) can be found in the Debug directory of the SecureApp project:

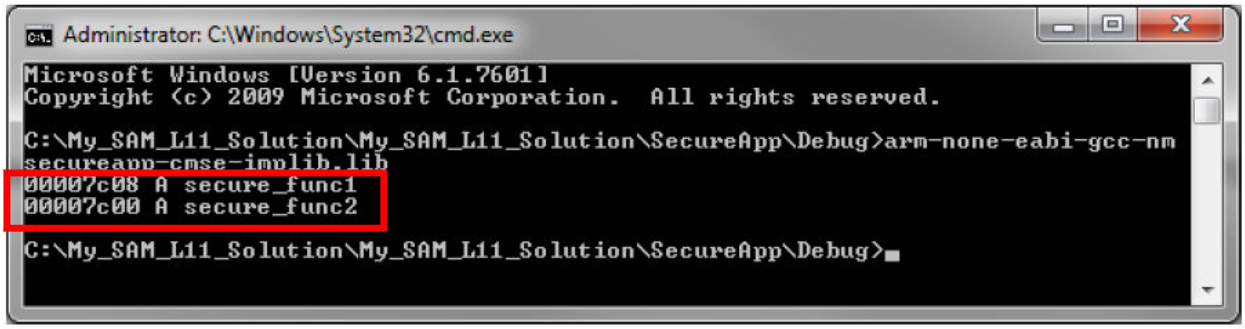
Figure 3-10. Generated Secure Gateway Import Library



Th Secure Gateway Import Library content can be verified using the `arm-none-eabi-gcc-nm` command located in the ARM GCC Toolchain install location:

C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\arm\arm-gnu-toolchain\bin.

Figure 3-11. arm-none-eabi-gcc-nm Output



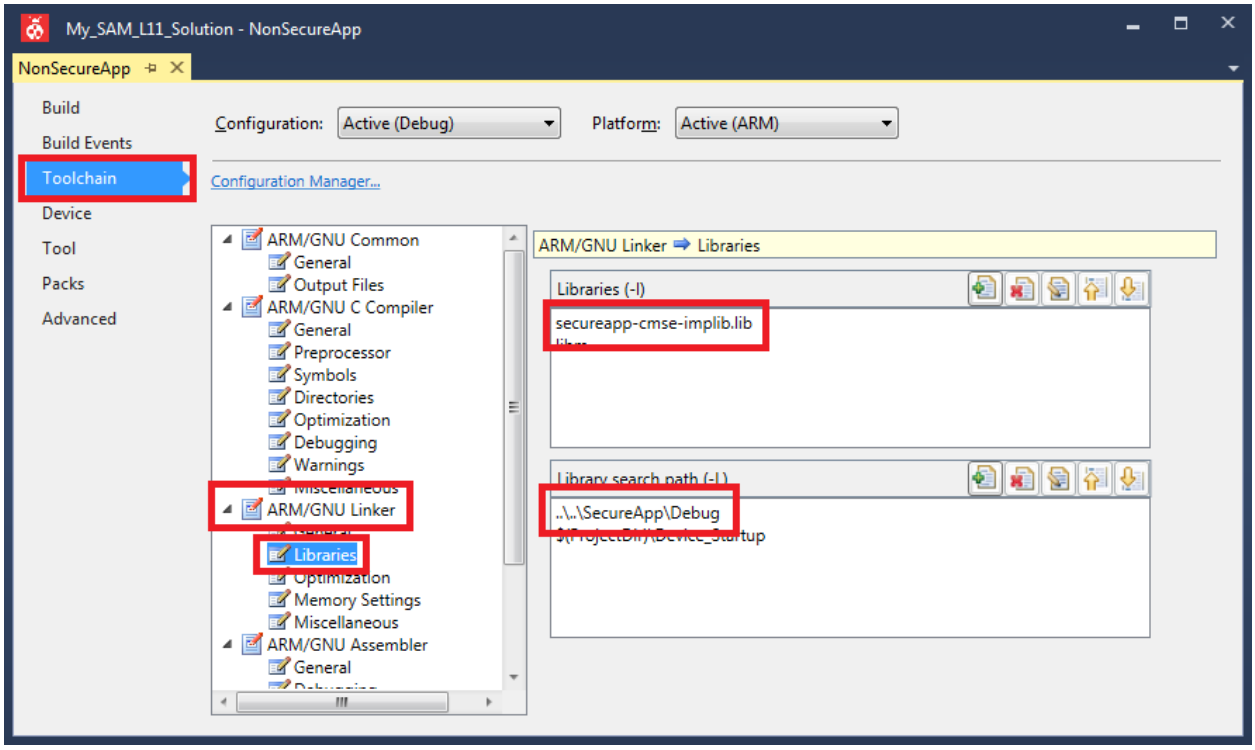
Note: The ARM GCC Toolchain install path can be added to the windows environment variable path to call the `arm-none-eabi-gcc-nm` command from the `.lib` location, as shown in the previous figure.

In a Non-Secure project, the use of TrustZone for ARMv8-M is transparent for the developer.

Only the Secure Gateway Import Libraries and its associated header files should be included.

This setting is accessible in the project properties window under *Toolchain>ARM/GNU Linker>Libraries*.

Figure 3-12. SAM L11 Non-Secure Project Linker Libraries Settings



For additional information, refer to the Create and Configure a Non-Secure Project (Customer B) chapter.

3.1.2.4 Resources Attribution

The Secure project is in charge of allocating the SAM L11 resources to both the Secure and the Non-Secure worlds.

Prior to starting any development and customization of the projects, it is mandatory to verify the system security resources attribution.

System resources can be allocated to the Secure or Non-Secure worlds by setting the SAM L11 NVM fuses. These fuses are in charge of defining the configuration of Boot modes, ChipErases, system peripherals (BOD and watchdog), IDAU (Memory security attribution), and PAC (Peripheral security attribution).

Any change to fuse configuration requires a restart of the device, as fuses are handled by the Boot ROM executed at device start-up. The Boot ROM is responsible for copying the configuration of the fuses in the different peripheral registers, then locking the configuration to any users (including Customer A) until the next boot.

The configuration of the NVM fuses can be done through some definitions at the beginning of the Secure `main.c` file, or in the Device Programing tool available in Atmel Studio 7 by clicking on *Tools>Device Programming*.

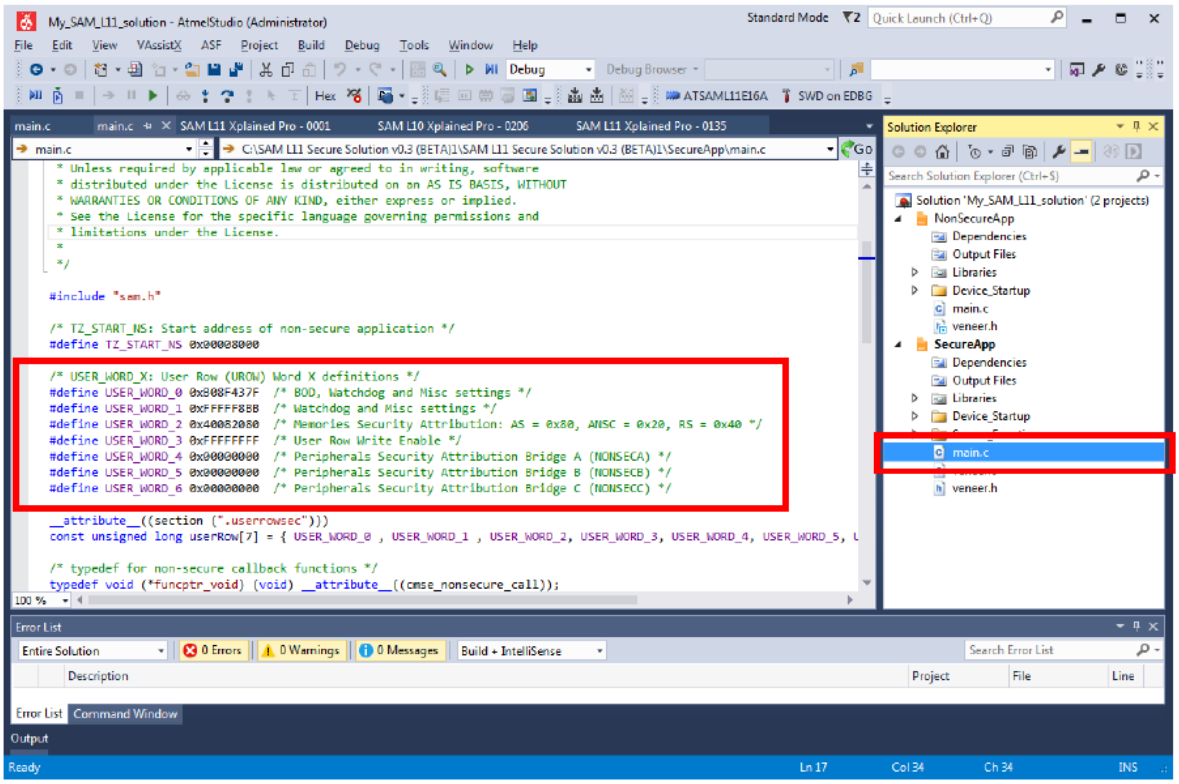
Note:

- The description of each USER_WORD can be found in section *10.2.1.2 SAM L11 User Row* of the product data sheet.
- The complete description of the SAM L11 Boot ROM can be found in section *14. Boot ROM* of the product data sheet.

To configure the NVM fuses, perform these actions:

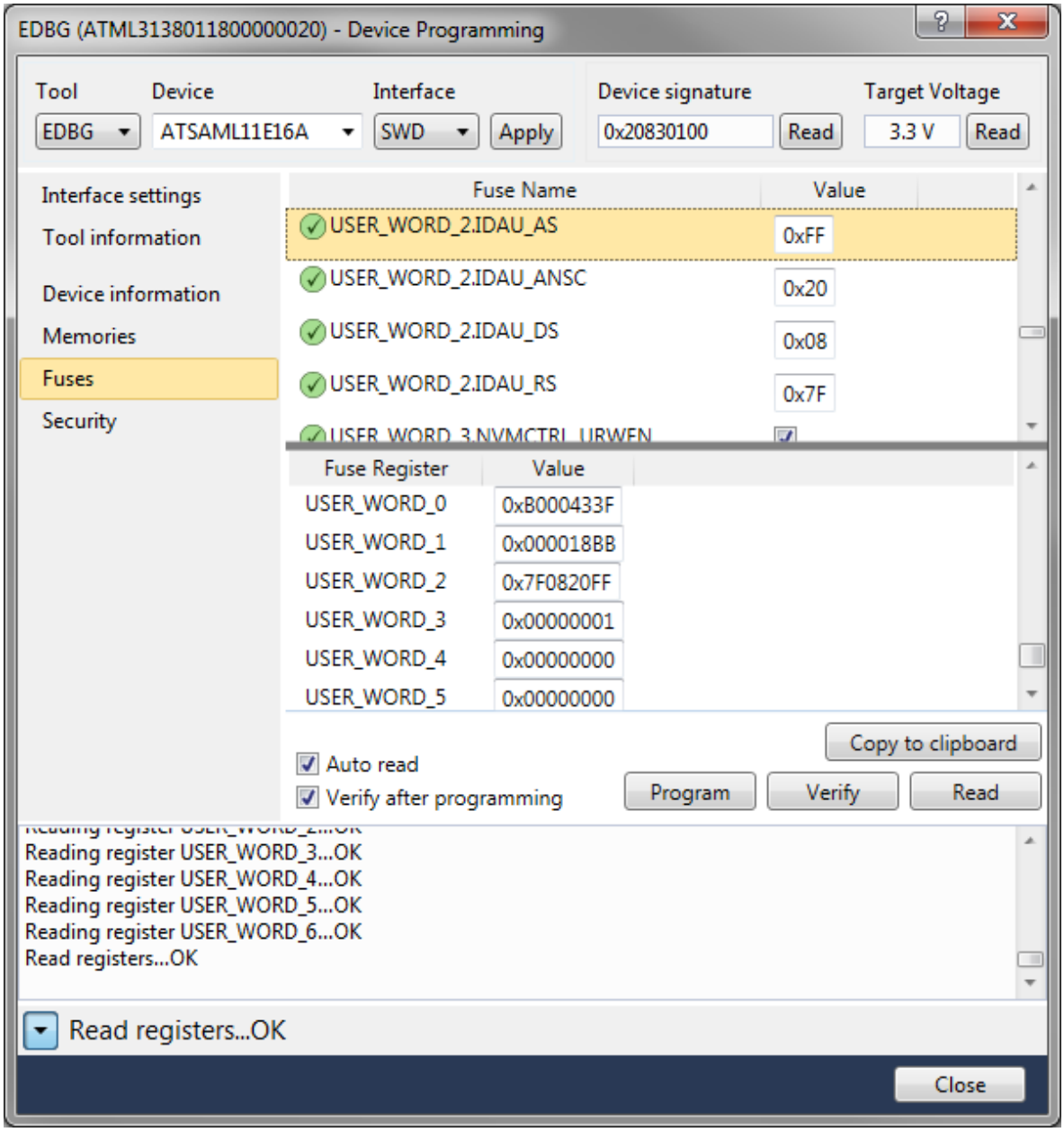
1. Fuses definitions in the Secure project `main.c` file:
   To ease the management of the fuses in charge of Memory and Peripheral security attribution, the Secure `main.c` file includes USER_WORD_x definitions. Any modifications to these definitions allows the Secure application developer to easily setup the security attribution of a specific peripheral, and manage memory security partitioning.

Figure 3-13. Fuses definition in Secure Project Main.c File



2. Fuse definitions in the device programming tool:
   The device programing tool available in Atmel Studio provides a graphical interface for managing the whole set of fuses through various standard device programing tools, such as EDBG, SAM-ICE, J-Link, and so on.

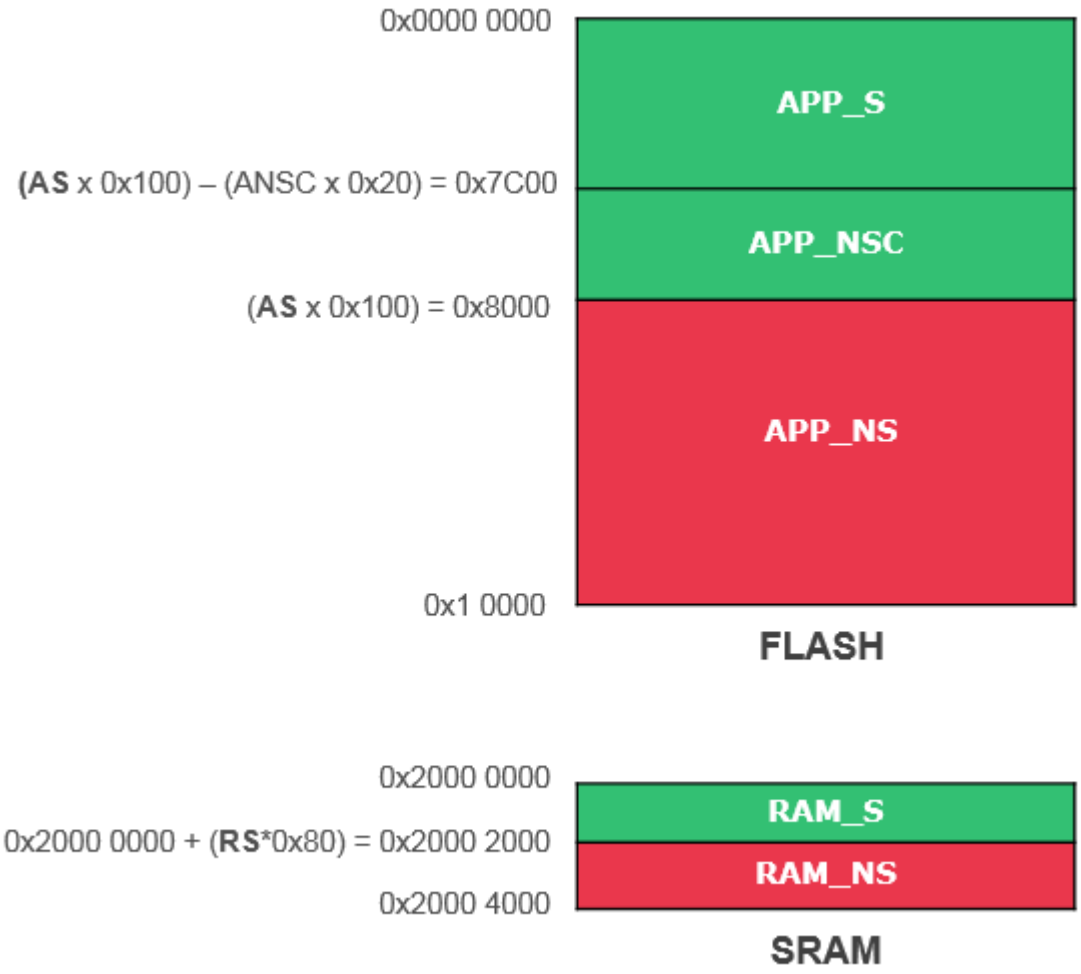
Figure 3-14. Fuses Definitions in Programming Tool



By default, the Secure Solution template sets the following memory and peripheral security attributions in the Secure `main.c` file.

- Memory secure attribution fuses (User Row – UROW) with:
  - AS = 0x80
  - ANSC = 0x20
  - RS = 0x40
  - DS = 0x0

Figure 3-15. SAM L11 Secure Template Memory Attribution



Note: No boot section is defined in the template, therefore the boot parameters fuses (BOOTPROT, BS, BNSC) are set to 0x00.

- All peripherals allocated to the Secure application (NONSECA = 0x00 ; NONSECB = 0x00 ; NONSECC = 0x00)

Important:
These definitions should be modified in any applications that require different Secure or Non-Secure resources attribution.

3.1.2.5 Project Linker Files

Secure and Non-Secure projects have their own linker file available in the *Device_Startup* directory. Both linker files should be aligned to the memory mapping defined in the product fuses. The following sections illustrate how to configure Secure and Non-Secure project linker files according to the following default configuration set in the SAM L11 Secure Solution Template, see SAM L11 Secure Template Memory Attribution.

3.1.2.5.1 Secure Project Linker File Content

The Scure project linker file should define at least four secure memory sections which must be in line with the memory attribution of each secure area defined by the NVM fuses (Secure `main.c` file).

- rom: Defines the Secure Application section of the FLASH memory.
- rom_nsc: Defines the Non-Secure Callable section of the FLASH memory.
- ram: Defines the Secure section of the SRAM memory.
- userrow: Defines the NVM Fuses section.

```
/* Memory Spaces Definitions based on Memories Security Attribution: AS = 0x80, ANSC = 0x20,
RS = 0x40 */
MEMORY
{
  rom      (rx)  : ORIGIN = 0x00000000, LENGTH = 0x00007C00
  rom_nsc  (rx)  : ORIGIN = 0x00007C00, LENGTH = 0x00000400
  ram      (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00002000
  userrow  (rw)  : ORIGIN = 0x00804000, LENGTH = 0x00000100
}
```

Secure `saml11e16a_flash.ld`

- The following code should be present after the rom segments to link all the functions defined with the attribute `arm-none-eabi-gcc-nm`.

```
.ARM.exidx :
{
  *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > rom
PROVIDE_HIDDEN (__exidx_end = .);

. = ALIGN(4);
_etext = .;

. = ALIGN(4);
.gnu.sgstubs :
{
_ssgstubs = .;
} > rom_nsc
```

Secure `saml11e16a_flash.ld`

- The following code should be present at the end of the Secure linker file to link the `main.c` User Row (UROW) definition in the previously defined User Row (UROW) memory section.

```
        . = ALIGN(8);
        _estack = .;
    } > ram

    . = ALIGN(4);
    _end = . ;

 .userRowBlock :{
    KEEP(*(.userrowsec))
    } > userrow
}
```

Secure `saml11e16a_flash.ld`

3.1.2.5.2 Non-Secure Project Linker File Content

The use of TrustZone is transparent in the Non-Secure project linker file.

The Non-Secure rom and ram section definitions should be defined as they are in any standard GCC executable projects.

Important: Developers need to ensure that there is no overlapping between the Non-Secure and Secure memory space definitions.

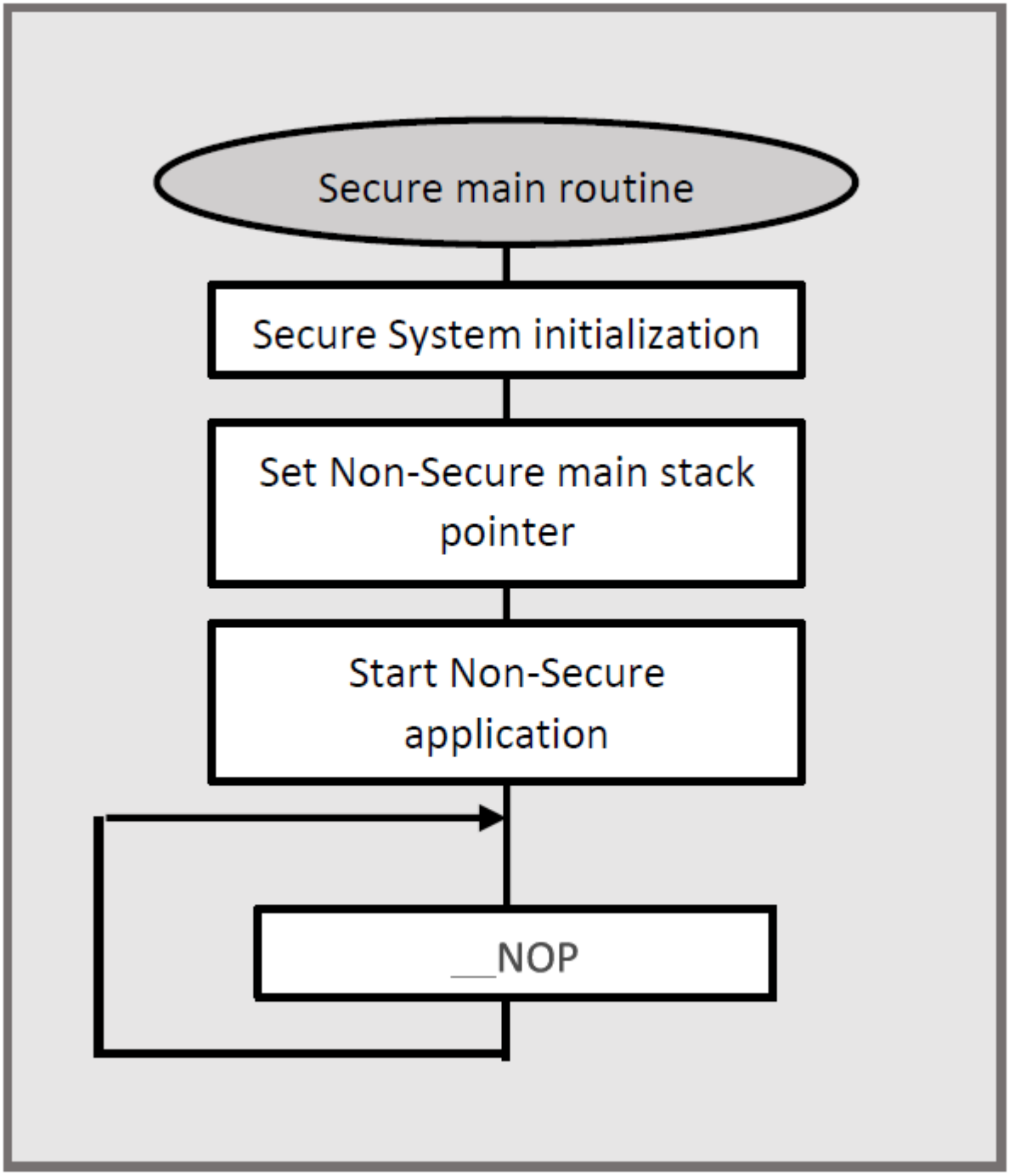
```
…
/* Memory Spaces Definitions based on Memories Security Attribution: AS = 0x80, ANSC = 0x20,
RS = 0x40 */
MEMORY
{
  rom      (rx)   : ORIGIN = 0x00008000, LENGTH = 0x00008000
  ram      (rwx)  : ORIGIN = 0x20002000, LENGTH = 0x00002000
}
…
```

Non-Secure `saml11e16a_flash.ld`

For more details, refer to the Create and Configure a Non-Secure project (Customer B) chapter.

3.1.2.6 Secure Main Function

The Secure main function flowchart from the Secure Solution template is shown in the following figure.



The Secure main routine is in charge of: the following:

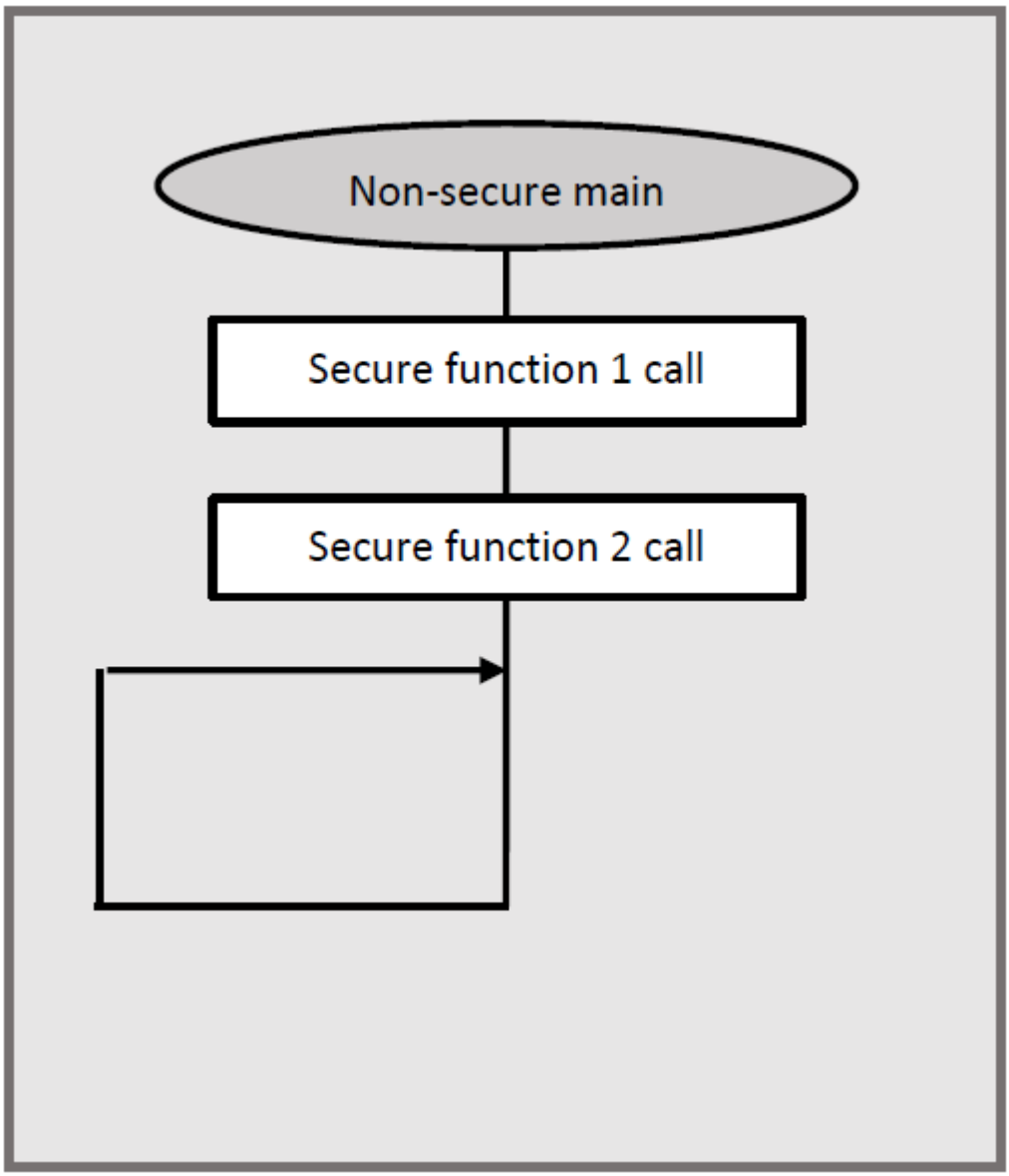
- Configuring system resources, security attribution of the system clocks, GPIO and mix Secure peripherals (system_init function)
- Prepares the Non-Secure main stack pointer (MSP_NS) before jumping to the Non-Secure application

This Secure `main.c` file can be used as a starting point for any secure applications development.

Note: The `system_init` function is empty by default, keeping the SAM L11 registers at their reset state (CPU running at 4MHz) . This function should be customized according to the Secure and Non-Secure application requirements.

3.1.2.7 Non-Secure Main Function

The Non-Secure main function flowchart from the Secure Solution Template is shown in the following figure.



The Non-Secure main function illustrates calls to the Secure functions through each of the veneer, which are provided by the Secure application.

This Non-Secure `main.c` file can be used as a starting point for any Non-Secure applications development.

3.1.2.8 Secure and Non-Secure Functions Call (secure.c/.h; veneer.c/.h)

The SAM L11 Secure Solution Template illustrates the declaration, definition and use of Secure functions across the Secure and Non-Secure projects.

For more information on Secure and Non-Secure Function call, refer to the Secure and Non-Secure Functions Call.

The following code example from `veneer.c` and `secure.h` illustrates the declaration and definition of a Secure function and its veneer:

```
#ifndef SECURE_H_
#define SECURE_H_

extern int func1(int x);

#endif /* SECURE_H_ */
```

```
#include "secure.h"

int func1(int x)
{
    return x + 3;
}
```

```
#ifndef VENEER_H_
#define VENEER_H_

/* Non-secure callable functions */
extern int secure_func1(int x);

#endif /* VENEER_H_ */
```

```
/* Non-secure callable (entry) function */
int __attribute__((cmse_nonsecure_entry)) secure_func1(int x)
{
    return func1(x);
}
```

Thanks to the `cmse_nonsecure_entry` attribute, the GCC compiler will automatically manage the generation of the Secure gateway section and linker will link it to the defined Non-Secure Callable region.

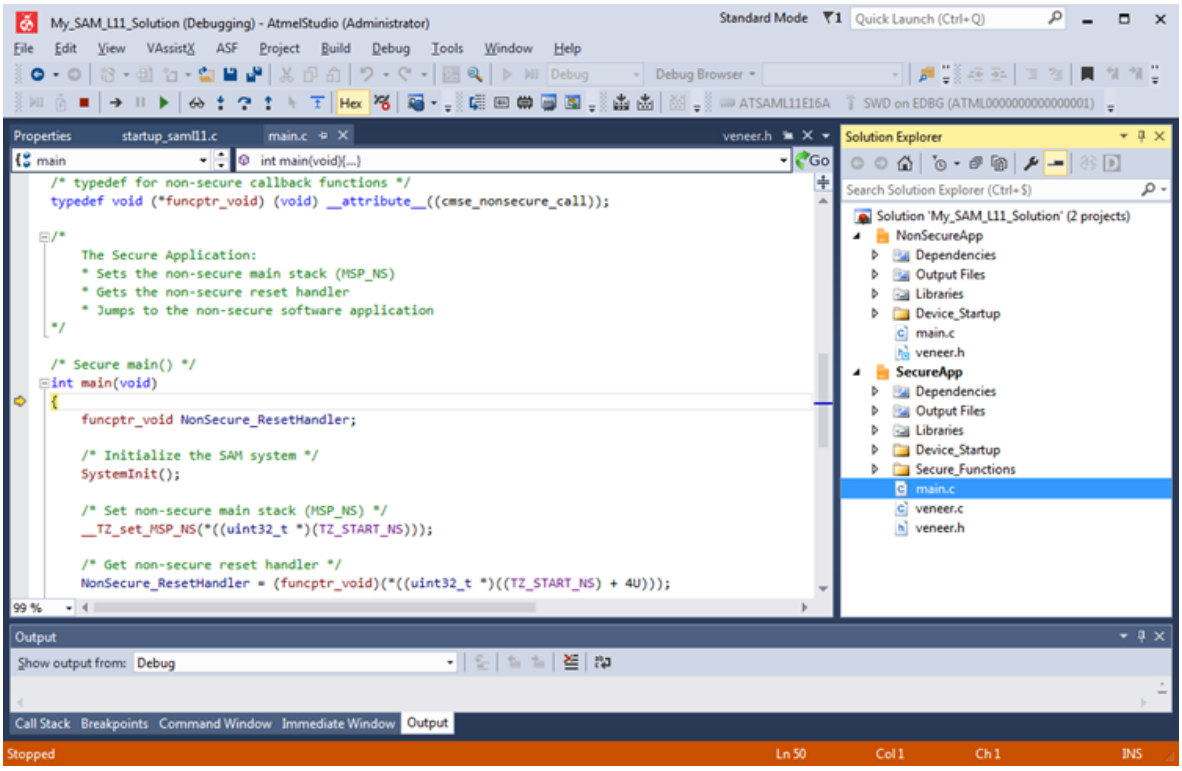
Important: When updating only the Secure project of an application that is flashed on the target MCU, do not change the veneer function addresses. Any modification in veneer addresses will lead to a misalignment between Non-Secure and Secure applications. This would require a re-link to the Non-Secure application and to update the whole solution in the SAM L11 Flash. Developers should ensure that the Secure gateways are linked to at a permanent address.

3.1.3 Debug the Solution

When the device debug access level is set to two (full debug access), Atmel Studio 7 supports the full debug of the Secure and Non-Secure projects allowing stepping through both projects and evaluating the Secure to Non-Secure transition. The following steps illustrate the debug capabilities of the Atmel Studio 7 integrated development environment.

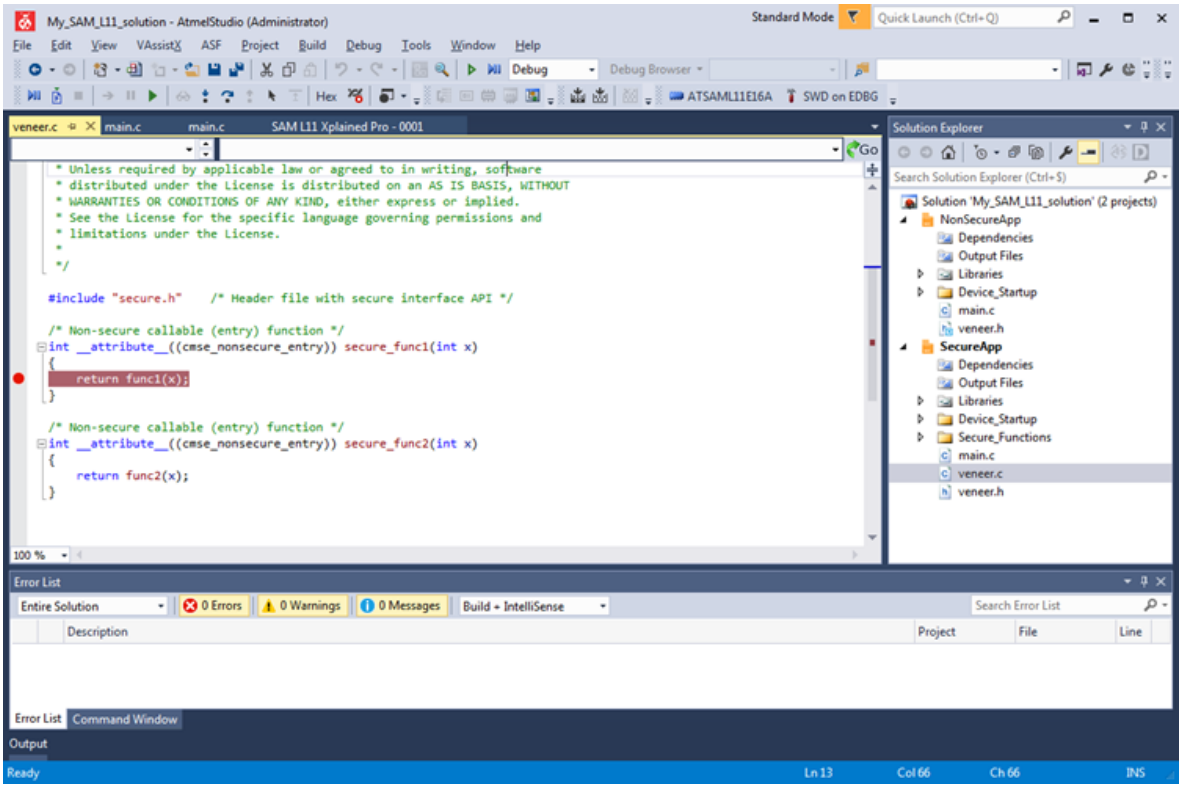
1. Power-up SAM L11 Xplained Pro, and then click  or (Alt+F5) to start debugging, and automatically break on the Secure main function.

Figure 3-16. Start Debugging and Break on Secure Main Function



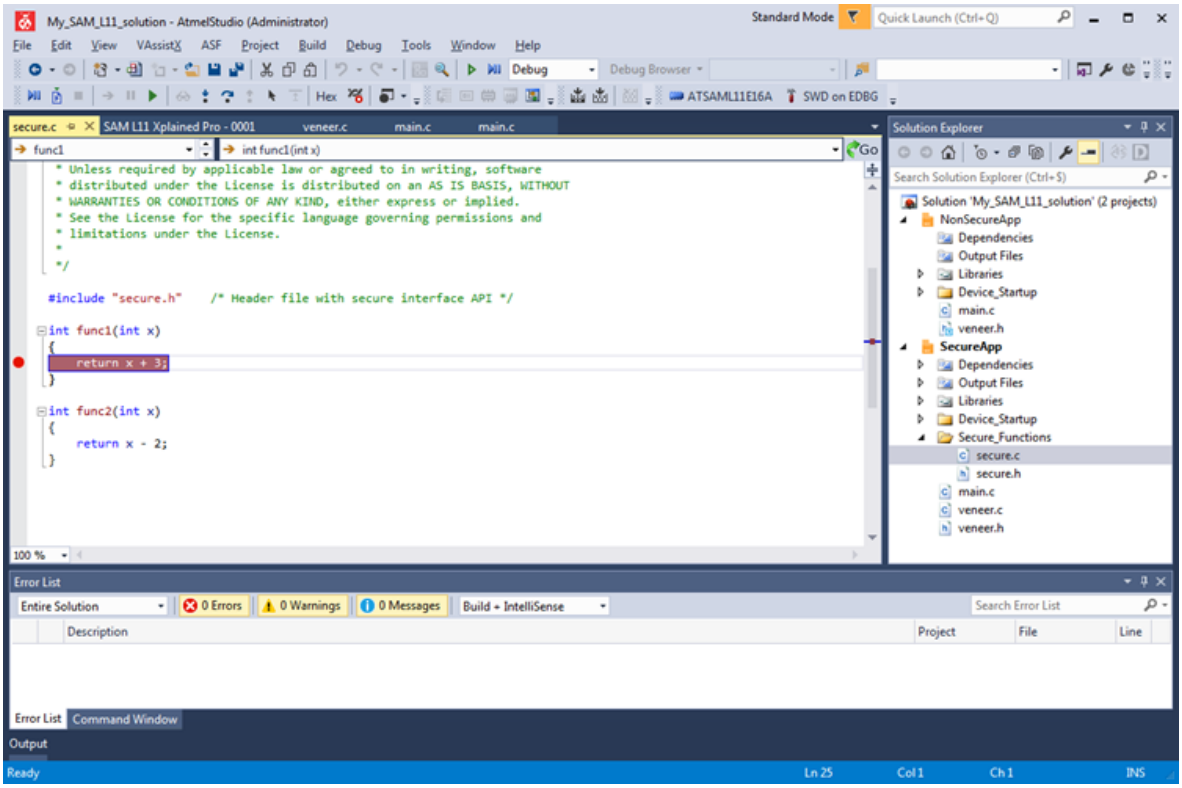
2. Add a breakpoint on the return line of *secure_func1* in the Secure project `veneer.c` file.

Figure 3-17. Breakpoint on secur_func1 Return (Secure Project)



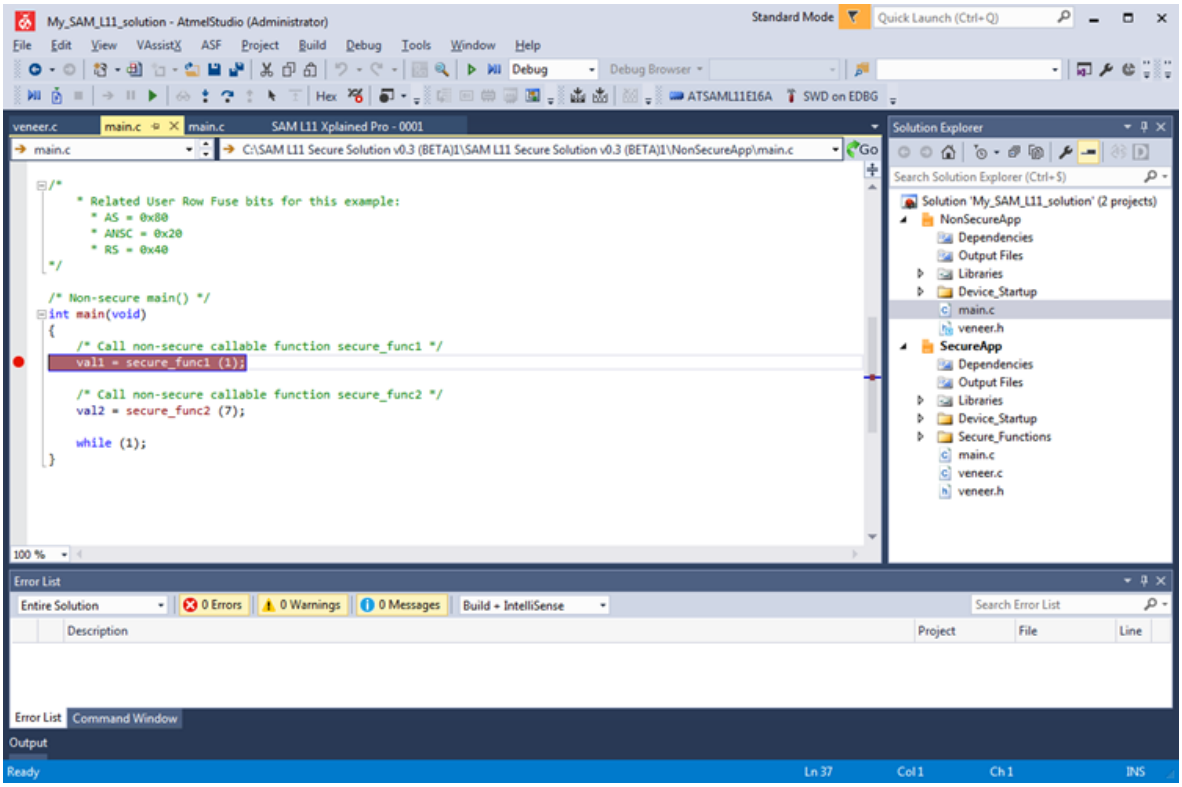
3. Add a breakpoint on the return line of *func1* in the Secure project `Secure_Functions/secure.c` file.

Figure 3-18. Breakpoint on func1 Call (Secure Project)



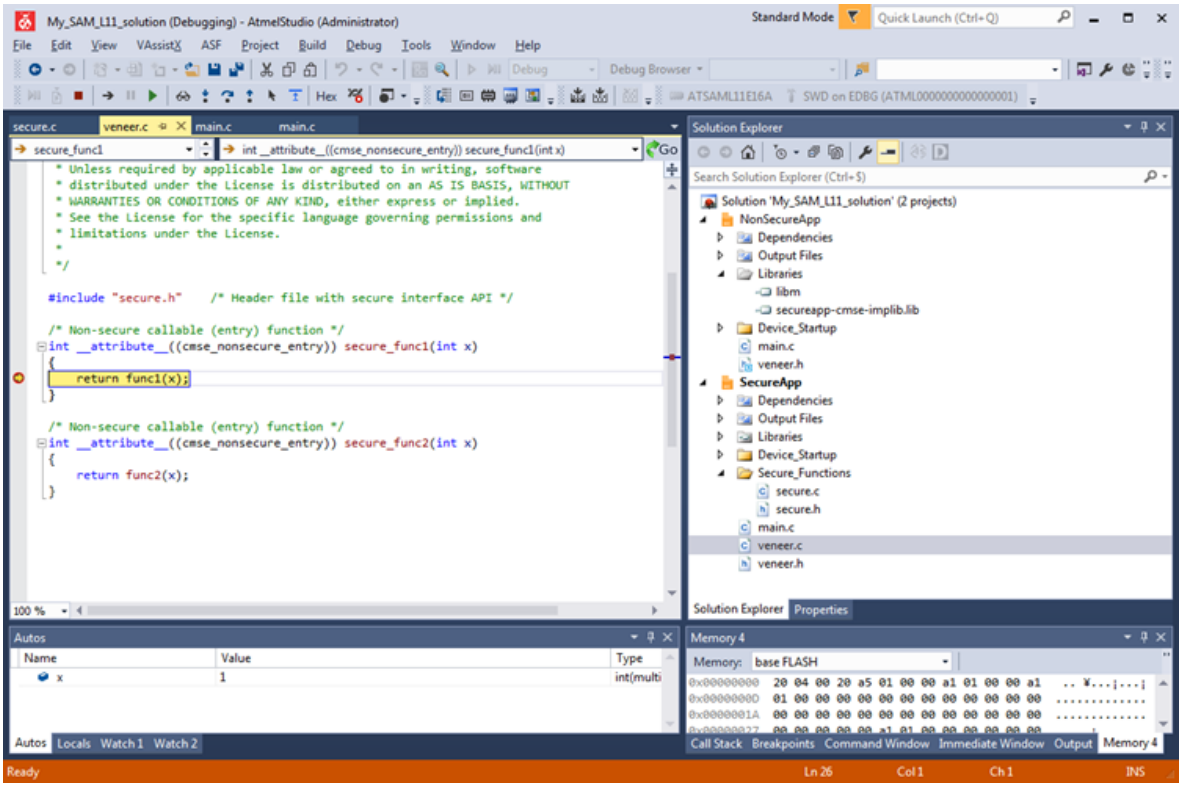
4. Add a breakpoint on the *secure_func1* call in the main function of the Non-Secure project (Non-Secure `main.c` file).

Figure 3-19. Breakpoint on secure_func1 Call (Non-Secure Project)



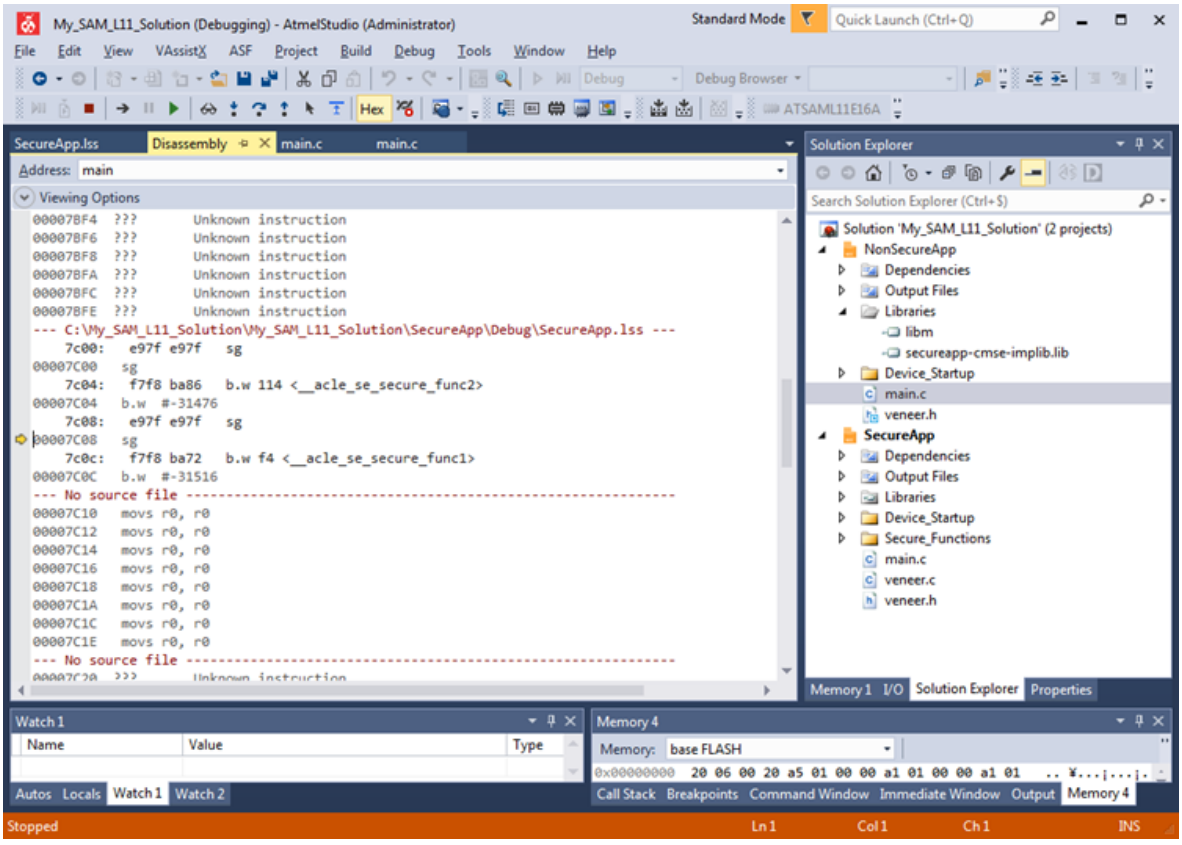
5. Continue the debug by clicking  or press <F5>.
   As a result of this process, the debugger should stop successively on:
   - 5.1. The Secure function veneer call (Non-Secure project).
   - 5.2. The Secure function veneer (Secure project).
   - 5.3. The Secure function (Secure project).

Figure 3-20. Break on secure_func1 Return



Note: A disassembly step-by-step debug is available by selecting the *Debug* > *Windows* > *Disassembly* or press <Alt+8>.

Figure 3-21. AS7 Disassembly Window



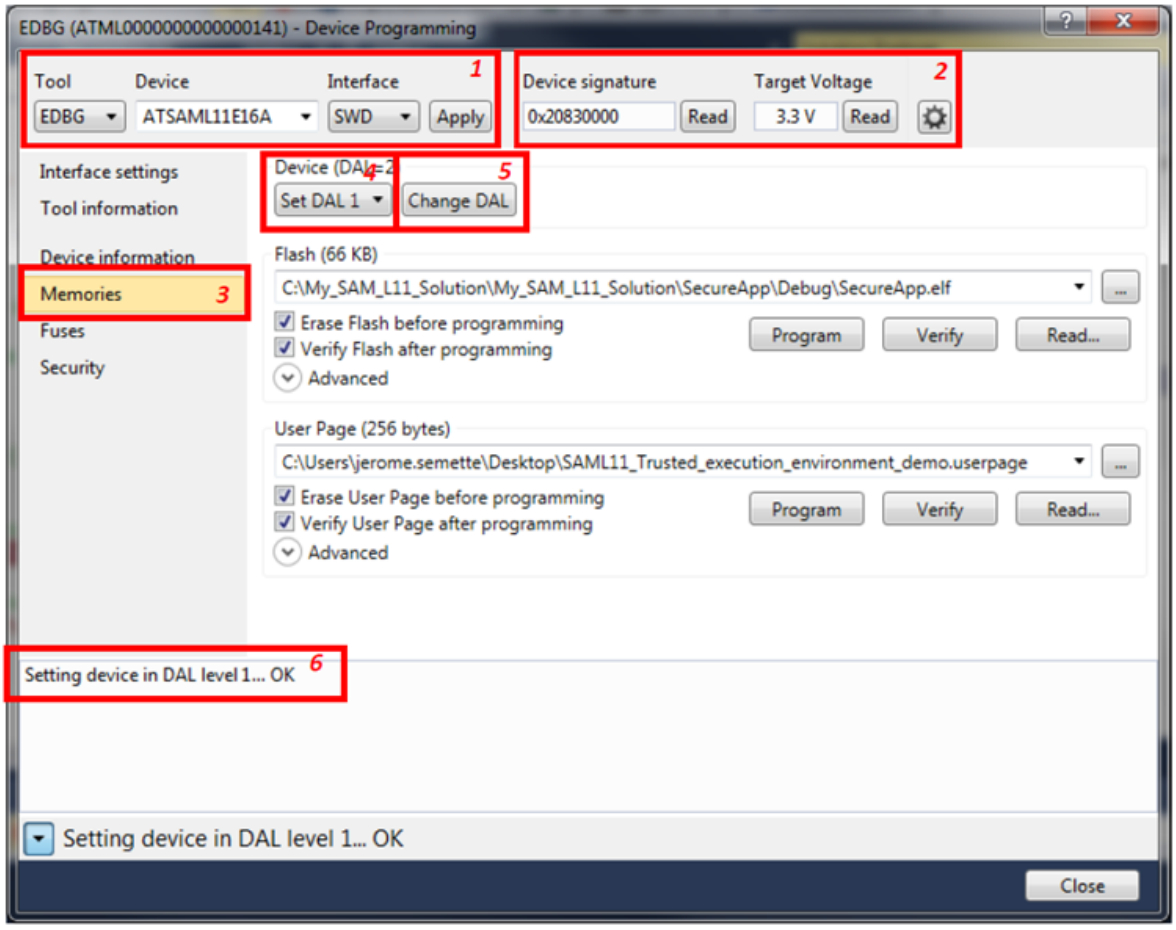
3.1.4 Protect the Secure Application Using Debug Access Levels

In a dual developer deployment approach, it is important to protect the secure application from further debugger accesses prior to delivering the pre-programmed chip to Customer B.

This can be done by changing the debug access level (DAL) to one. Changing the debug access level can be done using the Device Programing Tool by following the steps below:

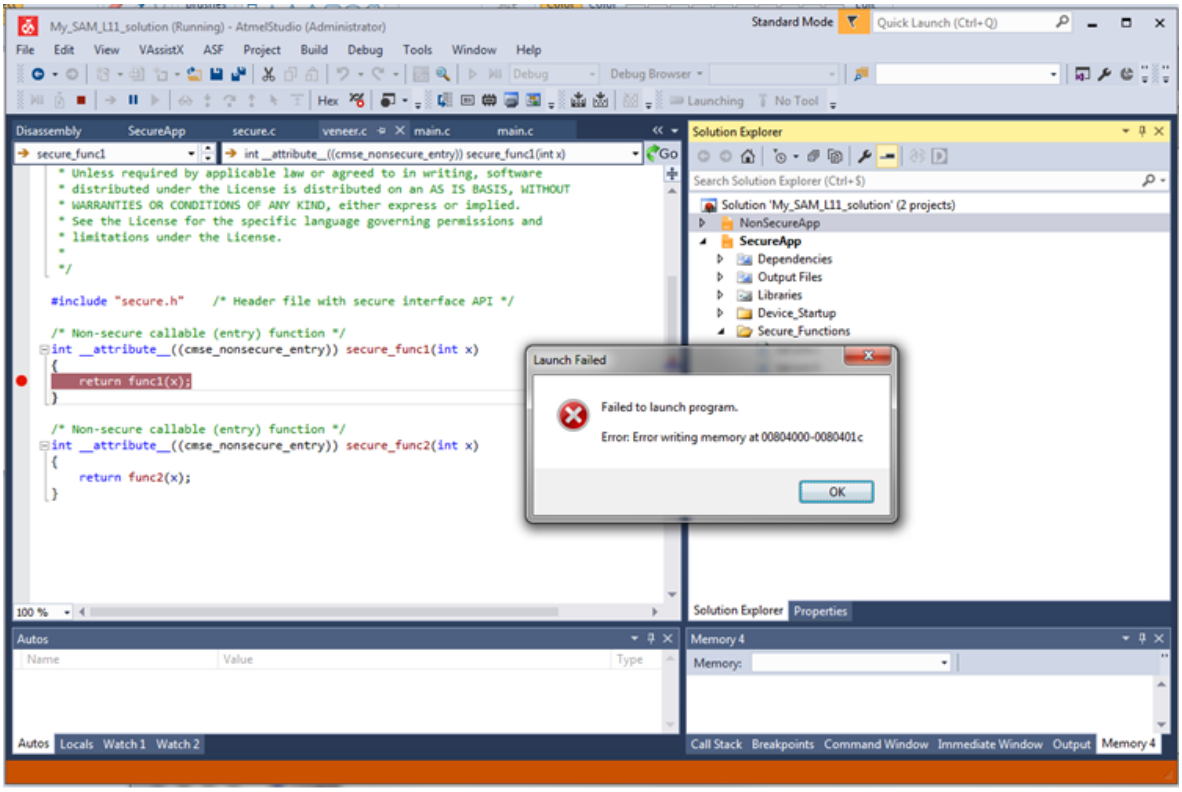
1. Close the debug session (if running).
2. Open the Device Programming tool by selecting the *Tools* > *Device Programming*.
3. Send the DAL1 command to the target SAM L11 Device:
    3.1. Select the Programming tool and click **Apply**.
    3.2. Select Memories.
    3.3. Select Set DAL 1.
    3.4. Click Change DAL
    3.5. Verify that no problem is reported by the Device Programing tool.

Figure 3-22. Changing DAL Using the AS7 Device Programming Tool



As a result, setting DAL to one (DAL1) prevents any future debug access to the Secure application and requires a ChipErase All command to re-enable the access to the Secure memories (DAL2). This can be tested by relaunching a debug session and running the code.

Figure 3-23. Launch Failed Error on DAL Protected Area



Important: Further development with the device requires the use of a standalone Non-Secure project. Refer to the Create and Configure a Non-Secure Project (Customer B). A ChipErase_ALL command (CE2) can be issued if the Secure application still needs to be debugged.

3.2 Create and Configure a Non-Secure Project (Customer B)

In the Customer B context, the development starts with a preprogrammed SAM L11 device that contains a DAL1 protected Secure application with predefined veneers.

It is mandatory for Customer A to provide some Non-Secure resource attribution descriptions, and Non-Secure callable function API information to Customer B.

Ideally, the approach should be for Customer A to provide a Non-Secure project template to Customer B.

The following sections explain how to create and configure a Non-Secure project for a preprogrammed SAM L11 device with a DAL1 protected Secure application.

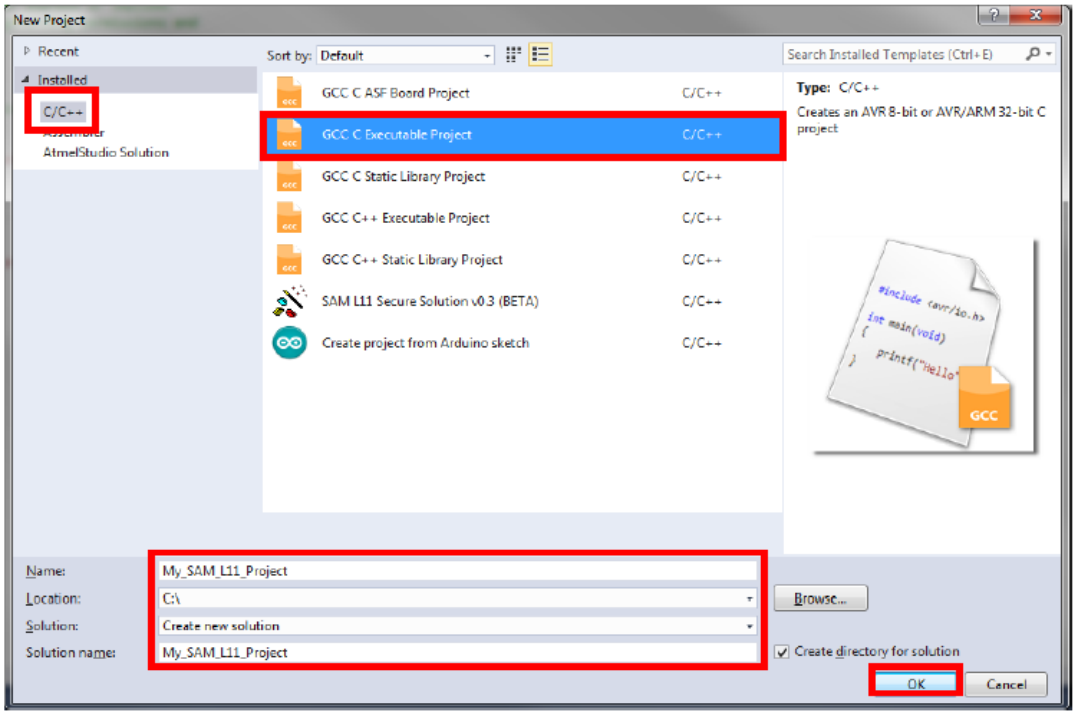
3.2.1 Project Creation

To create the project perform these actions::

1. Open a new Atmel Studio 7 instance.
2. Select *File* > *New* > *Project*.

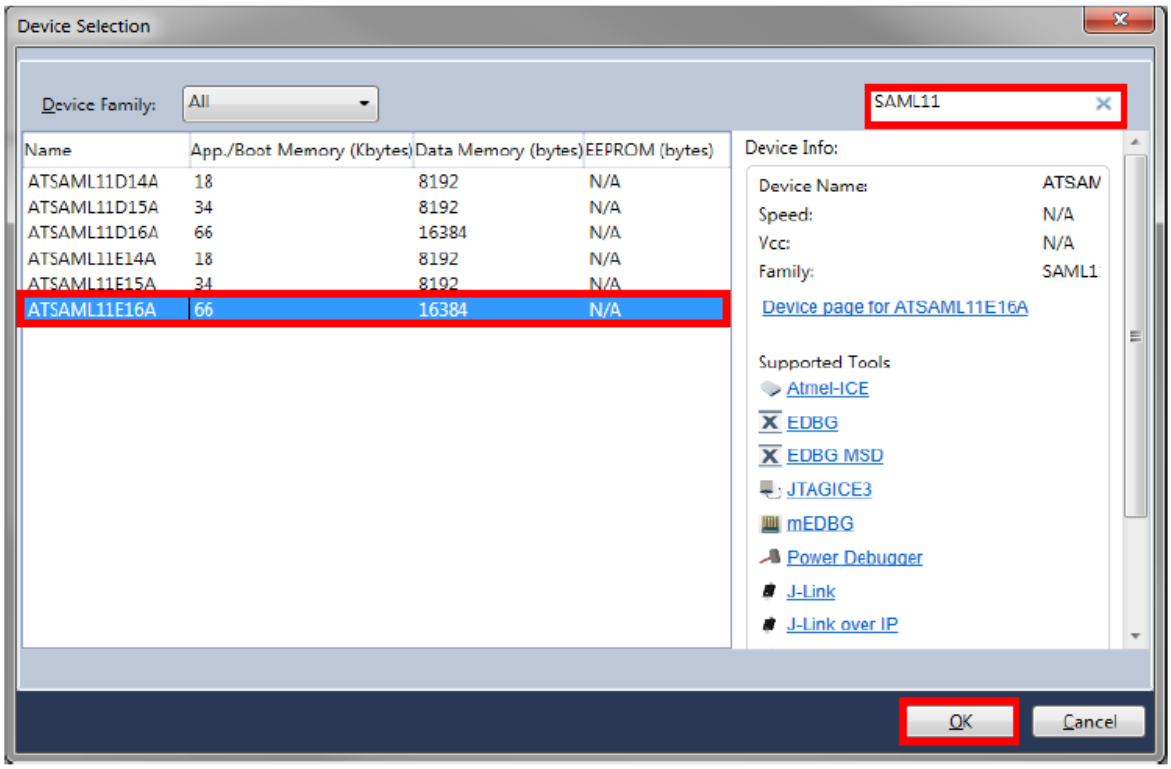
3. Configure the new project in the *New Project* window:
   3.1. Access the C/C++ tab.
   3.2. Select GCC C Executable Project.
   3.3. Enter details for Name, Location, Solution, and Solution Name.
   3.4. Click **OK**.

Figure 3-24. SAM L11 Standalone Non-Secure Project Creation Under AS7



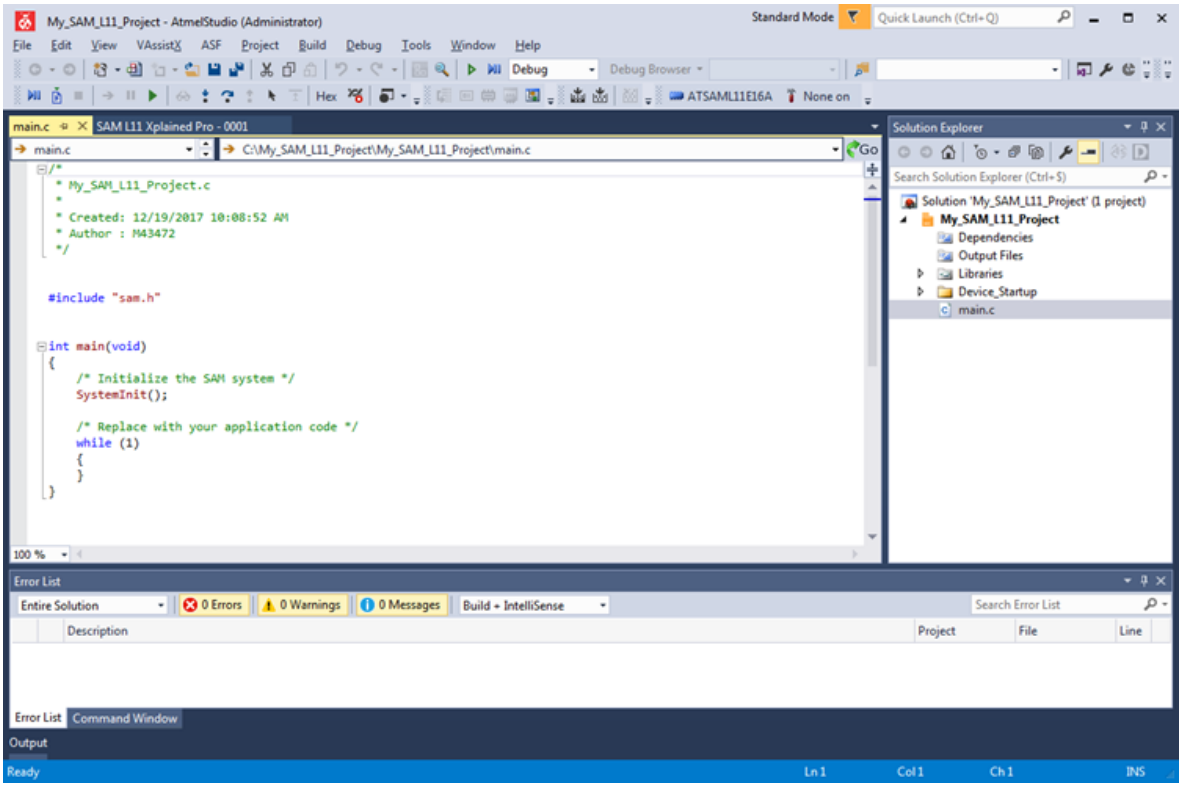
4. Select the ATSAML11E16A device in the Device Selection window, and then click **OK**.

Figure 3-25. SAM L11 Product Selection for New SAM L11 Standalone Non-Secure Project



The result is the Non-Secure project appear in Atmel Studio IDE, see image below.

Figure 3-26. Standalone SAM L11 Non-Secure Project Under AS7



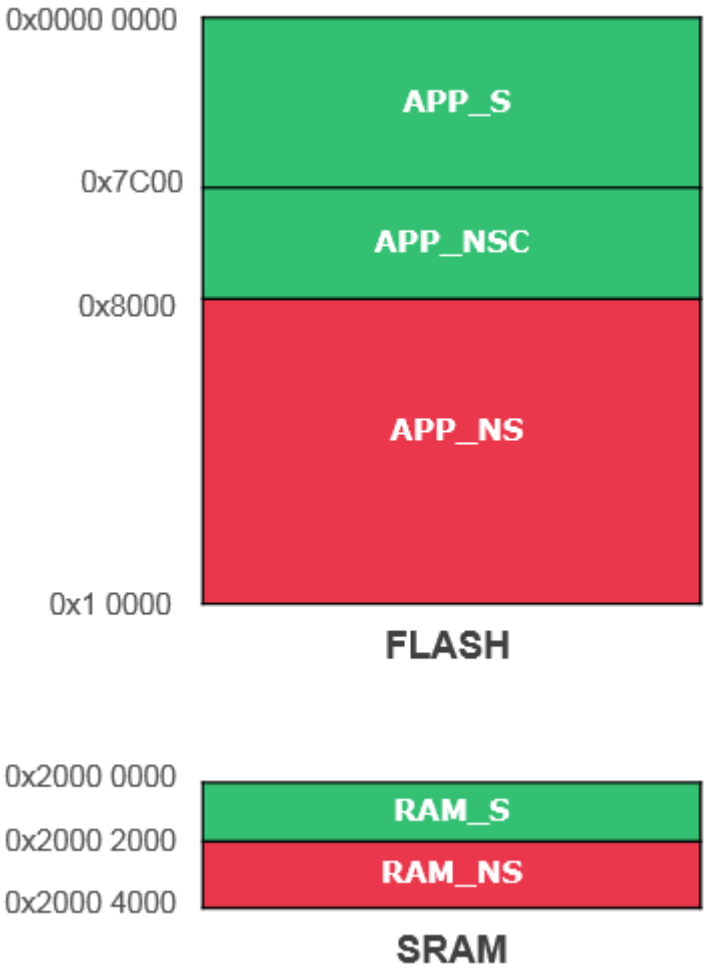
3.2.2 Project Configuration

Prior to starting Non-Secure project development for SAM L11, it is mandatory to perform these actions:

- Configure the project by aligning its linker file to the Secure and Non-Secure memories attribution predefined by Customer A.
- Add the Secure gateway library and veneer file and link them to the project.

3.2.2.1 Align Project Linker File to the SAM L11 Non-Secure Memories Attribution

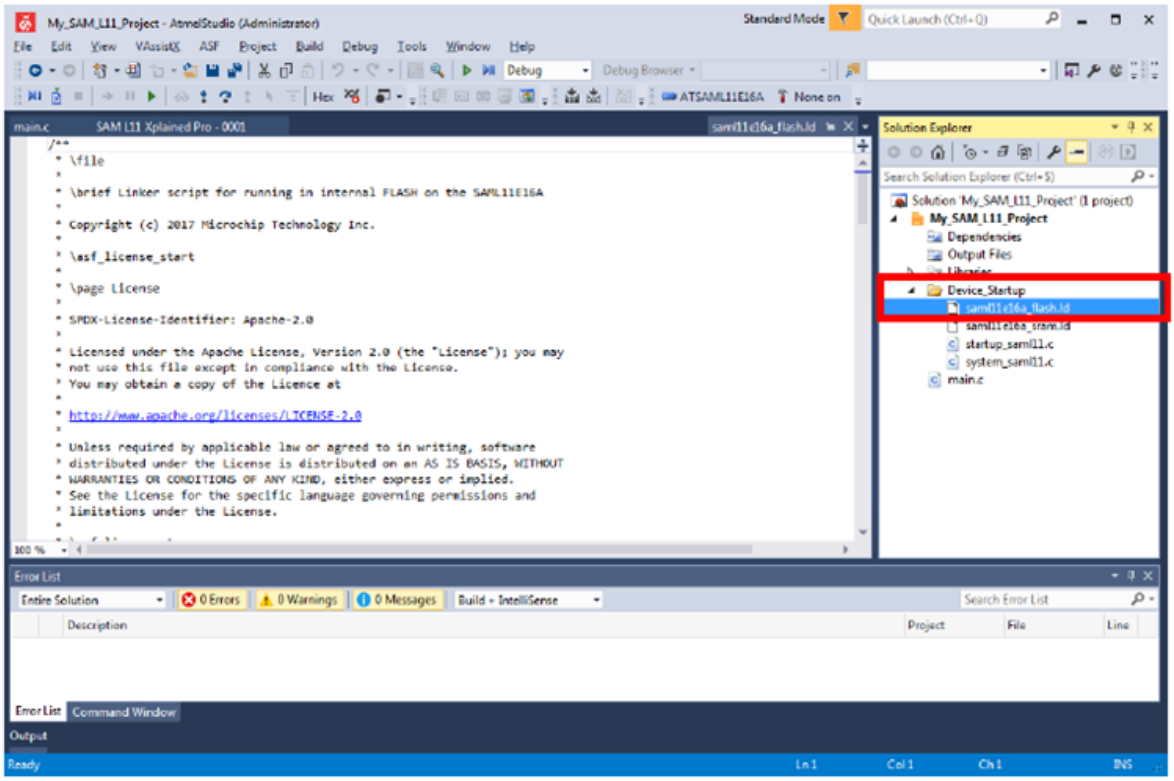
The following illustration provides how to modify the Non-Secure solution project linker file according to the following Secure and Non-Secure memory space.



Follow these steps to modify the Non-Secure solution project.

1. Open the project linker file: `Device Startup/saml11e16a_flash.ld`.

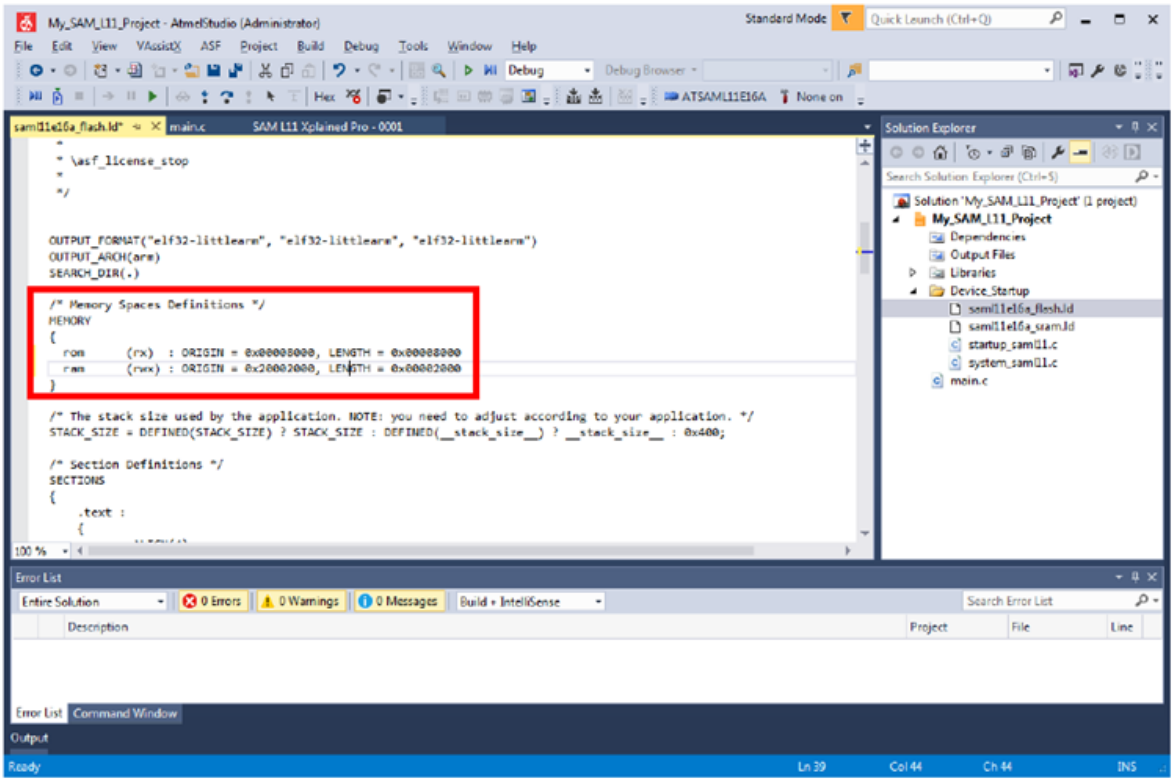
Figure 3-27. Non-Secure Project Linker File Location



2. Update the linker file memory space definitions according to the SAM L11 Non-Secure memory attribution.

```
/* Memory Spaces Definitions */
MEMORY
{
rom      (rx)   : ORIGIN = 0x00008000, LENGTH = 0x00008000
ram      (rwx)  : ORIGIN = 0x20002000, LENGTH = 0x00002000
}
```

Figure 3-28. Non-Secure Memory Address and Size Definition

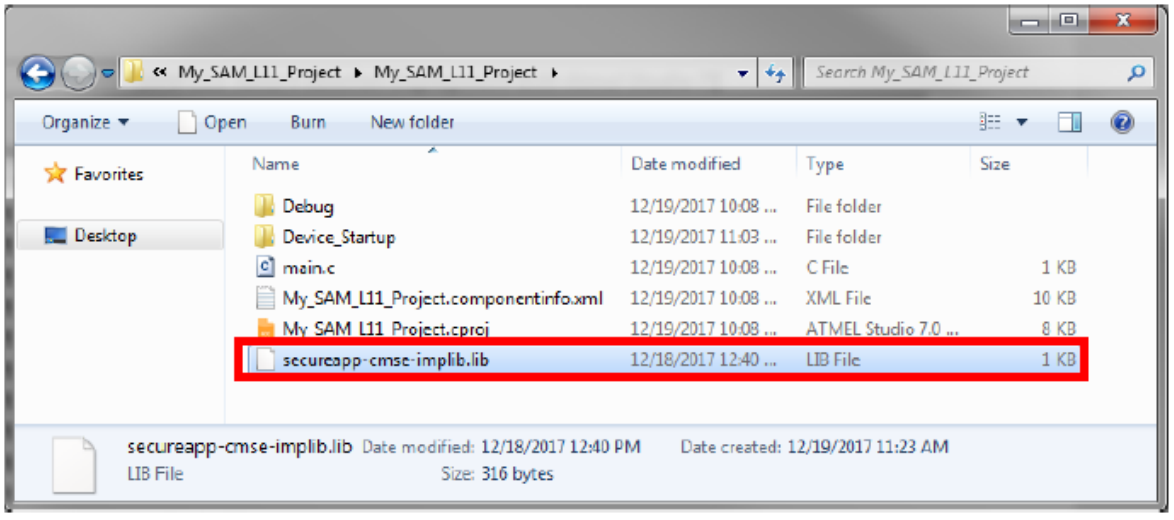


3.2.2.2 Add and Link the Secure Gateway Library to the Non-Secure Project

To add and link the Secure gateway library to the Non-Secure project, follow these steps:

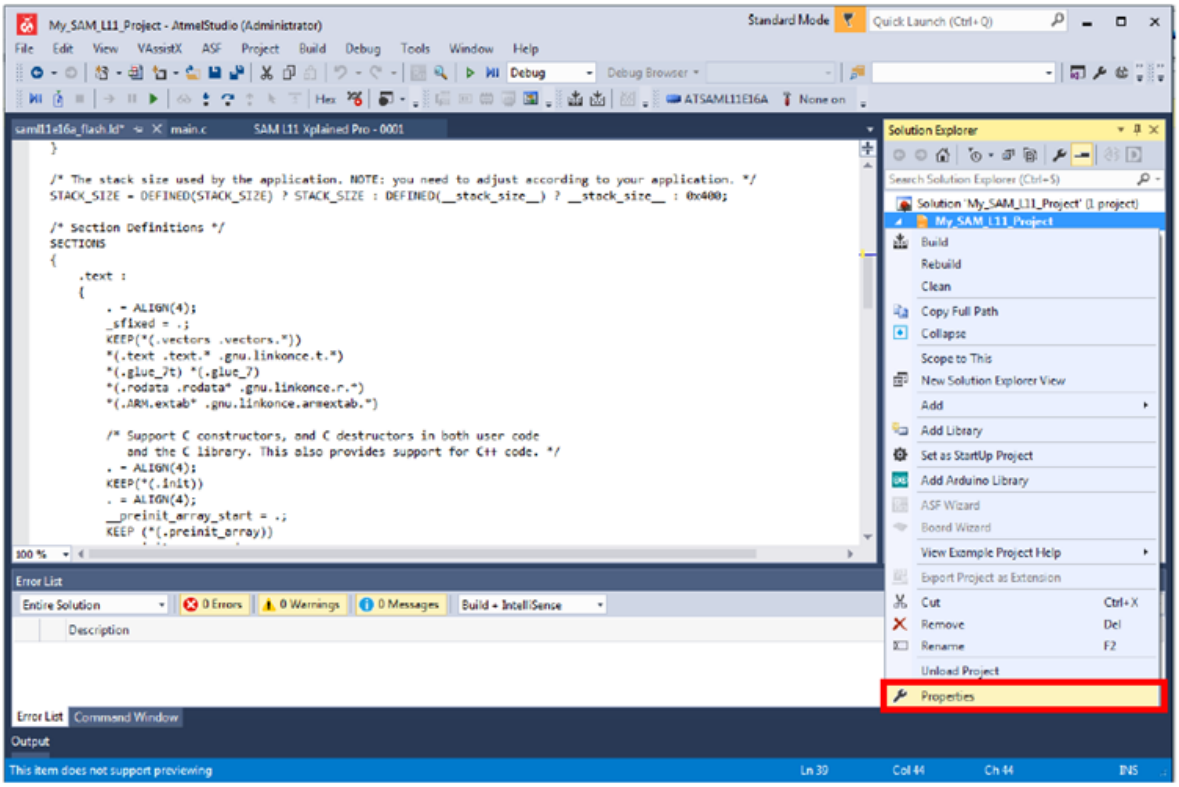
1. Copy the Secure project implib inside the Non-Secure project.

Figure 3-29. Secure Gateway Library File Inclusion in Non-Secure Project Sources



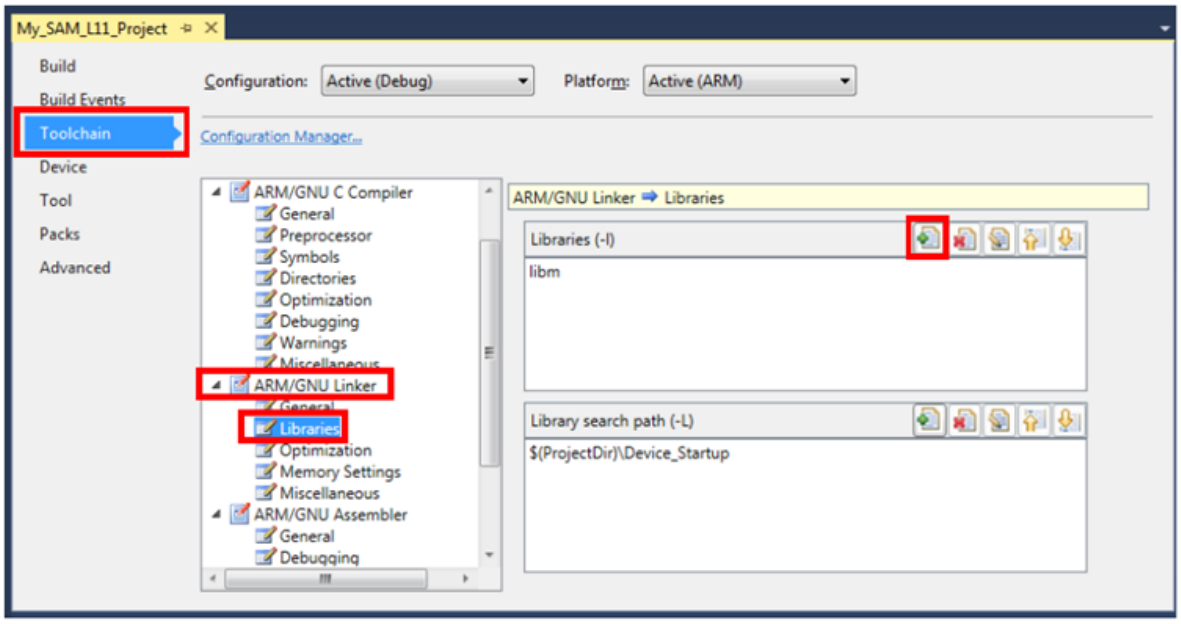
2. Under Atmel Studio 7, right click on the Non-Secure project and select Properties.

Figure 3-30. Access to Non-Secure Project Properties



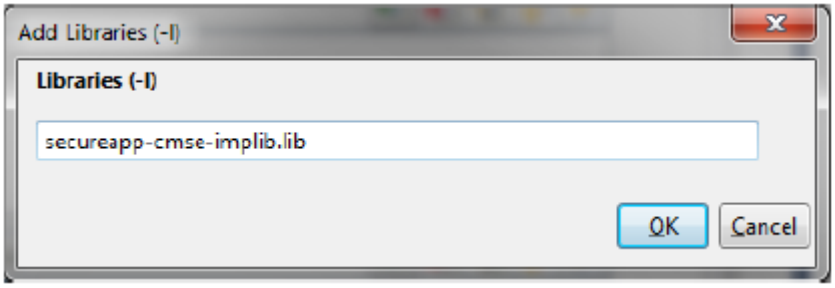
3. Add the Secure Project library by clicking the Add Item button in *Toolchain* > *ARM/GNU Linker* > *Libraries*.

Figure 3-31. Add New Library to the Link Option



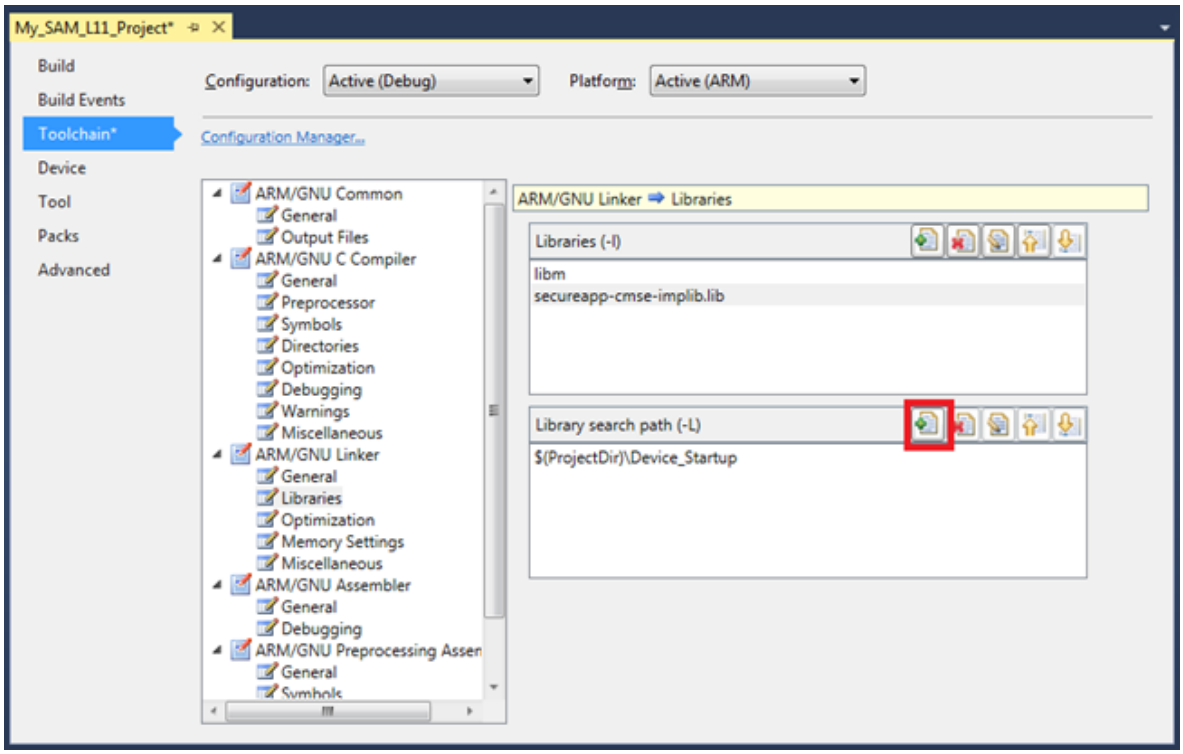
4. Enter the library name.

Figure 3-32. Adding Secure Gateway Library Name



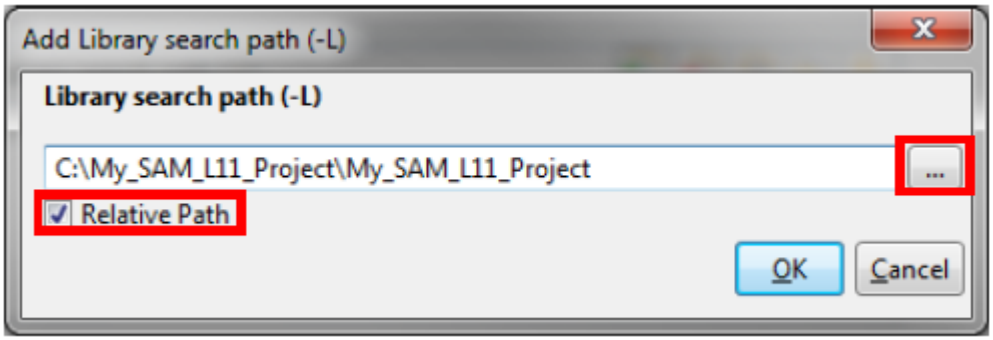
5. Add the Secure Project library path by clicking the *add Item* button in *Toolchain* > *ARM/GNU Linker* > *Libraries*.

Figure 3-33. Add New Library Search Path



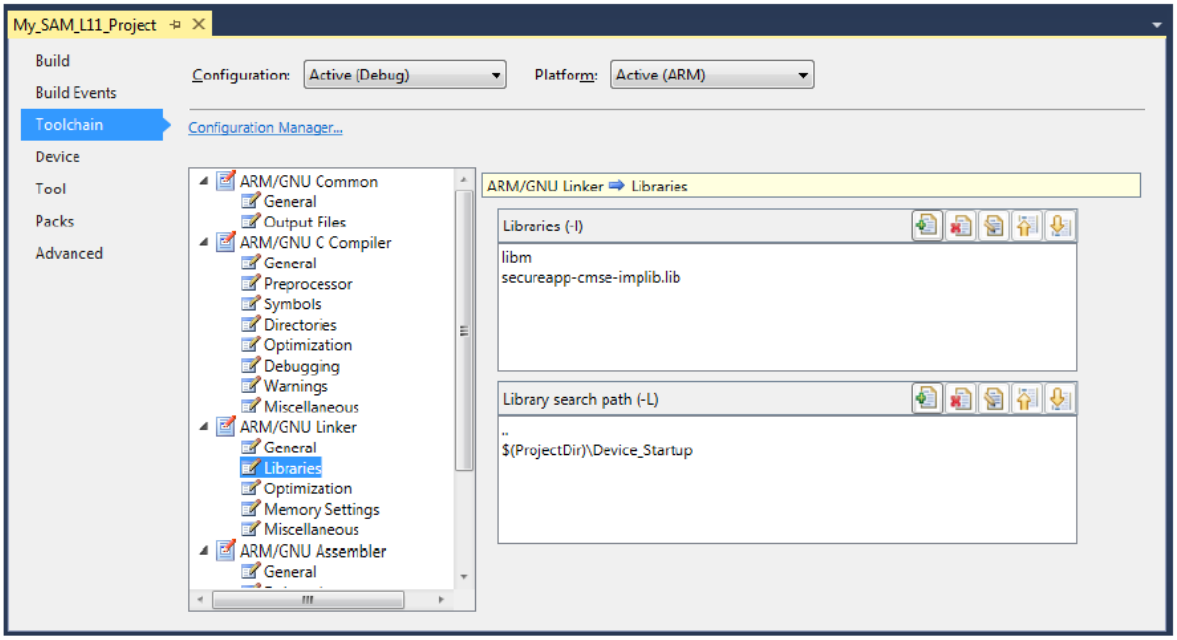
6. Click on the "…" button to browse and select the location of the secure project implib. Select "Relative Path" to ensure project portability.

Figure 3-34. Enter Relative Path to the Secure Gateway Library



7. Linker Libraries properties should be displayed as given in the image below.

Figure 3-35. Non-Secure Project Linker Libraries Configuration



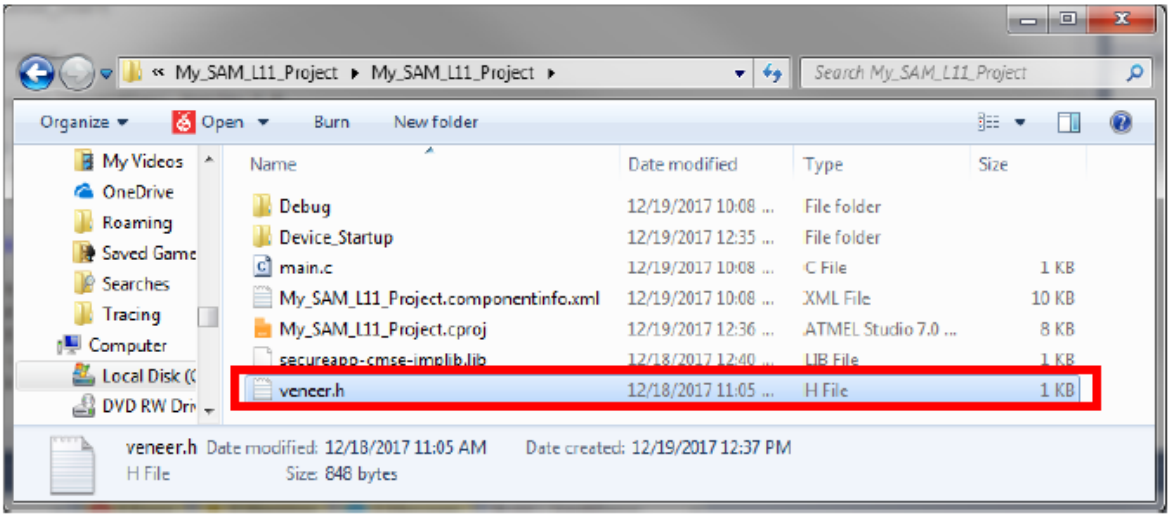
8. Clcik  (Save button) to save the project settings.

3.2.2.3 Add and Include Secure Gateway Header File

To add and include a secure gateway header file, perform these actions:

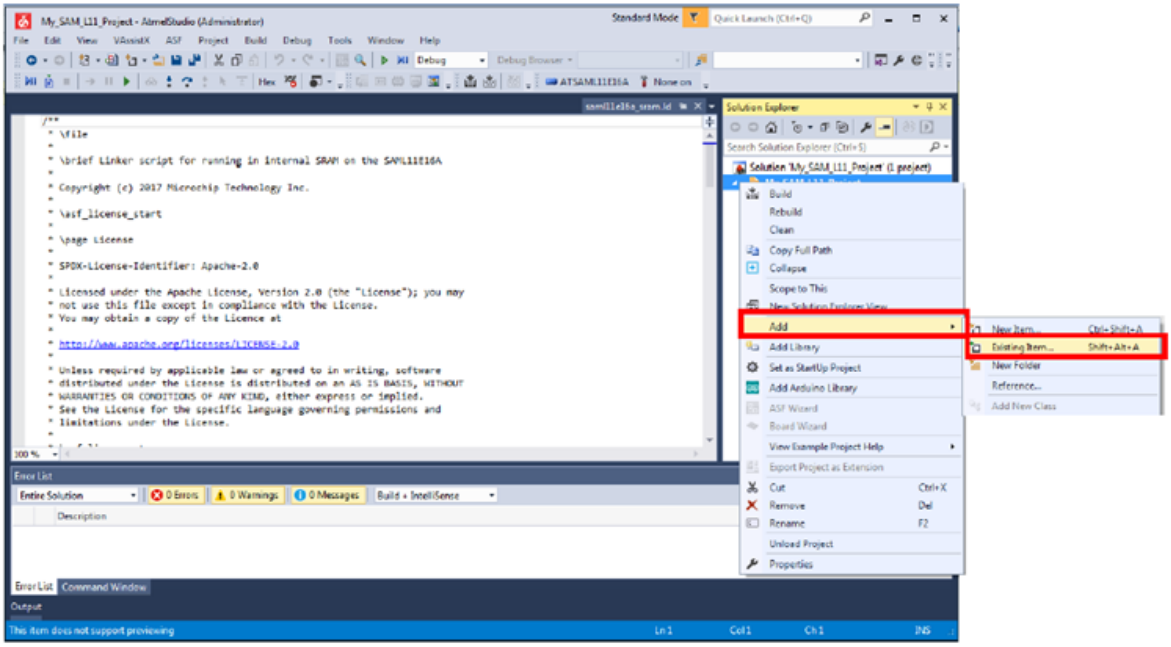
1. Copy the Secure gateway header file from the Secure project to the Non-Secure project.

Figure 3-36. Secure Gateway Header File Inclusion in Non-Secure Project Sources



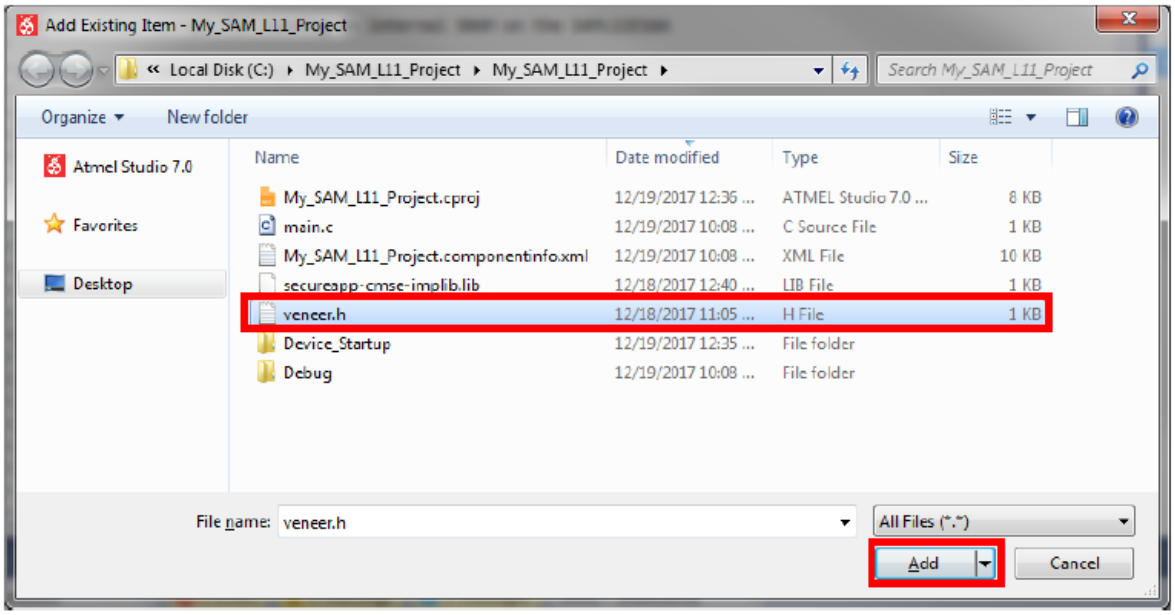
2. Right click "Non-Secure project" in the solution explorer, and then select *Add* > *Existing Item*.

Figure 3-37. Secure Gateway Header File Inclusion in AS7 Solution Explorer



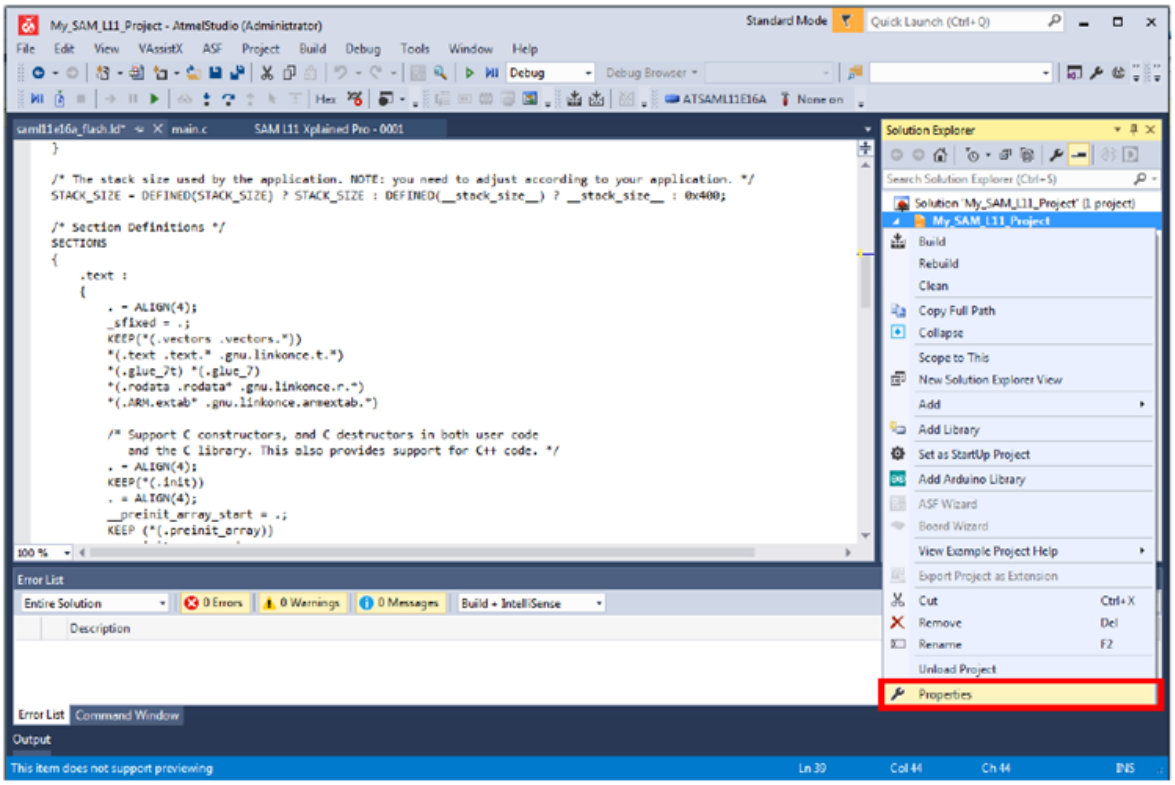
3. Select the Secure gateway header file, and then click **Add**.

Figure 3-38. Secure Gateway Header File Inclusion in Non-Secure Project



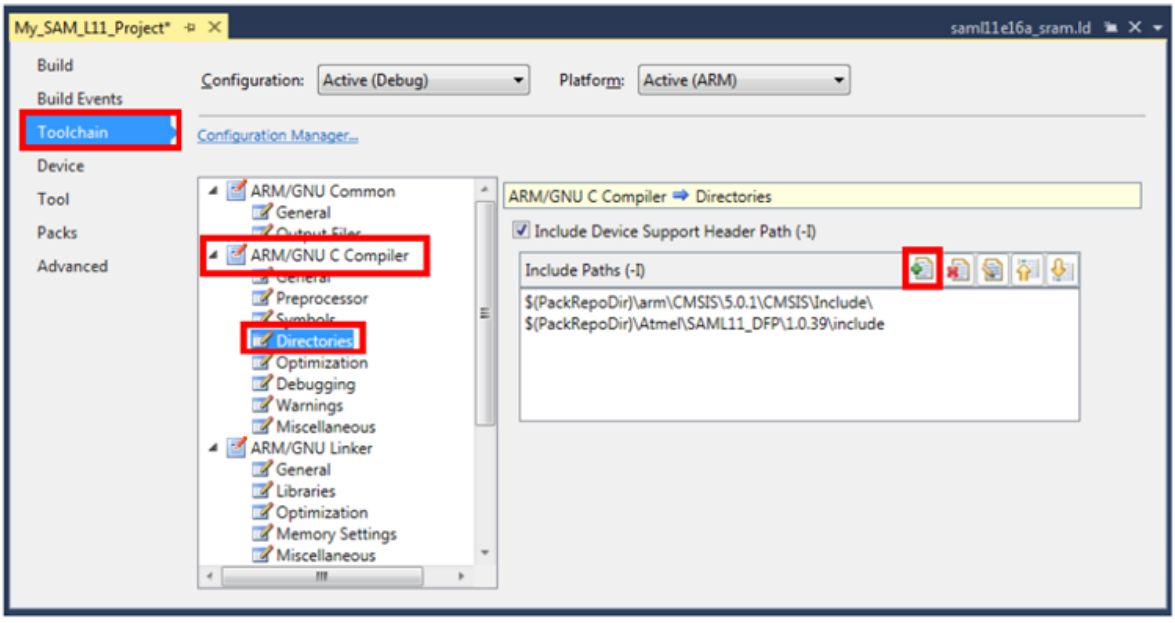
4. Right click "Non-Secure project" in the Solution explorer, and then select Properties.

Figure 3-39. Accessing Non-Secure Project Properties Under AS7



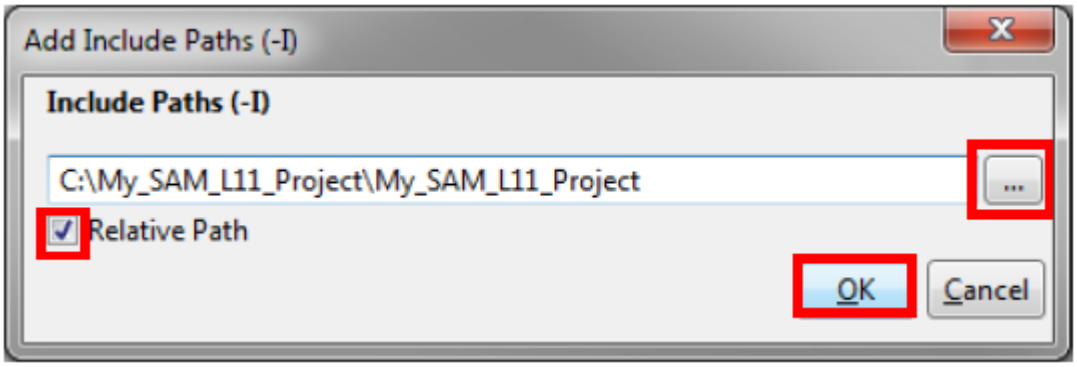
5. In the Non-Secure project property window, select *Toolchain* > *ARM/GNU C Compiler* > *Directories* and then click **Add Item**.

Figure 3-40. Adding New Compiler Directory to Non-secure Project



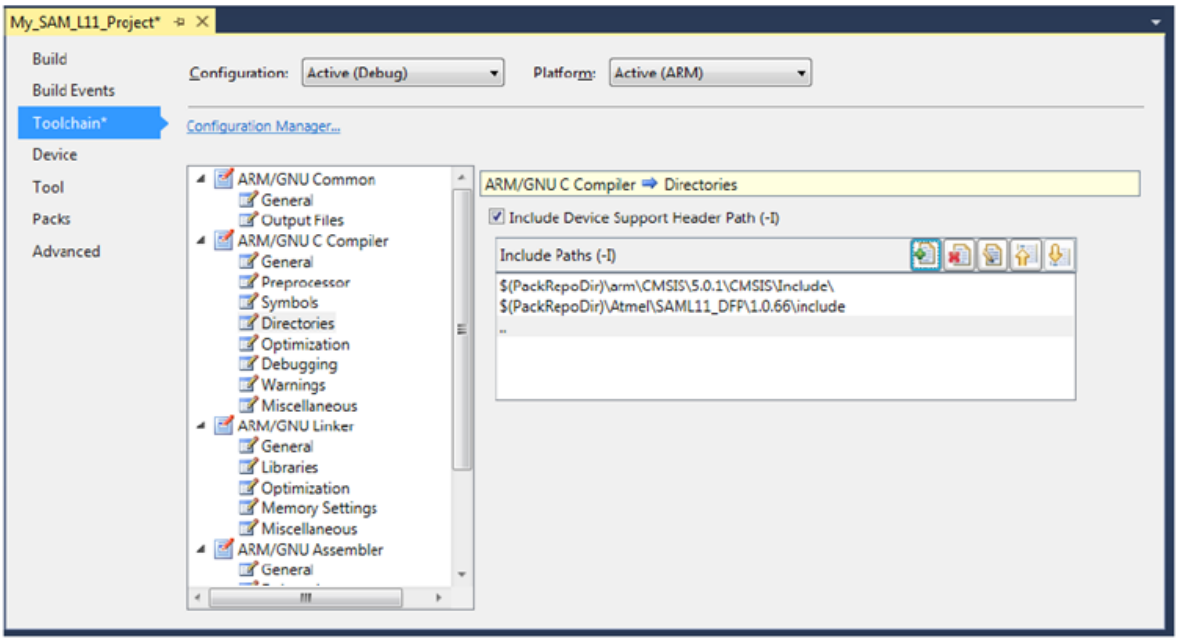
6. Click on the "…" button to browse, and then select the location of the `veneer.h` file. Select "Relative Path" to ensure project portability and then click **OK**.

Figure 3-41. Include Secure Gateway Library Path in Compiler Directory



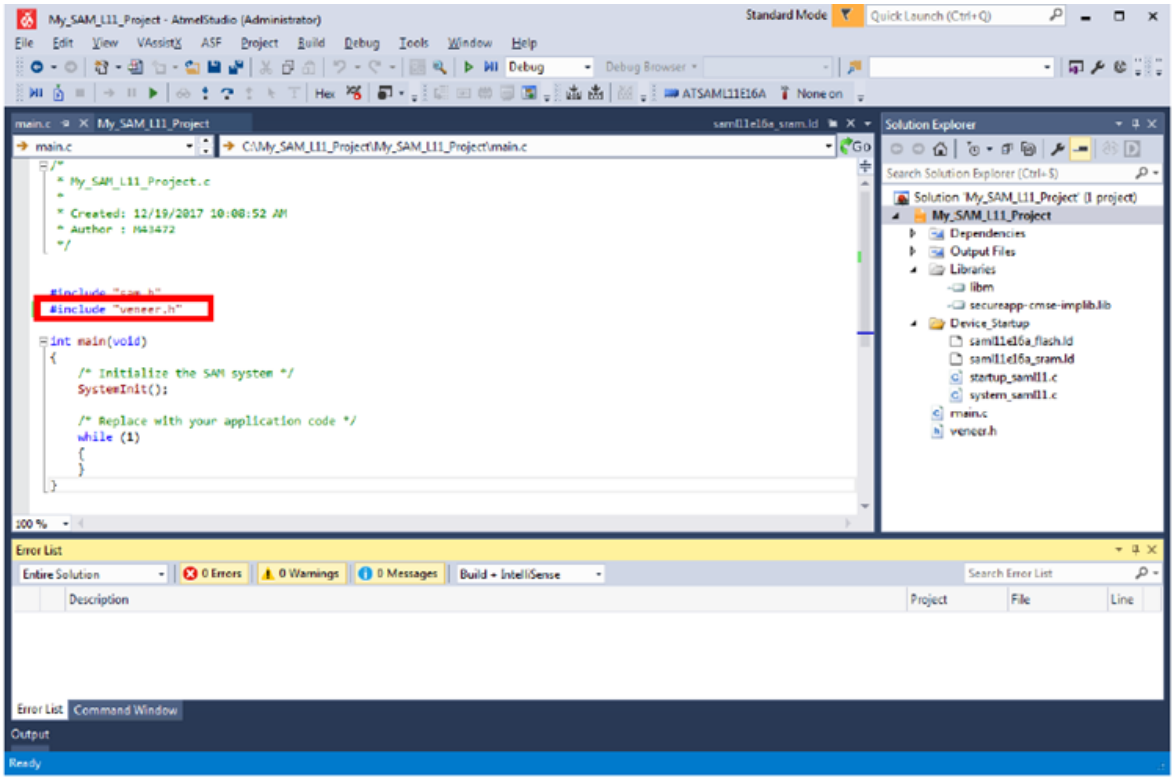
7. The Compiler Directories properties will be displayed as follows.

Figure 3-42. Non-Secure Project Compiler Directories Parameters



8. Press  (Save button) to save the project settings.
9. Add the following line at the beginning of the `main.c` file to include the Secure gateway library:

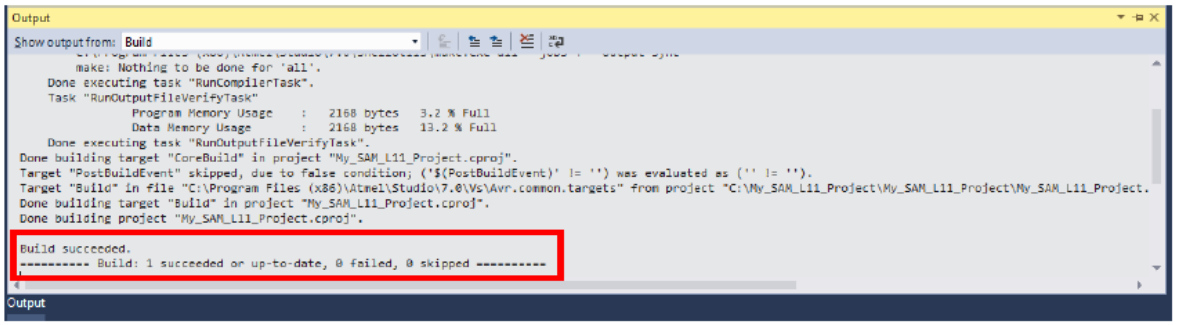
Figure 3-43. veneer.h Inclusion in Non-Secure Project main.c File



10. Click  (Save button) to save the modification to the `main.c` file.
11. Click  (Build Project button).
12. Verify that no error is reported by the build process.
13. Launch debug session and confirm it is working.

Important: This requires the previous Secure application to run, if not the application will hang and do not jump to the Non-Secure one.

Figure 3-44. Non-Secure Project Successful Build



14. Launch the debug session and check the project is working.

Important: Debugging the Non-Secure project requires a compatible preprogrammed Secure application that configures and starts the Non-Secure execution. If this Secure application is not present on the chip, the debug process will hang.

4. How to Define and Use Secure and Non-Secure Peripherals

4.1 TrustZone for ARMv8-M Extension to Integrated Peripherals

The SAM L11 extends the concept of TrustZone to the ARMv8-M memory partitioning.

The management of the peripheral security attribution is done through the Peripheral Access Controller (PAC).

Each peripheral security attribution is defined by programming their related User Row (UROW) fuse.

During Boot ROM execution, the NONSECx fuses from the NVM User row are copied in the PAC peripheral NONSECx registers so that they can be read by the application.

Figure 4-1. PAC.NONSECx Register Description

NONSECA	7:0	GCLK	SUPC	OSC32KCTRL	OSCCTRL	RSTC	MCLK	PM	PAC
	15:8			AC	PORT	FREQM	EIC	RTC	WDT
	23:16								
	31:24								
NONSECB	7:0				HMATRIXHS	DMAC	NVMCTRL	DSU	IDAU
	15:8								
	23:16								
	31:24								
NONSECC	7:0	ADC	TC2	TC1	TC0	SERCOM2	SERCOM1	SERCOM0	EVSYS
	15:8			TRAM	OPAMP	CCL	TRNG	PTC	DAC
	23:16								
	31:24								

Important: The peripherals security attribution cannot be changed during application run-time. Any changes to the User Row fuses require a reset of the SAM L11 device.

Peripherals can be categorized in three groups depending on their PAC security attribution and their internal secure partitioning capabilities (standard/mix-secure):

- Non-Secure peripheral: A standard peripheral configured as Non-Secure in the PAC. The security attribution of the whole peripheral is defined by the associated NONSECx fuse set to one. Secure and Non-Secure accesses to the peripheral are granted.
- Secure peripheral: A standard peripheral configured as Secure in the PAC. The security attribution of the whole peripheral is defined by the associated NONSECx fuse set to zero. Secure accesses to the peripheral are granted where Non-Secure accesses are discarded (Write is ignored, Read 0x0), and a PAC error is triggered.
- Mix-Secure peripherals: The SAM L11 embeds five mix-secure peripherals, such as PAC, NVMCTRL, PORT, EIC and EVSYS, that allow part of their internal resources to be shared between the Secure and Non-Secure applications:
  - When a mix-secure peripheral is secured (NONSECx fuse set to zero), the Secure world can allocate internal peripheral resources to the Non-Secure world using dedicated registers.

– When a mix-secure peripheral is Non-Secure (NONSECx fuse set to one), the peripheral behaves as a standard Non-Secure peripheral. Secure and Non-Secure accesses to the peripheral register are granted.

Note: For additional information, refer to the "Security" Chapter of the SAM L11 Family Data Sheet.

4.2 Peripherals Interrupts Handling

The code examples given in the following section shows how to allocate a Non-Secure handler and set the interrupt priority of a specific interrupt vector.

4.2.1 Non-Secure Interrupt Handling

Secure: `main.c` or `driver.c`

```
…
/* Set EIC EXTINT[1] Interrupt as Non-Secure at core level */
NVIC_SetTargetState(EIC_1_IRQn);

/* Set EIC EXTINT[1] as Non-Secure interrupt (Mix-Secure Use) */
EIC_SEC->NONSEC.reg = (EIC_NONSEC_EXTINT(1<<1));
EIC_SEC->NSCHK.reg = (EIC_NSCHK_EXTINT(1<<1));
…
```

Non-secure: `main.c` or `driver.c`

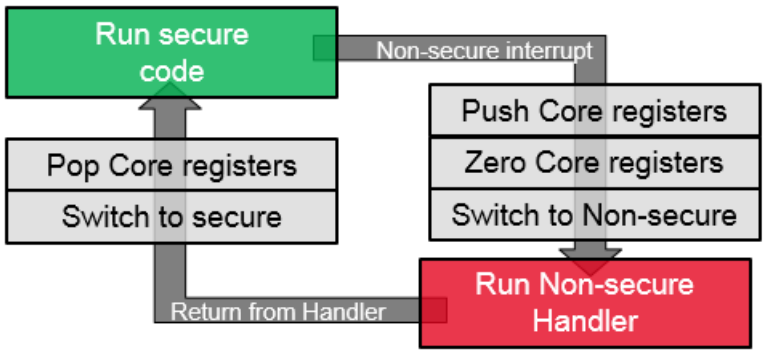
```
…
/* Enable Interrupt at peripheral level*/
EIC->INTENSET.bit.EXTINT = EIC_INTENSET_EXTINT(1<<1);

/* Enable EXTINT[1] Non-Secure Interrupt */
NVIC_EnableIRQ(EIC_1_IRQn);
…
```

Non-Secure `main.c` or `driver.c` or `interrupt.c`

```
/* Enable EXTINT[1] Non-Secure Interrupt */
void EIC_1_Handler(void) {
    /* Clear EIC EXTINT[1]interrupt flag */
EIC->INTFLAG.reg |= EIC_INTFLAG_EXTINT(1<<1);
…
}
```

The following figure displays the automatic clear of the CPU registers on the Secure to Non-Secure handler transition:

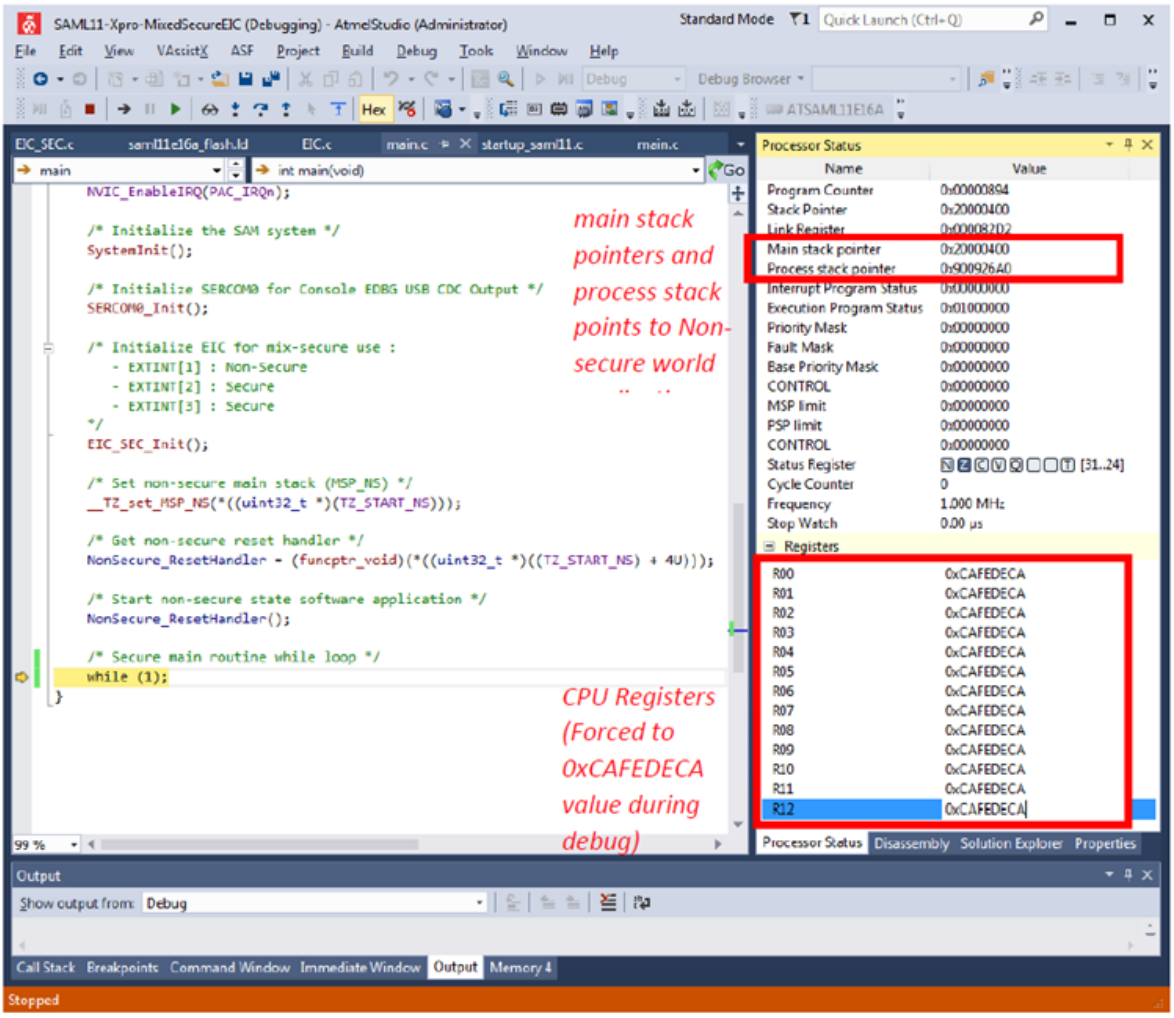
Figure 4-2. Cortex-M23 Interrupt Mechanism



Follow these steps for Interrupt Mechanism:

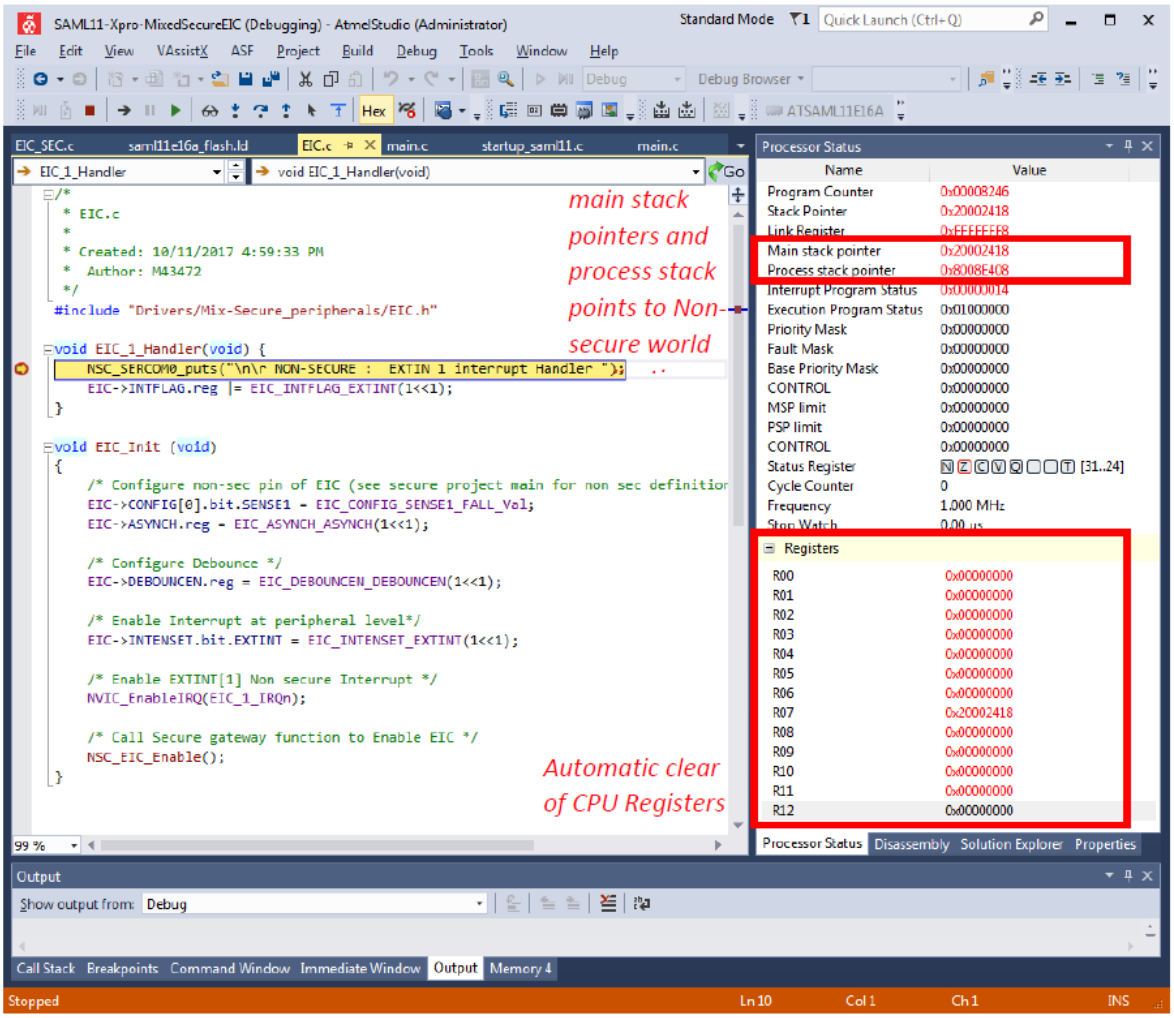
1. Processor status prior to Non-Secure interrupt.

Figure 4-3. CPU Registers Filled with Pattern in Secure Application



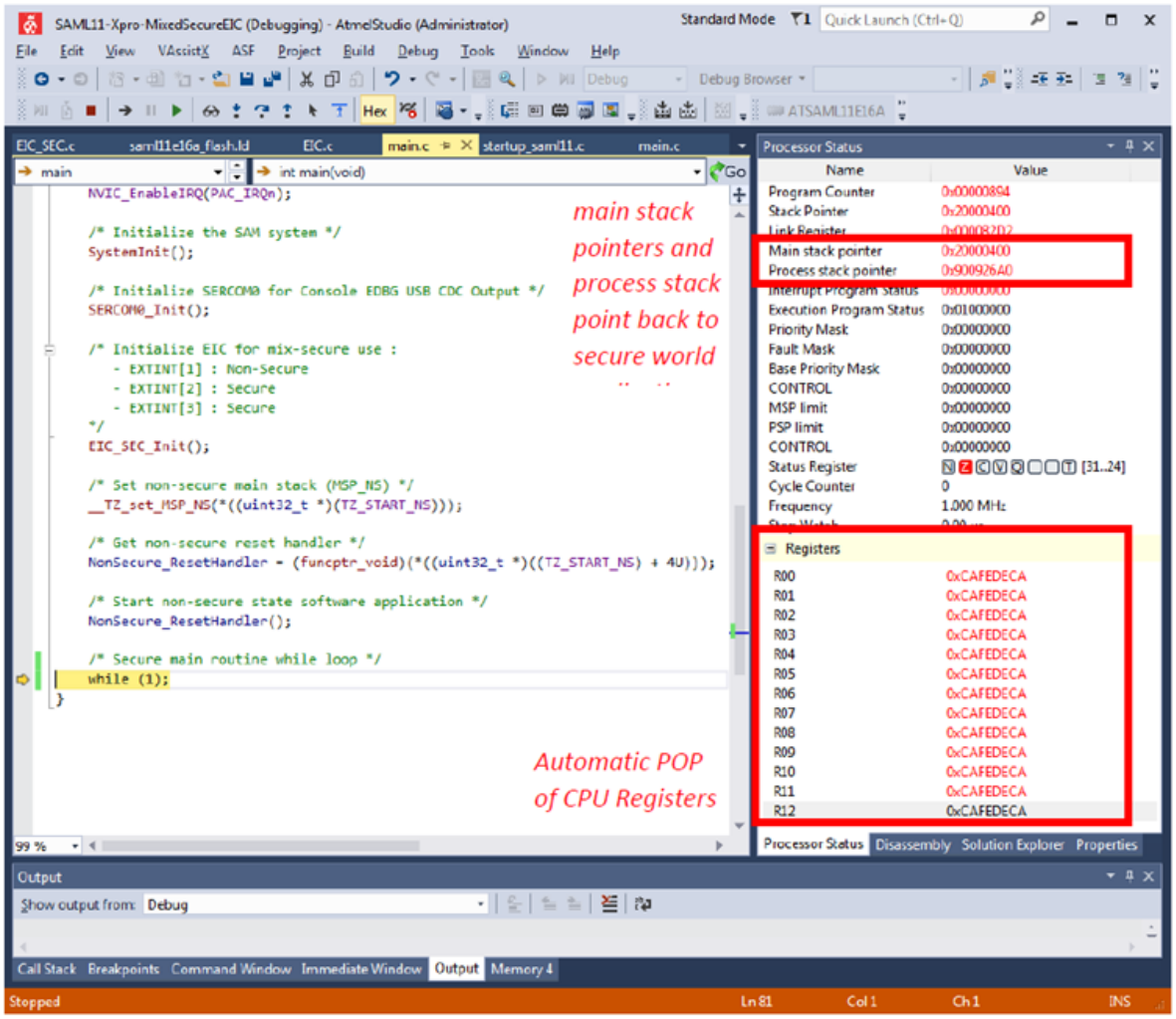
2. Processor status during Non-Secure interrupt handler execution.

Figure 4-4. Register Cleared Prior to Execute Non-Secure Interrupt Handler



3. Processor status during Non-Secure interrupt handler execution.

Figure 4-5. Automatic Pop of Register Content When Back to Secure World



4.3 How to Use Non-Secure Peripherals

When a peripheral is allocated to the Non-Secure world, both Secure and Non-Secure applications can access the peripheral registers.

At the Non-Secure world level, TrustZone for ARMv8-M considerations are totally transparent for the developer. On the secure world side, the application should ensure that all the system resources required by the peripheral are preconfigured or available to the Non-Secure world, such as PORT I/O, NVIC, DMA, EVSYS, and so on.

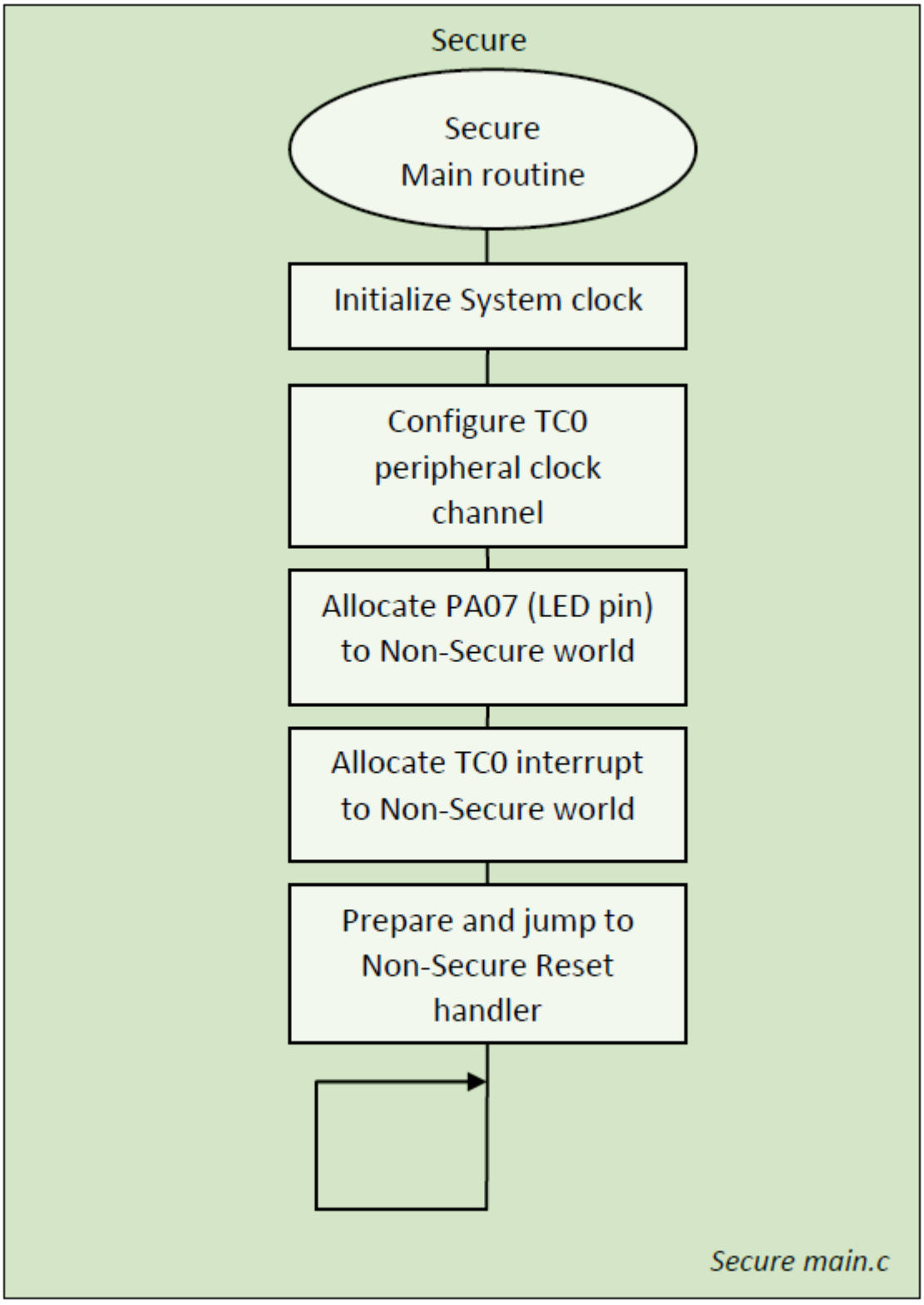
4.3.1 Non-Secure Timer Counter 0 (TC0) Peripheral Use Case Example

This section provide an example of a Non-Secure TC use case.

In this use case, the Secure project is in charge of allocating PORT and TC peripherals to the Non-Secure world, setting system clocks, and then jumping to the Non-Secure application.

The following figure displays the flowchart for the use case.

Figure 4-6. Secure Main Routine Flow Chart



The key software aspects of the Secure code are as follows:

- TC0 allocation to the Non-Secure world in fuses definition (define USER_WORD_6 as 0x00000010 in Secure application).

```
/* USER_WORD_X: User Row (UROW) Word X definitions */
#define USER_WORD_0 0xB08F437F    /* BOD, Watchdog and Misc settings */
#define USER_WORD_1 0xFFFFF8BB    /* Watchdog and Misc settings */
#define USER_WORD_2 0x40082080    /* Memories Security Attribution: AS = 0x80, ANSC = 0x20,
RS = 0x40 */
#define USER_WORD_3 0xFFFFFFFF    /* User Row Write Enable */
#define USER_WORD_4 0x00000000    /* Peripherals Security Attribution Bridge A (NONSECA) */
#define USER_WORD_5 0x00000000    /* Peripherals Security Attribution Bridge B (NONSECB) */
#define USER_WORD_6 0x00000010    /* Peripherals Security Attribution Bridge C (NONSECC) */
```

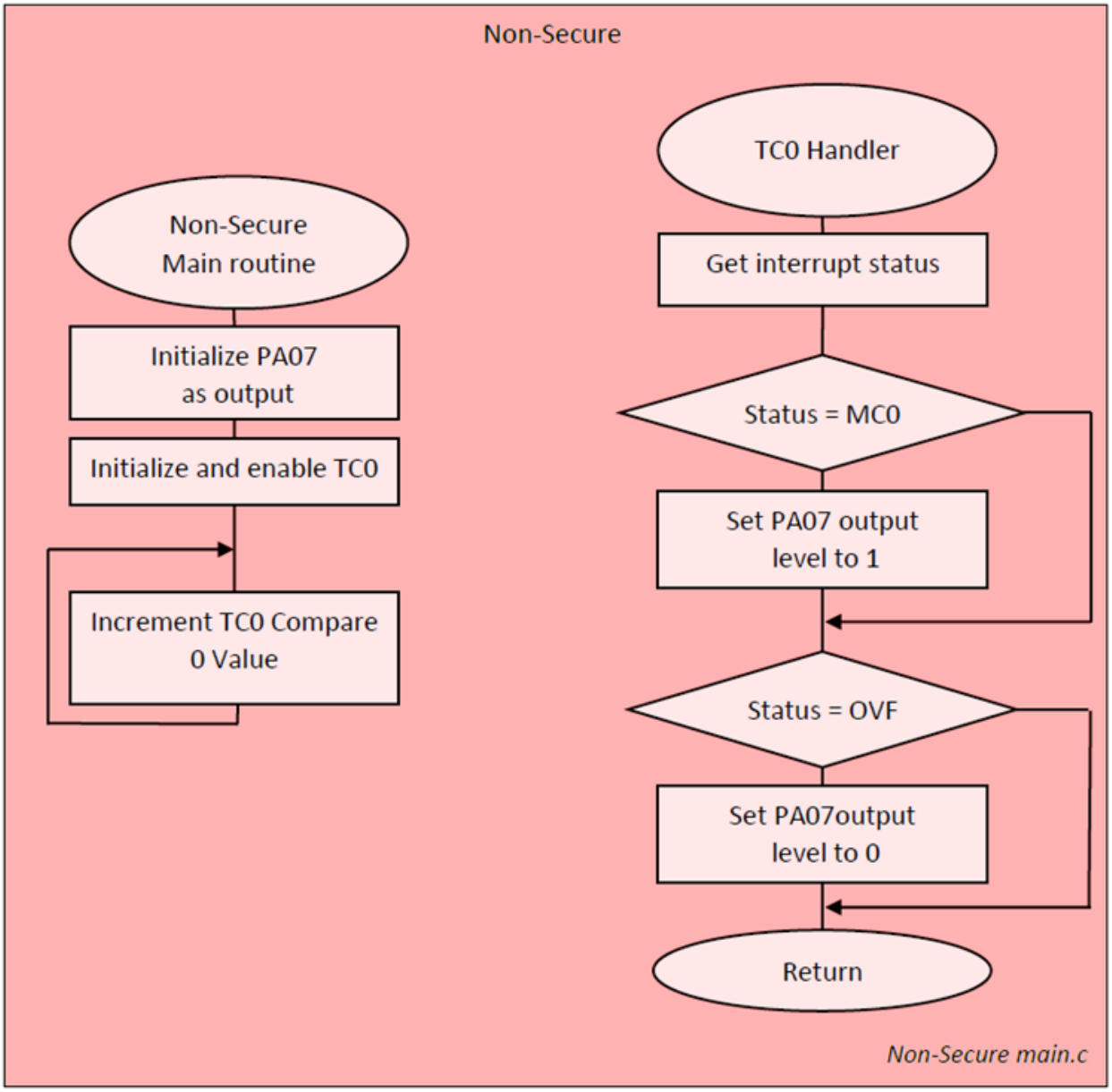
- TC0 peripheral clock configuration and interrupt allocation to the Non-Secure world (Secure application).

```
/* Secure main() */
int main(void)
{
    uint32_t ret;
    funcptr_void NonSecure_ResetHandler;

    /* Initialize the SAM system */
SystemInit();

/* Configure TC0 peripheral clock channel */
    GCLK->PCHCTRL[14].reg =(GCLK_PCHCTRL_GEN(0) | //
       GCLK_PCHCTRL_CHEN);  // Enable Generator

    /* Allocate PA07 (LED pin) to Non Secure world */
    PORT_SEC->Group[0].NONSEC.reg = (PORT_PA07);

    /* Allocate TC0 interrupt to Non-Secure world*/
    NVIC_SetTargetState(TC0_IRQn);

    /* Set Non-Secure main stack (MSP_NS) */
    __TZ_set_MSP_NS(*((uint32_t *)(TZ_START_NS)));

    /* Get Non-Secure reset handler */
    NonSecure_ResetHandler = (funcptr_void)(*((uint32_t *)((TZ_START_NS) + 4U)));

    /* Start Non-Secure state software application */
    NonSecure_ResetHandler();

    while (1)
    {
        __NOP();
    }
}
```

This Non-Secure application illustrates the use of a Non-Secured TC and I/O port by generating a PWM signal on the PA07 pin.

The following figure illustrates the flowchart of this process.

Figure 4-7. Non-Secure Main Routine Flow Chart



The TC0 peripheral and PORT PA07 are allocated to the Non-Secure world. The Non-Secure application can access them as standard peripherals without interaction with the Secure world.

4.4 How to Use Secure Peripherals

When a peripheral is allocated to the Secure world, only Secure accesses to its registers are granted, and interrupt handling should be managed in the Secure world only. Two different software development approaches can be followed depending on the software interaction requirements between Secure and Non-Secure projects to use this peripheral.

4.4.1 Driving Secure Peripheral Without Non-Secure SW Interactions

When working with peripherals that do not require specific interaction with the Non-Secure world, the Secure world will drive them as a standard peripheral without any specific TrustZone for Cortex-M considerations.

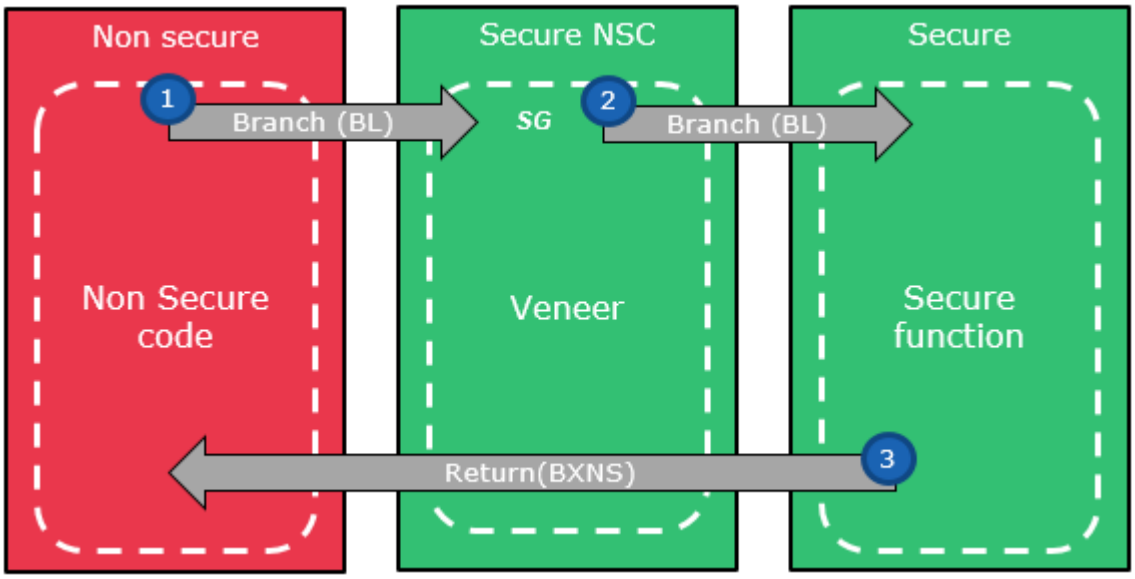
4.4.2 Driving Secure Peripheral With Non-Secure SW Interactions

If interactions between Non-Secure and Secure worlds are required to drive the Secure peripheral, the Secure application must provide Non-Secure callable APIs and callbacks to the Non-Secure world.

4.4.2.1 Non-Secure Callable APIs

The Secure gateway decouples the addresses of the Non-Secure callable APIs (stored in NSC regions) from the rest of the Secure code. All the project Secure gateways are expected to be placed in NSC memory, where all other code from the Secure executable is expected to be placed in the Secure memory regions. This limits the amount of code that can potentially be accessed by the Non-Secure state. This placement is under the control of the developer.

Figure 4-8. Non-Secure Callable APIs Mechanism

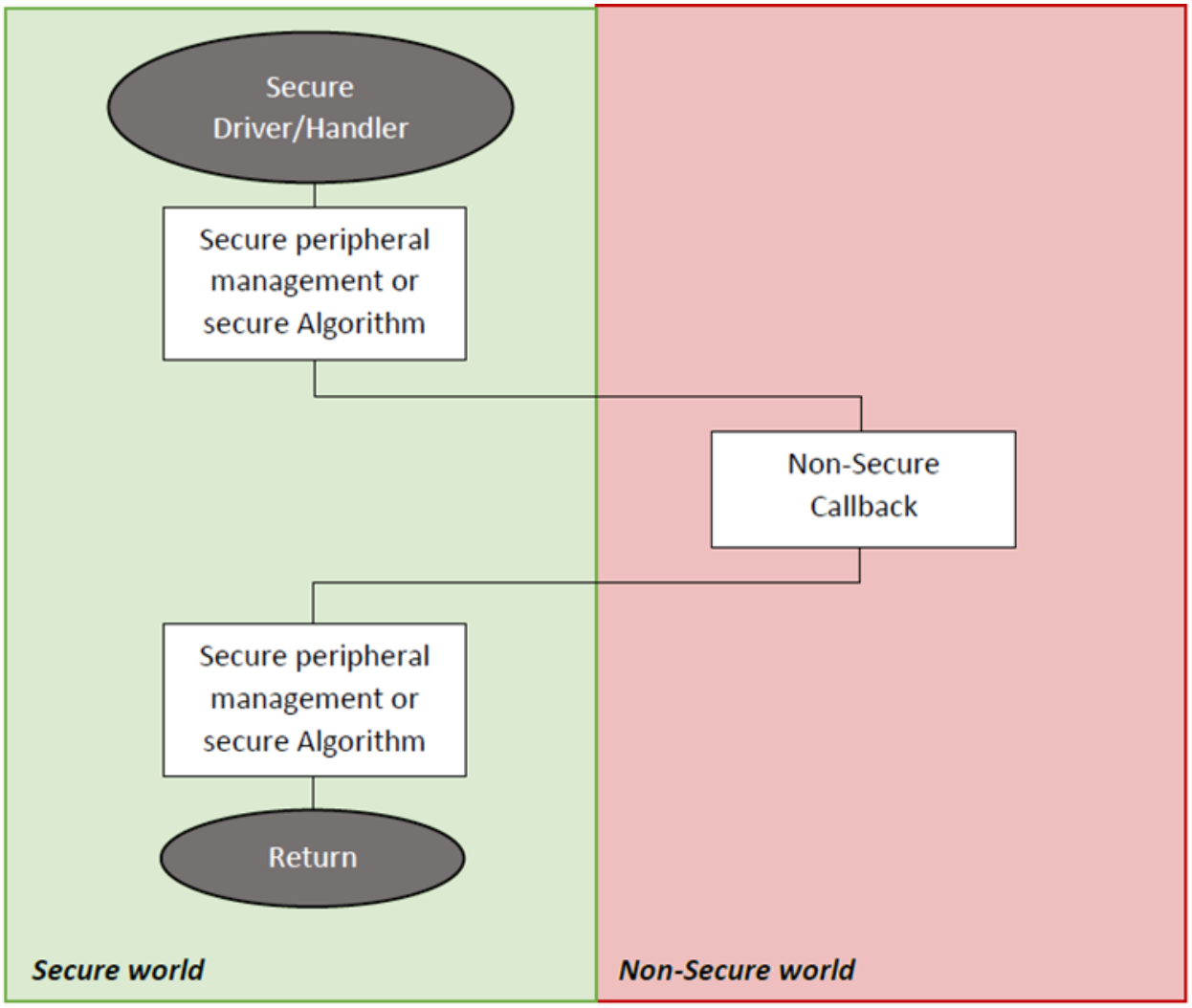


Refer to the Secure and Non-Secure Functions Call for more details.

4.4.2.2 Non-Secure Software Callbacks

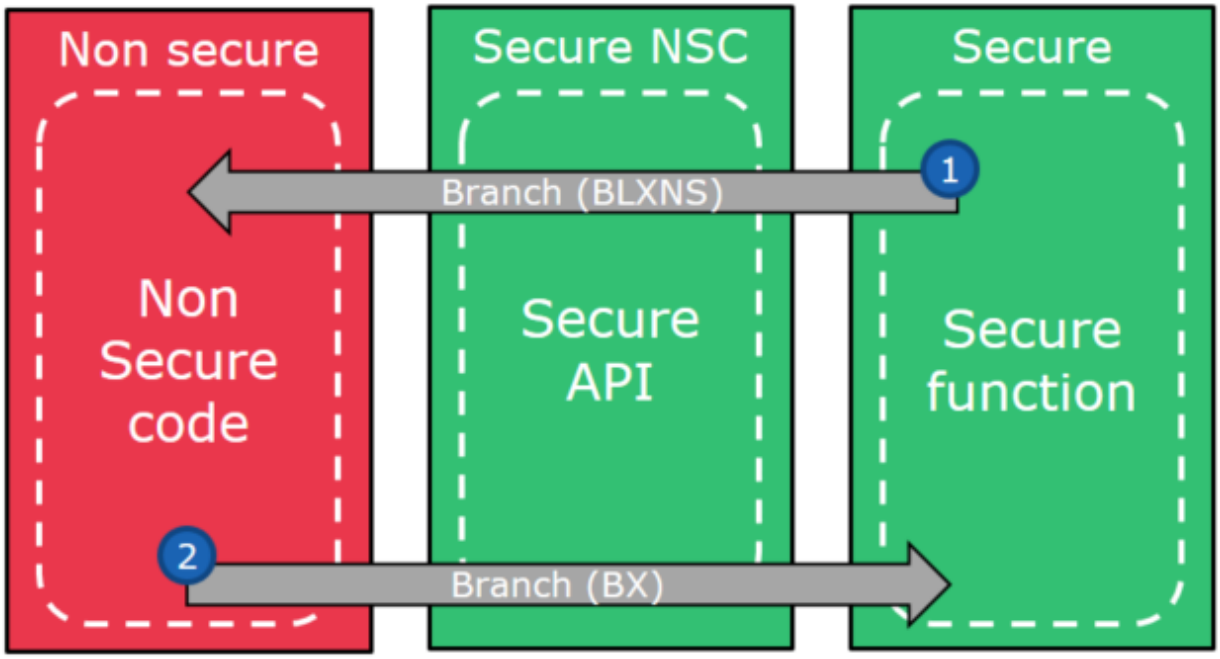
The Secure project should define and use software callbacks to execute functions from the Non-Secure world. This is a consequence of separating Secure and Non-Secure code into separate executable files.

Figure 4-9. Non-Secure Software Callbacks Flow Chart



The management of callback functions is done using the BLXNS instruction. The following figure and code illustrate the process.

Figure 4-10. Non-Secure Software Callback Mechanism



Note: A wrong use of pointers can lead to security weakness by enabling execution of any Secure functions by the Non-Secure code. To overcome this disadvantages, ARM provides a set of CMSE functions based on the new Cortex-M23 TT instructions.

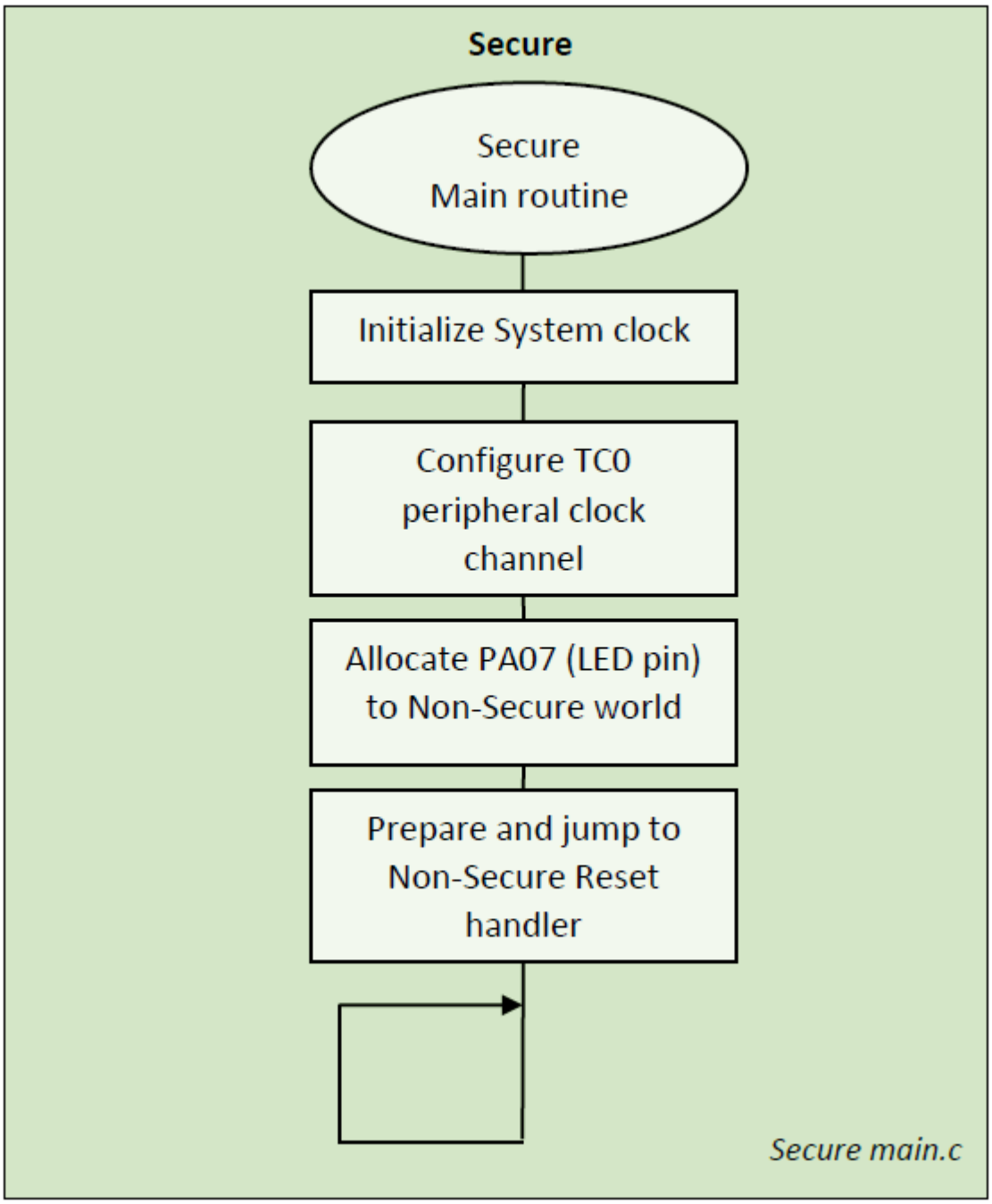
In the previous figure, the CMSE function, *cmse_check_pointed_object*, is used to return the Secure state of a specific address based on the product Secure memory attribution.

4.4.2.3 Secure Timer Counter 0 (TC0) Peripheral Use Case

This section provides an example of a Secure TC use in Secure and Non-Secure world . In this use case, the Secure project is in charge of configuring system resources and managing the TC peripheral.

It provides specific TC0 APIs and Non-Secure callbacks to the Non-Secure world. The following figure displays the secure main function flowchart:

Figure 4-11. Secure Main Routine Flowchart



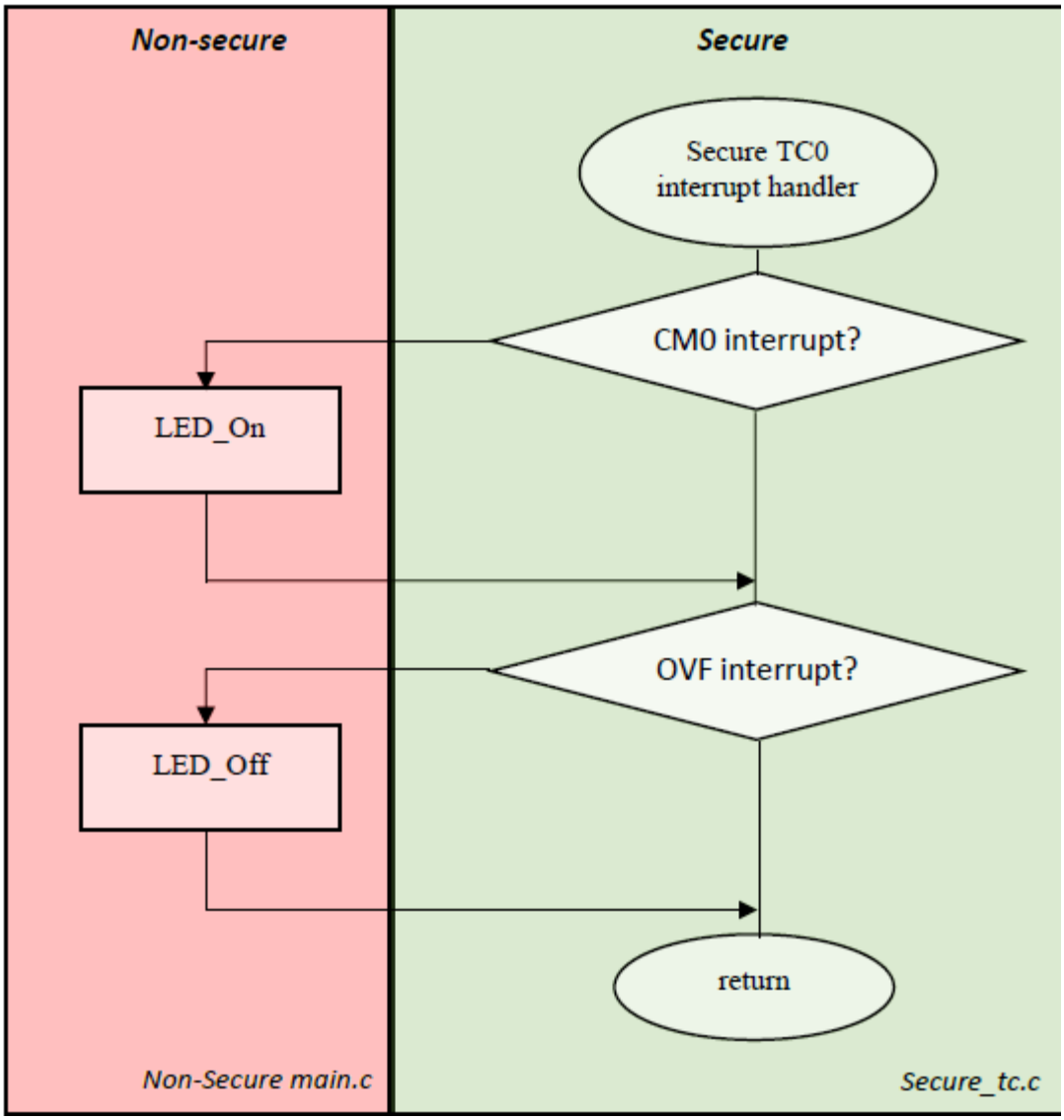
The following APIs or veneers are provided to Non-Secure world to drive TC0 peripheral from Non-Secure world:

- `tc0_compare_0_interrupt_callback_register(secure_void_cb_t pfunction);`
- `tc0_overflow_interrupt_callback_register(secure_void_cb_t pfunction);`
- `tc0_init(void);`
- `tc0_set_duty_cycle(uint8_t duty_cycle);`

The Non-Secure makes use of secured TC0 through APIs/veneers provided by the secure world and generates a PWM signal on PA07 pin. The following figures display the flowcharts of the application and the interaction with the secure world.

Figure 4-12. Non-Secure Main Routine Flow Chart

Figure 4-13. Secure TC Handler Flow Chart



4.5 How to Use Mix-Secure Peripherals

The SAM L11 embeds five Mix-Secure peripherals, which allow part of their internal resources to be shared between Secure and Non-Secure worlds.

The complete list of SAM L11 Mix-Secure peripherals and their shared resources are as follows:

- Peripheral Access Controller (PAC): Manages the peripherals security attribution (secure or non-secure).
- Non-volatile Memory Controller (NVMCTRL): Handles Secure and Non-Secure Flash region programming.
- I/O Pin controller (PORT): Supports individual allocation of each I/O to the Secure or Non-Secure applications.
- External Interrupt Controller (EIC): Supports individual assignment of each external interrupt to the Secure or Non-Secure applications.

- Event System (EVSYS): Supports individual assignment each event channel to the Secure or Non-Secure applications.

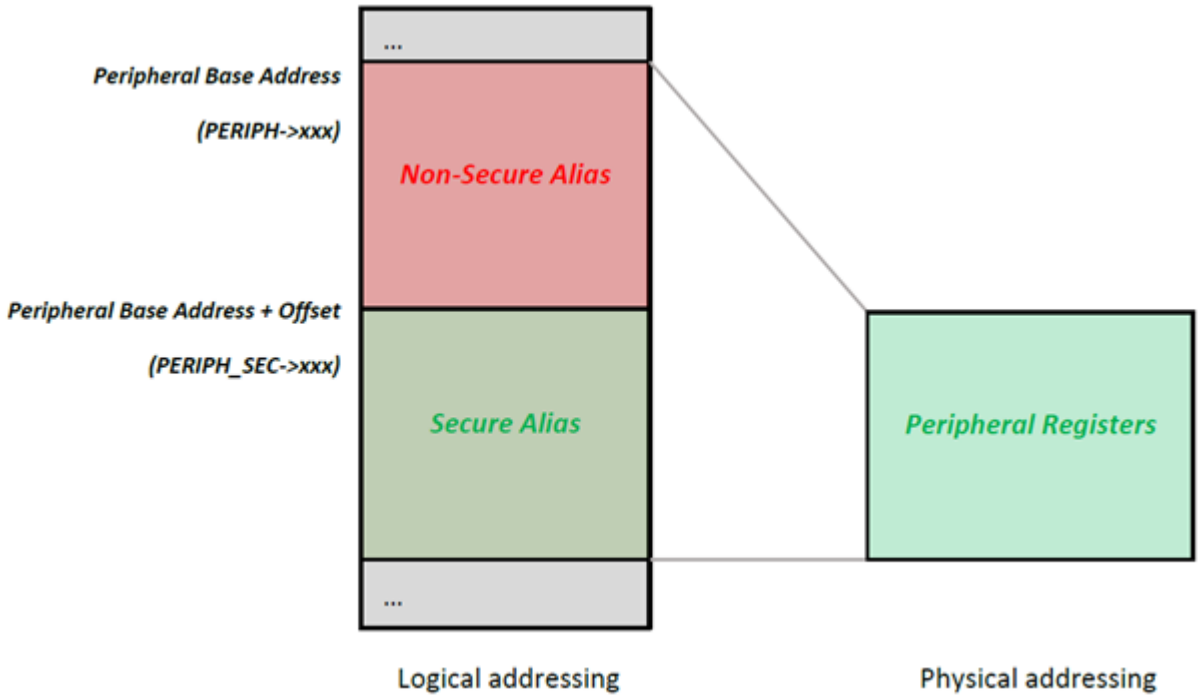
Refer to the Chapter "ARM TrustZone Technology for ARMv8-M" of the SAM L11 product data sheet for more details.

The capability for a mix-secure peripheral to share its internal resources depends on the security attribution of that peripheral in the PAC peripheral (PAC Secured or Not PAC Secured).

4.5.1 Mix-Secure Peripheral (PAC Secured)

When a mix-secure peripheral is PAC secured (associated PAC NONSECx fuses set to 0), the peripheral register is banked and accessible through two different memory aliases as shown in the following figure:

Figure 4-14. PAC Secured Mix-Secure Peripheral Registers Addressing



The Secure world can then independently enable Non-Secure access to the internal peripheral resources using its NONSEC register.

The following figure shows the External Interrupt Controller (EIC) NONSEC register.

Figure 4-15. NONSEC Register

Name: NONSEC
Offset: 0x40 [ID-00000c8b]
Reset: 0x00000000
Property: PAC Write-Protection, Write-Secure

Bit	31	30	29	28	27	26	25	24
	NMI							
Access	RW/R/RW							
Reset	0							

Bit	23	22	21	20	19	18	17	16
Access								
Reset								

Bit	15	14	13	12	11	10	9	8
Access								
Reset								

Bit	7	6	5	4	3	2	1	0
	EXTINT[7:0]							
Access	RW/R/RW	RW/R/RW	RW/R/RW	RW/R/RW	RW/R/RW	RW/R/RW	RW/R/RW	RW/R/RW
Reset	0	0	0	0	0	0	0	0

The NONSEC register content can only be modified by the Secure world through the peripheral register Secure alias (for example, EIC_SEC.NONSEC).

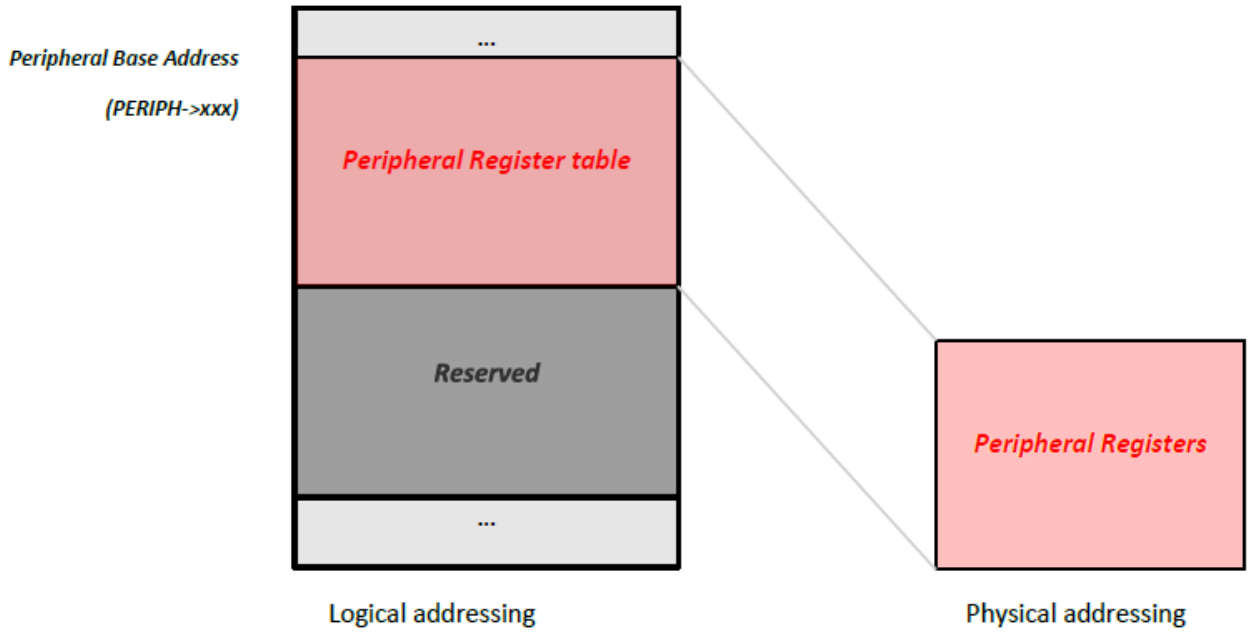
Setting a specific internal feature bitfield in the NONSEC register enables the access to the different bitfields associated to this feature in the peripheral Non-Secure alias.

4.5.2 Mix-Secure Peripheral (PAC Non-Secured)

When a Mix-Secure peripheral is PAC Non-Secured (associated NONSECx fuses set to 1), the peripheral behaves as a standard Non-Secure peripheral.

Secure and Non-Secure accesses to the peripheral register are granted. The Peripheral register mapping is shown in the following figure:

Figure 4-16. PAC Non-Secured Mix-Secure Peripheral Registers Addressing



Management of PAC Non-Secured, Mix-Secured peripherals at the application level is similar to the management of a standard Non-Secure peripheral.

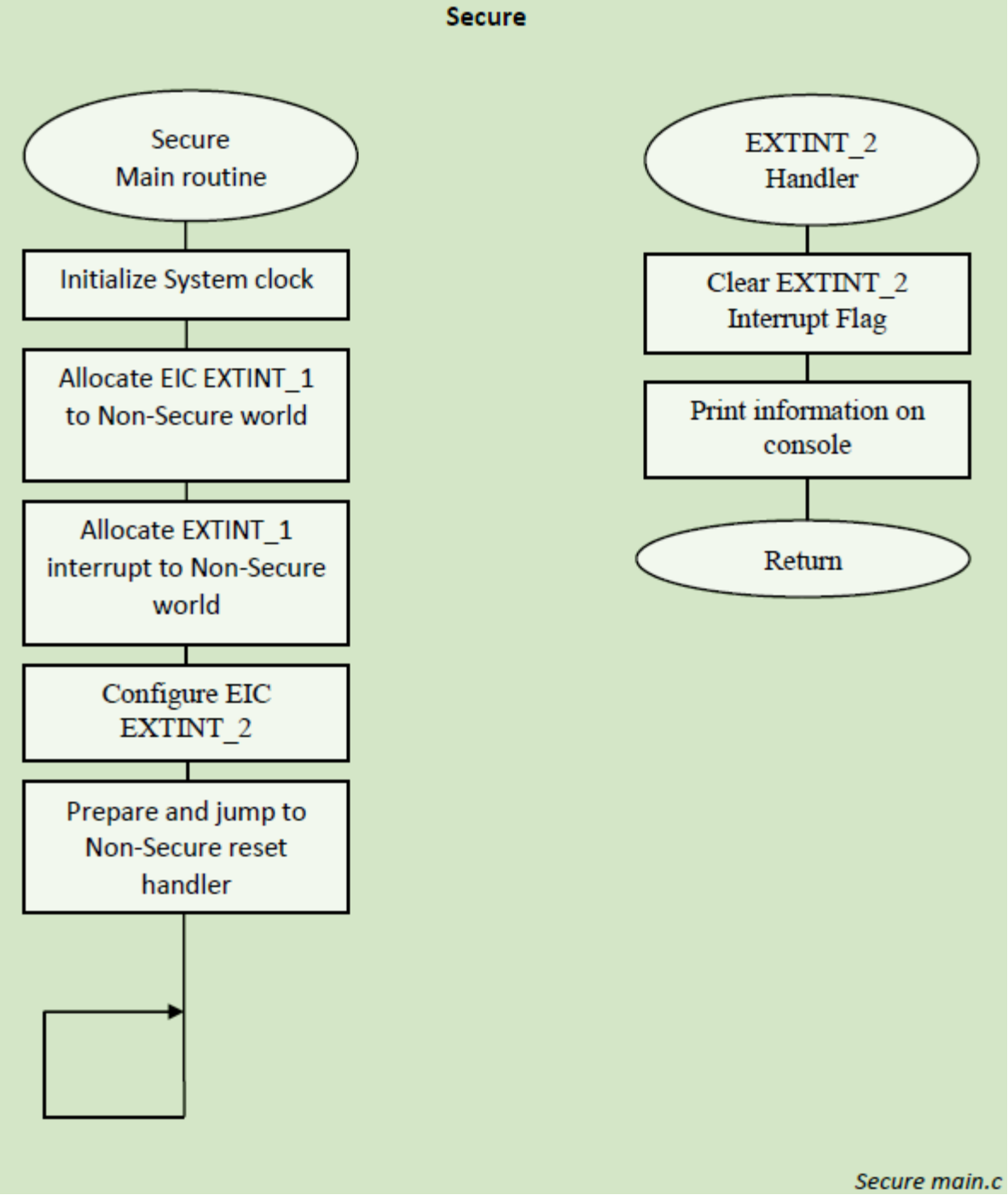
Refer to the How to Use Non-Secure Peripherals for more information.

4.5.3 Mix-Secure Peripheral (PAC Secure) Use Case

The Secure EIC use case displays an example of a Secure External Interrupt Controller (EIC) in use.

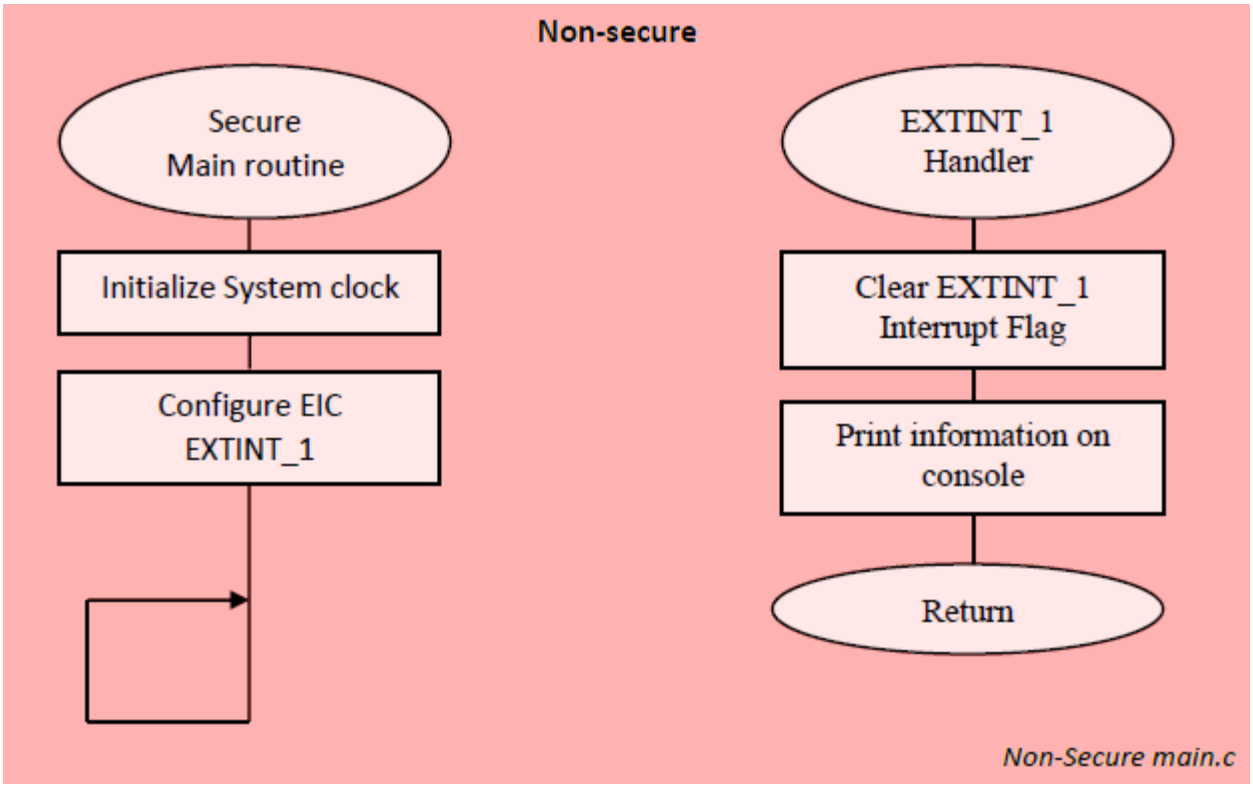
In the example, the Secure project is in charge of configuring system resources, allocating EIC interrupt line 1 to the Non-Secure world and managing the external interrupt on Secured interrupt line 2. The following figure shows the Secure main function flowchart.

Figure 4-17. Secure Application Flow Chart



In the example, the Non-Secure project is in charge of configuring and handling the EIC interrupt line 1, which has been allocated to the Non-Secure world by the Secure application. The following figure displays flowchart for this process:

Figure 4-18. Non-Secure Application Flow Chart



5. SAM L11 Security Features Use Cases

5.1 TrustRAM (TRAM)

The TrustRAM (TRAM) embedded in the SAM L11 offers a set of advanced security features for Secure information storage:

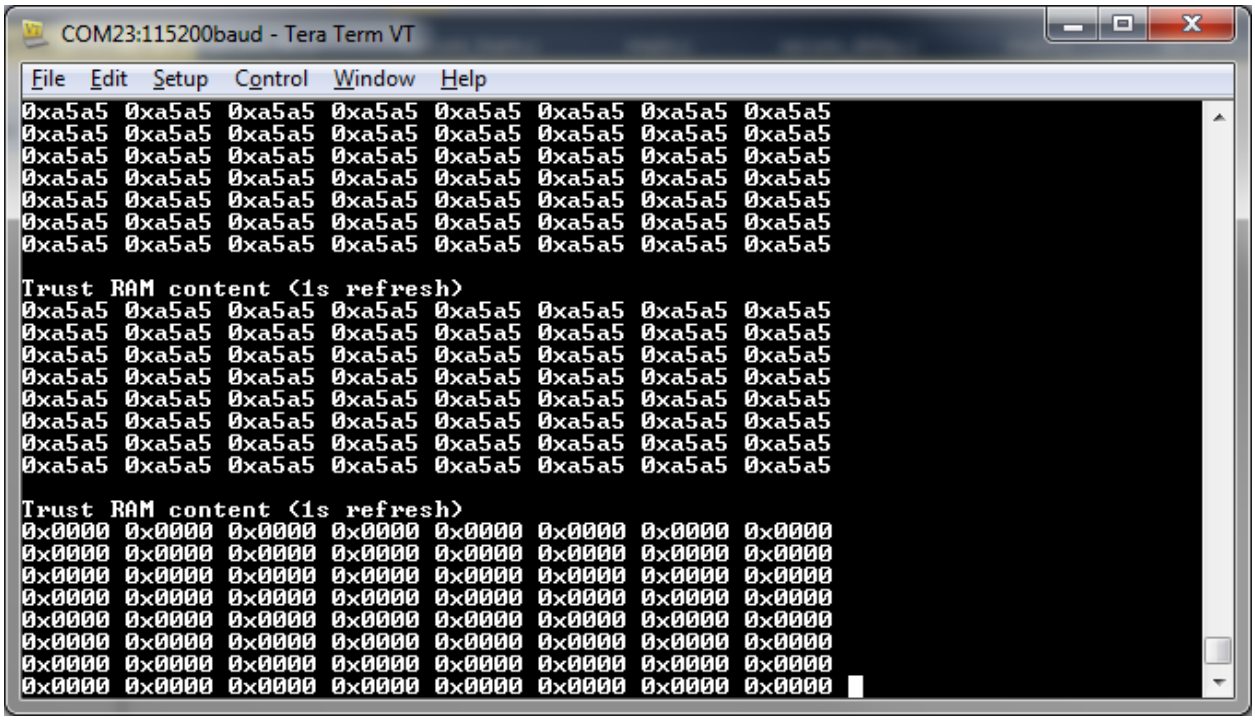
- Address and data scrambling
- Silent access
- Data remanence
- Active shielding and tamper detection
- Full erasure of scramble key and RAM data on tamper detection

The TrustRAM example provided with this document illustrates the configuration of TrustRAM with security features configured as follow:

- Address and data scrambling activated with key: 0xCAFE
- Silent access enabled
- Data remanence enabled
- RTC static tamper detection enabled on PA8
- Full erasure of scramble key and RAM data on tamper detection enabled

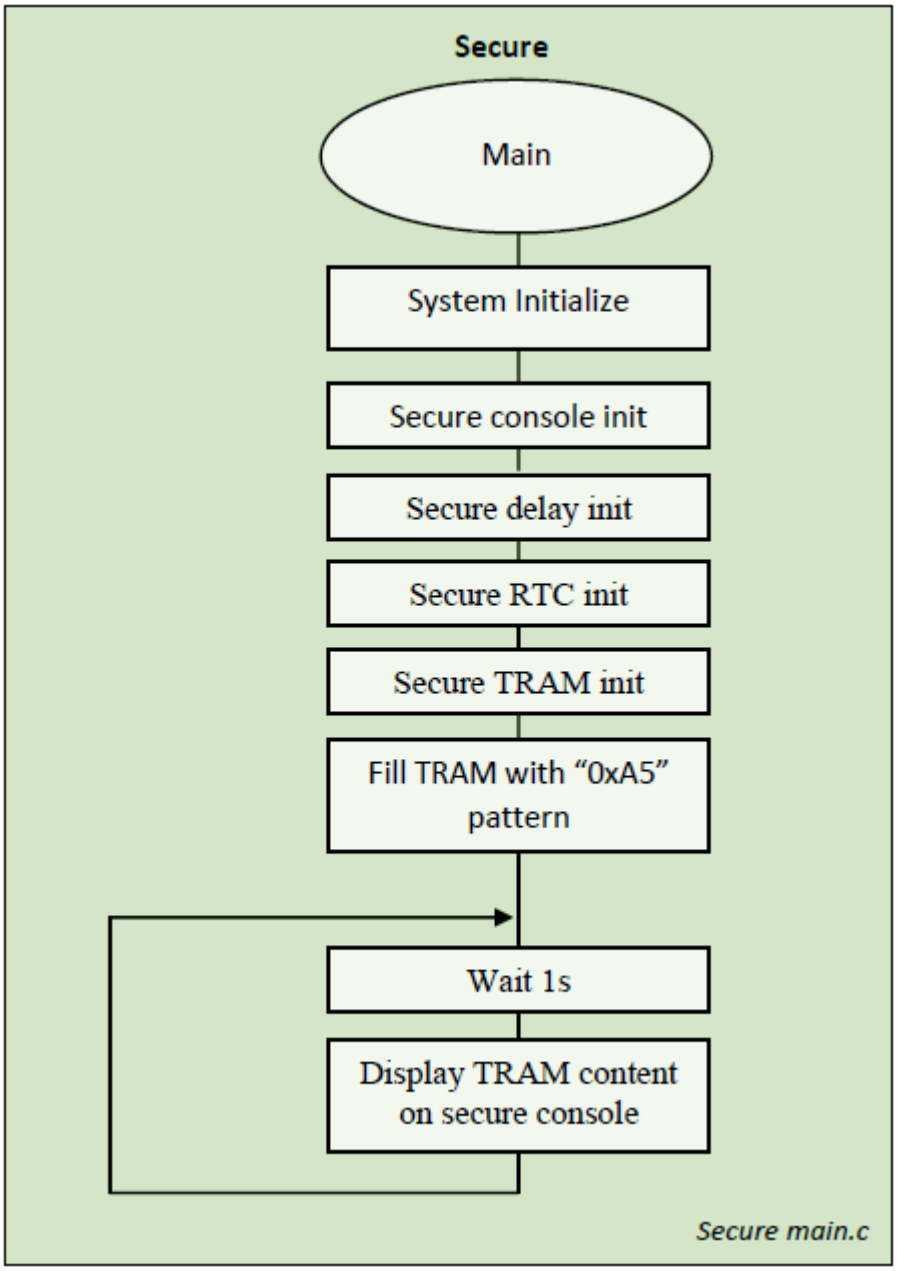
In this example, the TrustRAM content is displayed and refreshed every second on a Secure console (USART0) allowing user to experiment with static and dynamic tamper detections coupled with a TrustRAM full erase.

Figure 5-1. Use Case Application Output



The following flowchart illustrates the secure main function with TRAM:

Figure 5-2. Use Case Application Flow Chart



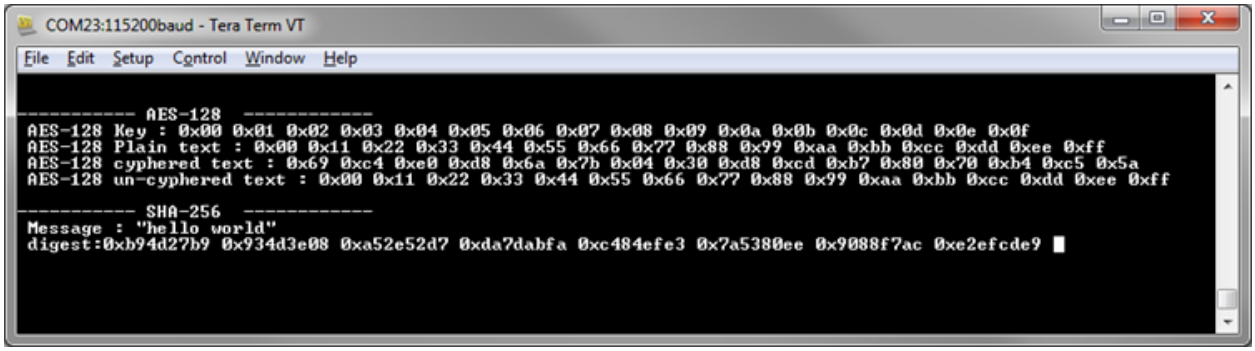
5.2 Cryptographic Accelerator (CRYA)

The SAM L11 embeds a hardware Cryptographic Accelerator (CRYA) with associated software functions stored in Boot ROM which provide the hardware acceleration for:

- Advanced Encryption Standard (AES-128) encryption and decryption
- Secure Hash Algorithm 2 (SHA-256) authentication
- Galois Counter Mode (GCM) encryption and authentication

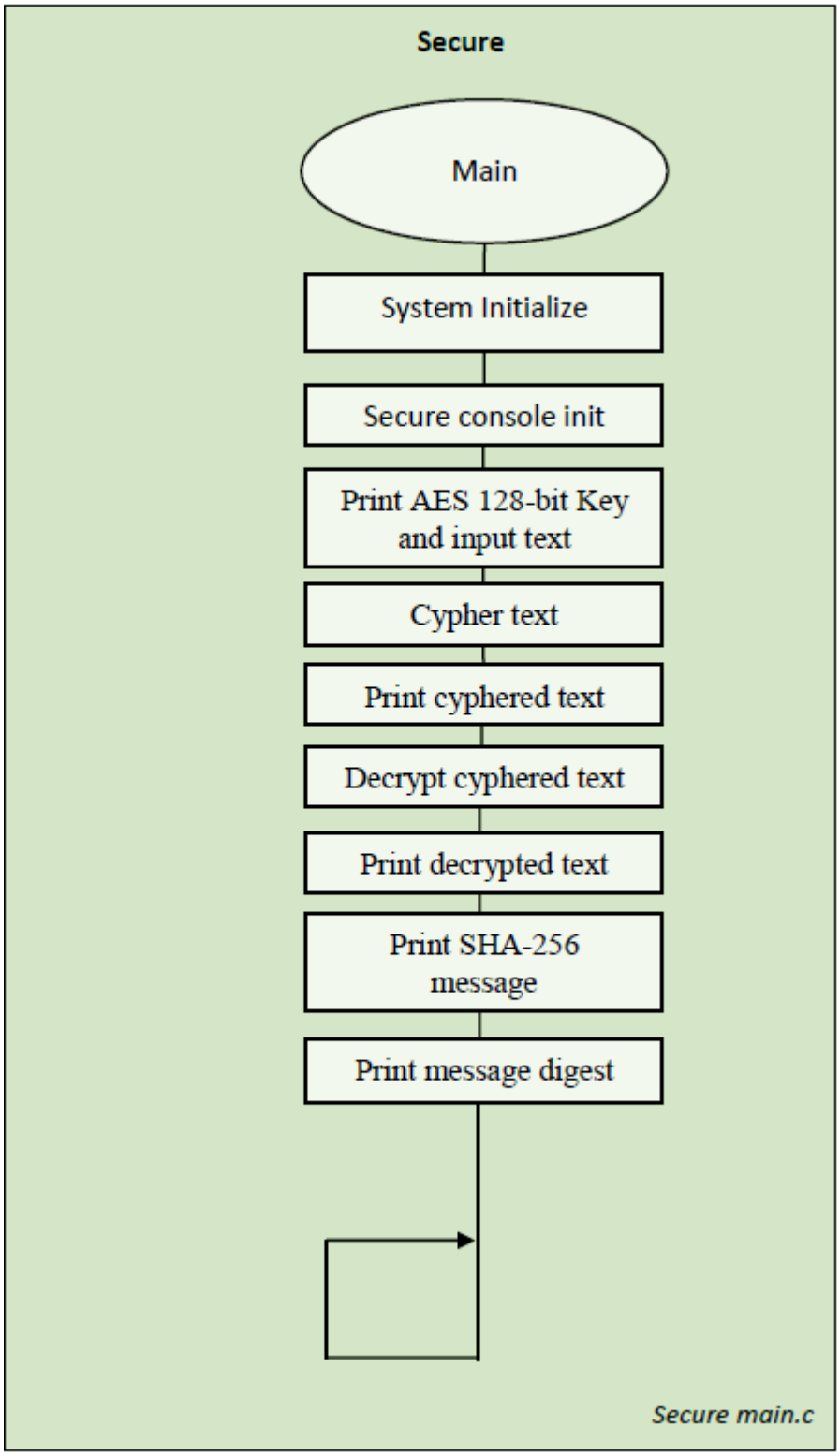
The CRYA example shown in the following figure illustrates the use of the CRYA for AES 128-bit key length and the SHA-256 cryptographic algorithm.

Figure 5-3. Use Case Application Output



The following figure shows the flowchart for this process:

Figure 5-4. Use Case Application Flow Chart



5.3 Data Flash

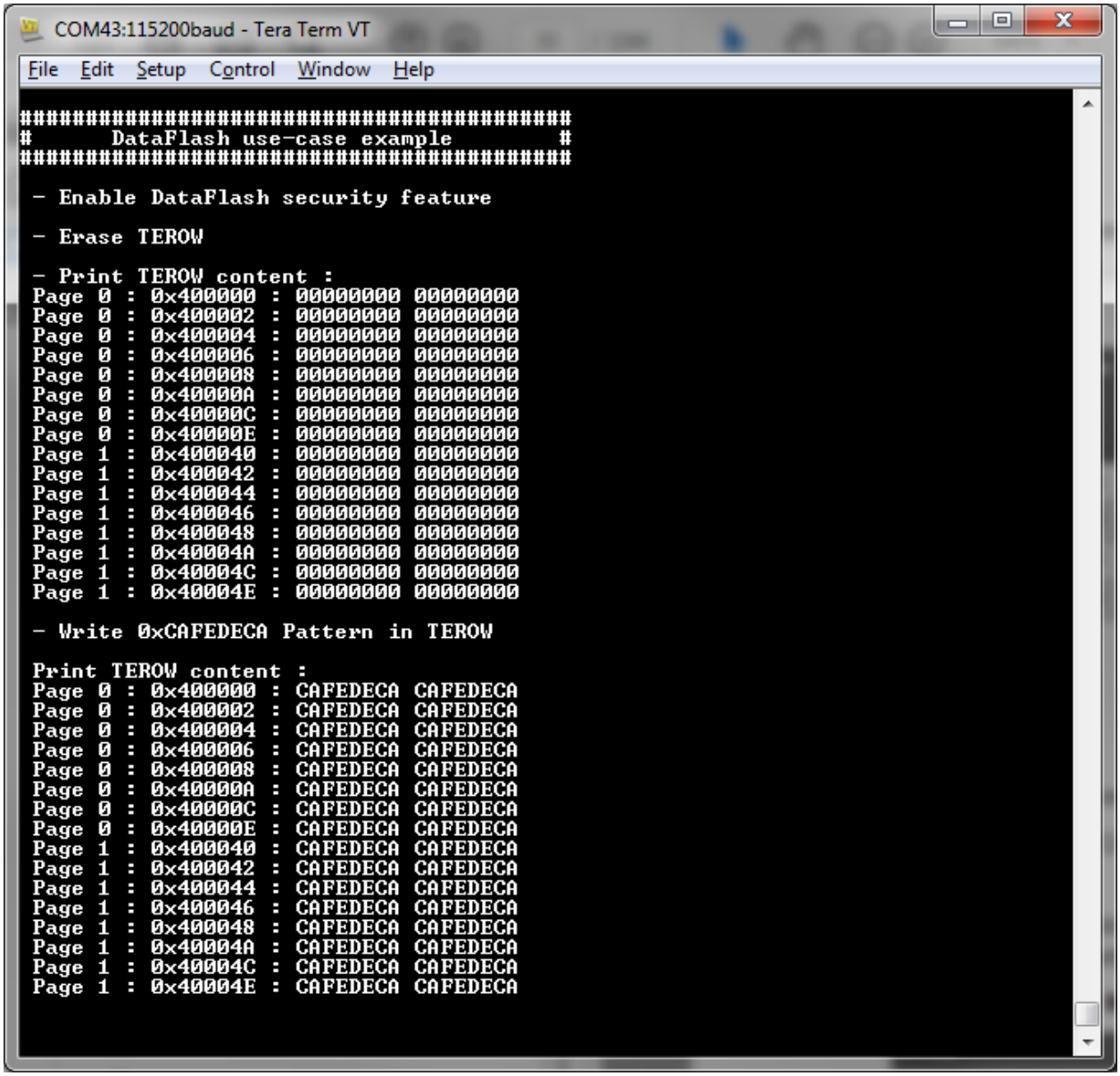
The Data Flash embedded in the SAM L11 offers a set of advanced security features for secure information storage:

- Data scrambling
- Silent access to selected row (TEROW)
- Tamper erase of selected row (TEROW) on tamper detection

The Data Flash use case shown in the following figure, illustrates the configuration of NVMCTRL for secure Data Flash management:

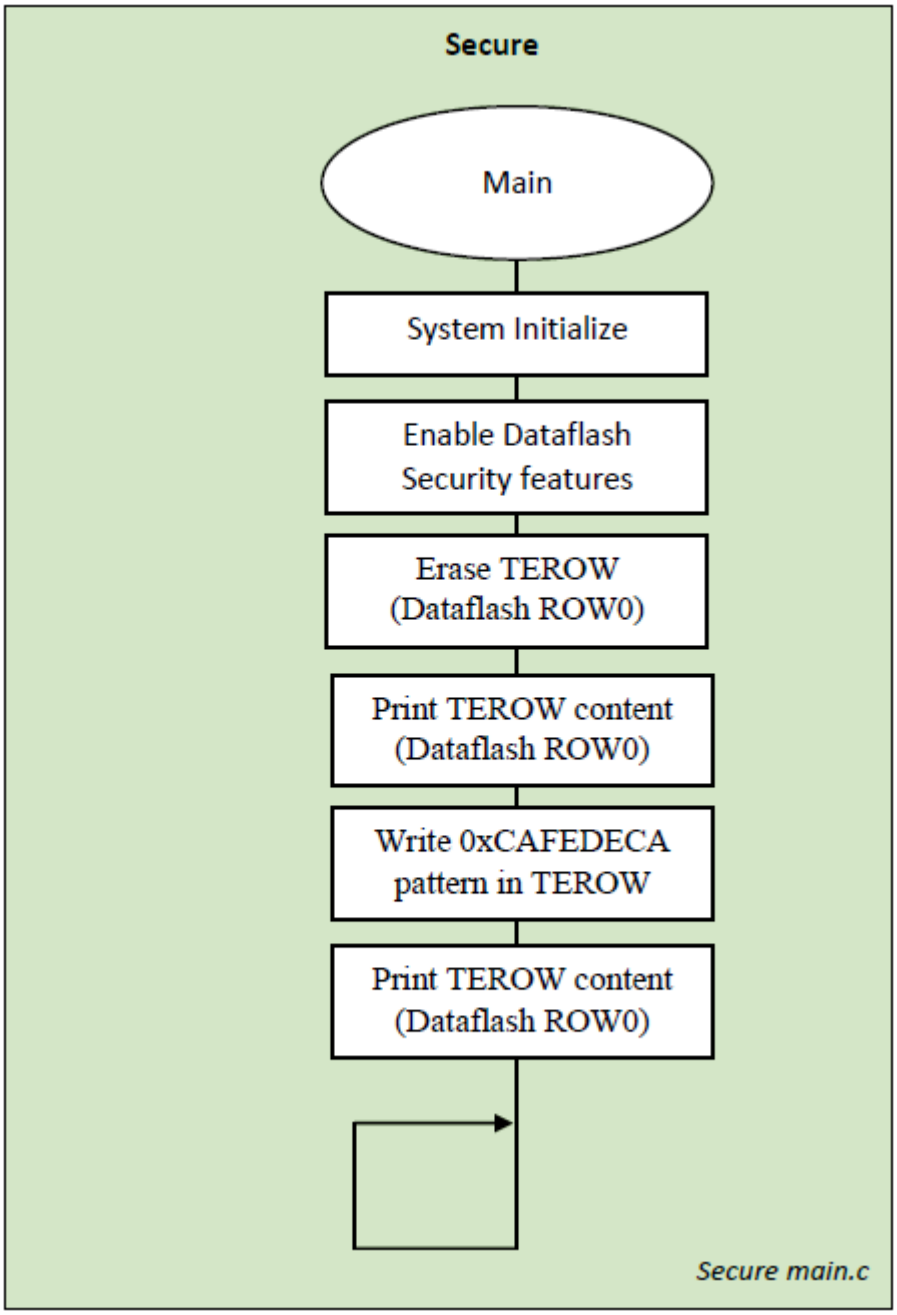
- Data scrambling activated with key: “0x1234”
- Silent access enabled on first Data Flash ROW

Figure 5-5. Use Case Application Output



The following figure illustrates the flowchart for this process:

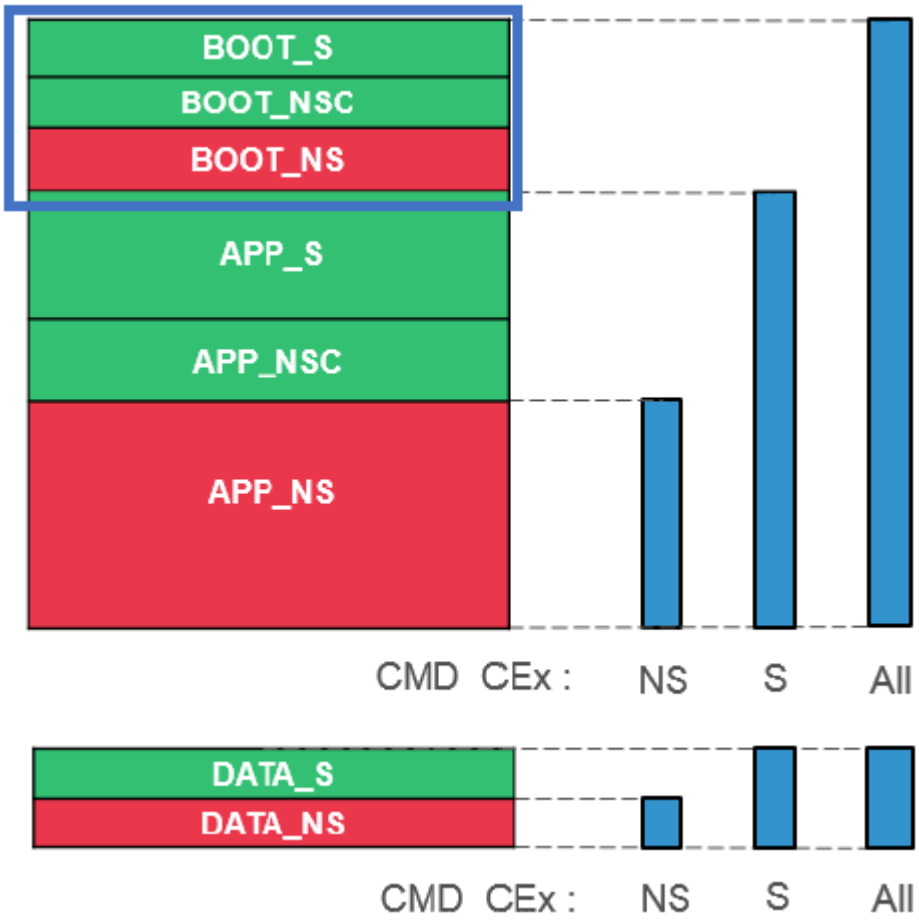
Figure 5-6. Use Case Application Flow Chart



6. Application Deployment with Secure and Non-Secure Bootloaders

The boot sections of the Flash memory allow the storage of Boot programs for a Secure and Non-Secure application in a dedicated memory section, which is protected against the ChipErase_NS and ChipErase_S commands.

Figure 6-1. SAM L11 Boot Sections

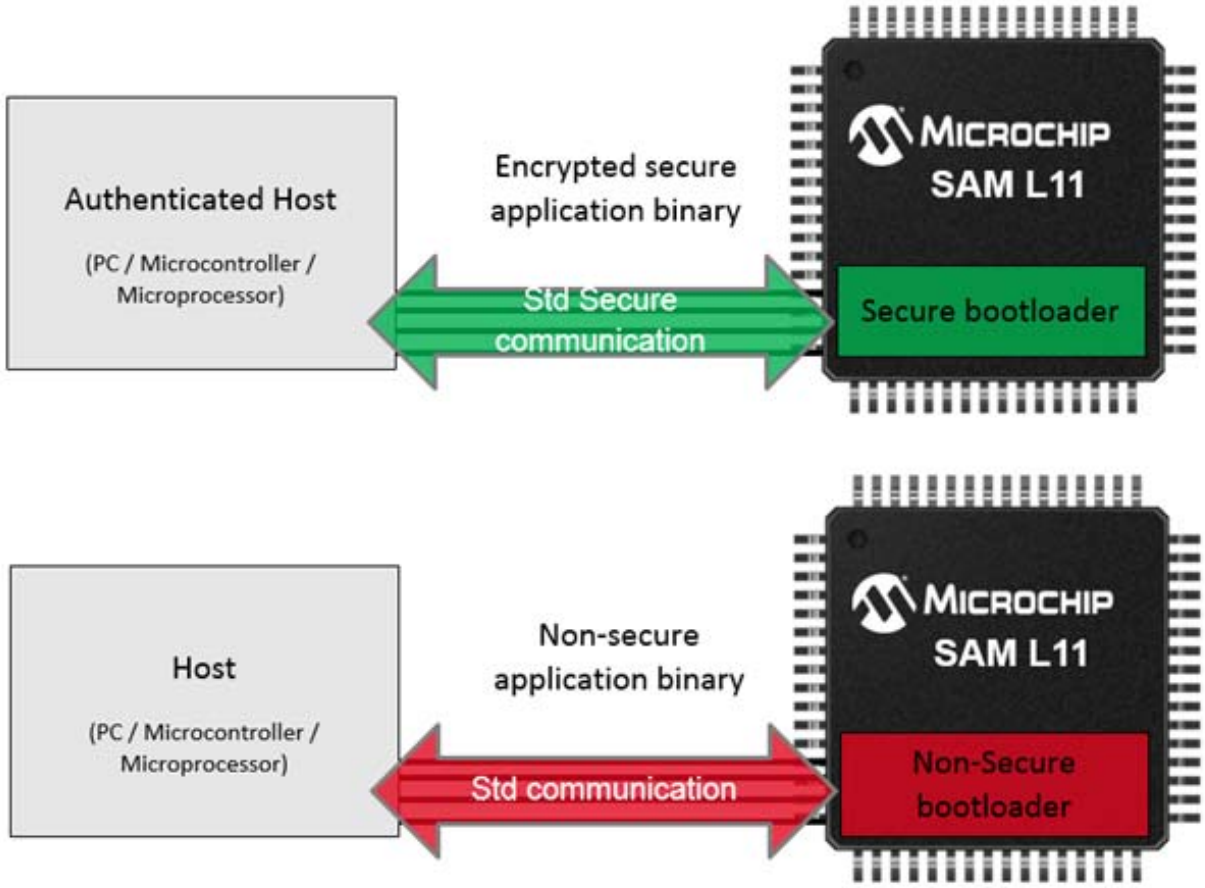


The SAM L11 Boot sections are mainly designed to store In Application Programing solutions, such as Secure and Non-Secure bootloaders. The following sections of this document explain the principle of Secure and Non-Secure application deployment on the SAM L11.

6.1 Software Secure and Non-Secure Bootloaders Principle

A lot of modern embedded systems require application image updates to fix errors or support new features. The main task of the software Secure and Non-Secure bootloaders is to download the respective Secure and Non-Secure programs stored in the SAM L11 memories. This software makes use of standard communication peripherals embedded in the product. This principle is called In Application Programing, as it allows the software upgrade in-situ without the need of a SWD programming interface. Firmware to be stored in the device can be sent by any host that is capable of communicating with the SAM L11 through one of the interfaces supported by the software bootloader (i.e USART, TWI or SPI).

Figure 6-2. Secure and Non-Secure Bootloaders

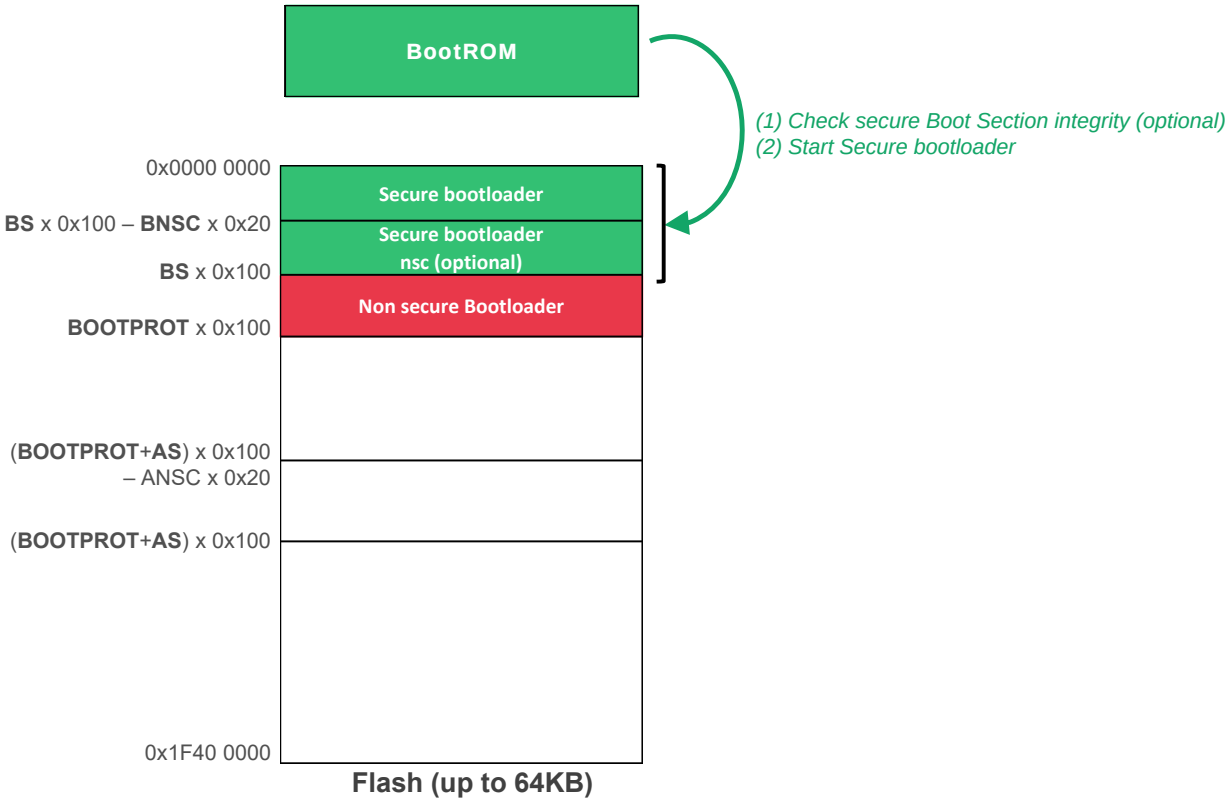


6.2 SAM L11 Secure Boot

The SAM L11 Boot ROM is always executed at product startup. This software is ROM coded into the device and cannot be avoided. Depending on the Boot Configuration Row (BOCOR) fuses setting, the Boot ROM knows if a Secure boot code is used in the system. The Boot ROM then offers the possibility to perform an integrity check or authenticate the firmware stored in the Secure Boot section prior to executing it.

The verification mechanism provided in the Boot ROM is a key element to consider for ensuring root of trust in the deployment and execution of Secure firmware.

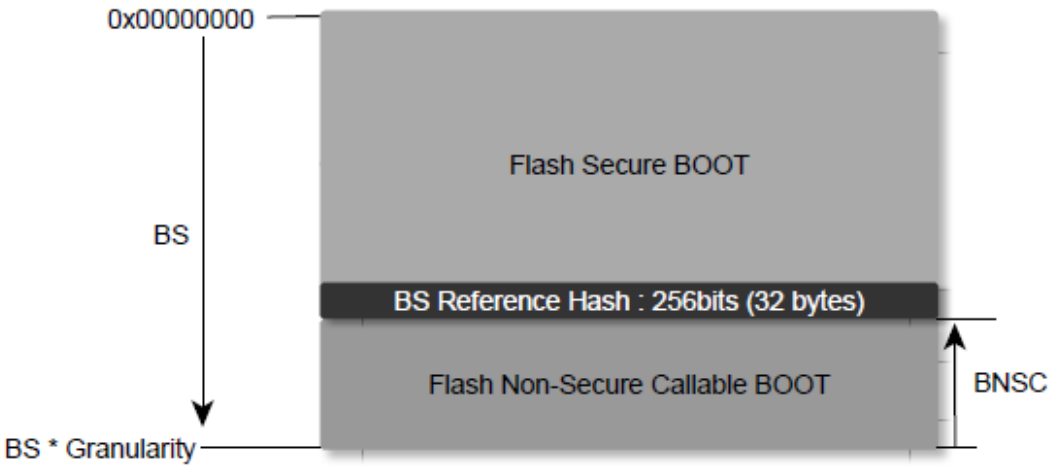
Figure 6-3. Verification Mechanism



The Secure Boot code verification is done using the standard SHA256 hash algorithm that uses product cryptography accelerator (CRYA). Both the Flash BS region and NVM BOCOR row hashes are computed on row, and the memory area is defined by the BOOTPROT, BS and BSNC fuses.

Verification results are compared to their respective reference hash (256 bits/32 bytes) and stored by the developer of the Secure bootloader in BOCORHASH fuses, and at the end of the Flash Secure Boot section.

Figure 6-4. Boot Secure Reference Hash Location



Any mismatch in the value will reset the device and restart the Boot ROM process if no debugger is detected by the SAML11 Debug Service Unit (DSU) (reset loop with no Flash code execution). The SAML11 will put the Boot ROM in Boot Interactive mode if a debugger is detected (This is done to issue the ChipErase_ALL command to clear the whole device content and reprogram it).

If the verification result is equal to the reference hashes, the Boot ROM starts the Secure bootloader execution.

The following figure shows the definitions of the fuses used for configuring Secure boot process.

Figure 6-5. Secure Boot Process Configuration Fuses

Bit Pos.	Name	Usage	Factory Setting	Related Peripheral
7:0	Reserved	Reserved	Reserved	Reserved
15:8	BS	Boot Flash Secure Size = BS*0x100	0x0	IDAU
21:16	BNSC	Boot Flash Non-Secure Callable Size = BNSC*0x20	0x0	IDAU
23:22	Reserved	Reserved	Reserved	Reserved
31:24	BOOTOPT	Boot Option	0xA0	Boot ROM
39:32	BOOTPROT	Boot Protection size = BOOTPROT*0x100	0x00	NVMCTRL
47:40	Reserved	Reserved	Reserved	
48	BCWEN	Boot Configuration Write Enable	0x1	NVMCTRL
49	BCREN	Boot Configuration Read Enable	0x1	NVMCTRL
63:50	Reserved	Reserved	Reserved	Reserved
95:64	BOCORCRC	Boot Configuration CRC for bit 63:0	0xC1D7ECC3	Boot ROM
127:96	ROMVERSION	ROM Code Version	0x0000003A	Boot ROM
255:128	CEKEY0	Chip Erase Key 0	All 1s	Boot ROM
383:256	CEKEY1	Chip Erase Key 1	All 1s	Boot ROM
511:384	CEKEY2	Chip Erase Key 2	All 1s	Boot ROM
639:512	CRCKEY	CRC Key	All 1s	Boot ROM
895:640	BOOTKEY	Secure Boot Key	All 1s	Boot ROM
1791:896	Reserved	Reserved	Reserved	Reserved
2047:1792	BOCORHASH	Boot Configuration Row Hash	All 1s	Boot ROM

BOOTPROT, BS and BSNC: Defines the configuration of the boot section in product Flash. The size of the Secure, Non-Secure and Non-Secure-Callable boot sections can be customized according to the application need. These fuses are used for security memory allocation in product IDAU and for integrity and authentication mechanisms when configured in the BOOTOPT fuse. Any change of the fuse setting requires a reset to be considered by the device as only the Boot ROM is allowed to change IDAU setting.

BOOTOPT : Defines the type of verification to be performed as either Secure and Non-Secure.

- 0: No verification method
- 1: Integrity check SHA256
- 2: Authentication check SHA-256 with BOOTKEY

Note: The use of the Secure Boot Authentication feature has an impact on the product startup time. Refer to the product data sheet for more information.

BOOTKEY: Stores the SHA-256 result to be compared with the result of the selected Boot ROM verification method execution. This value should be calculated and stored in advance by Customer A.

6.3 Custom Secure Software Bootloader

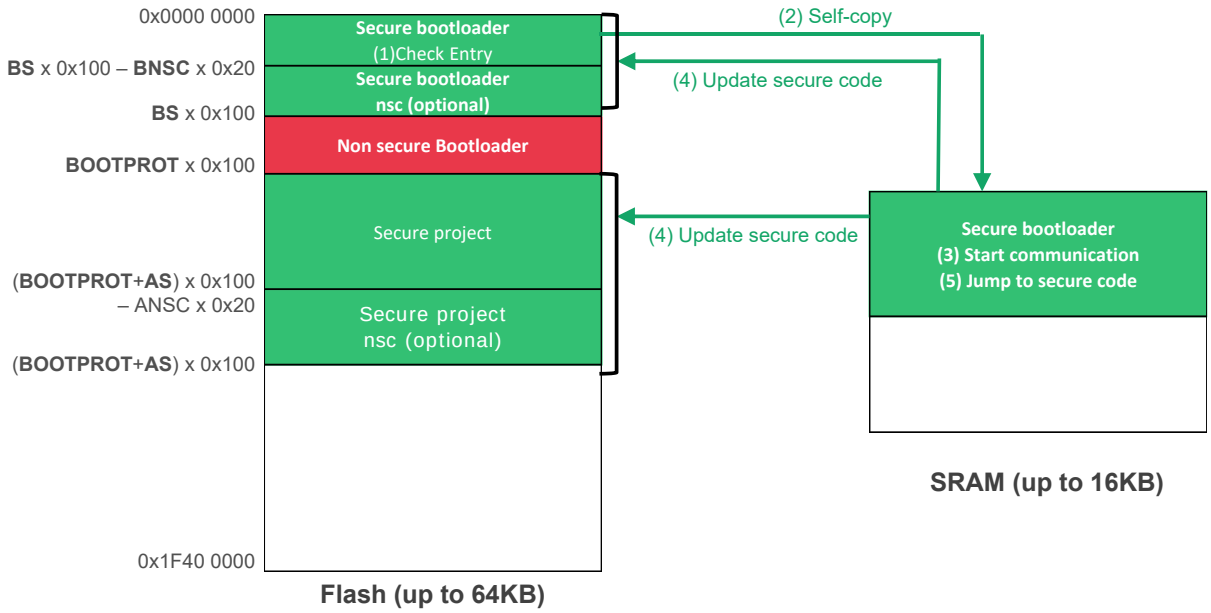
When required by the application, a Secure software bootloader should be stored in the Flash Boot secure region to benefit from the Boot ROM verification mechanism and ChipErase protection.

Specific care should be taken when updating the Secure application using the bootloader. As the Secure application may contain critical Secure code and data, the application firmware should not be vulnerable to interception during the data transfer from the external source.

Note: Refer to the "Secure UART Bootloader for SAM L11" Application Note for additional information.

The following diagram shows general Secure bootloader execution steps:

Figure 6-6. Secure Software Bootloader Execution Steps



The Software Bootloader Execution steps are as follows:

1. Bootloader Entry detection. Defines if the bootloader should be executed or not. For example, the secure bootloader can run automatically if there is no valid application in the product secure application, Flash memory region, and runs on the detection of an external request on a dedicated HW Entry pin.
2. Self copy to secure SRAM. As the Flash technology does not support the read-while-write operation, and most bootloaders should have the possibility to update their own software, the bootloader should be self-copied and executed from the SRAM. For this purpose the RXN (RAM is eXecute Never) must be cleared in the device fuse setting.
3. Enable secure communication with host. Care should be taken at this step to not disclose critical secure information, or allow unauthenticated host access to Secure bootloader features. Therefore, it is recommended to manage host authentication, data encryption, and data integrity during the transfer.
4. Update secure code section. Decrypt and check the integrity of new code blocks sent by the host and write them to the Secure memory regions.
5. Jump to secure code. A reference software can be found in *Secure UART Bootloader for SAM L11* application note, which is available for download at www.microchip.com.

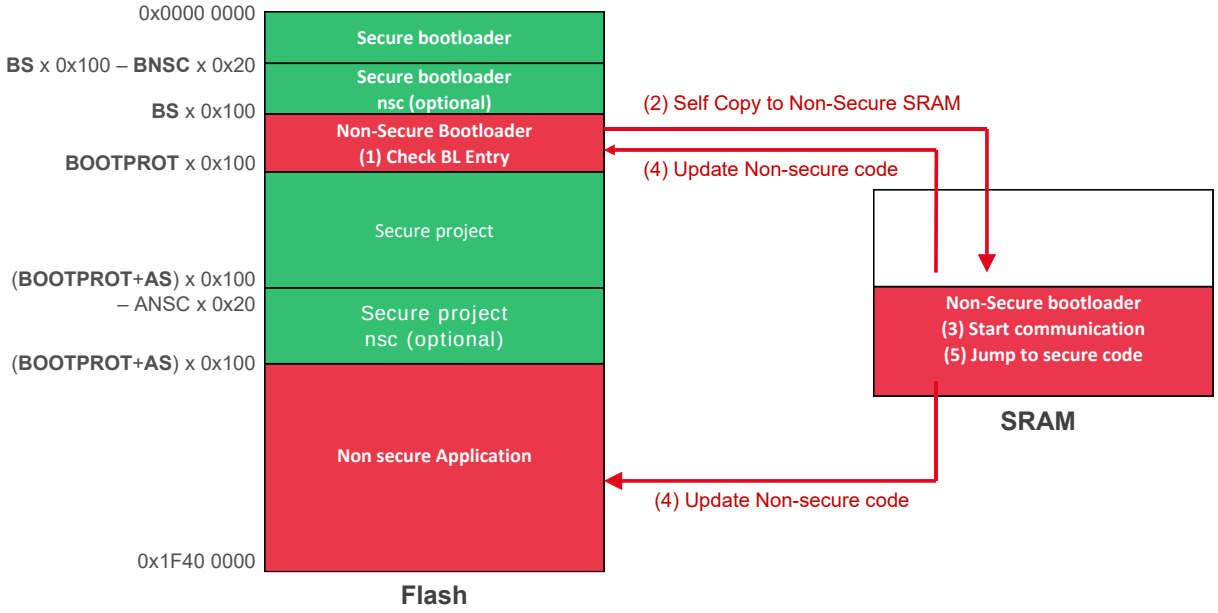
6.4 Custom Non-Secure Software Bootloader

When required by the application, a Non-Secure software bootloader should be stored in the Flash Non-Secure region of the boot region for ChipErase protection. The software architecture of a Non-Secure bootloader for SAM L11 is similar to the standard Cortex-M device bootloader.

This bootloader is executed prior to the Non-Secure application execution, and offers the possibility to upgrade the Non-Secure application stored in the device.

The following figure shows standard Non-Secure bootloader execution steps.

Figure 6-7. Non-Secure Software Bootloader Execution Steps



Follow these Non-Secure bootloader execution steps:

1. Bootloader Entry detection: Defines if the bootloader should be executed or not. For example, the Non-Secure Bootloader can run automatically if there is no valid application in the product Non-Secure application Flash memory region, or run on detection of external request on a dedicated entry pin. SSelf copy to secure SRAM:As Flash technology does not support the read-while-write operation, and most bootloaders should have the possibility to update their own software, the bootloader should be self-copied and executed from the SRAM. For this purpose the RXN fuse (RAM is eXecute Never) from the UROW row must be cleared in the device fuse setting.
2. Enable communication with host.
3. Update Non-Secure code section.
4. Jump to Non-secure code.

The Secure boot can share functionality, such as communication protocol management with the Non-Secure bootloader using the Non-Secure Callable boot region.

A reference software can be found along with *UART Bootloader for SAM L10 / SAM L11* Application note, which is available for download at www.microchip.com.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

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

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Trademarks

ISBN: 978-1-5224-3234-0

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820