



AN1374: Series 2 TrustZone

ARMv8-M TrustZone is a technology that provides a foundation for improved system security in embedded applications. It allows the ARMv8-M to be aware of the security states of the system. Series 2 devices use the Cortex-M33 core to implement the ARMv8-M TrustZone security extension, which provides the ability to restrict access to peripherals and memory regions based on the processor security attribute. TrustZone works with the MPU, which controls privileged/unprivileged execution of code to provide a complete security solution.

ARMv8-M TrustZone is an extensive topic. The references below are publicly available on the [ARM Developer Documentation](#) website.

- [ARMv8-M Architecture Reference Manual](#)
- [ARMv8-M Architecture Technical Overview](#)
- [ARM Cortex-M33 Processor Technical Reference Manual](#)
- [System Design with ARMv8-M](#)
- [TrustZone technology for ARMv8-M Architecture](#)
- [ARM Cortex-M33 Devices Generic User Guide](#)
- [Secure software guidelines for ARMv8-M](#)
- [Software Development in ARMv8-M Architecture](#)

Reading guides:

- Beginner - Minimal experience with TrustZone, starting with [TrustZone Basics](#)
- Intermediate - Have a basic understanding of the TrustZone technology, starting with [Bus Level Security](#)
- Advanced - Developed experience on TrustZone, starting with [TrustZone Implementation](#)
- Demo - Starting with [TrustZone Platform Examples](#)

KEY POINTS

- TrustZone Basics
- Bus Level Security (BLS)
- Secure and Privileged Programming Model
- TrustZone Implementation
- Upgrade Existing Application to TrustZone
- TrustZone Platform Examples

1. Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure communication paths to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to UG103.05: IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
Secure Vault Mid (SVM)	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Note:

1. The features of different Secure Vault levels can be found in <https://www.silabs.com/security>.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug Access Control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

Series 2 devices require a specific [SE firmware version](#) to support the TrustZone implementation. Refer to [AN1222: Production Programming of Series 2 Devices](#) to learn how to upgrade the SE firmware and [UG103.05: IoT Endpoint Security Fundamentals](#) for the latest SE Firmware shipped with Series 2 devices and modules.

Series 2 devices use Cortex-M33 core to implement the ARMv8-M Mainline TrustZone security extension and refer to TrustZone as [Bus Level Security](#). The following table lists the configuration of TrustZone related components in the Series 2 Cortex-M33 core.

Component	Series 2 Configuration	Description
Security Extension (TrustZone)	Enabled	The security extension cannot be disabled, and the entire memory after RESET is Secure by default.
Memory Protection Unit (MPU)	16 regions (maximum)	The MPU regions for both Secure and Non-secure MPUs.
Security Attribution Unit (SAU)	8 regions (maximum)	The SAU regions for Non-secure and Non-secure Callable.

2. TrustZone Basics

2.1 Introduction

TrustZone for ARMv8-M adds extra states to the Cortex-M processor operations to ensure there is a Secure and Non-secure state. These security states are orthogonal to the existing Thread and Handler modes, thereby having both a Thread and Handler mode in both Secure and Non-secure states. The Thread mode can also be either Privileged or Unprivileged.

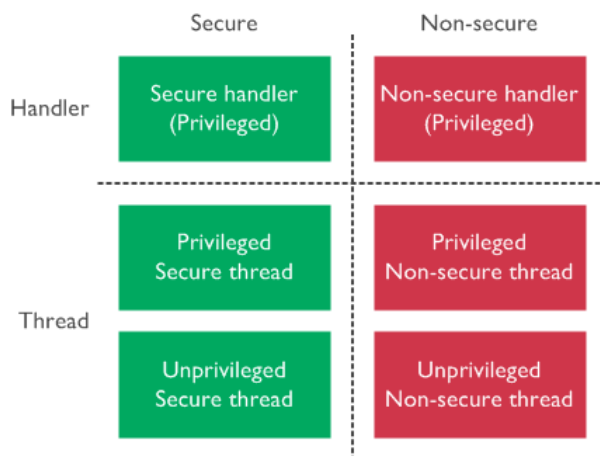


Figure 2.1. Operation States and Modes of TrustZone Implementation

Image: <https://documentation-service.arm.com/>. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

TrustZone for ARMv8-M is an optional architecture extension. By default, the system starts up in a Secure state if the processor implements the TrustZone security extension. The division of Secure and Non-secure worlds is memory-map based (security state depends on the address of the fetched instruction), and the transitions happen automatically. It is also possible to leave the Non-secure state unused and execute the whole application in the Secure state.

2.2 Memory Security Attributes

TrustZone classifies memory into four security attributes as described in the following table.

Security Attribute	Processor State	Description
Non-secure (NS)	Non-secure	Non-secure and Secure software can access these memory regions.
Secure (S)	Secure	Secure software can access these memory regions. Non-secure software cannot gain access to the Secure memory.
Non-secure Callable (NSC)	Secure	Secure memory with an NSC attribute provides entry points for Secure APIs that can be called from a Non-secure space. It is a region of memory that contains the Secure Gateway (SG) veneers that allow Non-secure code to call secure functions that exist in Secure code. Non-secure software cannot read/write to an NSC memory but can branch into it if the branch target is an SG instruction.
Exempted	Secure/Non-secure	Non-secure and Secure software can access these memory regions (exempted from security checking). Exempted regions are typically used by debugging components that do not pose any security risk (e.g., system ROM table) when accessed by the Non-secure software.

Note: The **Non-secure Callable** is also known as Secure Non-secure Callable (Secure NSC) to declare that this region resides in Secure memory.

2.3 Banked Register

The concept of a banked register in ARMv8-M between Secure and Non-secure states means that there are two copies of the register, and the core automatically uses the copy that belongs to the current security state. When a register is banked, the `_S` and `_NS` suffixes are used in the ARMv8-M architecture to identify whether the resource is for the Secure state or Non-secure state.

2.3.1 General-Purpose Registers

The Cortex-M processors have 16 general-purpose registers (R0 - R15) for data processing (R0 - R12) and control. The following figure shows the general-purpose register view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about these registers.

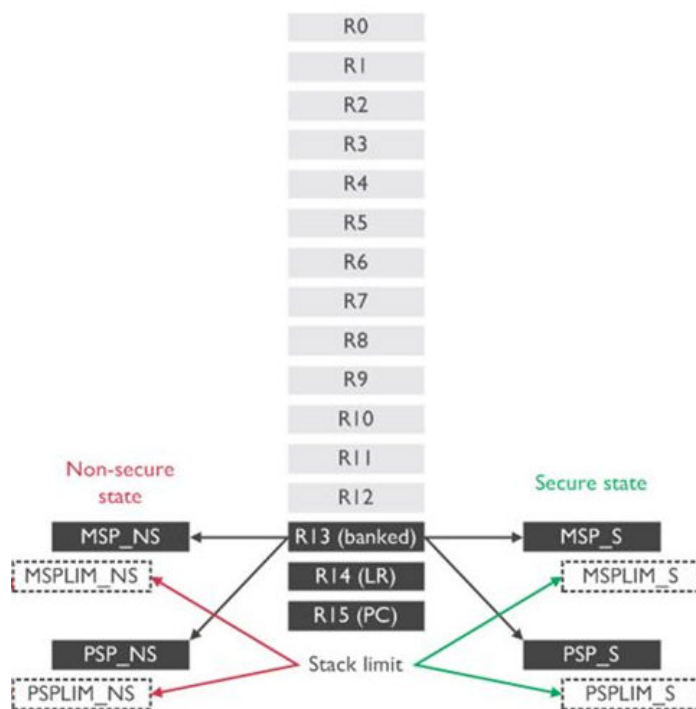


Figure 2.2. General-Purpose Register View with TrustZone

Image: <https://documentation-service.arm.com/>. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

The Secure or Non-secure state can access the data processing registers R0 - R12 and special usage registers R13 - R15. The register R13 (banked SP) is the stack pointer alias, and the actual stack pointer (MSP_NS, PSP_NS, MSP_S, PSP_S) accessed depends on the state (Secure or Non-secure) and mode (Handler or Thread) as described in the following figure.

In addition, stack limit registers ([special registers](#)) enable hardware to detect stack overflow conditions. Two pairs of [stack limit registers](#) (MSPLIM_NS and PSPLIM_NS, MSPLIM_S and PSPLIM_S) are implemented, one per security state, to protect the Main Stack Pointer (MSP) and Process Stack Pointer (PSP).

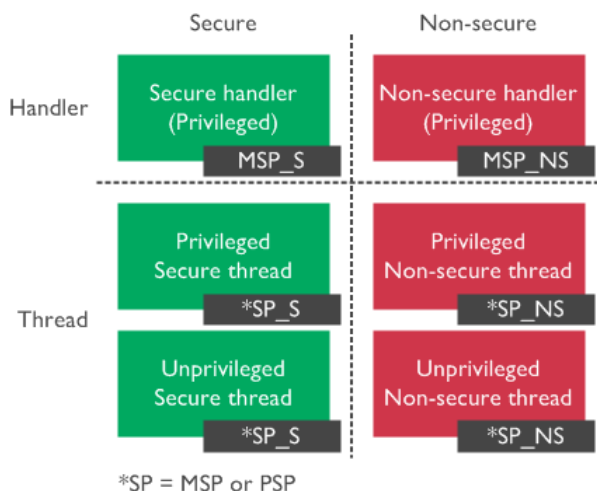


Figure 2.3. Banked Registers in the General-Purpose Registers

Image: <https://documentation-service.arm.com/>. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

In Thread mode, execution can be privileged or unprivileged. The stack pointer used can be the MSP or PSP, depending on the `SPSEL` bit in the `CONTROL` register. When in Handler mode, the processor is Privileged. The stack pointer is always MSP.

It is possible to directly [access](#) the stack pointers (MSP and PSP) and stack limit registers (MSPLIM and PSPLIM), providing that the processor is in a privileged state. If the processor is in a Secure privileged state, the software can also access the Non-secure stack pointers (`MSP_NS` and `PSP_NS`) through [Core Register Access Functions](#) in CMSIS-Core.

2.3.2 Special-Purpose Registers

Except for the general-purpose registers, there are several special-purpose registers for conditional flags, interrupt masking, control, and stack pointer limit. The following figure shows the special-purpose registers view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about these registers.

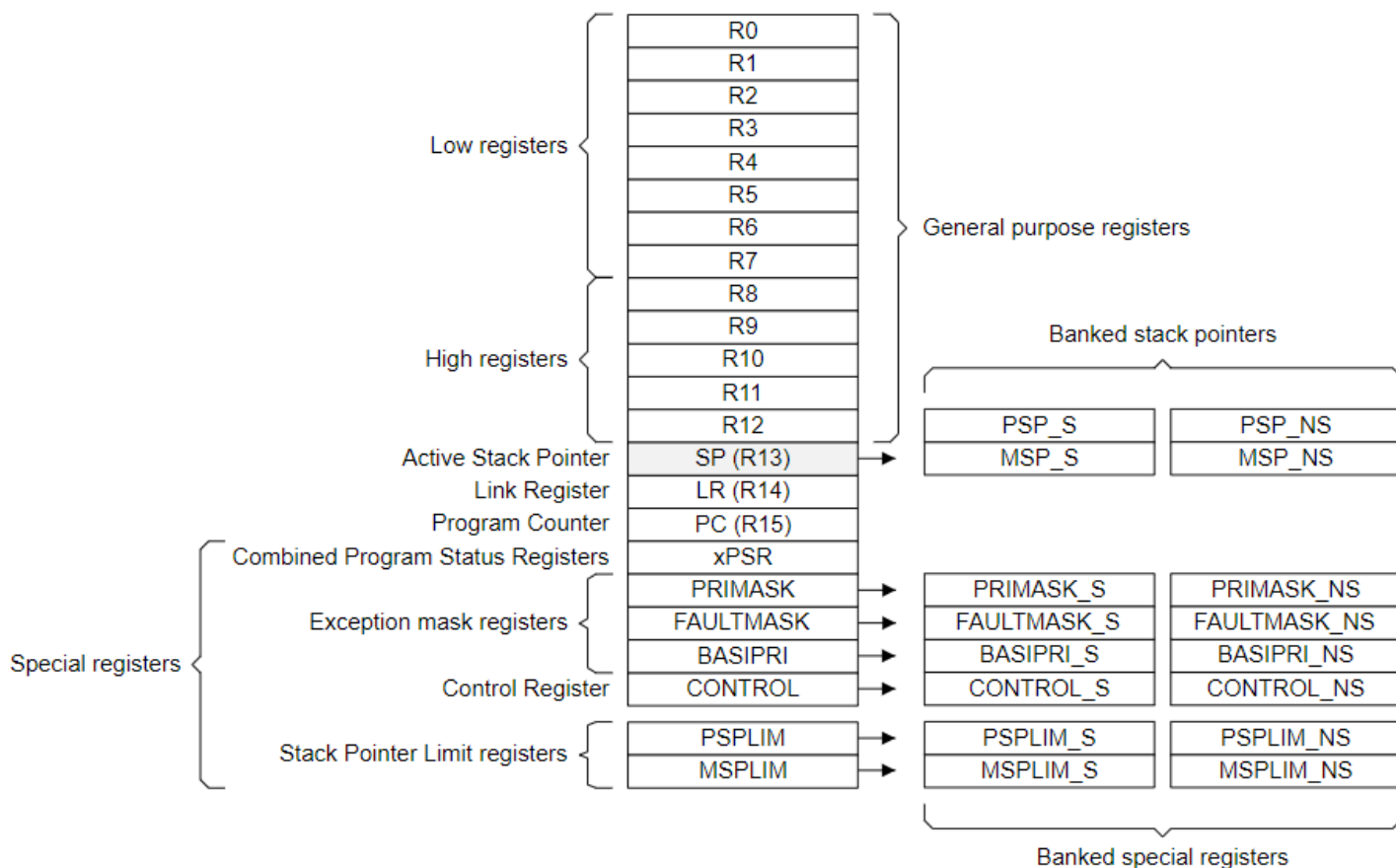


Figure 2.4. Special-purpose Registers View with TrustZone

Image: <https://documentation-service.arm.com/>. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

The Combined Program Status Register (xPSR) consists of the Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and Execution Program Status Register (EPSR).

Some of the special-purpose registers are banked between Secure and Non-secure states. Special-purpose registers are not memory-mapped and can be [accessed](#) using Core Register Access Functions in CMSIS-Core (except for EPSR in xPSR).

Secure privileged software can also access the Non-secure interrupt masking registers (`PRIMASK_NS`, `FAULTMASK_NS`, and `BASEPRI_NS`), `CONTROL` register (`CONTROL_NS`), and stack limit registers (`MSPLIM_NS` and `PSPLIM_NS`) through [Core Register Access Functions](#) in CMSIS-Core.

2.3.3 System Private Peripheral Bus (PPB)

The banking of registers is usually used to separate the Secure and Non-secure information of the system components inside the processor. The following figure shows the System Private Peripheral Bus (PPB) registers view of the ARMv8-M system with TrustZone. Refer to the [ARM Cortex-M33 Devices Generic User Guide](#) for details about the System PPB registers.

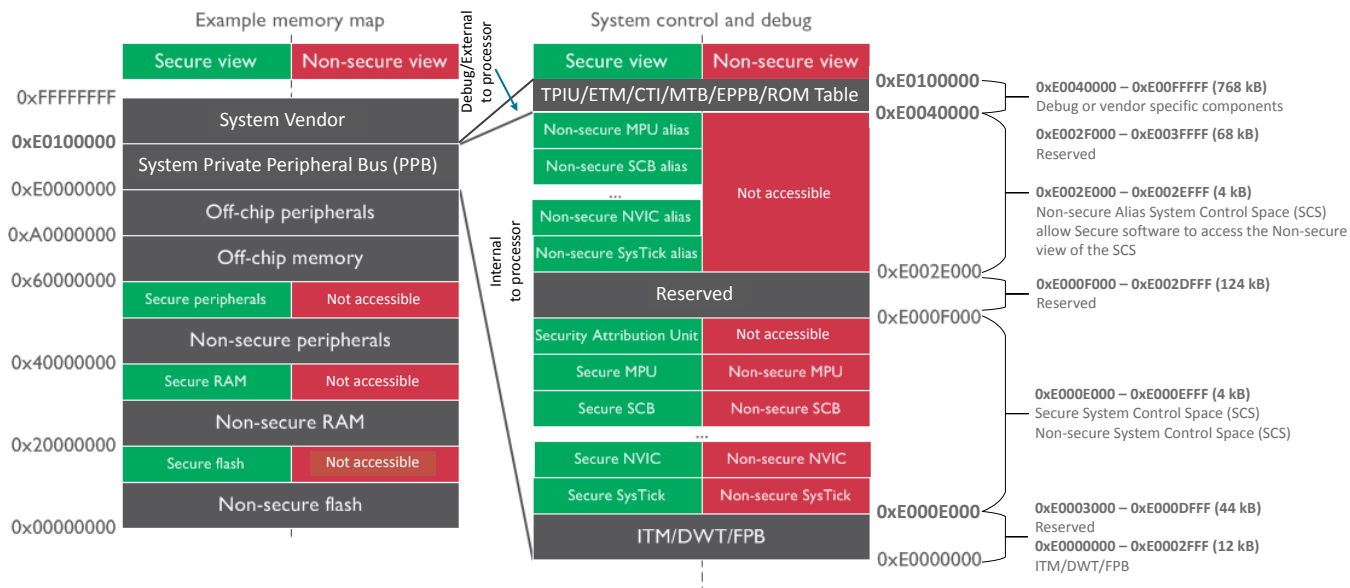


Figure 2.5. System Private Peripheral Bus (PPB) Registers View with TrustZone

System components for debugging and trace operations (0xE0000000 to 0xE002FFF):

- Instrumentation Trace Macrocell (ITM)
- Data Watch point and Trace unit (DWT)
- Flash Patch and Breakpoint unit (FPB)

System Control Space (SCS):

- The registers in SCS address spaces are memory-mapped and can be accessed using pointers in software
- Secure SCS (0xE000E000 to 0xE000EFFF) - Secure software using this address space to access the banked Secure SCS registers (e.g., SCB->CPUID)
- Non-secure SCS (0xE000E000 to 0xE000EFFF) - Non-secure software using this address space to access the banked Non-secure SCS registers (e.g., SCB->CPUID)
- Non-secure alias SCS (0xE002E000 to 0xE002EFFF) - Secure software using this address space to access the Non-secure SCS registers (e.g., SCB_NS->CPUID)

The following table describes some core peripherals in the SCS and corresponding [data structures](#) defined in the CMSIS-Core header file to access the registers of core peripherals in two SCS address spaces.

Core Peripheral	Data Structure for Secure and NS SCS	Data Structure for NS Alias SCS
Implementation Control Block	SCnSCB (0xE000E004)	SCnSCB_NS (0xE002E004)
SysTick Timer	SysTick (0xE000E010)	SysTick_NS (0xE002E010)
Nested Vectored Interrupt Controller	NVIC (0xE000E100)	NVIC_NS (0xE002E100)
System Control Block	SCB (0xE000ECFC)	SCB_NS (0xE002ECFC)
Memory Protection Unit	MPU (0xE000ED90)	MPU_NS (0xE002ED90)
Security Attribution Unit	SAU (0xE000EDD0)	—
Debug Control Block	CoreDebug (0xE000EDF0)	CoreDebug_NS (0xE002EDF0)
Software Interrupt Generation	STIR (0xE000EF00)	STIR_NS (0xE002EF00)
Floating-Point Extension	FPU (0xE000EF34)	FPU_NS (0xE002EF34)

Note:

- The SCB is a group of system control registers for the various usages below.
 - System Control Register (SCR) to configure processor low power mode
 - Fault Status Register (xFSR) to provide fault status information
 - [Vector Table Offset Register \(VTOR\)](#) for vector table relocation
- The [SAU](#) register is accessible from the Secure state only.
- The STIR register is not physically banked.
- Core peripherals such as SysTick, SCB, and MPU are duplicated. One instance is Secure and the other one is Non-secure.
- Secure software can use the corresponding functions for ARMv8-M in CMSIS-Core to configure the Non-secure [NVIC](#) and [SysTick](#) through the Non-secure alias SCS.

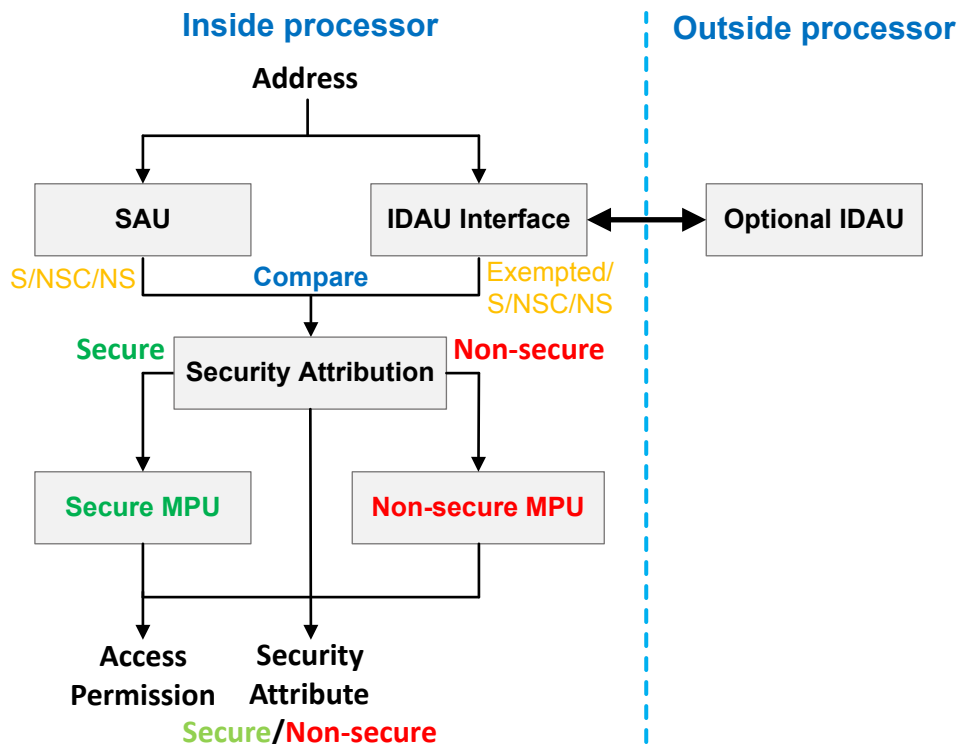
Debug or vendor specific components (0xE0040000 to 0xE00FFFFF):

- Optional debug components (e.g., ETM)
- [External Private Peripheral Bus \(EPPB\)](#) allows designers to add their own debug or vendor-specific components
- [System ROM Table](#) is a simple lookup table that enables debug tools to extract the addresses of debug and trace components

2.4 Secure Attribution Unit (SAU), Implementation Defined Attribution Unit (IDAU), and Memory Protection Unit (MPU)

Two units determine the security attribute of an address:

1. The internal programmable [Secure Attribution Unit \(SAU\)](#).
2. The external Implementation Defined Attribution Unit (IDAU), through the IDAU interface, returns the security attribute and region number of an address.



Three possible configurations to define the security attribute of an address:

1. Internal SAU only
2. External IDAU only
3. A combination of the internal [SAU](#) and external IDAU

Note:

- Series 2 devices use configuration 3.
- IDAU in Series 2 devices is the [External Secure Attribution Unit \(ESAU\)](#).

The [Memory Protection Unit \(MPU\)](#) is a programmable unit that allows privileged software to define memory access permission. If the TrustZone is enabled, there can be up to two MPUs, one for Secure and one for Non-secure.

- The number of [MPU regions](#) for the Secure and the Non-secure MPU can be different.
- The MPU registers are memory-mapped and are placed in the [System Control Space \(SCS\)](#).
- Secure software can use the [MPU Functions for ARMv8-M](#) in CMSIS-Core to configure the Non-secure MPU through the [Non-secure alias SCS](#) (0xE002ED90 - 0xE002EDC4).

Software	Non-secure MPU Registers	Secure MPU Registers	MemManage Fault
Non-secure privileged	0xE000ED90 - 0xE000EDC4	—	Non-secure MPU violation
Secure privileged	0xE002ED90 - 0xE002EDC4	0xE000ED90 - 0xE000EDC4	Secure MPU violation

2.5 Exceptions and Interrupts

2.5.1 Type of Exceptions

The following table describes the types of exceptions in the TrustZone implemented system.

Exception (IRQ) Number	Exception	Type	Default State
1 (—)	Reset	Secure only	Secure
2 (-14)	NMI	Configurable	Secure
3 (-13)	HardFault	Configurable	Secure
4 (-12)	MemManage Fault	Banked	Banked
5 (-11)	BusFault	Configurable	Secure
6 (-10)	UsageFault	Banked	Banked
7 (-9)	SecureFault	Secure only	Secure
11 (-5)	SVCall	Banked	Banked
12 (-4)	DebugMonitor	Configurable	Secure
14 (-2)	PendSV	Banked	Banked
15 (-1)	SysTick	Banked	Banked
16 - 495 (0 - 479)	IRQ0 - IRQ479	Configurable	Secure

Note:

- "Secure only" means the system exceptions can only trigger in the Secure state.
- "Configurable" means the system exceptions and interrupts can be configured to target either the Secure state or the Non-secure state.
- Banked means the system exceptions can have Secure and Non-secure versions. Both can be triggered and executed independently and have different priority level settings.

2.5.2 Exception Priorities

It may cause a security issue if the Non-secure software uses high priority levels to mask the Secure interrupts. To avoid this issue, TrustZone introduces a programmable bit in the `AIRCR` register called `PRIS` (Prioritize Secure exception) for Secure software to prioritize, if needed, Secure exceptions and interrupts.

The `AIRCR.PRIS` is set to 0 out of reset, which means Secure and Non-secure exceptions/interrupts share the same configurable programmable priority level space (columns 2 and 3 in the following table). When the `AIRCR.PRIS` is set to 1, all Non-secure configurable exceptions/interrupts are placed in the lower half of the priority level space so that Secure exceptions/interrupts can potentially have higher priorities (columns 2 and 4 in the following table).

Priority Value	Secure Priority	Non-secure Priority (PRIS = 0)	Non-secure Priority (PRIS = 1)
0	0	0 (0x00)	128 (0x80)
1	32	32 (0x20)	144 (0x90)
2	64	64 (0x40)	160 (0xA0)
3	96	96 (0x60)	176 (0xB0)
4	128	128 (0x80)	192 (0xC0)
5	160	160 (0xA0)	208 (0xD0)
6	192	192 (0xC0)	224 (0xE0)
7	224	224 (0xE0)	240 (0xF0)

Note: This table uses three bits (Bit [7:5]) of the group priority level (`AIRCR.PRIGROUP`) to limit the maximum number of preemption levels to 8. A lower priority value indicates a higher priority.

2.5.3 Vector Tables

The following figure shows two vector tables for Secure and Non-secure exceptions and interrupts. The vector table offset is defined by a Vector Table Offset Register (VTOR at 0xE000ED08), which can only be programmed in the privileged state.

Exception number	IRQ number	Secure Vector	Non-secure Vector	Offset
495	479	IRQ479	IRQ479	0x7BC
⋮	⋮	⋮	⋮	⋮
18	2	IRQ2	IRQ2	0x48
17	1	IRQ1	IRQ1	0x44
16	0	IRQ0	IRQ0	0x40
15	-1	SysTick_S	SysTick_NS	0x3C
14	-2	PendSV_S	PendSV_NS	0x38
13		Reserved	Reserved	0x30
12	-4	DebugMonitor	DebugMonitor	
11	-5	SVCall_S	SVCall_NS	0x2C
10		Reserved	Reserved	
9				
8				
7	-9	SecureFault		0x1C
6	-10	UsageFault_S	UsageFault_NS	0x18
5	-11	BusFault_S	BusFault_NS	0x14
4	-12	MemManage_S	MemManage_NS	0x10
3	-13	HardFault_S	HardFault_NS	0x0C
2	-14	NMI_S	NMI_NS	0x08
1		Reset		0x04
		Initial SP value		0x00

Image: [Vector Table](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

Note:

- The `VTOR_S` defines the address of the Secure vector table in Secure memory, and the [Secure Main Stack Pointer](#) (`MSP_S`) is the default stack for the Secure exception handler.
- The `VTOR_NS` defines the address of the Non-secure vector table in Non-secure memory, and the [Non-secure Main Stack Pointer](#) (`MSP_NS`) is the default stack for the Non-secure exception handler.
- Secure privileged software can access the `VTOR_NS` using the [Non-secure SCB alias](#) (0xE002ED08).
- The [System Control Space](#) contains registers for the SysTick timer, NVIC, and SCB.
- The interrupt masking registers (PRIMASK, FAULTMASK, and BASEPRI) are [banked](#) between security states. The priority level space is shared between the Secure and the Non-secure world, setting an interrupt mask register on one side can block some, or all, of the exceptions on the other side.
- Interrupts (IRQ0 - IRQ479) are defined as Secure by default. Each interrupt can be configured as Secure or Non-secure and is determined by the Interrupt Target Non-secure (`NVIC_ITNS`) register, which is only programmable in the Secure software.

2.5.4 State Transitions in Exceptions and Interrupts

The following figure shows transitions between the [processor states](#) in ARMv8-M TrustZone.

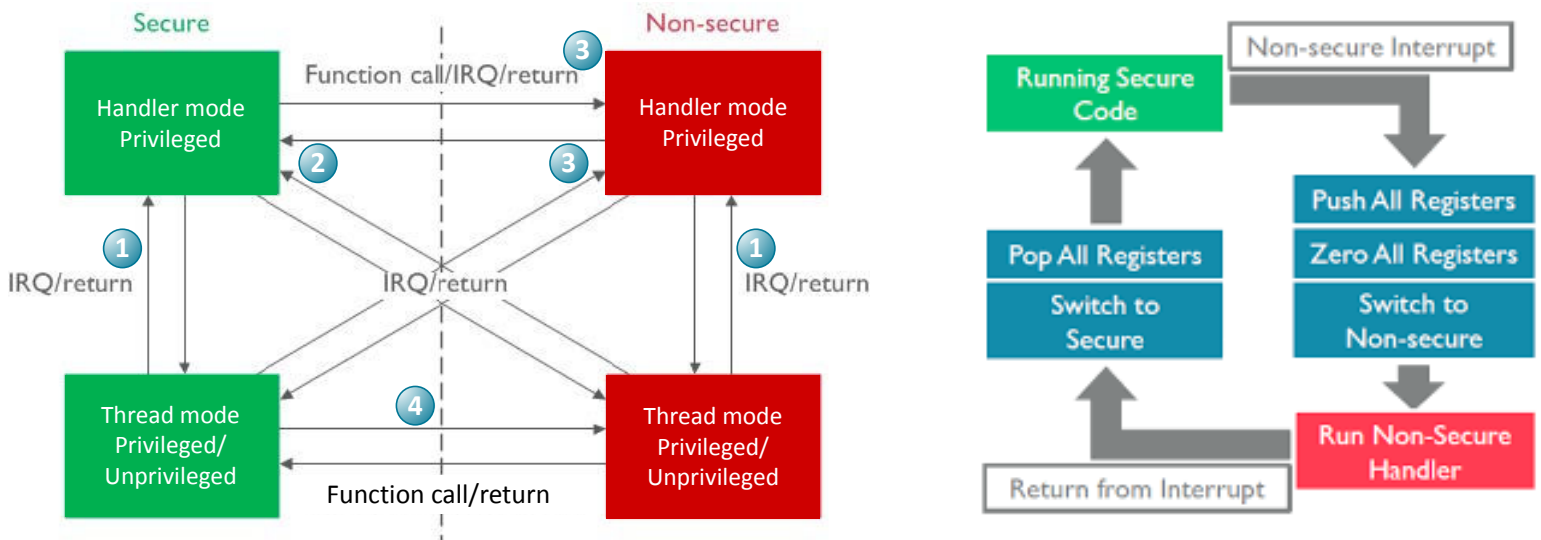


Figure 2.6. State Transitions

Image (left): [Switching-between-Secure-and-Non-secure-states](#). Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

1. Secure Thread → Secure Handler or Non-secure Thread to Non-secure Handler
 - No security state transition.
 - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.
 - The Interrupt Service Routine (ISR) is executed in the current security state (either Secure or Non-secure).
2. Non-secure Thread → Secure Handler or Non-secure Handler → Secure Handler
 - The transition from Non-secure to Secure state.
 - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.
 - The ISR is executed in a Secure state.
3. Secure Thread → Non-secure Handler or Secure Handler → Non-secure Handler
 - The transition from Secure to Non-secure state.
 - To avoid an information leak when transitioning from the Secure to Non-secure state. The processor automatically pushes all general-purpose registers into the Secure stack and erases the contents of all general-purpose registers before executing the Non-secure ISR. The processor pops the contents of all general-purpose registers from the Secure stack when returning from the Non-secure ISR (right side in [Figure 2.6 State Transitions on page 12](#)). It incurs a slightly longer interrupt latency.
 - The ISR is executed in a Non-secure state.
4. Secure Privileged Thread ↔ Non-secure Privileged Thread or Secure Unprivileged Thread ↔ Non-secure Unprivileged Thread
 - The transition from Secure to Non-secure state or Non-secure to Secure state.
 - The [Function calls and returns](#) can be used when the privileged level remains the same.

Note: Subject to interrupt priority, there are no restrictions regarding whether a Non-secure or Secure interrupt can occur when the processor runs Non-secure or Secure code.

2.6 Switching Between Secure and Non-secure States

The TrustZone allows direct calling between Secure and Non-secure software. The following figure shows how to use an API function call to trigger security state transitions. The state transitions can also happen because of [exceptions and interrupts](#).

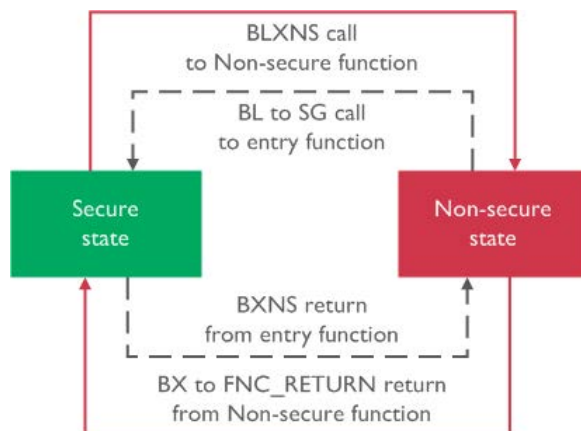


Image: *Switching-between-Secure-and-Non-secure-states*. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

2.6.1 Switching from Non-secure to Secure State

When the Non-secure program calls a Secure software, the first instruction must be a Secure Gateway (SG) instruction residing in Non-secure Callable memory. The Secure Gateway entry points (veneers) decouple the address of the SG instructions in the Non-secure Callable memory region from the rest of the Secure code. It can eliminate the risk of having inadvertent entry points when the Secure software contains a pattern that matches the opcode of the SG instruction.

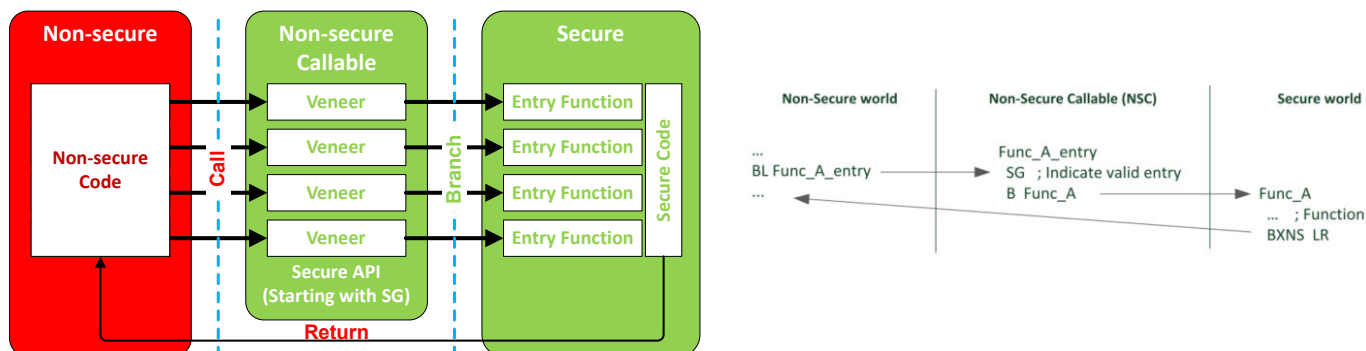


Image (right): *Whitepaper - ARMv8-M Architecture Technical Overview*. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

The bit 0 of the [Link Register \(LR\)](#) is cleared to zero by SG instruction to indicate that returning from this function transits from Secure to Non-secure. The processor is still in the Non-secure state when the SG instruction is executed. The BXNS LR instruction is used when returning since a normal BX LR instruction interprets it as an unsupported execution mode change. A [SecureFault](#) exception is triggered if the processor returns to a Secure address. It prevents a hacker from calling a Secure API with a fake return address pointing to a Secure program location. If bit 0 of LR is 1, the BXNS LR instruction behaves like a normal BX LR. Therefore, Secure code can call a Secure API in the NSC region even it is not a usual practice.

Program	Call Instruction	SG Instruction	Return Instruction
Non-secure call Non-secure	BL or BLX	—	BX LR (Return to Non-secure state)
Non-secure call Secure	BL or BLX	Clear bit 0 of LR	BXNS LR (Return to Non-secure state)
Secure call Secure	BL or BLX	Set bit 0 of LR	BXNS LR (Return to Secure state)

To help software developers create Secure APIs in C/C++, the [Cortex-M Security Extension \(CMSE\)](#) defines a C function attribute called `cmse_nonsecure_entry`.

- GCC — `__attribute__((cmse_nonsecure_entry))`
- IAR — `__cmse_nonsecure_entry`

2.6.2 Test Target (TT) Instruction

The software can use an ARMv8-M instruction called Test Target (TT) and the region number generated by the SAU or the IDAU to determine if a contiguous range of memory shares common security attributes and privilege levels.

The TT instruction returns the SAU/IDAU region number, security attributes (S/NS), and MPU region number after passing the start and end addresses of the memory range to the TT instruction. The software can determine whether the memory range has required security attributes and resides in the same region number.

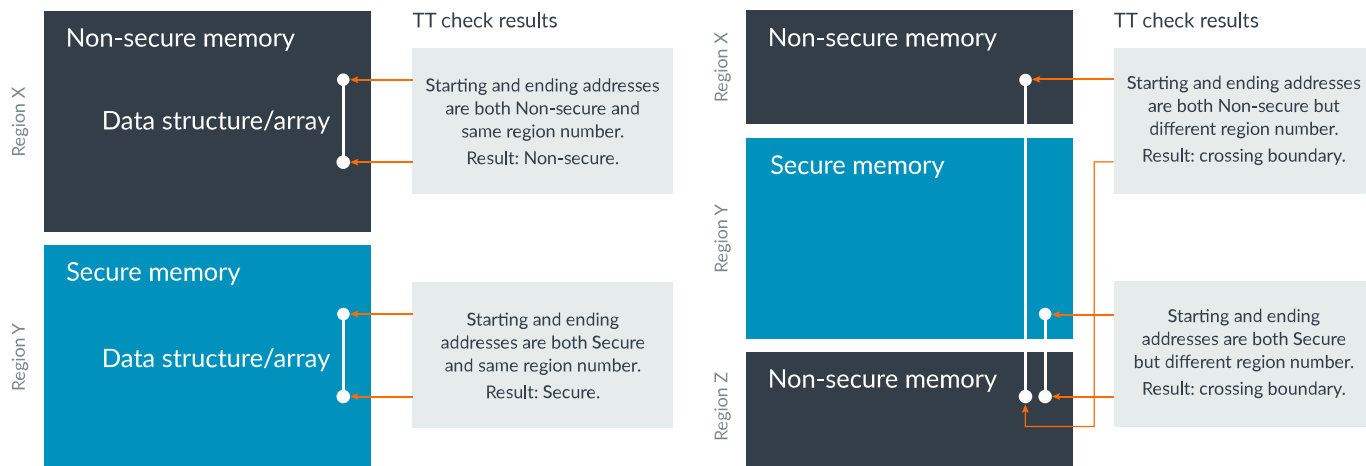


Image: *Test-target-instruction*. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

This mechanism allows security checking at the beginning of the API service (instead of during the operation) to determine if the memory referenced by a pointer from Non-secure software points to the Non-secure address. It prevents Non-secure software from using those APIs in Secure software to access or modify Secure data.

To make these operations easier in a C/C++ programming environment, the Cortex-M Security Extension (CMSE) has defined a range of [intrinsic functions](#) for dealing with pointer checks with the TT instructions.

2.6.3 Switching from Secure to Non-secure State

When the Secure program calls a Non-secure software, the Secure program must use a `BLXNS <reg>` instruction to invoke the process. If bit 0 of the `<reg>` is 0, the processor must switch to the Non-secure state when branching to the target address. During the state transition, the return address and some processor state information are pushed onto the Secure stack, while the return address on the [Link Register \(LR\)](#) is set to a special value called `FNC_RETURN (0xFEFFFFFF)`.

The Non-secure function completes by performing a branch (`BX LR`) to the `FNC_RETURN` address (bit 0 is 1 to indicate the function was called from the Secure state). It automatically triggers the unstacking of the actual return address from the Secure stack and returns to the calling function. The `FNC_RETURN` hides the return address of the Secure program from the Non-secure software to avoid the leakage of any secret information. It also prevents Non-secure software from modifying the Secure return address stored in the Secure stack.

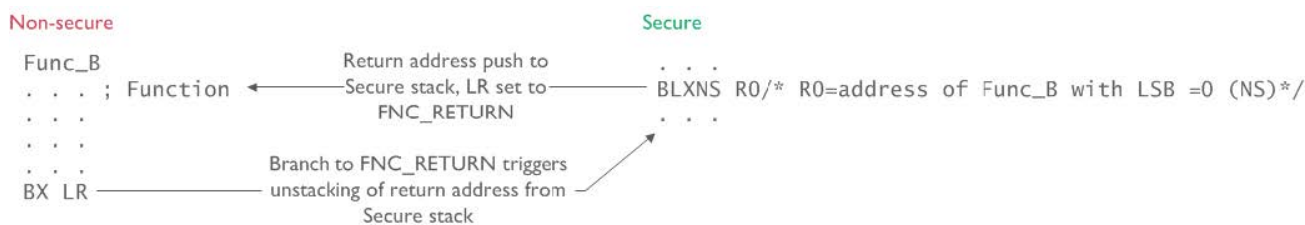


Image: *Switching-between-Secure-and-Non-secure-states*. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

To help software developers declare Non-secure function pointers in C/C++, the Cortex-M Security Extension (CMSE) defines a C function attribute called `cmse_nonsecure_call`.

- GCC: `__attribute__((cmse_nonsecure_call))`
- IAR: `__cmse_nonsecure_call`

2.7 Software Flow

The following figure describes a software flow example in a TrustZone implemented system.

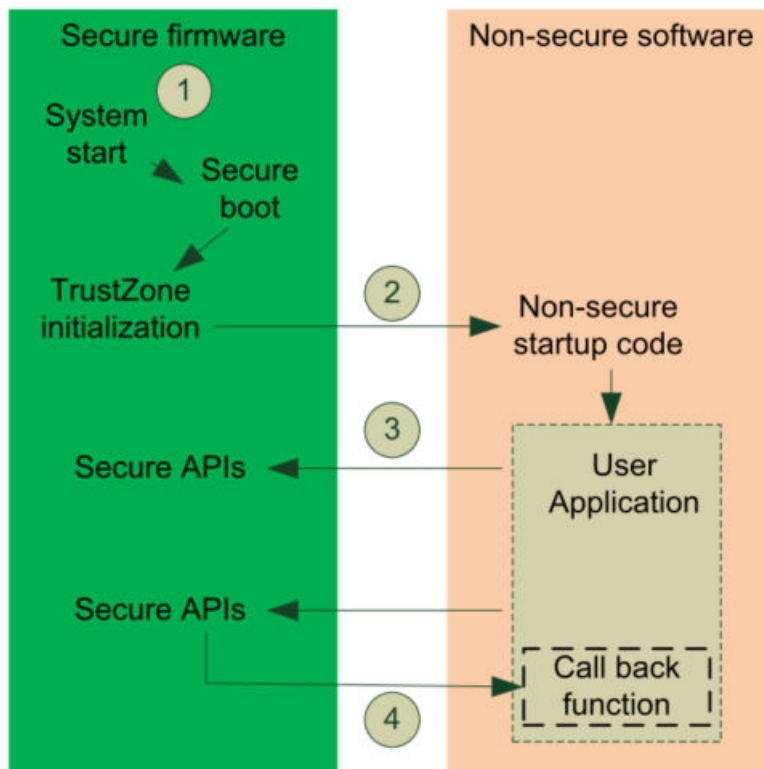


Figure 2.7. Execution Flow of a TrustZone Implemented System

Image: *Software Development in ARMv8-M Architecture*. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

1. The system starts executing code in the Secure state after a power-on or reset (Secure boot).
 - The Secure [stack pointer](#) (`MSP_S`) is set from the address of the Secure vector table (`VTOR_S`).
 - The Secure Reset Handler pointed by the `VTOR_S` is called.
 - Perform various initialization tasks such as C startup code.
 - Place peripherals and associated interrupts in either Secure or Non-secure applications.
 - Program [SAU/IDAU](#) to partition the entire memory into Secure, Non-secure Callable, and Non-secure regions.
 - Program the address of the Non-secure vector table (`VTOR_NS`).
 - Initialize the two first entries of the table for the Non-secure stack pointer (`MSP_NS`) and Reset Handler to emulate a Non-secure reset.
2. The Secure firmware branches to the entry point (Reset Handler pointed by the `VTOR_NS`) of the Non-secure application.
 - The Non-secure software has its Reset Handler.
 - Perform various initialization tasks such as C startup code and hardware initialization (e.g., Non-secure peripherals).
 - It does not conflict with initialization from the Secure code as the stack and heap spaces of Secure and Non-secure code are separated.
3. During the execution of Non-secure applications, the application could call Secure APIs through the [Secure Gateway \(SG\) veneer](#) in the Non-secure Callable region.
4. In some cases, Secure APIs might need to call [Non-secure call-back functions](#) (e.g., a hardware driver).

3. Bus Level Security (BLS)

3.1 System Design

The following figure shows two system designs:

- The sample system contains an ARMv8-M processor and the required components to support TrustZone.
- Bus Level Security (BLS) on Series 2 devices implements the concepts introduced in the ARM TrustZone sample system. BLS enforces Secure and privileged programming models and uses security components (colored blocks) to configure the security attribute and privileged level of peripherals and Bus Masters.

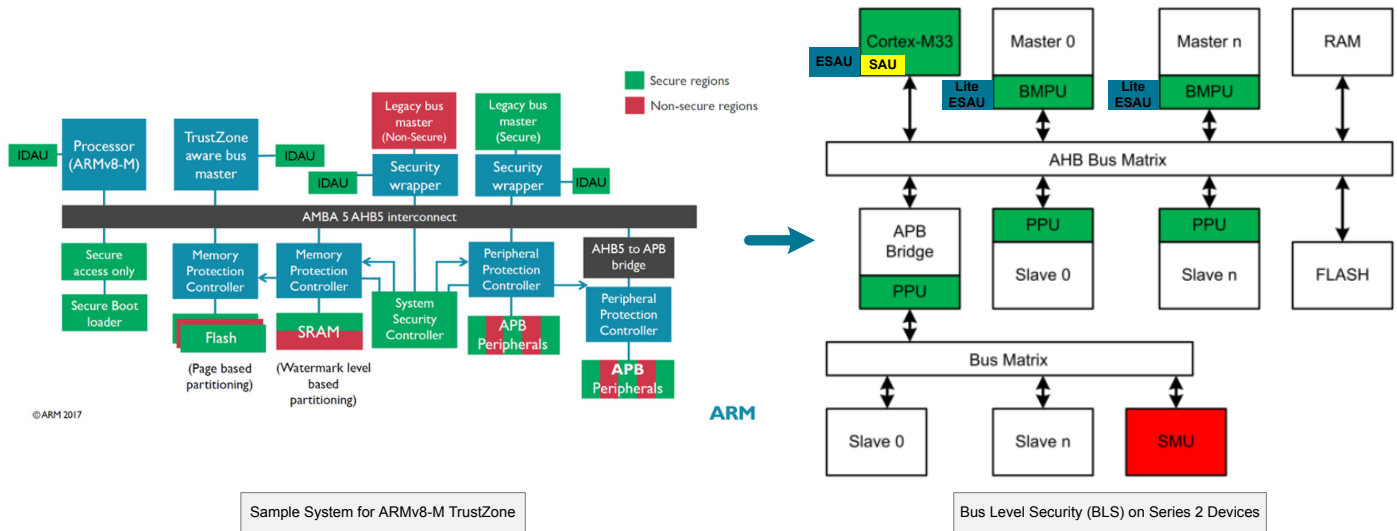


Figure 3.1. ARMv8-M TrustZone Implementation

ARMv8-M Processor

The ARMv8M processor is TrustZone capable of Secure and Non-secure states. It has a dedicated internal SAU that is fully programmable up to 8 different memory regions. Out of reset, the processor is in a Secure state and every transaction is a Secure transaction.

ARMv8-M Processor in Series 2 devices is the Cortex-M33.

System Security Controller

The system security controller is the central location for all security settings in the system. Each type of controller, IDAU, and wrapper receives its security configuration and bus response configuration from this block.

System Security Controller in Series 2 devices is the [Security Management Unit \(SMU\)](#).

Implementation Defined Attribution Unit (IDAU)

The IDAU generates the security attribute for a given address. All IDAUs in the system have the same memory partitioning. The IDAU is intended only for ARMv8-M cores and utilizes the entire IDAU interface for the core. The lite IDAU uses only the Secure and Non-secure interface from the IDAU and is intended for Non-ARMv8-M Bus Masters.

IDAU in Series 2 devices is the [External Secure Attribution Unit \(ESAU\)](#).

Security Wrapper

The Security Wrapper gives a legacy Bus Master the ability to drive security attribution. The security wrapper outputs the transaction address to the lite IDAU which returns the security attribute of the address. If the wrapper is configured as Non-secure, any transactions to a Secure address are blocked.

Security Wrapper in Series 2 devices is the [Bus Master Protect Unit \(BMPU\)](#).

Memory Protection Controller (MPC)

MPC has a security configuration for a per block of memory or memory above and below the watermark. If the security attribute of the block or memory region does not match the security attribute of the address, the transaction is blocked. This controller is used in a system that alias RAM or flash memory locations. This controller is not needed when the memory region size is programmable in an IDAU.

Series 2 devices have a programmable flash and RAM region in the [ESAU](#) (equivalent to IDAU) and are not implementing this block.

Peripheral Protection Controller (PPC)

PPC has a security configuration for every peripheral. If the security attribute of the selected peripheral does not match the security attribute of the address, the transaction is blocked. This controller is used in systems that alias the peripheral memory locations.

PPC in Series 2 devices is the [Peripheral Protection Unit \(PPU\)](#).

Hardware security is now extended to the peripheral bus system of the processor. Each component on the bus can verify and propagate the security level for each bus operation. The following sections describe the individual security component for BLS on Series 2 devices.

3.2 Security Management Unit (SMU)

The SMU is the only user-facing block in the BLS architecture and houses all the configuration and status for the [ESAU](#)s, [BMPUs](#), and [PPUs](#).

- Thirteen memory regions ([ESAU](#))
- Per Bus Master privileged and security attribute ([BMPU](#))
- Interrupt flag for Bus Master security fault (fault table in BMPU section)
- Per peripheral privileged and security attribute ([PPU](#))
- Interrupt flags for privileged, security, and instruction peripheral access faults (fault tables in PPU section)
- Separate Secure and Privileged IRQ

The SMU configurations can be [locked](#) down and protected from runaway code. The `SMU_LOCK` register resets to UNLOCK. Any write other than the unlock code (`0xACCCE55`) locks all SMU registers from further updates. The `SMU_STATUS` register contains a `SMULOCK` bit-field with the current lock state of the SMU.

The `SMU_M33CTRL` register can [lock](#) down internal security and privileged configurations below.

- Cortex-M33 SAU
- Non-secure MPU
- Secure MPU
- Non-secure Vector Table Offset Register (VTOR)
- Secure AIRCR register

Interrupt flags in the `SMU_IF` register can generate a [Secure or Privileged interrupt](#) in the table below when its corresponding interrupt enable bit in the `SMU_IEN` register is set and `IRQn` is enabled.

Enable Bit in SMU_IEN Register	IRQn	Interrupt Handler
BMPUSEC, PPUSEC	SMU_SECURE_IRQn	SMU_SECURE_IRQHandler()
PPUPRIV, PPUINST	SMU_PRIVILEGED_IRQn	SMU_PRIVILEGED_IRQHandler()

Each interrupt flag in the `SMU_IF` register can be cleared by writing 1 to the corresponding bit of the `SMU_IF_CLR` register.

3.3 External Secure Attribution Unit (ESAU)

The ESAU is responsible for determining the memory region and security attribute of a given address. Referring to [Figure 3.1 ARMv8-M TrustZone Implementation on page 16](#), the Cortex-M33 interfaces with an ESAU and the BMPUs of other Bus Masters interface with lite ESAUs to determine the security attribute of all transactions. The following figure describes the security attributes of different memory regions defined by the ESAU on Series 2 devices.

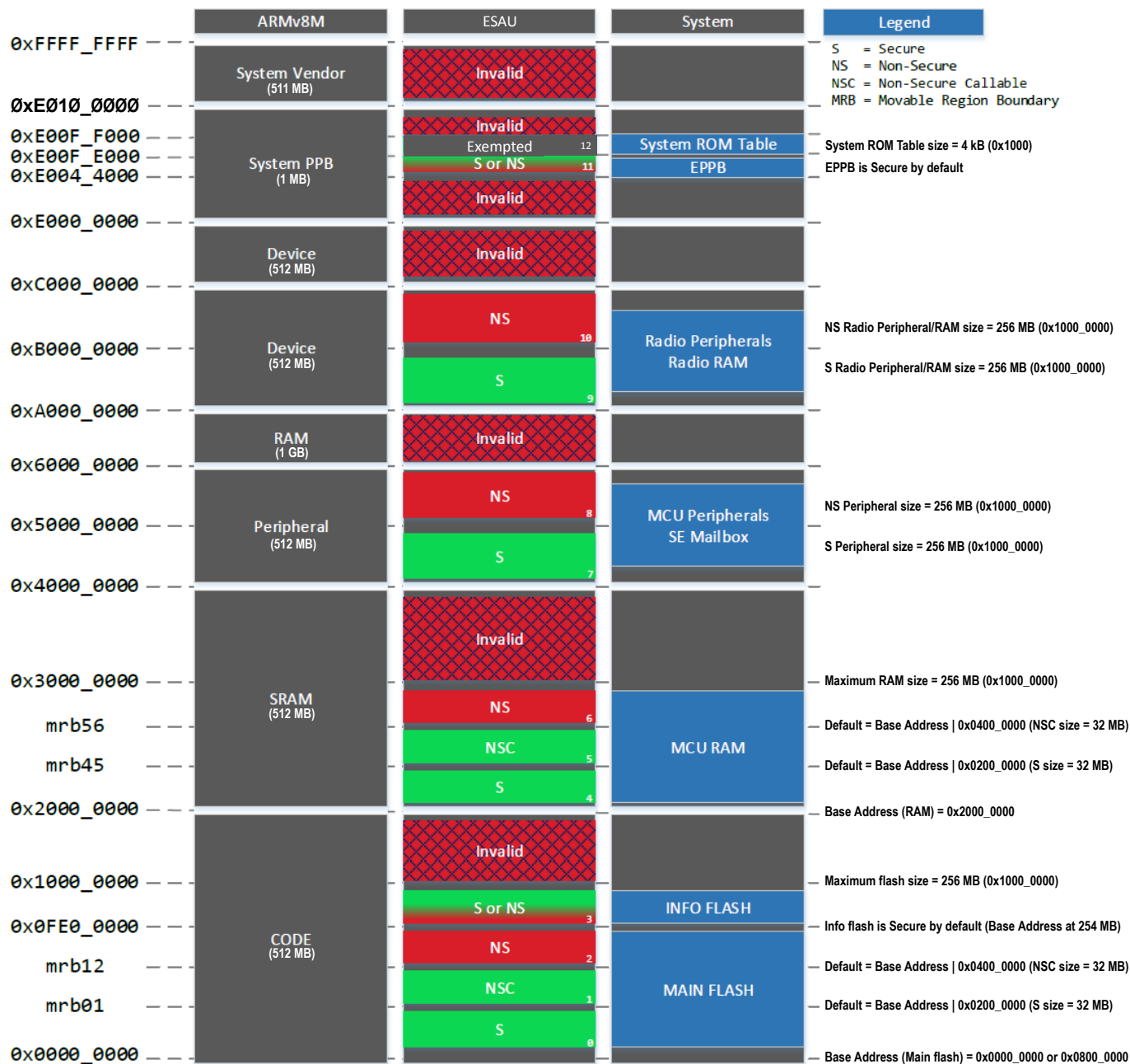


Figure 3.2. System Memory Map of Series 2 Device with TrustZone

Note:

- For Series 2 devices with base address 0x08000000 in region 0, the memory address from 0x0 to 0x07FFFFFFF is an invalid region.
- The invalid regions are deemed as Secure.
- The NSC and Exempted attributes are only available to the ESAU, and all lite ESAUs in the system view these attributes as Secure.

The ESAU divides the memory map into 13 memory regions and has a maximum of 6 Non-secure regions.

- Four Movable Region Boundaries (MRBs) determine the size of 6 regions.
- Two regions have configurable security attributes.
- Each memory region consists of a base address that specifies the start of the region and a limit address that specifies the end of the region plus one (+ 1).
- The address is valid if it falls between the base (\geq base) and limit ($<$ limit) of a region.
- If the memory region is not defined, it is deemed invalid and Secure.

The MRBs distinguish the Secure, Non-secure Callable, and Non-secure regions in flash and RAM. The two configurable regions determine if the Info flash and Cortex-M33 EPPB regions are Secure or Non-secure. The MRBs have a specific programming sequence. Any [misprogramming](#) results in a SMUPRGERR in the SMU_STATUS register.

ARMv8-M CODE Regions

- [Regions 0, 1, and 2](#) are in the Main space of flash. Region 3 is the info space of flash.
- The mrb01 (ESAUMRB01 in SMU_ESAURMBR01 register) determines the end of region 0 and the start of region 1.
- The mrb12 (ESAUMRB12 in SMU_ESAURMBR12 register) determines the end of region 1 and the start of region 2.
- The size of region 3 is device-dependent.
- Three regions' security attributes are static, and one region is configurable. Region 0 is always Secure, region 1 is always Non-secure Callable, and region 2 is always Non-secure. [Region 3](#) is configurable as either Secure or Non-secure (ESAUR3NS in SMU_ESAURTYPE0 register, default is secure after reset).
- Sizes of regions 0, 1, and 2 are adjusted in **4 kB increments** with the lower 12 bits of ESAUMRB## in SMU_ESAURMBR## ignored.
 - The Secure region can be set to size 0 when mbr01 = base address of region 0.
 - The Non-secure Callable regions can be set to size 0 when mbr01 = mbr12.
- The default value of mbr01 is equal to base address + 0x02000000, so the size of region 0 is 32 MB. Out of reset, all flash is Secure since all Series 2 devices have less than 32 MB of flash.

Region	Memory	Base Address	Limit Address	Security Attribute
0	Main flash	0x00000000 or 0x08000000	(0x00000000 or 0x08000000) mbr01	Secure
1	Main flash	(0x00000000 or 0x08000000) mbr01	(0x00000000 or 0x08000000) mbr12	Non-secure Callable
2	Main flash	(0x00000000 or 0x08000000) mbr12	0x0FE00000	Non-secure
3	Info flash	0x0FE0000	0x10000000	Secure or Non-secure

ARMv8-M RAM Regions

- [Regions 4, 5, and 6](#) cover the entire available RAM in the device.
- The mrb45 (ESAUMRB45 in SMU_ESAURMBR45 register) determines the end of region 4 and the start of region 5.
- The mrb56 (ESAUMRB56 in SMU_ESAURMBR56 register) determines the end of region 5 and the start of region 6.
- All three regions' security attributes are static. Region 4 is always Secure, region 5 is always Non-secure Callable, and region 6 is always Non-secure.
- Sizes of all three regions are adjusted in **4 kB increments** with the lower 12 bits of ESAUMRB## in SMU_ESAURMBR## ignored.
 - The Secure region can be set to size 0 when mbr45 = base address of region 4.
 - The Non-secure Callable region can be set to size 0 when mbr45 = mbr56.
- The default value of mbr45 is equal to 0x02000000, so the size of region 4 is 32 MB. Out of reset, all RAM is Secure since all Series 2 devices have less than 32 MB of RAM.

Region	Memory	Base Address	Limit Address	Security Attribute
4	SRAM	0x20000000	0x20000000 mbr45	Secure
5	SRAM	0x20000000 mbr45	0x20000000 mbr56	Non-secure Callable
6	SRAM	0x20000000 mbr56	0x30000000	Non-secure

ARMv8-M Peripheral Regions

- These regions are aliases to the [chip peripherals and SE mailbox](#) (a device with HSE).
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 7 is always Secure, and region 8 is always Non-secure.

Region	Memory	Base Address	Limit Address	Security Attribute
7	Chip Peripherals	0x40000000	0x50000000	Secure
8	Chip Peripherals	0x50000000	0x60000000	Non-secure

ARMv8-M Device Regions

- These regions are aliases to all radio peripherals and radio RAM.
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 9 is always Secure, and region 10 is always Non-secure.
- From the perspective of the device bus system, the [radio is one peripheral that is either Secure or Non-secure](#). So any Bus Master accessing the radio needs to know the security attribute of the radio. From the perspective of the radio, all of its radio bus peripherals are accessible regardless of the security attribute. However, the radio needs to know the security attribute of chip bus peripherals to access them through the correct alias.

Region	Memory	Base Address	Limit Address	Security Attribute
9	Radio Peripherals	0xA0000000	0xB0000000	Secure
10	Radio Peripherals	0xB0000000	0xC0000000	Non-secure

ARMv8-M System Private Peripheral Bus (PPB) Regions

- Both regions have a fixed size.
- [Region 11](#) is the Cortex-M33 EPPB memory region and is configurable as either Secure or Non-secure (`ESAU11NS` in `SMU_ESAURTPES1` register, default is secure after reset). It is important to note that the Cortex-M33 core is the only Bus Master that sees these memory regions. All other Bus Masters in the system do not have access to the System PPB, and it is an invalid region.
- Region 12 has a static security attribute of Exempted. It means that the Cortex-M33 core allows the transaction in all cases. It permits debuggers to read the system ROM Table regardless of the state of the Cortex-M33 core.

Region	Memory	Base Address	Limit Address	Security Attribute
11	EPPB	0xE0044000	0xE00FE000	Secure or Non-secure
12	System ROM Table	0xE00FE000	0xE00FF000	Exempted

Note:

- The regions in flash (0/1/2) and RAM (4/5/6) can only create in the order of Secure, Non-secure Callable, and Non-secure.
- The [ESAU and lite ESAUs](#) handle the transactions of Bus Masters and must have consistent security attribute mapping. Therefore, configurations in the SMU registers apply to ESAU and lite ESAUs.
- Unlike other Bus Masters using B MPU and lite ESAU, merging the address lookup results from the [internal SAU and ESAU](#) determines the [security attribute](#) of the Cortex-M33 transaction.

Bus Master	Security Attribution
Cortex-M33	SAU and ESAU
Other	Lite ESAU

3.4 Security Attribution Unit

In Series 2 devices, the combination of the integrated SAU in the Cortex-M33 processor and an ESAU determine the security attribute of a Cortex-M33 transaction.

The SAU consists of several [programmable registers](#). These registers are placed in the [System Control Space \(SCS\)](#) and are only accessible from the Secure privileged state.

- SAU Control Register (*SAU_CTRL*) — The SAU is disabled after RESET
- SAU Type Register (*SAU_TYPE*) — Indicates the number of [available regions](#) (read-only)
- SAU Region Number Register (*SMU_RNR*) — Assigns a region number
- SAU Region Base Address Register (*SAU->RBAR*) — Configures selected region base address
- SAU Region Limit Address Register (*SAU->RLAR*) — Configures selected region limit address and security attribute (NSC or NS), enable or disable the region

The following figure shows three different SAU configurations for determining the security attribute of a Cortex-M33 transaction.

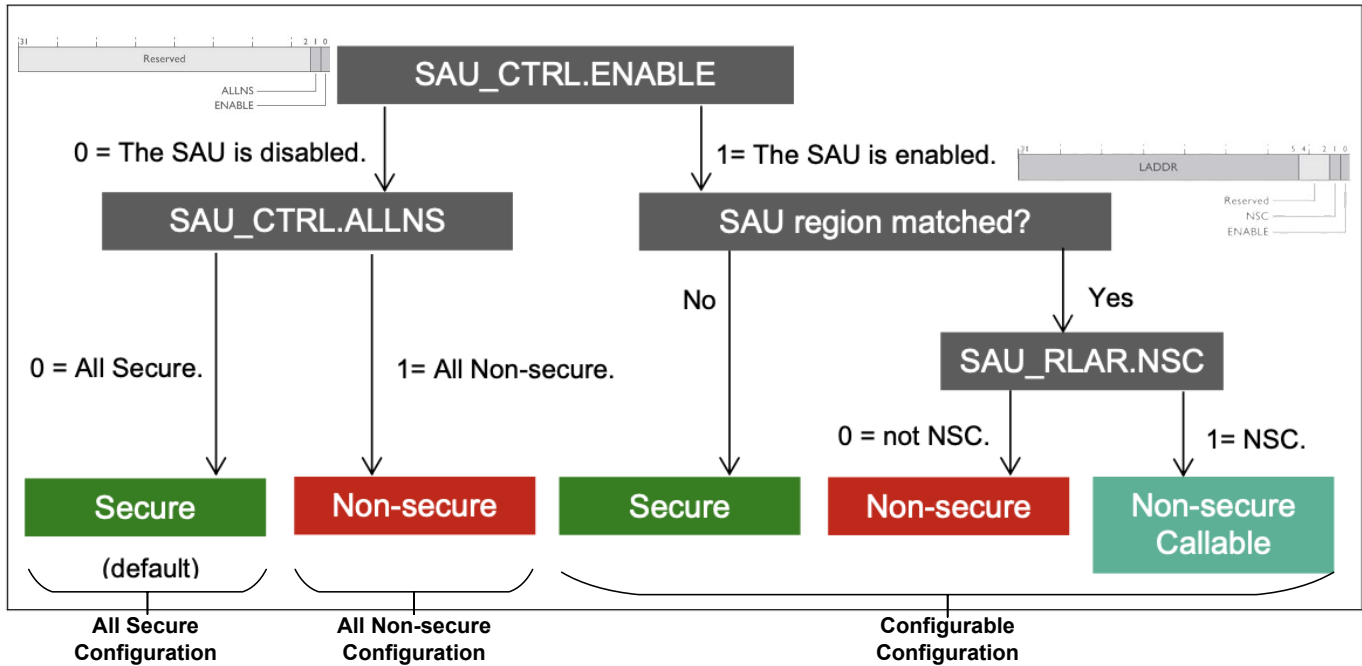
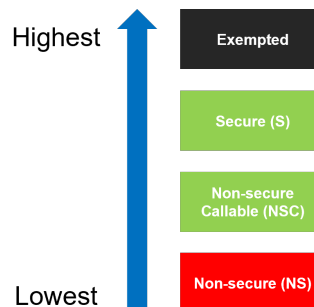


Figure 3.3. Configuration of SAU_CTRL Register

Note:

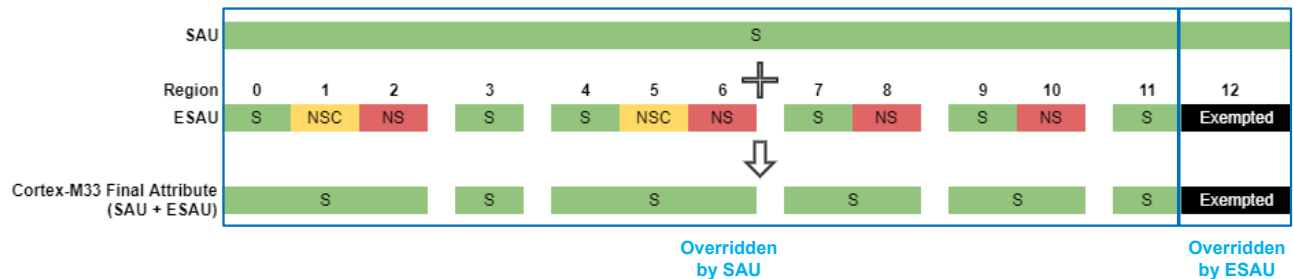
- All address ranges after RESET in SAU are Secure by default.
- The SAU can configure a **32 bytes aligned** region as Non-secure or Non-secure Callable. Any address not defined in the SAU defaults to Secure.
- An [ESAU](#) can configure or hard-code a region as Secure, Non-secure Callable, Non-secure, or Exempted. An [Exempted](#) region enables Non-secure debuggers to access debugging components and establish a debug connection to the processor before the SAU is configured.
- The processor determines the final attribute of the address based on the higher security attribute (Exempted > S > NSC > NS) from either the SAU or the ESAU.



3.4.1 All Secure Configuration

Highlights:

- SAU is disabled.
- ALLNS bit in the SAU Control register is clear.
- The whole memory is in a Secure state (highest security attribute apart from Exempted).
- All Cortex-M33 transactions in this configuration are Secure or Exempted and give the Cortex-M33 access to all memory locations through either the Secure or Non-secure alias after RESET.

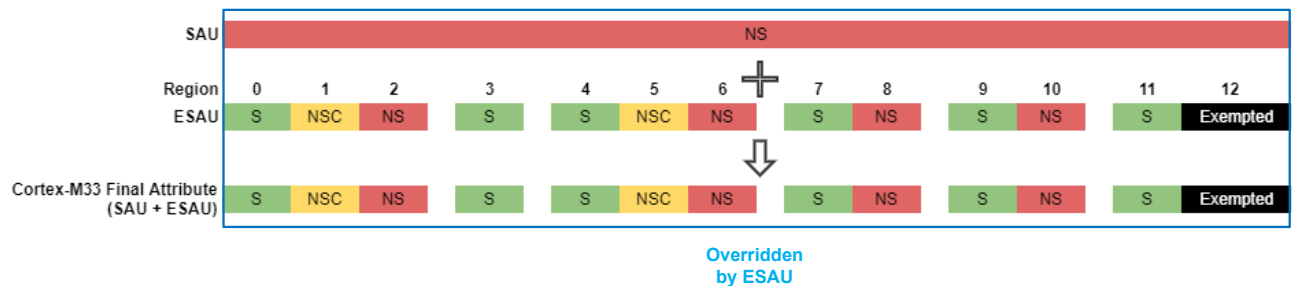


- It is up to the boot procedure in a Secure state to keep the current configuration or use other configurations once the boot process is complete.

3.4.2 All Non-secure Configuration

Highlights:

- SAU is disabled.
- ALLNS bit in the SAU Control register is set.
- The whole memory is in a Non-secure state (lowest security attribute).
- Therefore the [ESAU configuration](#) determines the security attribute of all Cortex-M33 transactions.

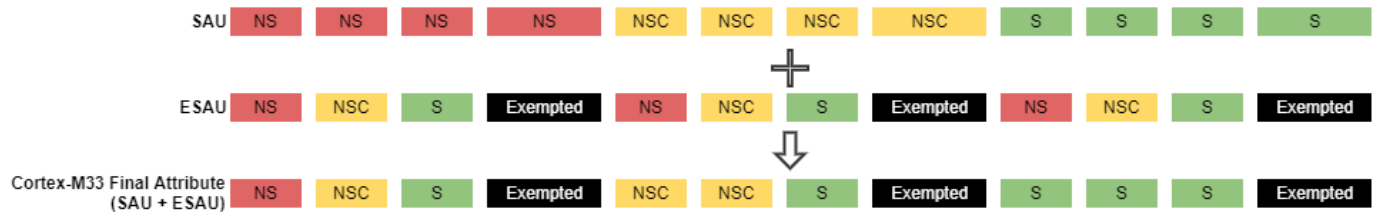


- Except for the [SAU_CTRL register](#), this configuration does not require programming on other SAU registers.

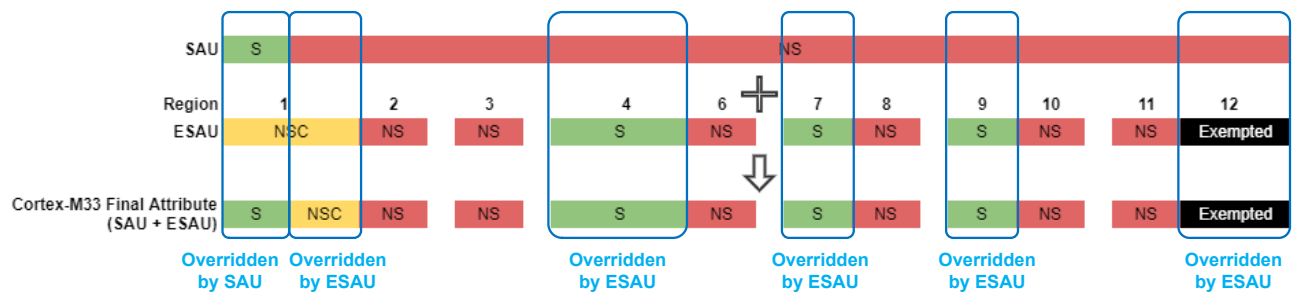
3.4.3 Configurable Configuration

Highlights:

- SAU is enabled.
- ALLNS bit in the SAU Control register can be 0 or 1 (do not care).
- The NSC bit on the SAU_RLAR register determines the security attribute of an address as Non-secure or Non-secure Callable if an address matches an SAU region.
- The security attribute of an address is Secure by default if the address does not match any SAU region.
- This configuration programs SAU_RNR, SAU_RBAR, and SAU_RLAR registers to correlate the Non-secure regions in ESAU.
- The SAU or ESAU overrides the attribute to a higher security level if any security attribute mismatch occurs in a memory region.



- The following figure is an example of a configurable configuration with the size of ESAU regions 0 and 5 are set to zero.



Note: The Cortex-M33 has an internal SAU that defaults all undefined addresses to Secure if enabled. If the Secure regions do not align between the Cortex-M33 (SAU + ESAU) and other Bus Masters (lite ESAU), the Cortex-M33 treats a memory region as Secure while other Bus Masters treat it as Non-secure. It can lead to the leaking of secure data if the Cortex-M33 stores secure data in what other Bus Masters think is a Non-secure area (Figure 5.1 Main Flash Layout on page 34).

3.5 Bus Master Protection Unit (BMPU)

The BMPU is a security wrapper used for assigning a Bus Master specific security and privileged states. Referring to [Figure 3.1 ARMv8-M TrustZone Implementation on page 16](#), the BMPU generally lies between the Bus Master and the Advanced High-performance Bus (AHB) Matrix. BMPU interfaces with a [lite ESAU](#) to determine the security attribute of all Bus Master transactions.

The registers below in SMU configure the [security](#) and [privileged](#) state of a Bus Master. The Bus Masters in group 0 are device-dependent. Out of reset, each Bus Master is Secure and privileged.

Register	Description
SMU_BMPUPATD0	Bitfields (privileged if set) for privileged attribute configuration on Bus Master group 0
SMU_BMPUSATD0	Bitfields (Secure if set) for security attribute configuration on Bus Master group 0

Note: The Bus Master privileged attribute only applies to peripheral accesses. Flash and RAM accesses ignore the privileged attribute of the Bus Master.

The BMPU generates a security fault when the security attribute of the bus transaction is Secure, and the security attribute (`SMU_BMPUSATD0`) for the BMPU is configured as Non-secure.

Below is the security fault table that shows how the security attribute on the bus is driven based on the lite ESAU attribute and the BMPU security configuration. The interrupt is triggered if `BMPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Lite ESAU Attribute	Secure Bus Master	Non-secure Bus Master
Non-secure	Non-secure	Non-secure
Secure	Secure	FAULT

Upon a BMPU fault, the registers in SMU below notify that a BMPU security fault occurred and on which Bus Master. The registers also identify the offending fault address. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and the corresponding interrupt flag is set in the `SMU_IF` register. The values in `SMU_BMPUFS` and `SMU_BMPUFSADDR` do not change until the BMPU fault (`BMPUSEC`) in the `SMU_IF` register is cleared by software.

Register	Bitfield	Fault
SMU_IF	BMPUSEC	Security Fault if set
SMU_BMPUFS	BMPUFMASTERID	ID of the Bus Master that triggered the fault
SMU_BMPUFSADDR	BMPUFSADDR	Access address that triggered the fault

Note: No privileged fault is generated because all the other Bus Masters in the system do not drive the privileged attribute.

3.6 Peripheral Protection Unit (PPU)

The PPU is a security wrapper used for assigning a Bus Slave peripheral specific security and privileged states. Referring to [Figure 3.1 ARMv8-M TrustZone Implementation on page 16](#), the PPU comes in the form of a PPU in Advanced High-performance Bus (AHB) and a PPU in Advanced Peripheral Bus (APB).

- The PPU AHB generally lies between the Bus Matrix and an AHB Bus Slave peripheral.
- The PPU APB lies between the output of an AHB to APB bridge and all of the APB Slaves on that APB bus.

The registers below in SMU configure the [security](#) and [privileged](#) state of a peripheral. The peripherals in groups 0 and 1 are device-dependent. Out of reset, each peripheral is Secure and privileged. While each peripheral in address `0x40000000` (region 7) or `0x50000000` (region 8) can be configured independently, the radio subsystem in `0xA0000000` (region 9) or `0xB0000000` (region 10) is configured as a [unit](#).

Register	Description
SMU_PPUPATD0	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUPATD1	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUSATD0	Bitfields (Secure if set) for security access configuration on peripheral group 0
SMU_PPUSATD1	Bitfields (Secure if set) for security access configuration on peripheral group 1

The PPU can generate three types of faults:

1. Privileged faults occur on unprivileged transactions to privileged peripherals. Below is the privileged fault table that shows when a privileged fault occurs based on the PPU peripheral privileged configuration and the bus transaction privileged attribute. The interrupt is triggered if `PPUPRIV` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	Privileged Peripheral	Unprivileged Peripheral
Privileged	SUCCESS	SUCCESS
Unprivileged	FAULT	SUCCESS

2. Security faults occur on Secure transactions to Non-secure peripherals and Non-secure transactions to Secure peripherals. Below is the security fault table that shows when a security fault occurs based on the PPU Peripheral security configuration and the bus transaction security attribute. The interrupt is triggered if `PPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Bus Attribute	Secure Peripheral	Non-secure Peripheral
Secure	SUCCESS	FAULT
Non-secure	FAULT	SUCCESS

3. Instruction faults occur on any transaction marked as an instruction fetch. Below is the instruction fault table that shows when a PPU instruction fault occurs based on the bus transaction instruction attribute. The interrupt is triggered if `PPUINST` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	PPU Output
Instruction	FAULT
Data	SUCCESS

Upon a [PPU fault](#), the registers below in [SMU](#) notifies which PPU fault occurred and on which peripheral. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and set the corresponding interrupt flag in the `SMU_IF` register. The values in `SMU_IF` and `SMU_PPUFS` do not change until all PPU faults in the `SMU_IF` register are cleared by software.

Register	Bitfield	Fault
SMU_IF	PPUPRIV	Privilege Fault if set
SMU_IF	PPUSEC	Security Fault if set
SMU_IF	PPUINST	Instruction Fault if set
SMU_PPUFS	PPUFSPERIPHID	ID of the peripheral that caused the fault

3.7 Compatibility

Secure software usually controls the SYSCFG and SMU peripherals to prevent Non-secure software from changing critical configurations in the Secure domain. It requires switching between Secure and Non-secure states when Non-secure software wants to update the registers in these peripherals. Therefore dedicated registers for Non-secure access are added to SYSCFG and SMU peripherals on newer Series 2 devices.

3.7.1 System Configuration (SYSCFG)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table 3.1. Dedicated Bitfield to Configure Access for Non-secure SYSCFG Registers

Bitfield (Register)	Description
SYSCFGCFGNS (SMU_PPUPATD0)	Bitfields (privileged if set) for privileged access configuration on NS SYSCFG registers
SYSCFGCFGNS (SMU_PPUSATD0)	Bitfields (Secure if set) for security access configuration on NS SYSCFG registers

Note: Reset SYSCFGCFGNS bit in SMU_PPUSATD0 to allow Non-secure software to access NS SYSCFG registers.

Table 3.2. Dedicated SYSCFG Registers for Non-secure State

SYSCFG Non-secure Registers	Description
SYSCFG_CFGNS_CFGNSTCALIB	NS SysTick calibration value register
SYSCFG_CFGNS_ROOTNSDATA0	NS root data register 0
SYSCFG_CFGNS_ROOTNSDATA1	NS root data register 1

3.7.2 Security Management Unit (SMU)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table 3.3. Dedicated Bitfield to Configure Access for Non-secure SMU Registers

Bitfield (Register)	Description
SMUCFGNS (SMU_PPUPATD1)	Bitfields (privileged if set) for privileged access configuration on NS SMU registers
SMUCFGNS (SMU_PPUSATD1)	Bitfields (Secure if set) for security access configuration on NS registers

Note: Reset SMUCFGNS bit in SMU_PPUSATD1 to allow Non-secure software to access NS SMU registers.

The SMU_CFGNS register file is for the TrustZone Non-secure state and has its register lock (NSLOCK). It allows hardware to maintain the privileged assignments for the NS state. The privileged configuration within the NS state is the same as the Secure state, except it has an "NS" to differentiate the registers.

Table 3.4. Dedicated SMU Registers for Non-secure State

SMU Non-secure Registers	Description
SMU_CFGNS_NSSTATUS	Lock status of SMU_CFGNS registers
SMU_CFGNS_NSLCOK	Lock and unlock the SMU_CFGNS registers
SMU_CFGNS_NSIF	Interrupt flags for NS privilege (PPUNSPRIVIF) and instruction (PPUNSINSTIF) faults
SMU_CFGNS_NSIEN	Interrupt enable flags for NS privilege (PPUNSPRIVIEN) and instruction (PPUNSINSTIEN) faults
SMU_CFGNS_PPUNSPATD0	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 0
SMU_CFGNS_PPUNSPATD1	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 1
SMU_CFGNS_PPUNSF	ID (PPUFSPERIPHID) of the NS peripheral that caused the fault
SMU_CFGNS_BMPUNSPATD0	Bitfields (privileged if set) for privileged attribute configuration on NS Bus Master group 0

Table 3.5. Fault Statuses Only for Secure State

Bitfield (Register)	Description
PPUPRIV (SMU_IF)	Fault status now limited only to Secure state
PPUINST (SMU_IF)	Fault status now limited only to Secure state
PPUPRIV (SMU_IEN)	Fault status now limited only to Secure state
PPUINST (SMU_IEN)	Fault status now limited only to Secure state
PPUFSPERIPHID (SMU_PPUFS)	Fault status now limited only to Secure state

Table 3.6. Dedicated SMU Interrupt for Non-secure State

Interrupt	Description
SMU_NS_PRIVILEGED_IRQHandler()	An interrupt flag in the SMU_CFGNS_NSIF register can generate an NS privileged interrupt when its corresponding interrupt enable bit in the SMU_CFGNS_NSIEN register is set and SMU_NS_PRIVILEGED_IRQn is enabled, and in which the peripheral (ID) that triggers the fault is in the SMU_CFGNS_PPUNSF register.

4. Secure and Privileged Programming Model

The implementation of BLS on Series 2 devices, both flash and RAM, use a programmable watermark to delineate Secure, Non-secure Callable, and Non-secure regions. On the other hand, peripherals exist in both a Secure and Non-secure alias of memory.

4.1 BLS SMU Programming

4.1.1 Enabling SMU Clock

Except for the EFR32xG21 devices, all Series 2 devices enable the SMU clock in CMU before programming the SMU registers.

```
#if (_SILICON_LABS_32B_SERIES_2_CONFIG > 1)
    CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
#endif
```

4.1.2 Cortex-M33 Lock Control

The Cortex-M33 security and privileged configurations can be locked by programming the SMU_M33CTRL register.

```
// Lock Secure MPU configuration
SMU->M33CTRL |= SMU_M33CTRL_LOCKSMU;
```

4.1.3 Locking SMU Configuration

The entire SMU configuration can be locked down to avoid runaway code. Below is an example of how to lock and unlock the SMU.

```
uint32_t lock_status;

// Lock Down SMU
SMU->LOCK = ~SMU_LOCK_SMULOCKKEY_UNLOCK;

// Grab Lock Status
lock_status = (SMU->STATUS & _SMU_STATUS_SMULOCK_MASK) >> _SMU_STATUS_SMULOCK_SHIFT;

// Unlock SMU
SMU->LOCK = SMU_LOCK_SMULOCKKEY_UNLOCK;
```

4.1.4 Interrupt Control

Each interrupt flag in SMU_IF can generate an interrupt when its corresponding interrupt enable flag in the SMU_IEN register is set. Each interrupt flag can be cleared by writing the clear alias of the SMU_IF register.

```
// Clear and enable the SMU PPUSEC and BMPUSEC interrupt
NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
NVIC_EnableIRQ(SMU_SECURE_IRQn);
SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
```

4.2 BLS ESAU Programming

4.2.1 Region Types

The SMU_ESAURTYESn registers are used to configure memory regions with a specific security attribute. All configurable memory regions reset to Secure. Below is an example of programming regions 3 and 11 to Non-secure.

```
// Region 3 (Info flash) is Non-secure
SMU->ESAURTYES0 = SMU_ESAURTYES0_ESAUR3NS;

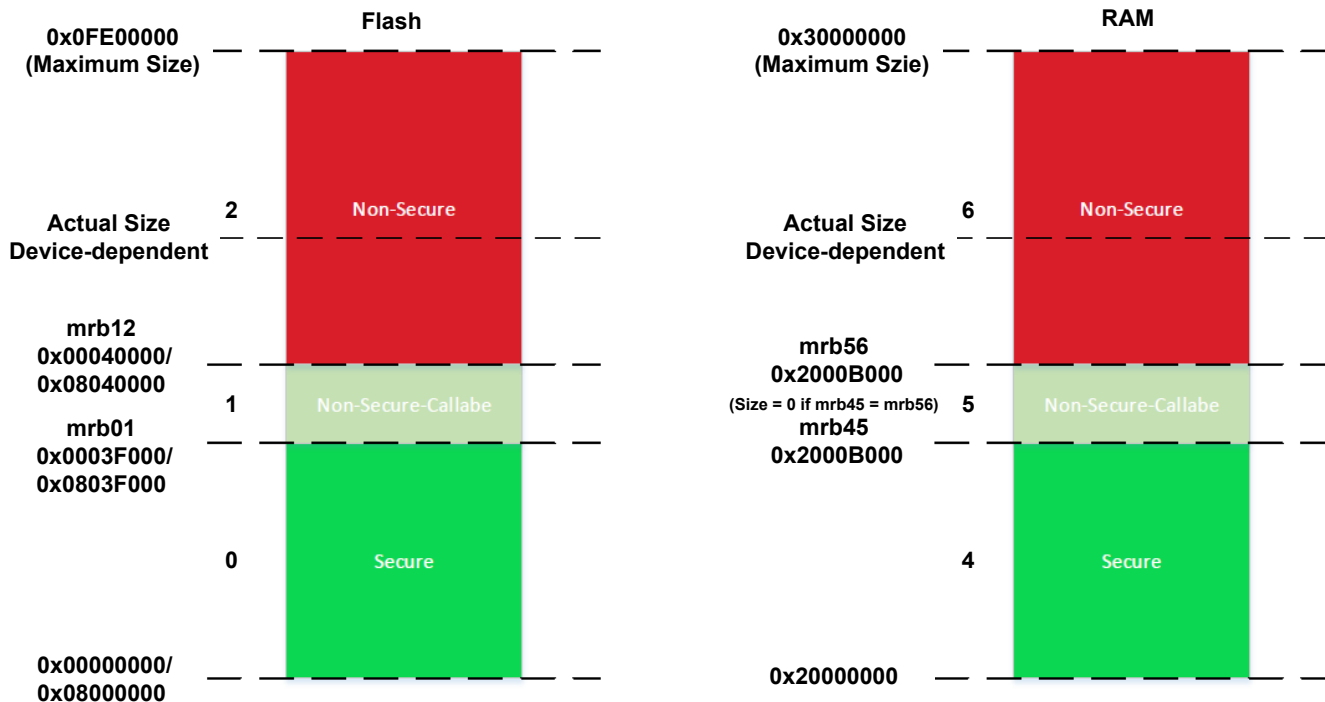
// Region 11 (EPPB) is Non-secure
SMU->ESAURTYES1 = SMU_ESAURTYES1_ESAUR11NS;
```

4.2.2 Region Sizes

The code and figure below highlight how to program the Movable Region Boundaries (MRBs) of ESAU.

```
// ESAU region 0/1/2 programming
// Boundary01 at 252kB and Boundary12 at 256kB
SMU->ESAUMRB01 = 0x0003F000U & _SMU_ESAUMRB01_MASK;
SMU->ESAUMRB12 = 0x00040000U & _SMU_ESAUMRB12_MASK;

// ESAU region 4/5/6 programming
// Boundary45 at 44kB and Boundary56 at 44kB (region 5 size = 0)
SMU->ESAUMRB45 = 0x0000B000U & _SMU_ESAUMRB45_MASK;
SMU->ESAUMRB56 = 0x0000B000U & _SMU_ESAUMRB56_MASK;
```



Note:

- The mrb12 (ESAUMRB12 in SMU_ESAUMBR12) has to be greater than or equal to mrb01 (ESAUMRB12 in SMU_ESAUMBR12).
- The mrb56 (ESAUMRB56 in SMU_ESAUMBR562) has to be greater than or equal to mrb45 (ESAUMRB45 in SMU_ESAUMBR45).
- If one of the rules above is violated, the SMU_STATUS.SMUPRGERR is asserted.
- When mrb01 and mrb12 are equal, region 1 (NSC) is a size of 0 and is not seen by the system.
- When mrb45 and mrb56 are equal, region 5 (NSC) is a size of 0 and is not seen by the system.

4.3 BLS SAU Programming

4.3.1 All Secure Configuration

All secure configuration is the default state after reset. It clears the SAU_CTRL.ENABLE and the SAU_CTRL.ALLNS bits in SAU, and the entire memory is in a Secure attribute.

4.3.2 All Non-secure Configuration

All Non-secure Configuration occurs when the SAU_CTRL.ENABLE bit is cleared, and the SAU_CTRL.ALLNS bit is set. The ESAU controls the security attribute of a Cortex-M33 transaction.

```
// Disable SAU (ALLNS = 1) and clear data and instruction pipe
SAU->CTRL = SAU_CTRL_ALLNS_Msk;
__DSB();
__ISB();
```

4.3.3 Configurable Configuration

Configurable configuration occurs when the SAU_CTRL.ENABLE bit is set (SAU_CTRL.ALLNS is irrelevant). Both SAU and ESAU determine the security attribute of a Cortex-M33 transaction. The code and figure below highlight how to program the SAU regions.

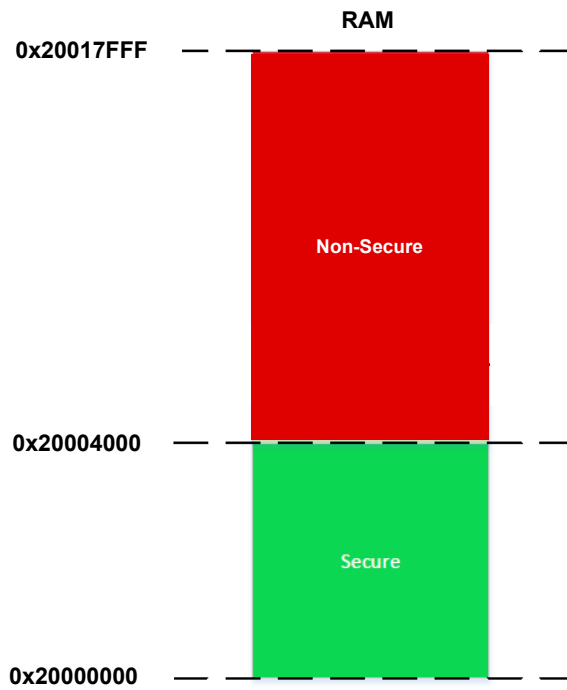
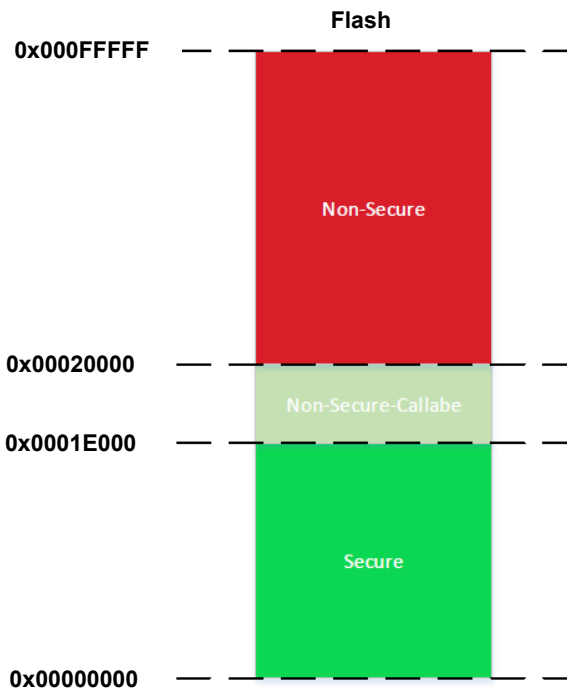
```
// Define all Non-secure (NS) and Non-secure Callable (NSC) Regions
#define REGION0_BASE 0x0001E000UL
#define REGION1_BASE 0x00020000UL
#define REGION2_BASE 0x20004000UL
#define REGION0_LIMIT 0x0001FFFFUL
#define REGION1_LIMIT 0x000FFFFFFUL
#define REGION2_LIMIT 0x20017FFFUL

// CMSIS calls to enable SAU Regions
// SAU region 0 - Flash NSC at 120 kB to 128 kB (0x0001E000 - 0x0001FFFF)
SAU->RNR = (0UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION0_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION0_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_NSC_Msk | SAU_RLAR_ENABLE_Msk;

// SAU region 1 - Flash NS at 128 kB to 1024 kB (0x00020000 - 0x000FFFFFF)
SAU->RNR = (1UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION1_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION1_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;

// SAU region 2 - RAM NS at 16 kB to 96 kB (0x20004000 - 0x20017FFF)
SAU->RNR = (2UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION2_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION2_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;

// CMSIS functions to enable SAU and clear data and instruction pipe
TZ_SAU_Enable();
__DSB();
__ISB();
```



4.4 BLS BMPU Programming

4.4.1 Bus Master Privileged Attribute

A Bus Master can be configured as either privileged (default) or unprivileged by programming the corresponding index in the SMU_BMPUPATDn register.

```
// Configure all odd Bus Masters unprivileged
for (i = 0; i < SMU_NUM_BMPUS; i++) {
    if (i & 0x01) {
        SMU->BMPUPATD0 &= ~(1 << i);
    }
}
```

4.4.2 Bus Master Security Attribute

A Bus Master can be configured as either Secure (default) or Non-secure by programming the corresponding index in the SMU_BMPUPATDn register. Configure a Bus Master as Non-secure results in the Bus Master only being able to access Non-secure addresses.

```
// Configure all odd Bus Masters Non-secure
for (i = 0; i < SMU_NUM_BMPUS; i++) {
    if (i & 0x01) {
        SMU->BMPUSATD0 &= ~(1 << i);
    }
}
```

4.4.3 Bus Master Fault Status

The Bus Master ID and the address that triggered the fault can be read from the SMU_BMPUFS and SMU_BMPUFSADDR registers.

```
uint32_t fs_bmpu_id;
uint32_t fs_bmpu_addr;
uint32_t fs_bmpu_secfault;

// Read Bus Master fault status
fs_bmpu_id = SMU->BMPUFS;
fs_bmpu_addr = SMU->BMPUFSADDR;
fs_bmpu_secfault = (SMU->IF & _SMU_IF_BMPUSEC_MASK) >> _SMU_IF_BMPUSEC_SHIFT;

// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_BMPUSEC;
```

4.5 BLS PPU Programming

4.5.1 Peripheral Privileged Attributes

A peripheral can be configured as either privileged (default) or unprivileged by programming the corresponding index in the SMU_PPUPATDn register.

```
// Configure all odd peripherals unprivileged
for (i = 0; i < SMU_NUM_PPU_PERIPHS; i++) {
    if (i & 0x01) {
        if (i >= 32){
            SMU->PPUPATD1 &= ~(1 << (i-32));
        } else {
            SMU->PPUPATD0 &= ~(1 << i);
        }
    }
}
```

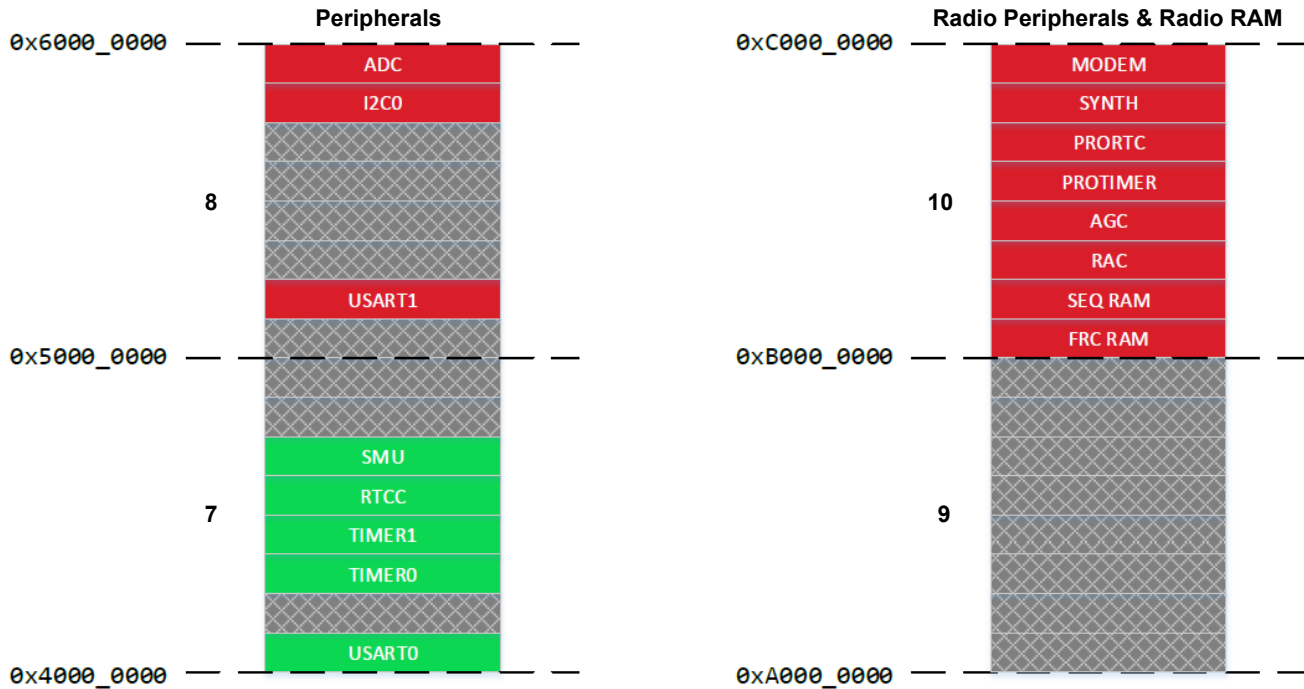
Note:

- The peripherals in SMU_PPUPATD0 and SMU_PPUPATD1 are device-dependent.
- The privileged attribute of the radio subsystem (AHBRADIO index) is configured as a unit.

4.5.2 Peripheral Security Attributes

A peripheral can be configured as either Secure (default) or Non-secure by programming the corresponding index in the SMU_PPUSATDn register. The figure below shows the memory map when the ADC, I2C0, USART1, and RADIO are configured as Non-secure and other peripherals (e.g., SMU, RTCC, TIMER1, TIMER0, USART0...) as Secure.

```
// Configure all the Non-secure peripherals
SMU->PPUSATD0 &= ~SMU_PPUSATD0_USART1;
SMU->PPUSATD1 &= ~(SMU_PPUSATD1_I2C0 | SMU_PPUSATD1_IADC0 | SMU_PPUSATD1_AHBRADIO);
```



- Note:**
- The peripherals in SMU_PPUSATD0 and SMU->PPUSATD1 are device-dependent.
 - The security attribute of the radio subsystem (AHBRADIO index) is configured as a unit.

4.5.3 Peripheral Fault Status

The peripheral ID that triggered the fault can be read from the SMU_PPUFS register.

```
uint32_t fs_ppu_periph_id;
uint32_t fs_sec_fault;
uint32_t fs_priv_fault;
uint32_t fs_inst_fault;

// Read peripheral fault status
fs_ppu_periph_id = SMU->PPUFS;
fs_sec_fault = (SMU->IF & _SMU_IF_PPUSEC_MASK) >> _SMU_IF_PPUSEC_SHIFT;
fs_priv_fault = (SMU->IF & _SMU_IF_PPUPRIV_MASK) >> _SMU_IF_PPUPRIV_SHIFT;
fs_inst_fault = (SMU->IF & _SMU_IF_PPUINST_MASK) >> _SMU_IF_PPUINST_SHIFT;

// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_PPUPRIV | SMU_IF_PPUINST;
```

4.6 Floating Point Unit (FPU) Programming

If the Non-secure application enables the FPU at initialization, the Secure software needs to set up the NSACR register in SCB to grant the FPU access for Non-secure software.

```
// Enable Non-secure access to the FPU
SCB->NSACR |= SCB_NSACR_CP10_Msk + SCB_NSACR_CP11_Msk;
```


5. TrustZone Implementation

The goal of TrustZone implementation is to provide Secure Key Storage that can keep access to keys limited to Secure applications while at the same time allowing Non-secure applications to exercise the keys. It is an added feature for the SVM devices that do not have dedicated hardware for [Secure Key Storage](#) as in SVH devices.

The [PSA Crypto](#) is placed in a Secure region to keep key material hidden from the Non-secure application. The exposed PSA Crypto APIs stay the same while the backend provides persistent key encryption and decryption similar to the key wrapping and unwrapping functionality of the SVH device.

The following items need to be considered when upgrading the existing system for Secure Key Storage with TrustZone.

- [System Configuration](#)
- [Gecko Bootloader](#)
- [Secure Library](#)
- [TrustZone Secure Key Storage](#)
- [PSA Attestation](#)
- [SE Manager](#)
- [Common Vulnerabilities and Exposures \(CVE\)](#)

5.1 System Configuration

The system configuration includes the following items:

- Enable system exceptions in the Secure state.
- Set the security attributes of different regions in the SAU and ESAU.
- Place peripherals and associated interrupts in either Secure or Non-secure applications.
- Assign the Bus Masters' security attributes.
- The system has two Secure/Non-secure pairs for the bootloader and application. The Secure part of each pair is responsible for properly configuring the split in its Secure application before branching to the Non-secure application.

Note: The secure application will issue a software reset at startup (fatal error) if the device's SE firmware version is lower than the [first version](#) that supports TrustZone.

5.1.1 System Exceptions

The following [system exceptions](#) are enabled in the Secure state for the bootloader and application.

- MemManage Fault
- BusFault
- UsageFault
- SecureFault

5.1.2 Main Flash Layout

The following figure is an overview of the main flash layout that covers the isolation requirements for the Secure Key Storage solution. The SAU and ESAU configurations provide the required security to the Cortex-M33 and other Bus Masters during boot and normal operation.

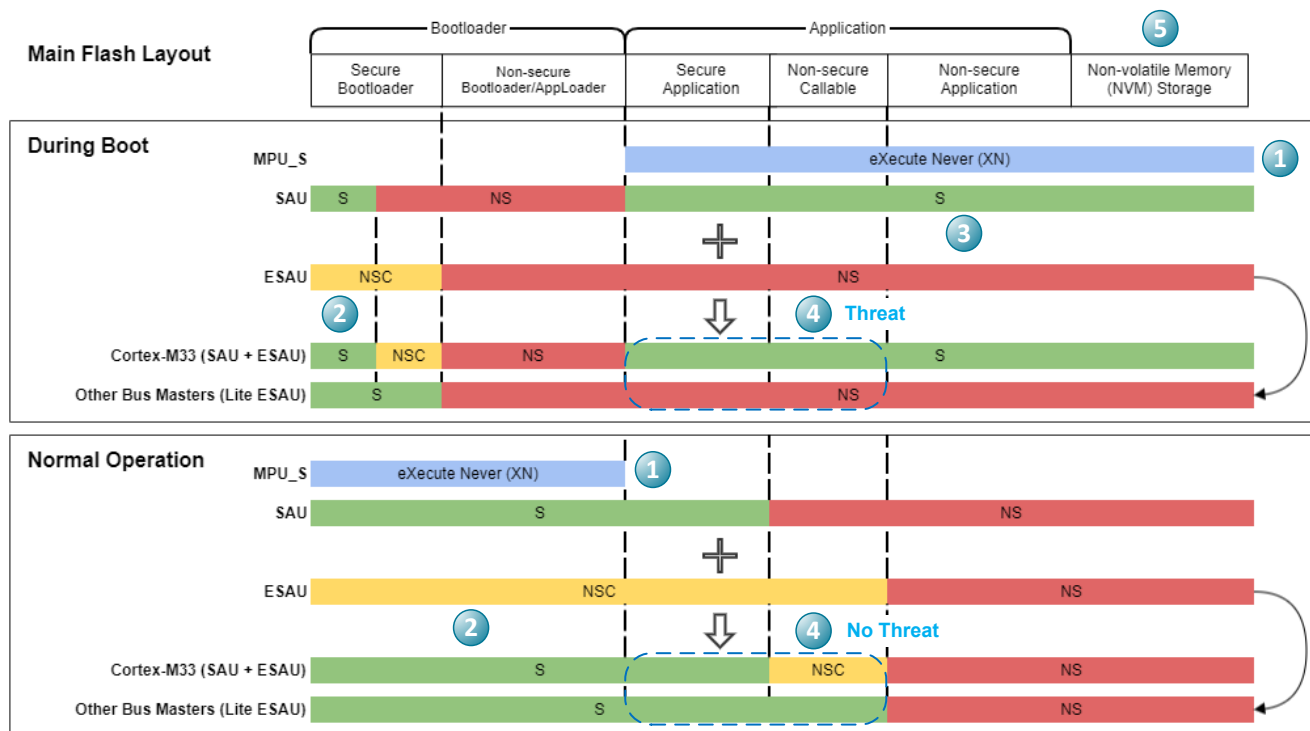


Figure 5.1. Main Flash Layout

1. Settings:

- The application is set to non-executable (XN) by Secure MPU to avoid any code execution in this area during boot.
 - The bootloader is set to non-executable (XN) by Secure MPU to avoid any code execution in this area during normal operation.
2. The ESAU configuration only uses the NSC section by setting mrb01 to the base address of region 0. The reason is that lite ESAU in other Bus Masters treats both S and NSC as a Secure attribute. For the Cortex-M33, the SAU upgrades the NSC in the ESAU to Secure. The 32 bytes region alignment of SAU also relaxes the 4 kB alignment restriction on the start address of the NSC in ESAU.
3. The whole application area is set to Secure in SAU for Cortex-M33 during boot to hide details from the bootloader NS part.
4. The ESAU cannot mark any region that comes after a Non-secure section as Secure (must be in the order of S/NSC/NS). Therefore the Secure application area does not align between the Cortex-M33 (SAU + ESAU) and other Bus Masters (lite ESAU) during boot. The secrets stored in that Secure region expose as Non-secure for other Bus Masters during boot (no such issue in normal operations). So the application must not save any plaintext secrets in that Secure region to overcome this limitation during boot.
5. The NVM storage is in the Non-secure region, so the application must encrypt the persistent keys before storing them in this area.

5.1.3 RAM Layout

The following figure is an overview of the RAM layout used for the bootloader and application. The SAU and ESAU are used to split the RAM into a Secure and Non-secure region (Non-secure Callable is not required).

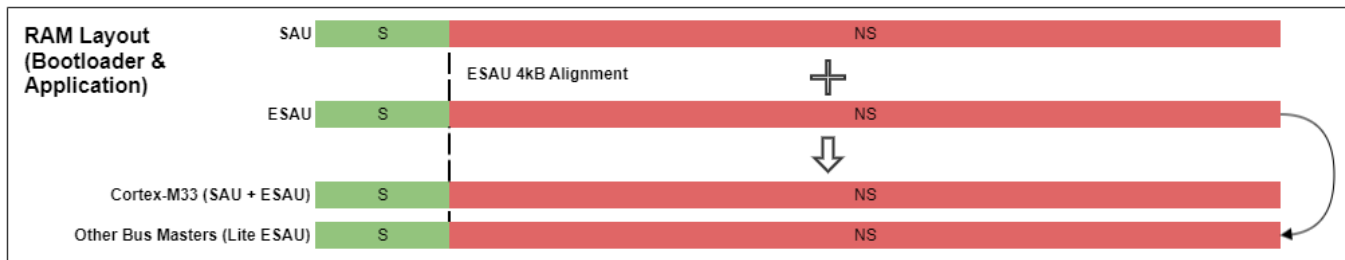


Figure 5.2. RAM Layout

In practice, the Secure part (bootloader or application) takes ownership of the amount of RAM it needs from the beginning of RAM and leaves the rest (up to the ESAU 4 kB alignment requirement) configured as Non-secure.

The bootloader does not know how the application partitions the RAM between Secure and Non-secure. So bootloader removes any secrets from RAM before handing control to the application.

5.1.4 Info Flash and EPPB

The following figure is an overview of the Info flash and EPPB layout for the application. The Cortex-M33 core is the only Bus Master that can access the EPPB region.

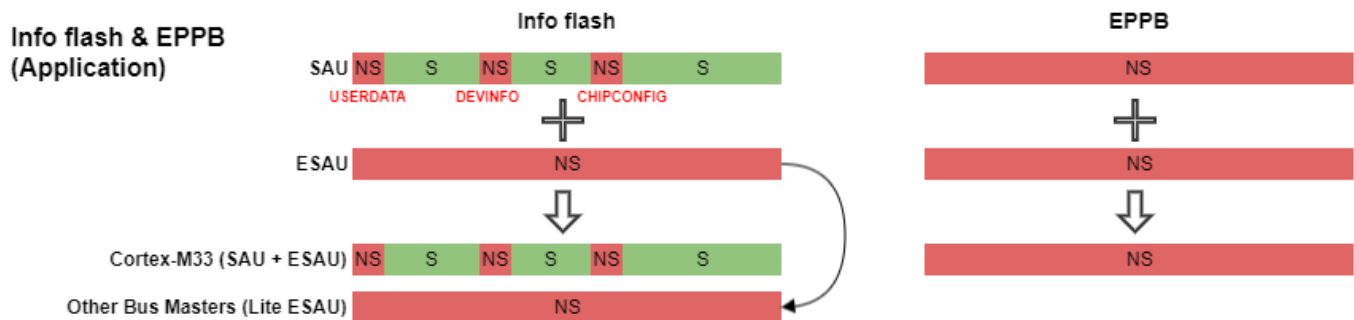


Figure 5.3. Info Flash and EPPB Layout

5.1.5 Peripheral and Device

The following figure is an overview of the peripheral and device layout for the bootloader and application. The SAU and ESAU are used to split the peripheral and device into a Secure and Non-secure region.

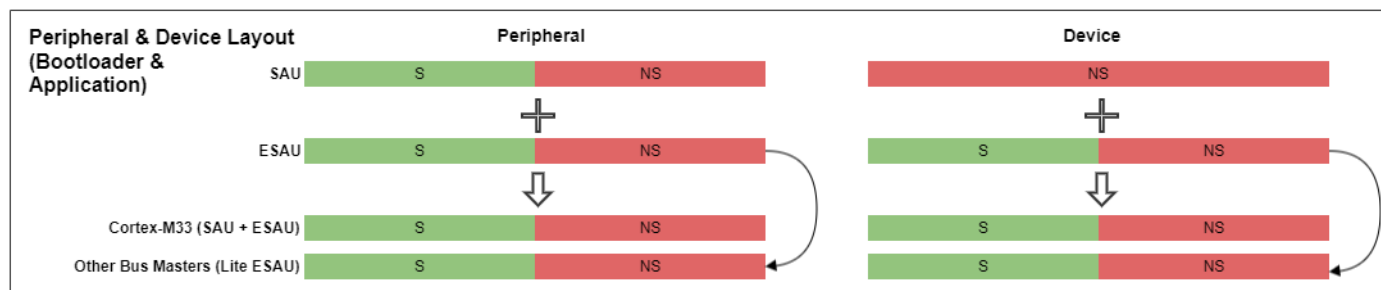


Figure 5.4. Peripheral and Device Layout

The Secure software is responsible for moving all peripherals and associated interrupts to the Non-secure state at startup, except for the peripherals and interrupts that need to be Secure. The Non-secure software must not include code that attempts to directly access any peripheral that is used by the Secure software.

Peripherals owned by the Secure software (application):

1. Security Management Unit (SMU)
 - It prevents Non-secure software from changing the configuration for the ESAUs, BMPUs, and PPU.
 - Except for EFR32xG21 devices, some features are also available in the dedicated [Non-secure version of SMU registers \(SMU_CFGNS\)](#).
2. CRYPTOACC (VSE devices) or SEMAILBOX (HSE devices)
 - The crypto engine is placed in the Secure domain for [Secure library](#).
3. System Configuration (SYSCFG)
 - It prevents Non-secure software from changing system configurations for Secure software.
 - Except for EFR32xG21 devices, some features are also available in the dedicated [Non-secure version of SYSCFG registers \(SYSCFG_CFGNS\)](#).
4. Memory System Controller (MSC)
 - It prevents Non-secure software from writing to Secure flash.

Peripheral interrupts owned by the Secure software:

Table 5.1. Secure Peripheral Interrupts (Application)

VSE Device	HSE Device
SMU_SECURE_IRQn	SMU_SECURE_IRQn
SYSCFG_IRQn	SYSCFG_IRQn
MSC_IRQn	MSC_IRQn
CRYTOACC_IRQn	SEMBRX_IRQn
TRNG_IRQn	SEMBTX_IRQn
PKE_IRQn	

The `PRIS` bit in the `AIRCR` register is set to 1 to place all Non-secure exceptions/interrupts in [lower priority level space](#). Therefore any Secure exceptions/interrupts can be programmed with higher priority than Non-secure ones.

The `BMPUSEC` and `PPUSEC` interrupt enable flags in the `SMU_IEN` register are set to enable the SMU security fault interrupts (`SMU_SECURE_IRQn`) on Bus Masters and peripherals.

Floating Point Unit (FPU):

The Secure application does not use the FPU. But the Secure startup code also enables the [FPU](#) for use by the Non-secure application.

5.1.6 Bus Masters

To keep all secrets from the Non-secure world, only the Bus Masters in the table below can access data in the Secure world. For the Bus Masters living in the Secure world, the secure application must configure their corresponding control interfaces in the peripheral space to Secure. The Cortex-M33 core as a Bus Master is split to run in Secure and Non-secure contexts.

Table 5.2. Secure Bus Masters (Application)

Device	Secure Bus Master	Control Interface of Bus Master
VSE	CRYPTOACC	CRYPTOACC
HSE	SEDMA or SEEXTDMA	SEMAILBOX

Note:

- Use `SMU_BMPUSATD0` register to [configure](#) the security attribute of a Bus Master.
- Use `SMU_PPUSATDn` register to [configure](#) the control interface of Bus Master as a Secure peripheral.
- LDMA is set as a Non-secure Bus Master to make sure it cannot be used to copy out data from the Secure memory.

5.1.7 Application Transitions

The system contains two Secure/Non-secure pairs.

1. The [bootloader pair](#) has a Secure bootloader and a Non-secure bootloader containing the communication interfaces.
2. The [application pair](#) has a Secure application and a Non-secure application consisting of the wireless stacks (if applicable) and application layers.

As described in the preceding sections, the Secure part of these pairs is responsible for setting the security configurations of the system during startup. For the handover between Secure/Non-secure pairs, the software must restore the system so the Secure part of the other pair can execute and reconfigure the system.

The software must reconfigure the following items before transitioning to the next Secure/Non-secure pair:

- Restored all peripherals and interrupts to Secure
- Reset ESAU to default configuration (all configurable regions to Secure)
- Reset SAU to default configuration (Secure for everything)
- Reset MPU to default configuration (removes any XN)

5.2 Gecko Bootloader

The [Gecko bootloader](#) ensures the Secure assets are protected during the boot flow and normal operation.

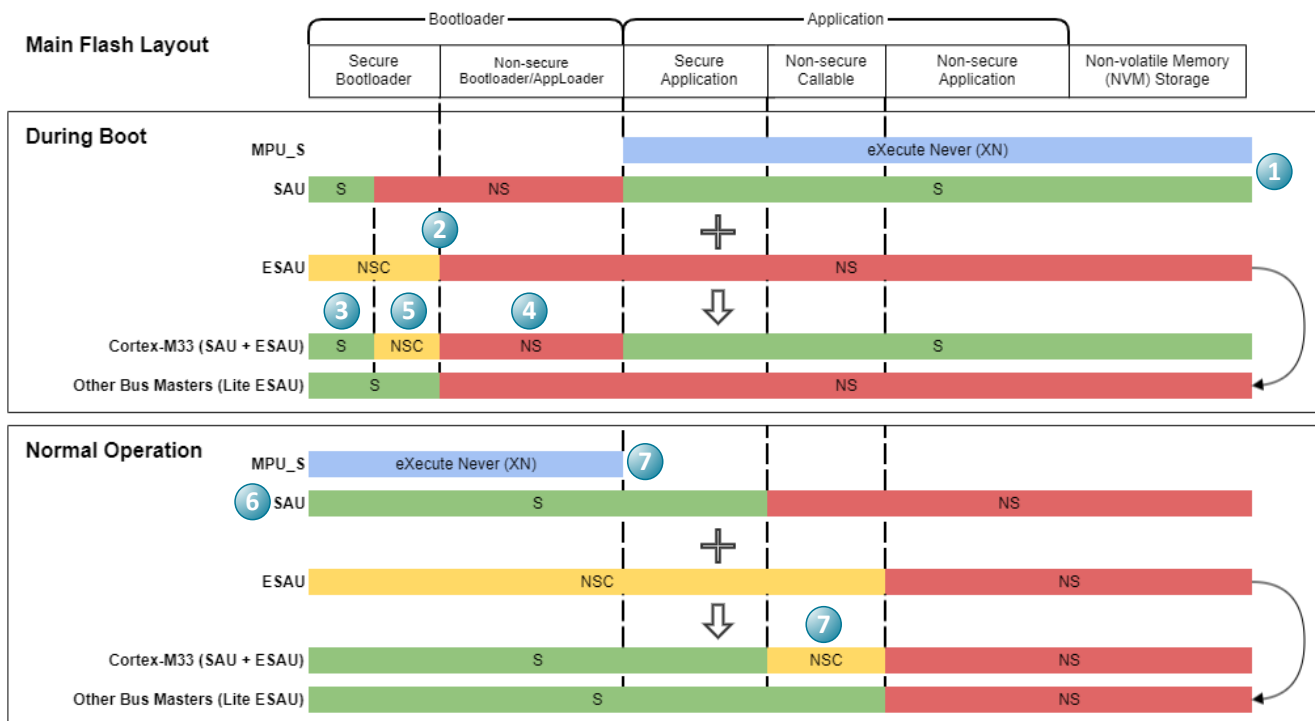


Figure 5.5. Gecko Bootloader Flow

1. The SAU and [Secure MPU](#) mark all the flash for application and NVM as Secure and non-executable (XN) during boot. It guards against bootloader NS code execution branching into the application code.
2. The bootloader needs to split into Secure and Non-secure software to protect secrets in the system. Secure code can access the entire flash to validate or upgrade the system.
3. For VSE devices, the [GBL Decryption Key \(AES-128 key\)](#) is moved from the NS memory (last page of the main flash) to the Secure part of the bootloader. The Simplicity Commander v1.13 or higher provides a feature to inject the AES-128 key to the bootloader binary file.

```
commander convert <BL image file> --aeskey <decryption key file> --outfile <BL image with decryption key>
```

4. The bootloader communication interfaces are placed in the NS area to support various [communication components](#) below for firmware upgrades.
 - BGAPI UART
 - EZSP-SPI
 - UART XMODEM
5. The NS communication functions call into the [bootloader APIs](#) placed in the bootloader NSC region. The Secure application [validates](#) all input arguments before processing the request.
6. Before transiting from bootloader to normal operation, it resets the SAU to default configuration to make all the flash for bootloader as Secure.
7. The Non-secure application software can call [bootloader APIs](#) through application NSC, and the corresponding Secure function releases the non-executable (XN) restriction on the bootloader during normal operation.

5.3 Secure Library

The goal of the Secure library is to keep the [PSA Crypto](#) key and [attestation token](#) protected from malicious code on the NSPE. The following figure overviews multiple components to support the Secure library.

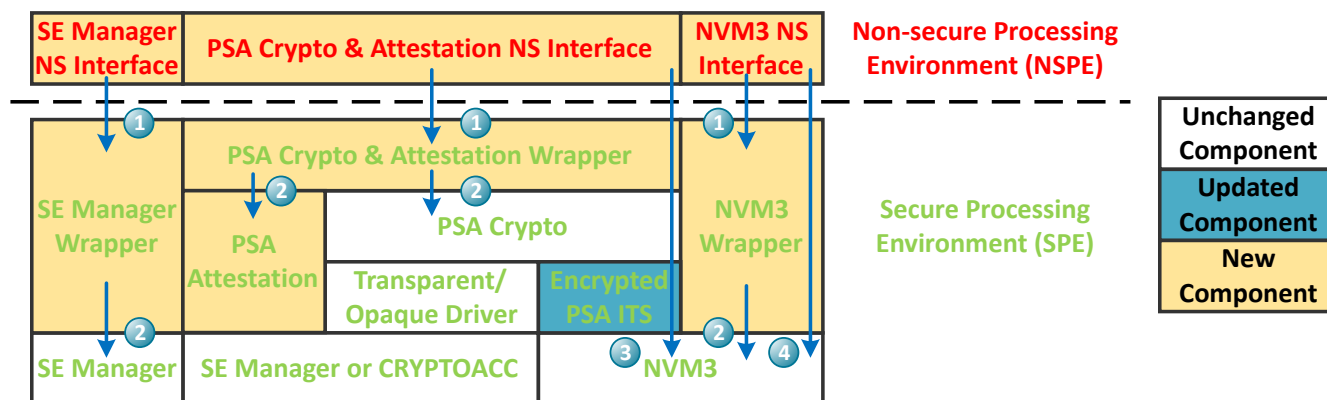


Figure 5.6. Secure Library Components

1. The NS interfaces in NSPE are responsible for packing and passing all input arguments over the NSC functions on wrappers in SPE.
2. The wrappers in SPE validate all input arguments before calling into the corresponding APIs in different drivers.
3. Because of the system memory layout limitation, the flash for NVM3 storage is located in the NSPE. Therefore the updated PSA Internal Trusted Storage (ITS) driver needs to encrypt all crypto keys before storing them in Non-secure NVM.
4. Data stored directly using the NVM3 APIs are not encrypted.

The following table describes the new and updated components of the Secure library.

Component	Description
SE Manager NS interface	This component contains SE Manager API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the SE Manager wrapper's NSC functions.
SE manager wrapper	This component contains the interface into SE Manager exposed to the NSPE. These NSC functions grant access to the SE Manager utility API and validate all input arguments before calling into SE Manager.
PSA Crypto & Attestation NS interface	This component contains PSA Crypto and attestation API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the PSA Crypto and attestation wrapper's NSC functions.
PSA Crypto & Attestation wrapper	This component contains the interface into PSA Crypto and attestation exposed to the NSPE. These NSC functions grant access to the entire PSA Crypto and attestation API and validate all input arguments before calling into PSA Crypto and attestation.
PSA attestation	This component in SPE provides the functionality required by the PSA attestation specification.
Encrypted PSA ITS	The PSA ITS layer builds on top of NVM3. This component is updated to support encrypted storage to secure stored keys. The encryption is based on the device's TrustZone Root Key.
NVM3 NS interface	This component contains NVM3 API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the NVM3 wrapper's NSC functions.
NVM3 wrapper	This component contains the interface into NVM3 exposed to the NSPE. These NSC functions grant access to the NVM3 API and validate all input arguments before calling into NVM3.

Note:

- The SE Manager NS interface, PSA Crypto NS interface, and NVM3 NS interface in the NSPE provide drop-in replacement on [SE Manager utility](#), [PSA Crypto](#), and [NVM3](#) APIs for existing wireless stacks and user applications.
- The NSC calls can only take a limited number of arguments, so all NSC functions take a pointer to a list of parameters to support a long list of arguments. All arguments must be validated using the [intrinsic functions](#) from CMSIS.

5.4 TrustZone Secure Key Storage

The TrustZone Secure Key Storage provides a solution to store a user key in Secure RAM or an encrypted form in Non-secure flash.

The TrustZone Root Key stored in the SE NVM for Secure Key Storage encryption is generated or renewed by following operations.

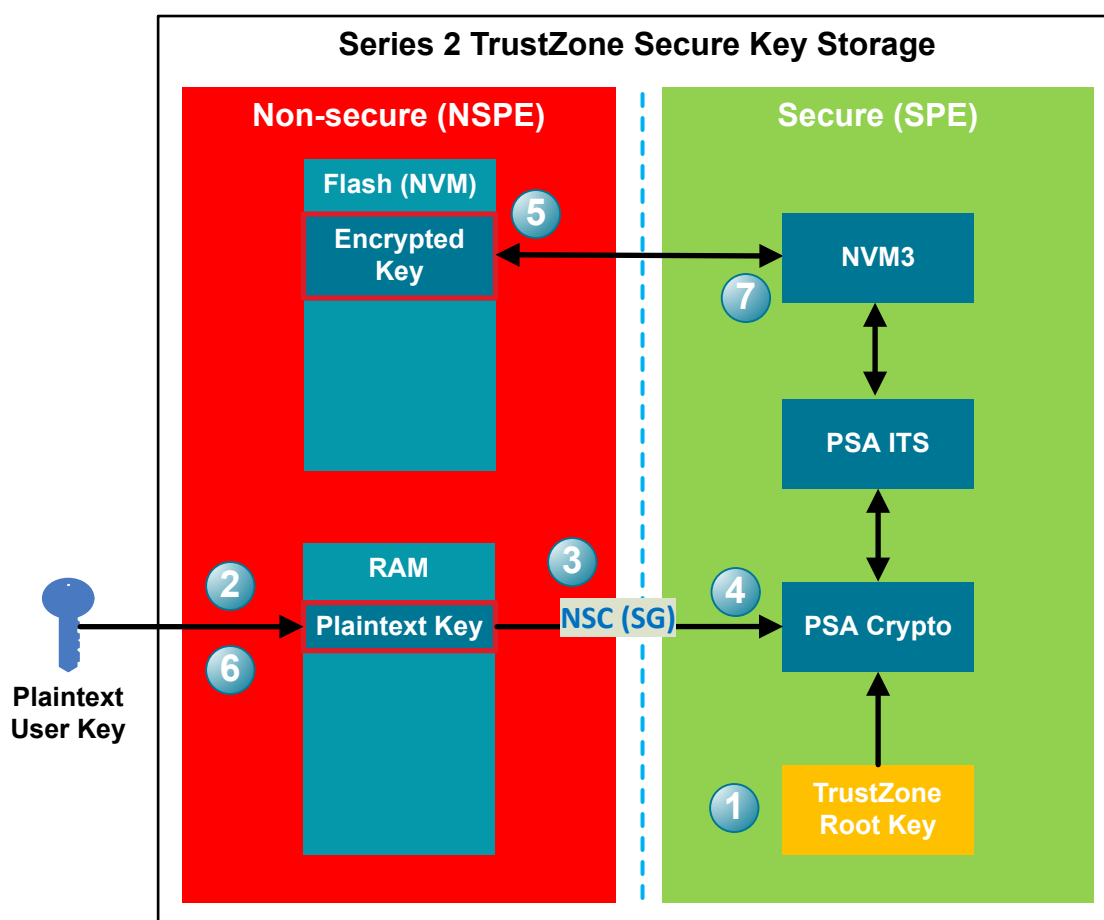
- The TrustZone Root Key had already existed if the shipped Series 2 device with [SE firmware version](#) supports this key.
- Generate a TrustZone Root Key when upgrading from a SE firmware version that did not support this key to the one that does.
- Renew a TrustZone Root Key after performing a [Device Erase](#).

Note: The TrustZone Root Key cannot be renewed if Device Erase is disabled.

The TrustZone Root Key is not exposable to the NSPE, and access to this key in SPE is different in HSE and VSE devices.

- HSE - The SPE can access the TrustZone Root Key as a built-in non-exportable key in HSE NVM.
- VSE - The SPE can access the TrustZone Root Key in Secure RAM, which is copied from VSE NVM during boot.

The TrustZone Root Key value after reset is identical to the value before reset. TrustZone Root Keys are unique on each device. The key allows a user to securely store a key in the Non-secure flash, limiting the number of keys that can be saved only by the amount of Non-secure storage. The following figure describes using the TrustZone Root Key to encrypt a plaintext key and store it in Non-secure NVM.



1. After power-on, the device's TrustZone Root Key is available for the SPE.
2. A user key is generated and imported into the device's Non-secure memory. In this example, the key is imported into Non-secure RAM for easy deletion, and the key is lost if device power is removed.
3. Call PSA Crypto API (`psa_import_key()` or `psa_generate_key()`) through SG in NSC to generate a key for crypto operations.
4. The plaintext key is passed to the PSA Crypto in SPE, where it is encrypted (AES-GCM) with the TrustZone Root Key.
5. The encrypted key is stored to the NVM in NSPE through the PSA ITS and NVM3 drivers.
6. The plaintext key can now be deleted from the Non-secure RAM.
7. Only the PSA Crypto in SPE can retrieve the encrypted key from NVM in NSPE and decrypt it for crypto operations in SPE.

Note: Ignore steps 2 and 6 if the plaintext key is randomly generated by the PSA Crypto.

The following tables describe the storage differences between SVM and SVH devices with and without TrustZone Secure Key Storage (SKS).

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (without TrustZone SKS)	RAM	RAM
Persistent Plaintext (without TrustZone SKS)	NVM	NVM
Volatile Wrapped (without TrustZone SKS)	Not supported	RAM (1)
Persistent Wrapped (without TrustZone SKS)	Not supported	NVM (1)

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (with TrustZone SKS)	Secure RAM (2)	Secure RAM
Persistent Plaintext (with TrustZone SKS)	Encrypted plaintext key in NS NVM (2)	Encrypted plaintext key in NS NVM
Volatile Wrapped (with TrustZone SKS)	Not supported	Secure RAM
Persistent Wrapped (with TrustZone SKS)	Not supported	Encrypted wrapped key in NS NVM

Note:

- The NVM or [NS NVM](#) is at the last part of the main flash.
- It is possible to replace the [wrapped key](#) solution on the SVH device (1) with TrustZone Secure Key Storage on the SVM device (2), but this is a less secure approach.

5.5 PSA Attestation

The device attestation service creates a token that contains a fixed set of device-specific data when requested by the caller. Each device must have a unique Initial Attestation Key (IAK) pair. The device uses the private IAK to sign the token, and the caller uses the public IAK to verify the token's authenticity.

The generation of the private IAK is different in SVM and SVH devices.

- SVM - If the private IAK does not exist in NVM3, it is randomly generated when requested from the [PSA Attestation](#) driver and saved to NVM3 through the [TrustZone Secure Key Storage](#).
- SVH - The private IAK is generated and securely stored in the HSE during chip production.

An Entity Attestation Token (EAT) is a mini-report that is cryptographically signed. An EAT is encoded in either one of two standardized data formats: a Concise Binary Object Representation (CBOR) or in the text-based format JSON. A digital signature is then used to protect its content. The technical specification defining the content of the EAT, which are claims about the hardware and the software running on a device, is specified by the Internet Engineering Task Force (IETF).

The EAT is a crypto-signed report card with claims. A claim is a data item that is represented as a Key-Value pair. Claims can relate to the device's pedigree or anything one wants the device to attest. Collected data can originate from the Root of Trust (RoT), any protected area, or non-protected areas.

The EAT must be signed following the structure of the CBOR Object Signing and Encryption (COSE) specification. For asymmetric key algorithms, the signature structure must be COSE-Sign1. A COSE-Sign1 is a CBOR encoded, self-secured data blob that contains headers, a payload, and a signature.

The primary need for EAT verification is to check correct formation and signing as for any token. In addition, though, the verifier can operate a policy where values of some of the claims in this profile can be compared to reference values that are registered with the verifier for a given deployment, to confirm that the device is endorsed by the manufacturer supply chain.

The [PSA attestation token](#) (aka Initial Attestation Token - IAT) is a profiled EAT. The Series 2 device will generate this token by request with a challenge (Nonce claim below) unless the SE OTP is uninitialized or the [SECURE_BOOT_ENABLE](#) option in SE OTP is disabled.

The following tables describe claims used in the PSA attestation token of the Series 2 device.

Table 5.3. Claims of PSA Attestation Token

Key	Claim Name (Present)	Claim Description	Claim Value
265 (-75000)	Profile Definition (Must)	The Profile Definition claim encodes the unique identifier corresponds to the EAT profile.	http://arm.com/psa/2.0.0
2394 (-75001)	Client ID (Must)	The Client ID claim represents the security domain of the caller.	See note below (2 bytes)
2395 (-75002)	Security Lifecycle (Must)	The Security Lifecycle claim represents the current lifecycle state of the PSA RoT.	Device dependent (2 bytes)
2396 (-75003)	Implementation ID (Must)	The Implementation ID claim uniquely identifies the implementation of the immutable PSA RoT.	Device dependent (32 bytes)
2397 (-75004)	Boot Seed (Optional)	The Boot Seed claim represents a value created at system boot time that will allow differentiation of reports from different boot sessions.	Device dependent (32 bytes)
2399 (-75006)	Software Components (Must)	The Software Components claim is a list of software components that includes all the software loaded by the PSA RoT.	See note below
10 (-75008)	Nonce (Must)	The Nonce claim is used to carry the challenge provided by the caller to demonstrate freshness of the generated token. The length must be either 32, 48, or 64 bytes.	Random nonce (32/48/64 bytes)
256 (-75009)	Instance ID (Must)	The Instance ID claim represents the unique identifier of the IAK. The length must be 33 bytes.	SHA-256 hash of public IAK (32 bytes) with header 0x01

Note:

- Some claims MUST be present in a PSA attestation token.
- The keys -7500x were defined in a previous version of the PSA attestation token specification ([PSA_IOT_PROFILE_1](#) profile) that is still used in the HSE-SVH firmware.
- The actual claims returned from the tokens on the SVH device are HSE firmware version-dependent.

- Key 2394: In PSA, a security domain is represented by a signed integer where negative values represent callers from the NSPE and positive IDs represent callers from the SPE. The value 0 is not permitted.
- Key 2395 (For the definitions of these lifecycle states, refer to the ARM [PSA Security Model](#)):
 - UNKNOWN (0x0000 - 0x00ff)
 - ASSEMBLY_AND_TEST (0x1000 - 0x10ff)
 - PSA_ROT_PROVISIONING (0x2000 - 0x20ff)
 - SECURED (0x3000 - 0x30ff)
 - NON_PSA_ROT_DEBUG (0x4000 - 0x40ff)
 - RECOVERABLE_PSA_ROT_DEBUG (0x5000 - 0x50ff)
 - DECOMMISSIONED (0x6000 - 0x60ff)
- Key 2396:
 - Word[0]: Die revision
 - Word[1]: SE OTP version (return 0 for VSE SE firmware < v1.2.14)
 - Word[2]: Security capability (not applicable to HSE-SVH device, always returns 1 in this word)
 - 0: Unknown security capability
 - 1: Security capability not applicable
 - 2: Basic security capability
 - 3: Root of Trust security capability
 - 4: HSE-SVM security capability
 - 5: HSE-SVH security capability (run HSE-SVM binary on HSE-SVH device)
 - Word[3]: Production version
 - Word[4:7]: Reserved (zeros)
- Key 2399: Each software component uses the attributes described in the following table, and some MUST be present in a software component claim.

Key	Attribute (Present)	Description	Value
1	Measurement Type (Optional)	The Measurement Type attribute is a short string representing the role of this software component.	See note below
2	Measurement Value (Must)	The Measurement Value attribute represents a hash of the invariant software component in memory at startup time.	SHA-256 hash (32 bytes) of the firmware
4	Version (Optional)	The Version attribute is the issued software version in the form of a text string.	A string of 8 bytes

The following measurement types may be used for Key 1:

- "BL": a Bootloader
- "PRoT": a component of the PSA Root of Trust
- "ARoT": a component of the Application Root of Trust
- "App": a component of the NSPE application
- "TS": a component of a Trusted Subsystem

The PSA Attestation API allows access to the PSA attestation token, so an external entity can cryptographically verify the identity and trust status of the device.

Table 5.4. PSA Attestation API

PSA Attestation API	Usage
psa_initial_attest_get_token	Retrieve the PSA attestation Token.
psa_initial_attest_get_token_size	Calculate the size of a PSA attestation Token.
sl_tz_attestation_get_public_key	Get the public IAk key for PSA attestation token signature verification.

Note: The `sl_tz_attestation_get_public_key` is a Silicon Labs custom API.

5.6 SE Manager

SE Manager is the foundation for the [Secure library](#) cryptographic operations on HSE devices. It means that SE Manager has to move into the SPE.

The following SE Manager [core](#) APIs are always available in the NSPE.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_init	Y	Y	Y
sl_se_deinit	Y	Y	Y
sl_se_init_command_context	Y	Y	Y
sl_se_deinit_command_context	Y	Y	Y
sl_se_set_yield	Y	Y	Y

The following SE Manager [core](#) APIs expose to the NSPE through the NSC interface for the VSE devices.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_read_executed_command	Y	—	—
sl_se_ack_command	Y	—	—

The following SE Manager [utility](#) APIs expose to the NSPE through the NSC interface for configuring the security features of HSE or VSE devices.

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_check_se_image	Y	Y	Y
sl_se_apply_se_image	Y	Y	Y
sl_se_get_upgrade_status_se_image	Y	Y	Y
sl_se_check_host_image	Y	Y	Y
sl_se_apply_host_image	Y	Y	Y
sl_se_get_upgrade_status_host_image	Y	Y	Y
sl_se_init_otp_key	Y	Y	Y
sl_se_read_pubkey	Y	Y	Y
sl_se_init_otp	Y	Y	Y
sl_se_read_otp	Y	Y	Y
sl_se_get_se_version	Y	Y	Y
sl_se_get_debug_lock_status	Y	Y	Y
sl_se_apply_debug_lock	Y	Y	Y
sl_se_get_otp_version	Y	Y	Y
sl_se_write_user_data	—	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_erase_user_data	—	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_reset_cause	—	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_status	—	Y	Y
sl_se_get_serialnumber	—	Y	Y
sl_se_enable_secure_debug	—	Y	Y
sl_se_disable_secure_debug	—	Y	Y

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_set_debug_options	—	Y	Y
sl_se_erase_device	—	Y	Y
sl_se_disable_device_erase	—	Y	Y
sl_se_get_challenge	—	Y	Y
sl_se_roll_challenge	—	Y	Y
sl_se_open_debug	—	Y	Y
sl_se_disable_tamper	—	—	Y
sl_se_read_cert_size	—	—	Y
sl_se_read_cert	—	—	Y

Note: The NSPE cannot access the other [SE Manager APIs](#) for cryptographic and attestation operations.

5.7 Common Vulnerabilities and Exposures (CVE)

At this writing, the following known TrustZone CVE had been fixed in the current implementation.

- [CVE-2020-16273](#): Stack sealing
- [CVE-2021-36465](#): VLLDM instruction/floating-point vulnerability

6. Upgrade Existing Application to TrustZone

The main concerns when upgrading existing deployment to the TrustZone solution are:

- The [Secure/Non-secure pair for the bootloader](#) (24 kB) does not fit inside the current allotted bootloader space (16 kB).
- The [Secure/Non-secure pair for the application](#) does not fit inside the current allotted application space.
- The [PSA ITS](#) moves from a non-encrypted to an encrypted format, so the existing stored cryptographic keys in NVM3 cannot be reused after upgrading the current application to TrustZone.

The [Secure Library](#) is based on PSA Crypto, so the existing application cannot integrate with the TrustZone if one of the following conditions is valid.

- Use [SE Manager APIs](#) for cryptographic and attestation operations.
- Use classic Mbed TLS APIs for cryptographic operations (except for X.509 certificate) and Transport Layer Security (TLS) protocol.

6.1 System Requirements

The following table lists the tools and software required for TrustZone development on Series 2 devices.

Tool/Software	Required Version	Description
GCC	v10.3.1	Fix a bug (ID 99271) on <code>cmse_nonsecure_call</code> attribute.
IAR EWARM	v9.20.4	Fix a bug (EWARM-9484) on <code>__cmse_nonsecure_call</code> attribute.
Segger J-Link	≥ v7.6.2c	v7.6.2c is the first version to add basic TrustZone support on Series 2 devices.
Simplicity Studio	≥ v5.6.3.0	v5.6.3.0 is the first version to support TrustZone software development on Series 2 devices.
Simplicity Commander	≥ v1.13.3	v1.13.3 includes a TrustZone-aware flash loader and supports features required for TrustZone development.
GSDK	≥ v4.2.2	GSDK v4.2.2 is the first version to support TrustZone software development on Series 2 devices.
SE Firmware	≥ v1.2.14	v1.2.14 is the first version to fully support TrustZone on xG21 (HSE) and xG22 (VSE) devices.
SE Firmware	≥ v2.2.1	v2.2.1 is the first version to fully support TrustZone on other Series 2 HSE and VSE devices.

Note:

- Required GCC and IAR EWARM versions are GSDK-dependent.
- [Bug list of GCC v10.3](#)
- [IAR EWARM release note](#)
- [Segger J-Link release note](#)
- [Simplicity Studio user guide](#)
- [Latest version of Simplicity Commander](#)
- [GSDK release note](#)
- Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required TrustZone features. The latest SE firmware image and release notes after installing the GSDK (Windows folder):

```
C:\Users\\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public
```

6.2 Peripheral Addresses in Device Header Files

The device header files (e.g., `efr32mg21b020f1024im32.h`) need to be configurable for different situations. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` definitions specify whether the compilation is for Secure or Non-secure applications. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` should be exclusive. If none of the definitions are true, the state should be similar to the Non-secure configuration, but the **startup code** (`SystemInit()` in `system_*.c`) will be responsible for reconfiguring the system.

Define (Software Component)	Default Peripheral Pointer	Startup Code
<code>SL_TRUSTZONE_SECURE</code> (TrustZone Secure)	Point to Secure peripherals (*_BASE = *_S_BASE)	No effect on SystemInit()
<code>SL_TRUSTZONE_NONSECURE</code> (TrustZone Non-Secure)	Point to Non-secure peripherals (*_BASE = *_NS_BASE)	No effect on SystemInit()
None of the above (—)	Point to Non-secure peripherals (*_BASE = *_NS_BASE)	SystemInit() moves peripherals to Non-secure

When building a Secure application (`SL_TRUSTZONE_SECURE` is true), all peripherals shall have their non-suffixed default address pointing to the Secure location of the peripheral (e.g., EMU). But the definitions in `sl_trustzone_secure_config.h` can force the addresses of specific peripherals pointing to the Non-secure location.

```
#ifndef SL_TRUSTZONE_SECURE_CONFIG_H
#define SL_TRUSTZONE_SECURE_CONFIG_H

// Specify security configuration of peripherals. Peripherals that are not
// included here will automatically have their _BASE addresses point to their
// secure address. This might not be true, since most peripherals are configured
// to be non-secure -- but it's also not a problem if the peripheral is not
// accessed from the S app.

// Used in multiple places.
#define SL_TRUSTZONE_PERIPHERAL_CMU_S (0)

// Used by SE Manager service.
#define SL_TRUSTZONE_PERIPHERAL_AHBRADIO_S (0)

// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMA_S (1)

// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMAXBAR_S (1)

#endif // SL_TRUSTZONE_SECURE_CONFIG_H
```

```
#if defined(SL_CATALOG_TRUSTZONE_SECURE_CONFIG_PRESENT)
#include "sl_trustzone_secure_config.h"
#endif

#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_EMU_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_EMU_S) && (SL_TRUSTZONE_PERIPHERAL_EMU_S != 0)))
#define EMU_BASE          (EMU_S_BASE)          /* EMU base address */
#else
```

In other cases (`SL_TRUSTZONE_NONSECURE` is true or both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), all peripherals shall have their non-suffixed default address pointing to the Non-secure location of the peripheral (e.g., EMU).

```
#define EMU_BASE          (EMU_NS_BASE)          /* EMU base address */
```

Note: Do not install the **TrustZone Secure** or **TrustZone Non-Secure** software component to the **TrustZone-unaware** application.

6.3 Startup Code

The startup code moves peripherals from Secure to Non-secure to support the [default peripheral locations](#). In a TrustZone-aware application (either `SL_TRUSTZONE_SECURE` or `SL_TRUSTZONE_NONSECURE` is true), this is the application's responsibility (skip lines 168 to 194 in `SystemInit()`) and is done in the [Secure firmware](#) of the system.

For the TrustZone-unaware application (both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), the `SystemInit()` in `system_*.c` (e.g., `system_efr32mg21.c`) moves peripherals to the Non-secure location.

- The `SystemInit()` sets the accesses of all peripherals to Non-secure except for the [SMU](#) and HSE SEMAILBOX (lines 172 to 178).
- The `SystemInit()` sets the SAU in [All Non-secure](#) configuration (lines 180 to 187).
 - It ensures Non-secure access to Non-secure peripherals.
 - The device component files (e.g., `efr32mg21b020f1024im32.slcc`) enable the [CMSE](#) compiler option (`-mcmse` for GCC and `--cmse` for IAR) to pass the condition in line 181 to program the SAU.
 - To catch the missing CMSE compiler option, it will generate a preprocessor error (line 186) if the CMSE flag is not set when manually upgrading a project from GSDK v4.0.x to \geq v4.1.x for the TrustZone-unaware application.
- The `SystemInit()` does not program the [ESAU](#) (default Secure flash is 32 MB), so the whole program is run in the **Secure** state.
- The `SystemInit()` also enables the [BMPUSEC](#) and [PPUSEC](#) interrupts in the SMU (lines 189 to 193). It ensures the TrustZone-unaware application catches any violations of Bus Master and peripheral security access permissions.

```

144 void SystemInit(void)
145 {
146 #if defined ( _VTOR_PRESENT) && ( _VTOR_PRESENT == 1U)
149
150 #if defined(UNALIGNED_SUPPORT_DISABLE)
153
154 #if ( _FPU_PRESENT == 1) && ( _FPU_USED == 1)
158
159 /* Secure app takes care of moving between the security states.
160 * SL_TRUSTZONE_SECURE MACRO is for secure access.
161 * SL_TRUSTZONE_NONSECURE MACRO is for non-secure access.
162 * When both the MACROS are not defined, during start-up below code makes sure
163 * that all the peripherals are accessed from non-secure address except SMU,
164 * as SMU is used to configure the trustzone state of the system. */
165 #if !defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_NONSECURE) \
166 && defined(__TZ_PRESENT)
167
168 #if ( _SILICON_LABS_32B_SERIES_2_CONFIG >= 2)
169 CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
170 #endif
171
172 /* config SMU to Secure and other peripherals to Non-Secure. */
173 SMU->PPUSATD0_CLR = _SMU_PPUSATD0_MASK;
174 #if defined (SEMAILBOX_PRESENT)
175 SMU->PPUSATD1_CLR = ( _SMU_PPUSATD1_MASK & (~SMU_PPUSATD1_SMU & ~SMU_PPUSATD1_SEMAILBOX));
176 #else
177 SMU->PPUSATD1_CLR = ( _SMU_PPUSATD1_MASK & ~SMU_PPUSATD1_SMU);
178 #endif
179
180 /* SAU treats all accesses as non-secure */
181 #if defined(__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
182 SAU->CTRL = SAU_CTRL_ALLNS_Msk;
183 __DSB();
184 __ISB();
185 #else
186 #error "The startup code requires access to the CMSE toolchain extension to set proper SAU settings."
187 #endif /* __ARM_FEATURE_CMSE */
188
189 /* Clear and Enable the SMU PPUSEC and BMPUSEC interrupt. */
190 NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
191 SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
192 NVIC_EnableIRQ(SMU_SECURE_IRQn);
193 SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
194 #endif /*SL_TRUSTZONE_SECURE */
195 }

```

The `SMU_BASE` and `HSE_SEMAILBOX_HOST_BASE` in device header files must point to the Secure location regardless of the `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` settings to avoid security violations on peripherals in the TrustZone-unaware application (SMU and HSE SEMAILBOX are set to Secure peripherals).


```

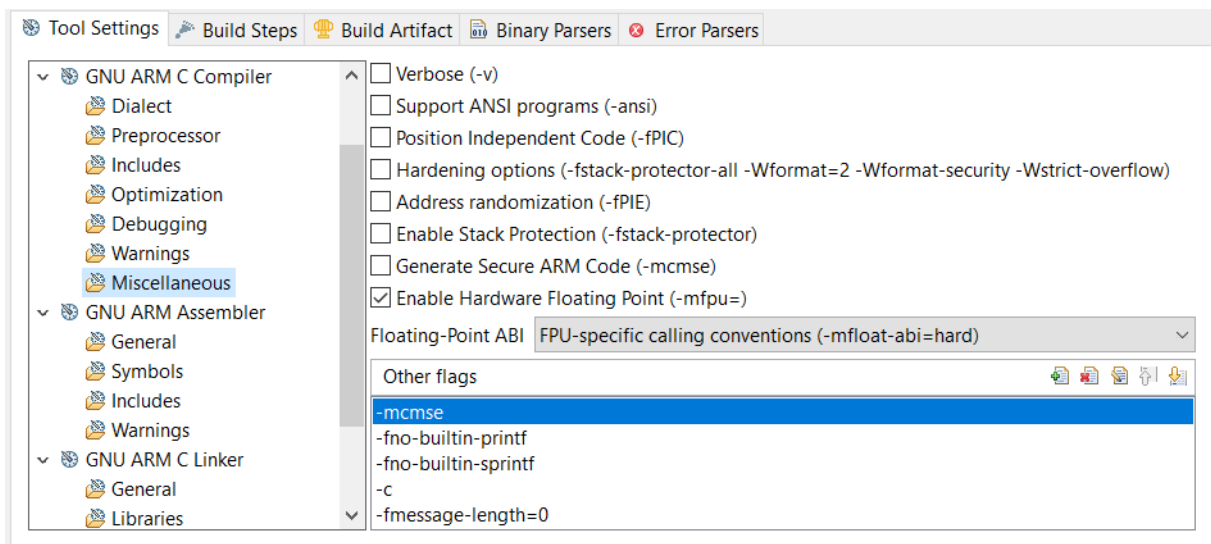
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SMU_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_SMU_S) && (SL_TRUSTZONE_PERIPHERAL_SMU_S != 0)))
#define SMU_BASE          (SMU_S_BASE)          /* SMU base address */
#else
#define SMU_BASE          (SMU_S_BASE)          /* SMU base address */

#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S) && (SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S != 0)))
#define SEMAILBOX_HOST_BASE (SEMAILBOX_S_HOST_BASE) /* SEMAILBOX_HOST base address */
#else
#define SEMAILBOX_HOST_BASE (SEMAILBOX_S_HOST_BASE) /* SEMAILBOX_HOST base address */

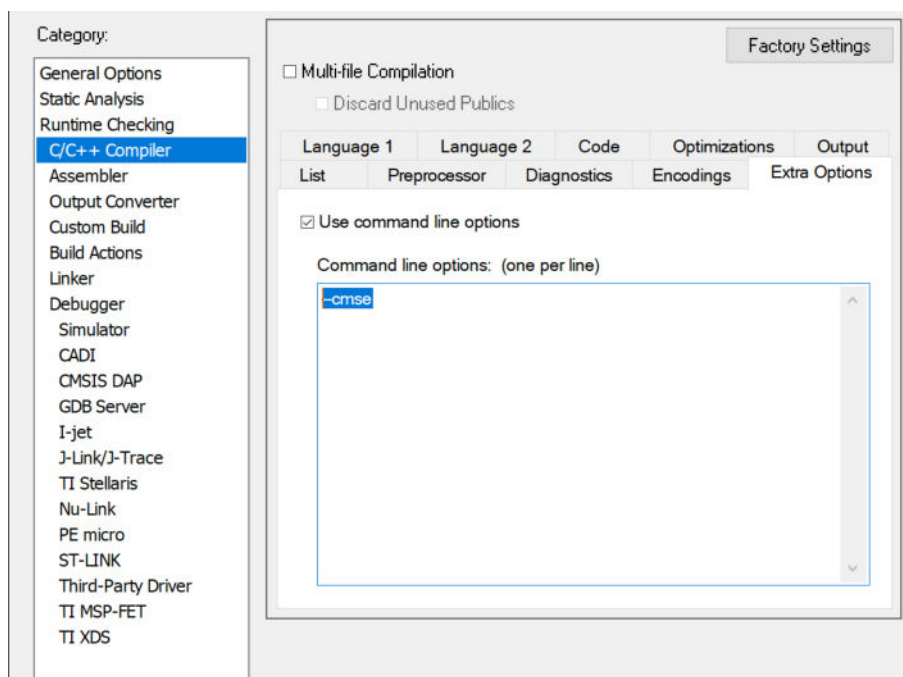
```

Note:

- The CMSE compiler option of GCC is in the Other flags window under C/C++ Build → Settings → Tool Settings → GNU ARM C Compiler → Miscellaneous.



- The CMSE compiler option of IAR is in the Command line options: (one per line) window under Options... → C/C++ Compiler → Extra Options.



6.4 Linker File

The `template_contribution` defined in the `s1cp` files of [Secure and Non-secure projects](#) will override the default memory settings defined in the device component files (e.g., `efr32mg21b020f1024im32.s1cc`) to generate the linker files for [Secure](#) and [Non-secure](#) applications.

Memory Region	Default Setting in Device Component File	Override Setting in <code>template_contribution</code>
Flash start address	<code>device_flash_addr</code>	<code>memory_flash_start</code>
Flash size	<code>device_flash_size</code>	<code>memory_flash_size</code>
RAM start address	<code>device_ram_addr</code>	<code>memory_ram_start</code>
RAM size	<code>device_ram_size</code>	<code>memory_ram_size</code>

Default setting (efr32mg21b020f1024im32.s1cc)

```
- name: device_family
  value: efr32mg21
- name: device_flash_addr
  value: 0
- name: device_flash_size
  value: 1048576
- name: device_flash_page_size
  value: 8192
- name: device_ram_addr
  value: 536870912
- name: device_ram_size
  value: 98304
```

template_contribution example (Secure)

```
template_contribution:
- name: memory_flash_start
  value: 0x0
- name: memory_flash_size
  value: 0x2C000
- name: memory_ram_start
  value: 0x20000000
- name: memory_ram_size
  value: 0x3000
```

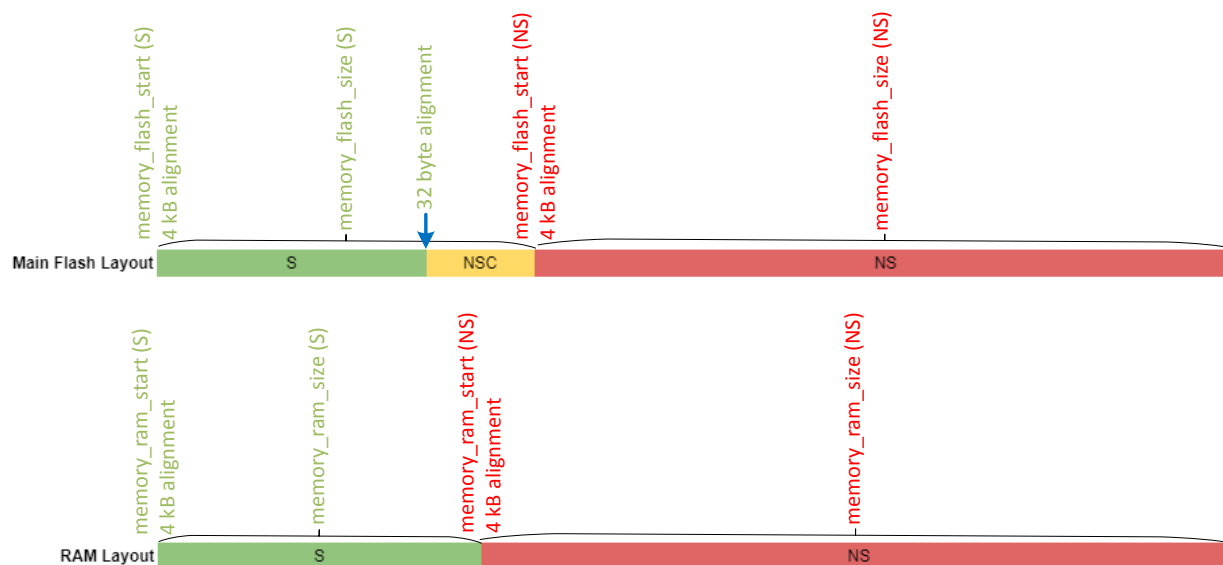
template_contribution example (Non-secure)

```
template_contribution:
- name: memory_flash_start
  value: 0x2C000
- name: memory_flash_size
  value: 0x54000
- name: memory_ram_start
  value: 0x20003000
- name: memory_ram_size
  value: 0x5000
```

The [ESAU](#) sets the flash and RAM start address, so these addresses should be alignment at **4 kB** (0x1000). The Secure project linker file needs to have a section for [NSC](#) (Secure Gateway) at the end of the Secure flash section. The [SAU](#) sets the start address of the NSC section, so this section only needs to be **32 bytes** aligned.

- GCC NSC: The `.gnu.sgstubs` region in the Secure application map file (`.map`)
- IAR NSC: The `Veneer$CMSE` region in the Secure application map file (`.map`)

The Secure and Non-secure flash and RAM sizes are incremented or decremented in **4 kB**. The memory configurations in Secure and Non-secure applications are correlated, so the flash and RAM settings are in pairs.



Note: Users should not directly edit the `template_contribution` in the `s1cp` file, but rather use the [Memory Editor](#) in Simplicity Studio to update the memory configuration.

6.5 Debugger

Simplicity Studio supports two debuggers:

- GNU Debugger (GDB) client and SEGGER's GDB server
- Simplicity Studio Debugger

The [TrustZone-unaware](#) and [TrustZone-aware](#) applications enable the `PPUSEC` interrupts in the SMU. The debugger will trigger the `SMU_SECURE_IRQHandler` if the **[Registers]** or **[Peripherals]** view feature violates peripheral security access permission.

Simplicity Studio Debugger

The **[Registers]** view of Simplicity Studio Debugger can only access the Secure location of a peripheral. The following figure demonstrates the `Default_Handler` (`SMU_SECURE_IRQHandler` not defined) is triggered (`PPUSEC` in `SMU->IF = 1`) when viewing the registers of GPIO peripheral (`PPUFSPERIPHID = 13`) that is set to Non-secure access in the SMU.

The debugger can access the registers of the SMU since this peripheral is set to Secure access in the SMU.

This limitation does not apply to **GSDK < v4.1.0** since no peripherals are configured for Non-secure access.

The screenshot shows the Simplicity Studio Debugger interface. On the left, the source code for `startup_efr32mg21.c` is displayed, with the `Default_Handler` function highlighted. On the right, the **Registers** window is open, showing a list of SMU registers. The `PPUFSPERIPHID` register is selected, showing its value as `0xD` and its description as `Peripheral I`. Below the register list, the details for `PPUFSPERIPHID` are shown: Name: PPUFSPERIPHID, Hex: 0xD, Decimal: 13, Octal: 015, Binary: 0b1101, Default: 0xD.

The Simplicity Studio Debugger is not the preferred choice for TrustZone debugging since it has limitations on viewing Non-secure access peripherals.

GNU Debugger (GDB)

The [Peripherals] view of GNU Debugger can access either the Secure or Non-secure location of the peripheral to avoid conflicts on security access permission. The following figure shows the registers of GPIO on Secure (GPIO at 0x4003C000) and Non-secure (GPIO_NS at 0x5003C000) addresses. The GPIO peripheral is set to Non-secure access in the SMU, so the registers in the Secure address are displayed as zero.

The figure consists of two side-by-side screenshots of the GNU Debugger's Peripherals view. Each screenshot shows a list of peripherals and a detailed view of the selected peripheral's registers.

Left Screenshot: GPIO (Secure Address)

Peripheral	Address	Description
CMU_NS	0x50008000	CMU Registers
DPLL0	0x4001C000	DPLL Registers
DPLL0_NS	0x5001C000	DPLL Registers
EMU	0x40004000	EMU Registers
EMU_NS	0x50004000	EMU Registers
FRC	0xA8004000	FRC Registers
FRC_NS	0xB8004000	FRC Registers
FSRCO	0x40018000	FSRCO Registers
FSRCO_NS	0x50018000	FSRCO Registers
GPCRC	0x40088000	GPCRC Registers
GPCRC_NS	0x50088000	GPCRC Registers
GPIO	0x4003C000	GPIO Registers
GPIO_NS	0x5003C000	GPIO Registers
HFRCO0	0x40010000	HFRCO Registers
HFRCO0_NS	0x50010000	HFRCO Registers
HFRCOEM23	0x4A014000	HFRCO Registers
HFRCOEM23_NS	0x5A014000	HFRCO Registers
HFXO0	0x4000C000	SYXO Registers
HFXO0_NS	0x5000C000	SYXO Registers
I2C0	0x4A010000	I2C Registers
I2C0_NS	0x5A010000	I2C Registers

Register	Address	Value
GPIO	0x4003C000	
PORTA_CTRL	0x4003C000	0x00000000
SLEWRATE	[6:4]	0x0
DINDIS	[12]	0x0
SLEWRATEALT	[22:20]	0x0
DINDISALT	[28]	0x0
PORTA_MODEL	0x4003C004	0x00000000
PORTA_DOUT	0x4003C010	0x00000000
PORTA_DIN	0x4003C014	0x00000000
PORTB_CTRL	0x4003C030	0x00000000
PORTB_MODEL	0x4003C034	0x00000000
PORTB_DOUT	0x4003C040	0x00000000
PORTB_DIN	0x4003C044	0x00000000
PORTC_CTRL	0x4003C060	0x00000000
PORTC_MODEL	0x4003C064	0x00000000
PORTC_DOUT	0x4003C070	0x00000000
PORTC_DIN	0x4003C074	0x00000000

Right Screenshot: GPIO_NS (Non-secure Address)

Peripheral	Address	Description
CMU_NS	0x50008000	CMU Registers
DPLL0	0x4001C000	DPLL Registers
DPLL0_NS	0x5001C000	DPLL Registers
EMU	0x40004000	EMU Registers
EMU_NS	0x50004000	EMU Registers
FRC	0xA8004000	FRC Registers
FRC_NS	0xB8004000	FRC Registers
FSRCO	0x40018000	FSRCO Registers
FSRCO_NS	0x50018000	FSRCO Registers
GPCRC	0x40088000	GPCRC Registers
GPCRC_NS	0x50088000	GPCRC Registers
GPIO	0x4003C000	GPIO Registers
GPIO_NS	0x5003C000	GPIO Registers
HFRCO0	0x40010000	HFRCO Registers
HFRCO0_NS	0x50010000	HFRCO Registers
HFRCOEM23	0x4A014000	HFRCO Registers
HFRCOEM23_NS	0x5A014000	HFRCO Registers
HFXO0	0x4000C000	SYXO Registers
HFXO0_NS	0x5000C000	SYXO Registers
I2C0	0x4A010000	I2C Registers
I2C0_NS	0x5A010000	I2C Registers

Register	Address	Value
GPIO_NS	0x5003C000	
PORTA_CTRL	0x5003C000	0x00400040
SLEWRATE	[6:4]	0x4
DINDIS	[12]	0x0
SLEWRATEALT	[22:20]	0x4
DINDISALT	[28]	0x0
PORTA_MODEL	0x5003C004	0x00004000
PORTA_DOUT	0x5003C010	0x00000008
PORTA_DIN	0x5003C014	0x00000008
PORTB_CTRL	0x5003C030	0x00400040
PORTB_MODEL	0x5003C034	0x00000000
PORTB_DOUT	0x5003C040	0x00000000
PORTB_DIN	0x5003C044	0x00000000
PORTC_CTRL	0x5003C060	0x00400040
PORTC_MODEL	0x5003C064	0x00000000
PORTC_DOUT	0x5003C070	0x00000000
PORTC_DIN	0x5003C074	0x00000000

The GNU Debugger is the preferred choice for TrustZone debugging and is the default debugger for Simplicity Studio \geq v5.5.0.0.

7. TrustZone Platform Examples

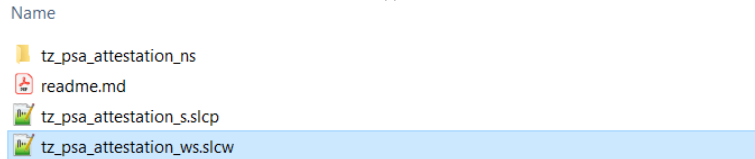
The following TrustZone platform examples located in the `C:\Users\\SimplicityStudio\SDKs\gecko_sdk\app\common\example` folder (Windows) demonstrate the TrustZone implementation on Series 2 devices. All TrustZone platform examples do not include [Gecko Bootloader](#).

TrustZone PSA Attestation

tz_psa_attestation_ws
This example workspace demonstrates TrustZone for PSA Attestation.

[View Project Documentation](#) CREATE

SimplicityStudio > SDKs > gecko_sdk > app > common > example > tz_psa_attestation



Example Folder	Description
tz_psa_attestation	The workspace description file (tz_psa_attestation_ws.slc) creates the TrustZone PSA Attestation example. The project description file (tz_psa_attestation_s.slc) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_attestation_ns	The project description file (tz_psa_attestation_ns.slc) configures a Non-secure application for the TrustZone PSA Attestation example.

Note:

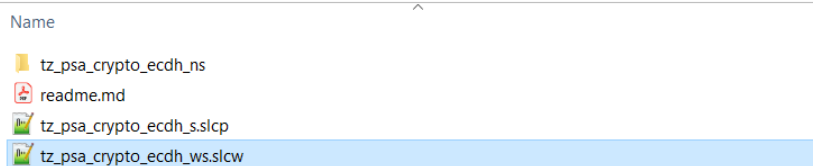
- This example cannot run if the `SECURE_BOOT_ENABLE` (root of trust of the attestation) option in SE OTP is disabled.
- The combined image of Secure and Non-secure applications is signed by the `example_signing_key.pem` (private key) in `C:\Users\\SimplicityStudio\SDKs\gecko_sdk\platform\common` folder (Windows). The `example_signing_pubkey.pem` (public key) in the same folder is installed to the SE OTP to verify the image signature during [Secure Boot](#).

TrustZone PSA Crypto ECDH

tz_psa_crypto_ecdh_ws
This example workspace demonstrates TrustZone for ECDH key agreement.

[View Project Documentation](#) CREATE

SimplicityStudio > SDKs > gecko_sdk > app > common > example > tz_psa_crypto_ecdh >



Example Folder	Description
tz_psa_crypto_ecdh	The workspace description file (tz_psa_crypto_ecdh_ws.slc) upgrades the existing Platform - PSA Crypto ECDH example to TrustZone-aware. The project description file (tz_psa_crypto_ecdh_s.slc) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_crypto_ecdh_ns	The project description file (tz_psa_crypto_ecdh_ns.slc) configures the existing Platform - PSA Crypto ECDH example as a Non-secure application. The source code can be reused without changes.

The following sections use Simplicity Studio v5.6.3.0 and GSDK v4.2.2. The procedures and pictures may be different if using higher versions of Simplicity Studio 5 and GSDK.

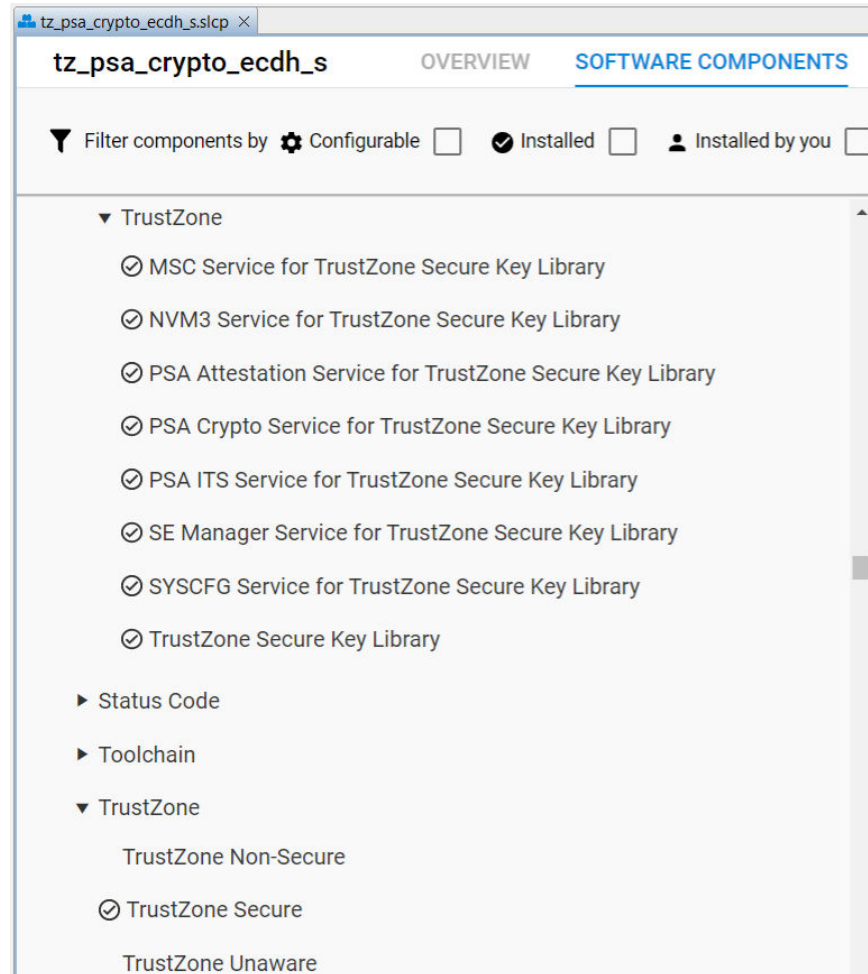
7.1 Project Description File

The project description file (`.slcp`) contains references to the GSDK used and a list of components to use from these. The TrustZone-aware application requires separate `slcp` files for the Secure and Non-secure applications.

Users should not directly edit the `slcp` files, but rather use the [Memory Editor](#) and Post Build Editor in Simplicity Studio to update the [memory configuration](#) and post-build actions.

7.1.1 Secure Application

The following figure describes which TrustZone software components are installed for the TrustZone Secure library of the [TrustZone PSA Crypto ECDH](#) example.

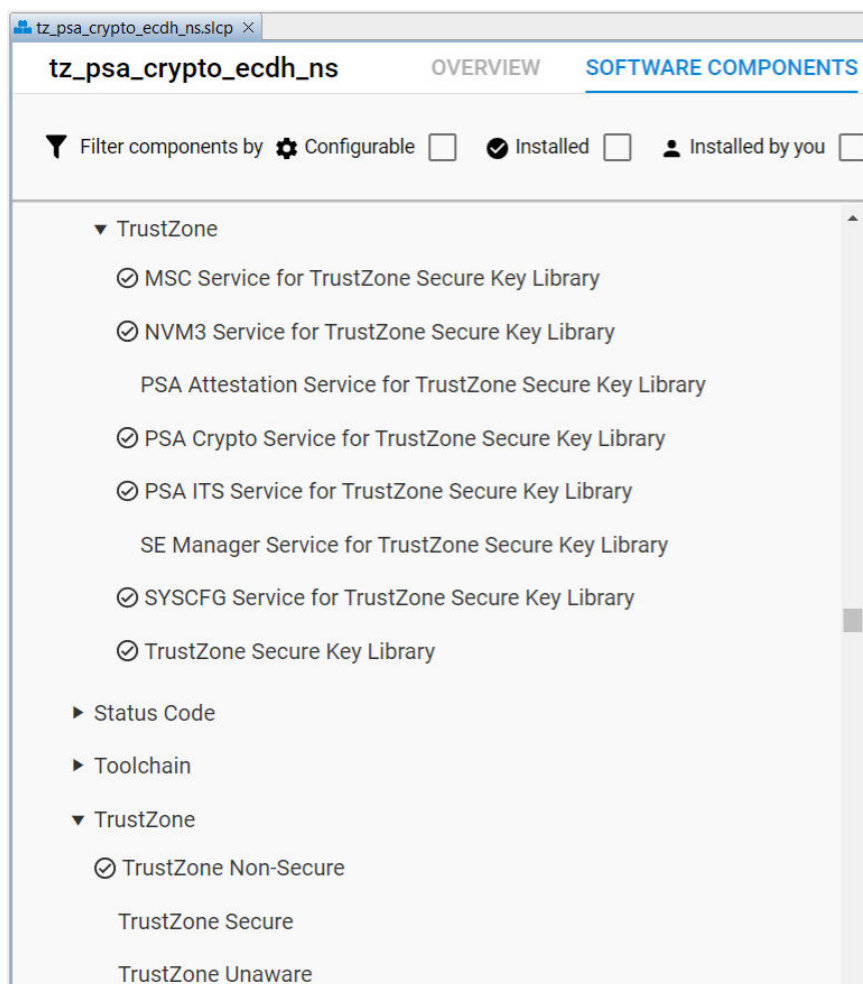


Note:

- The services provided by the Secure library are standardized.
- The source files for the Secure library will be automatically added to the application when generating the Secure project from the `slcp` file. For the current TrustZone implementation, modifications of the source files of the Secure library are not recommended.

7.1.2 Non-secure Application

The following figure describes which TrustZone software components are installed for the Non-secure application of the [TrustZone PSA Crypto ECDH](#) example.



Note:

- The following software components are automatically installed when [PSA Crypto and ITS](#) services are used on the Non-secure application.
 - MSC Service for TrustZone Secure Key Library
 - NVM3 Service for TrustZone Secure Key Library
 - PSA Crypto Service for TrustZone Secure Key Library
 - PSA ITS Service for TrustZone Secure Key Library
 - SYSCFG Service for TrustZone Secure Key Library
- The following software components can be installed to the Non-secure application when those services are required.
 - [PSA Attestation Service for TrustZone Secure Key Library](#)
 - [SE Manager Service for TrustZone Secure Key Library](#)

7.2 Workspace

A workspace is a structure that can contain multiple projects. 'Workspace' is a generic term for this construct. In the context of Simplicity Studio, where workspace has a different, Eclipse-based, meaning, workspaces are referred to as [Solutions](#).

The workspace description file (`.slcw`) contains references to projects (`.slcp`) that make up the workspace. Users should not directly edit the `slcw` file, but rather use the Post Build Editor in Simplicity Studio to update the post-build actions.

7.3 Memory Configuration

The [memory configurations](#) in the TrustZone platform examples are based on the Series 2 radio board with minimum flash (512 kB) and RAM (32 kB), so these configurations can run on all Series 2 radio boards. Users can customize the settings when more flash and RAM are available on the selected device.

- Memory flash size (total) = `memory_flash_size (S)` + `memory_flash_size (NS)` = 512 kB
- Memory RAM size (total) = `memory_ram_size (S)` + `memory_ram_size (NS)` = 32 kB

7.3.1 Secure Application

The project description file of the Secure application (`*_s.s1cp`) uses the default memory setting below to generate the [Secure linker file](#) (`linkerfile.ld` for GCC and `linkerfile.icf` for IAR in the project `autogen` folder).

The actual memory usage during software development is unknown, so it needs to reserve enough flash (`memory_flash_size`: 176 kB) and RAM (`memory_ram_size`: 12 kB) for the Secure part of all TrustZone platform examples. The bigger RAM size (including stack and heap) is mainly for the software fallback on cryptographic operations in PSA Crypto.

Default Memory Setting (Secure)	xG21 and xG22 Devices	Other Series 2 Devices
<code>memory_flash_start</code>	0x00000000	0x08000000
<code>memory_flash_size</code>	0x0002C000 (176 kB)	0x0002C000 (176 kB)
<code>memory_ram_start</code>	0x20000000	0x20000000
<code>memory_ram_size</code>	0x00003000 (12 kB)	0x00003000 (12 kB)

```
MEMORY
{
  FLASH   (rx)  : ORIGIN = 0x0, LENGTH = 0x2c000
  RAM     (rwx) : ORIGIN = 0x20000000, LENGTH = 0x3000
}
```

7.3.2 Non-secure Application

The project description files of the Non-secure application (`*_ns.s1cp`) use the default memory setting below to generate the [Non-secure linker file](#) (`linkerfile.ld` for GCC and `linkerfile.icf` for IAR in the project `autogen` folder).

The actual memory usage during software development is unknown, so the remaining flash (`memory_flash_size`: 336 kB) and RAM (`memory_ram_size`: 20 kB) should be big enough for the Non-secure part of all TrustZone platform examples.

Default Memory Setting (Non-secure)	xG21 and xG22 Devices	Other Series 2 Devices
<code>memory_flash_start</code>	0x0002C000 (176 kB)	0x0802C000 (176 kB)
<code>memory_flash_size</code>	0x00054000 (336 kB)	0x00054000 (336 kB)
<code>memory_ram_start</code>	0x20003000 (12 kB)	0x20003000 (12 kB)
<code>memory_ram_size</code>	0x00005000 (20 kB)	0x00005000 (20 kB)

```
MEMORY
{
  FLASH   (rx)  : ORIGIN = 0x2c000, LENGTH = 0x54000
  RAM     (rwx) : ORIGIN = 0x20003000, LENGTH = 0x5000
}
```

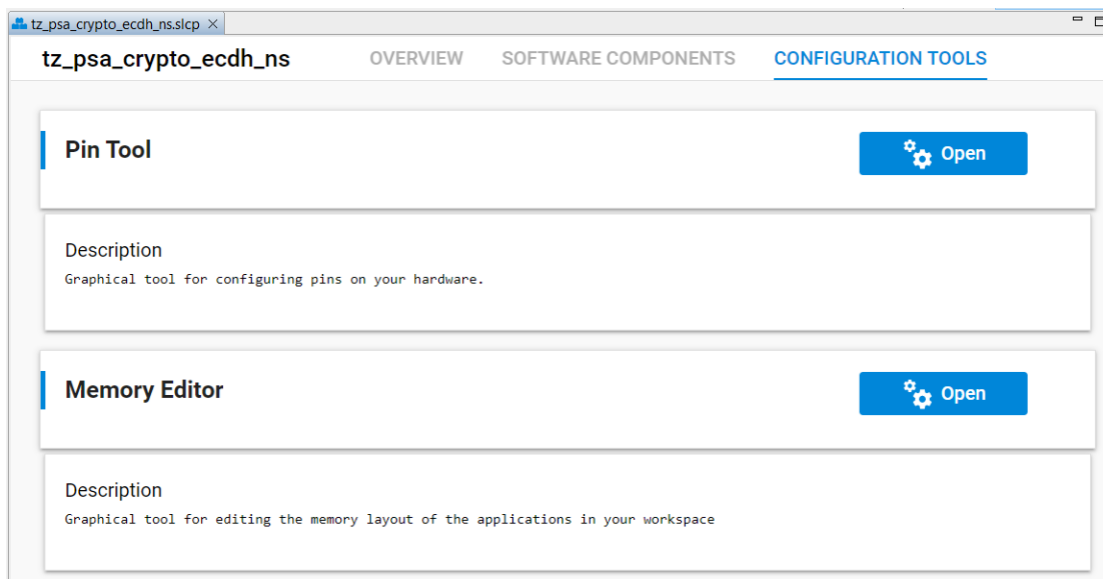
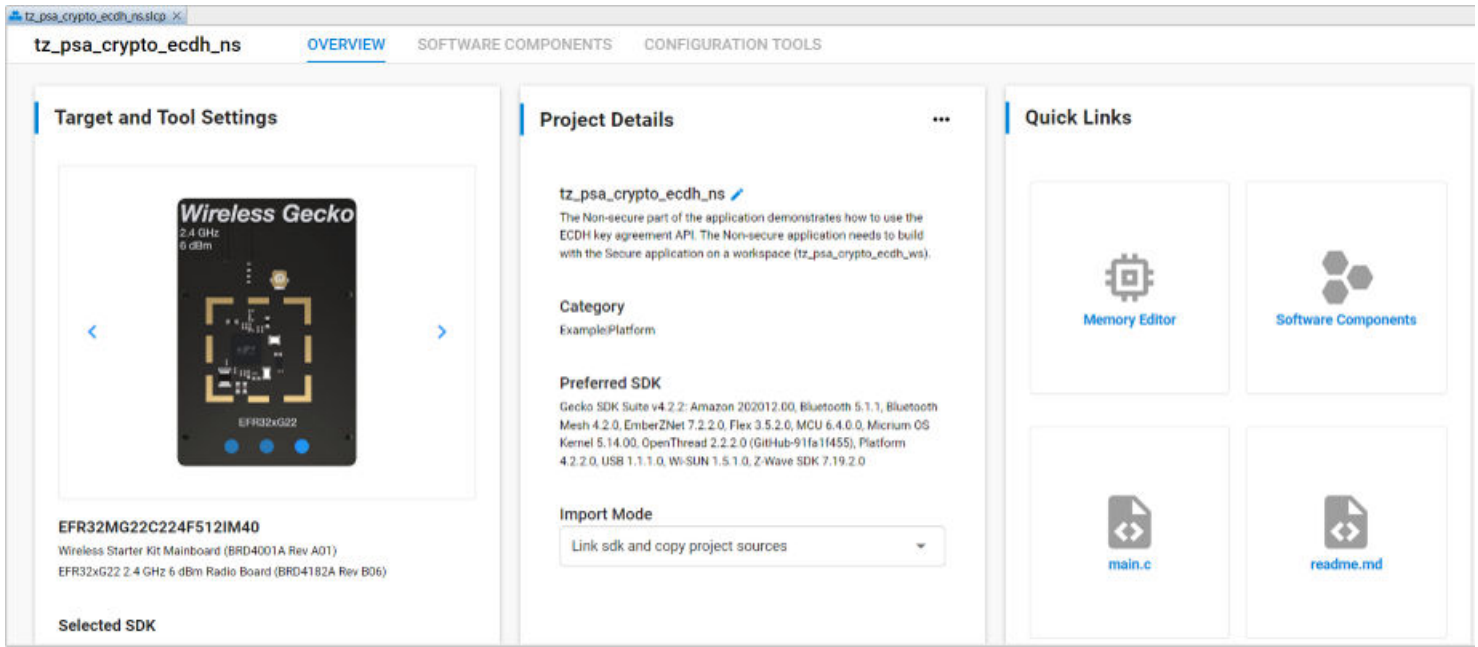
Note: The usable flash for Non-secure code should be equal to `memory_flash_size - NVM size` (default is 40 kB) if NVM3 storage is required.

7.3.3 Memory Editor

The default memory setting of **Secure** and **Non-secure** applications are good enough for software development and debugging. The final memory layouts of Secure and Non-secure projects are deduced by inspecting the flash and RAM usage in the Secure application memory map file (.map).

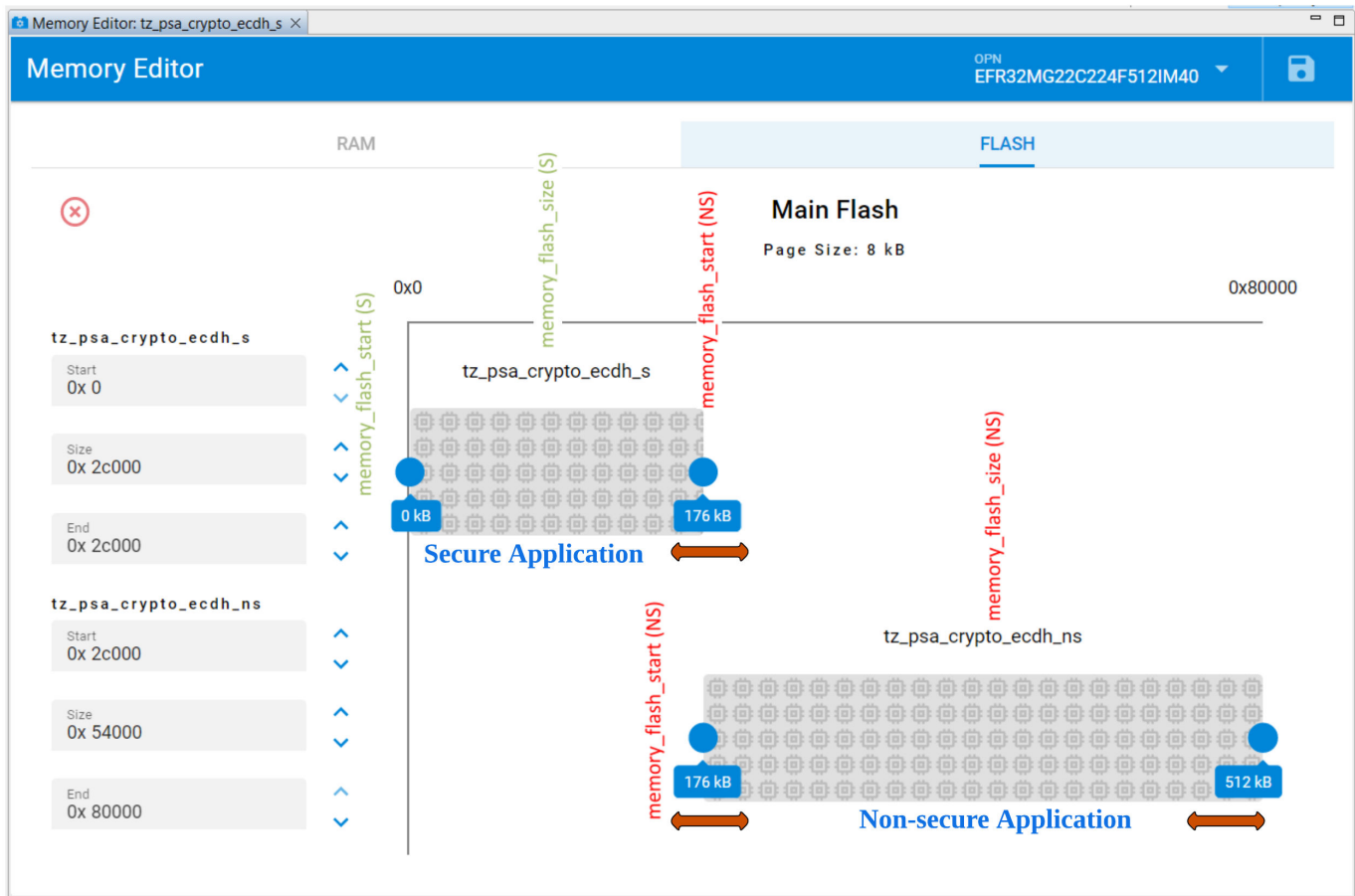
The **Memory Editor** in Simplicity Studio 5 is a graphical tool for editing the memory layout (flash and RAM) of the applications in the workspace. The Memory Editor will update the linker file in the project `autogen` folder with the custom settings. **Rebuild** the projects to use the new memory configurations in the linker files.

The Memory Editor is located at the **Quick Links** and **CONFIGURATION TOOLS** of Secure or Non-secure `slcp` file.



The following items will be determined by the flash usage in the Secure application memory map file:

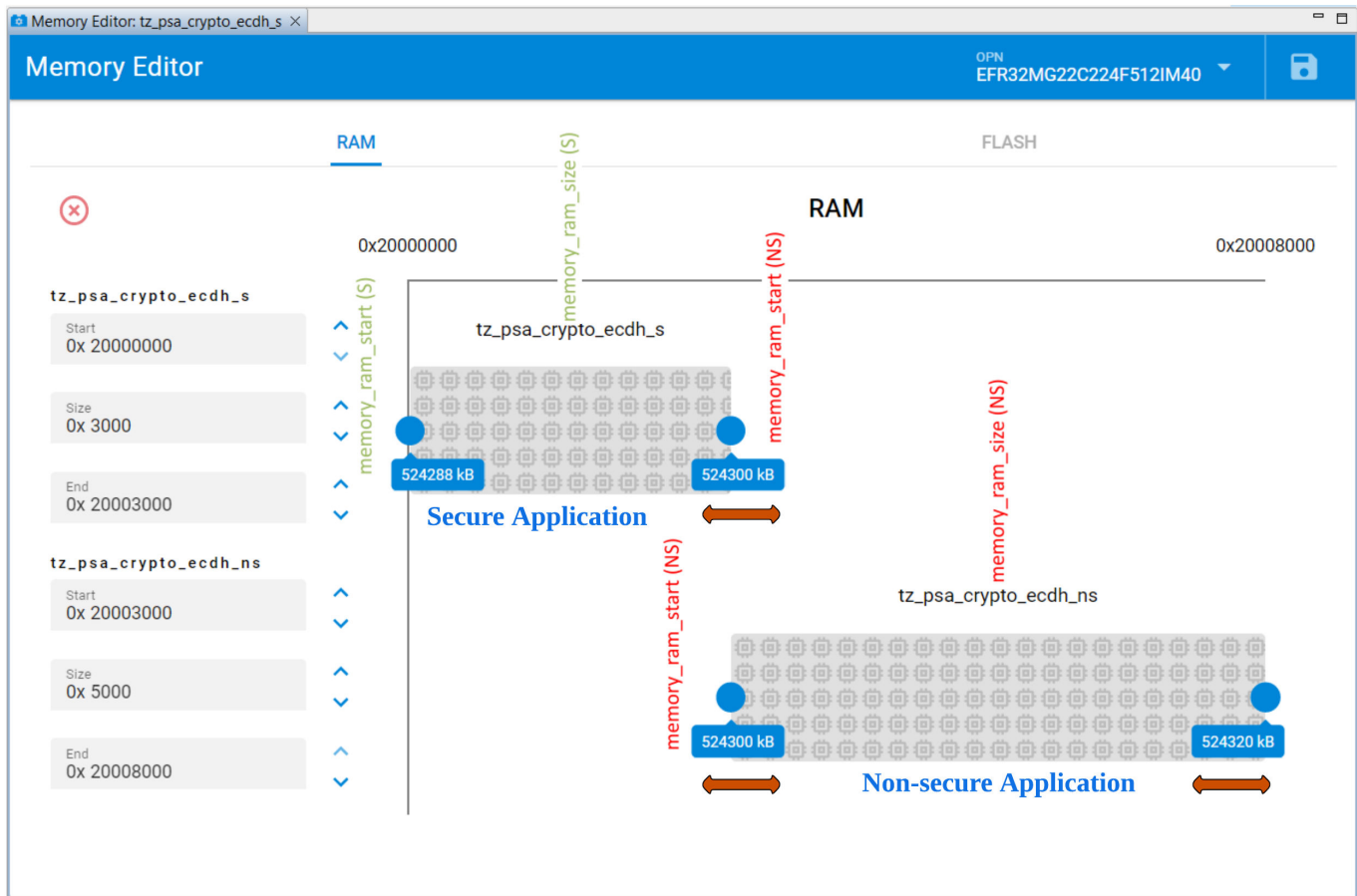
- memory_flash_size (S)
- memory_flash_start (NS)
- memory_flash_size (NS)



Note: The Memory Editor in Simplicity Studio v5.6.3.0 can only adjust the flash size in **8 kB** (page size) alignment, which may not fit the **4kB alignment** between the Secure and Non-secure flash boundary.

The following items will be determined by the RAM usage in the Secure application memory map file:

- memory_ram_size (S)
- memory_ram_start (NS)
- memory_ram_size (NS)



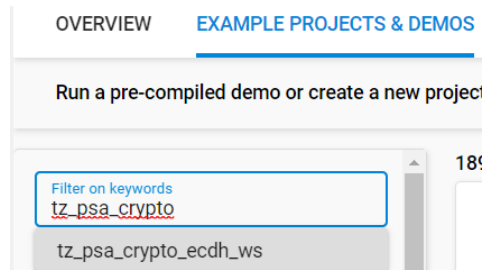
7.4 Build

The Secure project must be built first to create the Secure object library (`trustzone_secure_library.o`) with function entries for the Non-secure project. Both projects need to be rebuilt if any changes in the Secure project. Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to build the TrustZone platform examples.

7.4.1 Simplicity IDE

The following procedures are based on the **TrustZone PSA Crypto ECDH** example on BRD4182A Radio Board (EFR32MG22C224F512IM40).

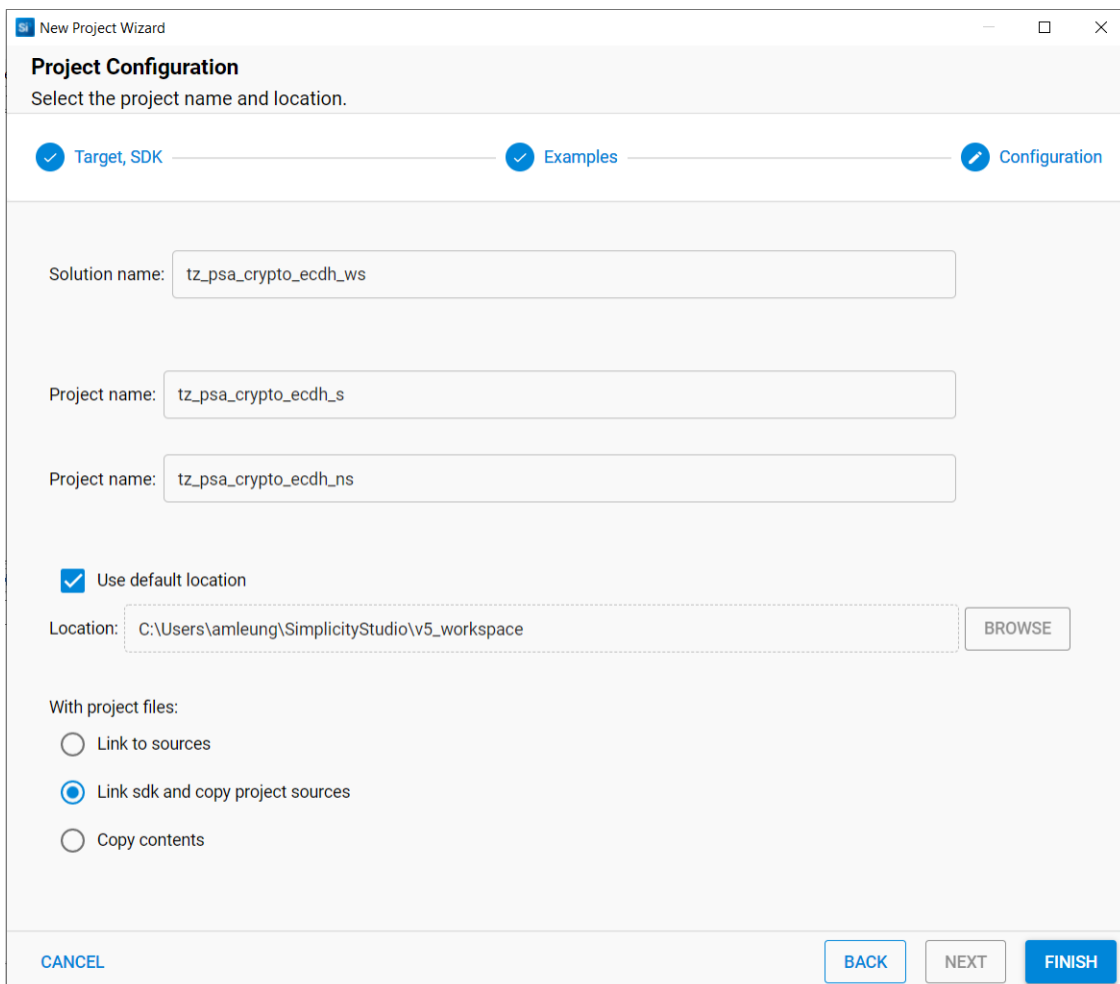
1. Use the `tz_psa_crypto` keyword to search in **EXAMPLE PROJECTS & DEMOS** tab. Select the `tz_psa_crypto_ecdh_ws` example.



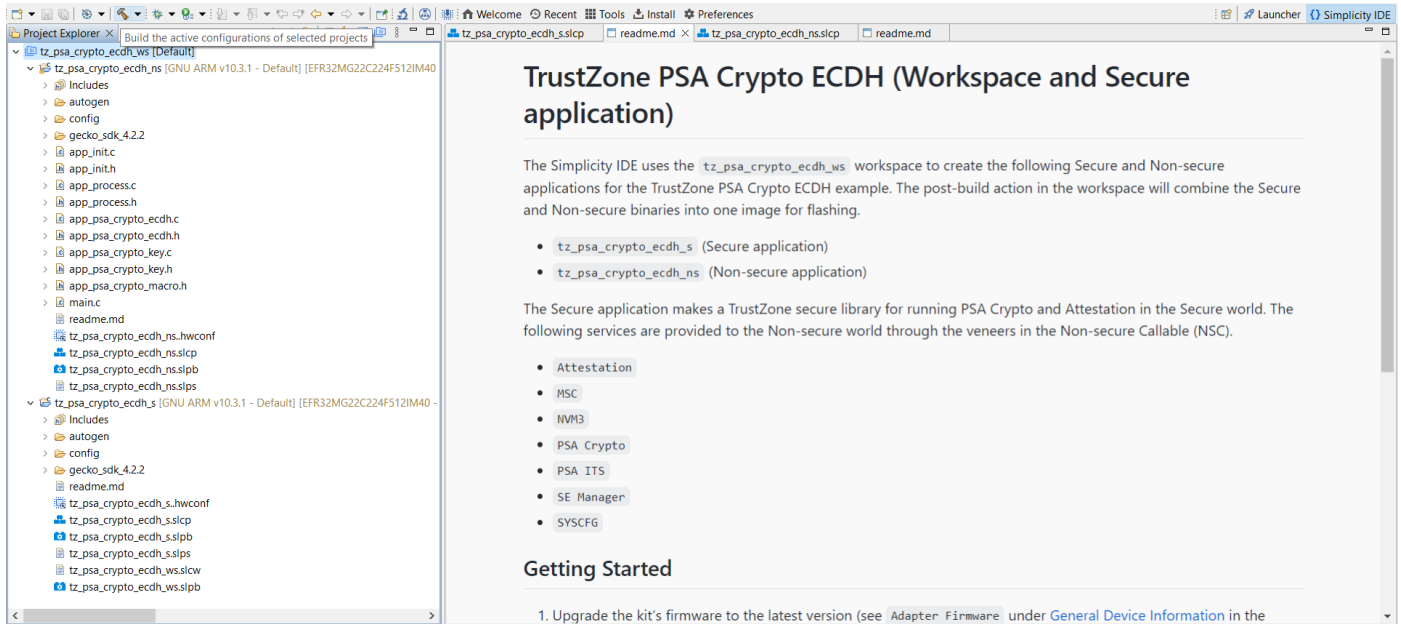
2. Click **[CREATE]** to generate the solution.



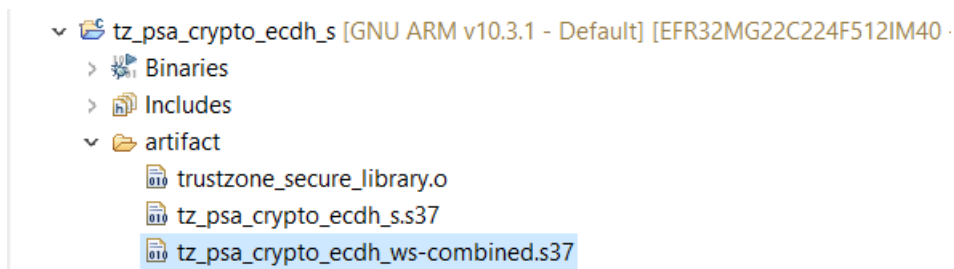
3. The **Project Configuration** dialog shows the Secure and Non-secure projects in the target solution. Click **[FINISH]** to start the creation process.



4. The Simplicity IDE perspective opens after finishing the solution creation. Click **Build** on the Simplicity IDE perspective toolbar to build the projects of a selected solution in order (Secure then Non-secure).



5. The post-build actions (.slpb files) of the Secure project, Non-secure project, and workspace will be processed in sequence if the solution is successfully built. The combined image (tz_psa_crypto_ecdh_ws-combined.s37) in the Secure project artifact folder can be used for programming the device or debugging.



6. Use **Memory Editor** to finalize the memory layouts of Secure and Non-secure applications and rebuild the solution to update the memory configurations.

Note: The Simplicity IDE can only apply the post-build action to a particular project if multiple Secure or Non-secure projects exist in the solution.

7.4.2 IAR EWARM

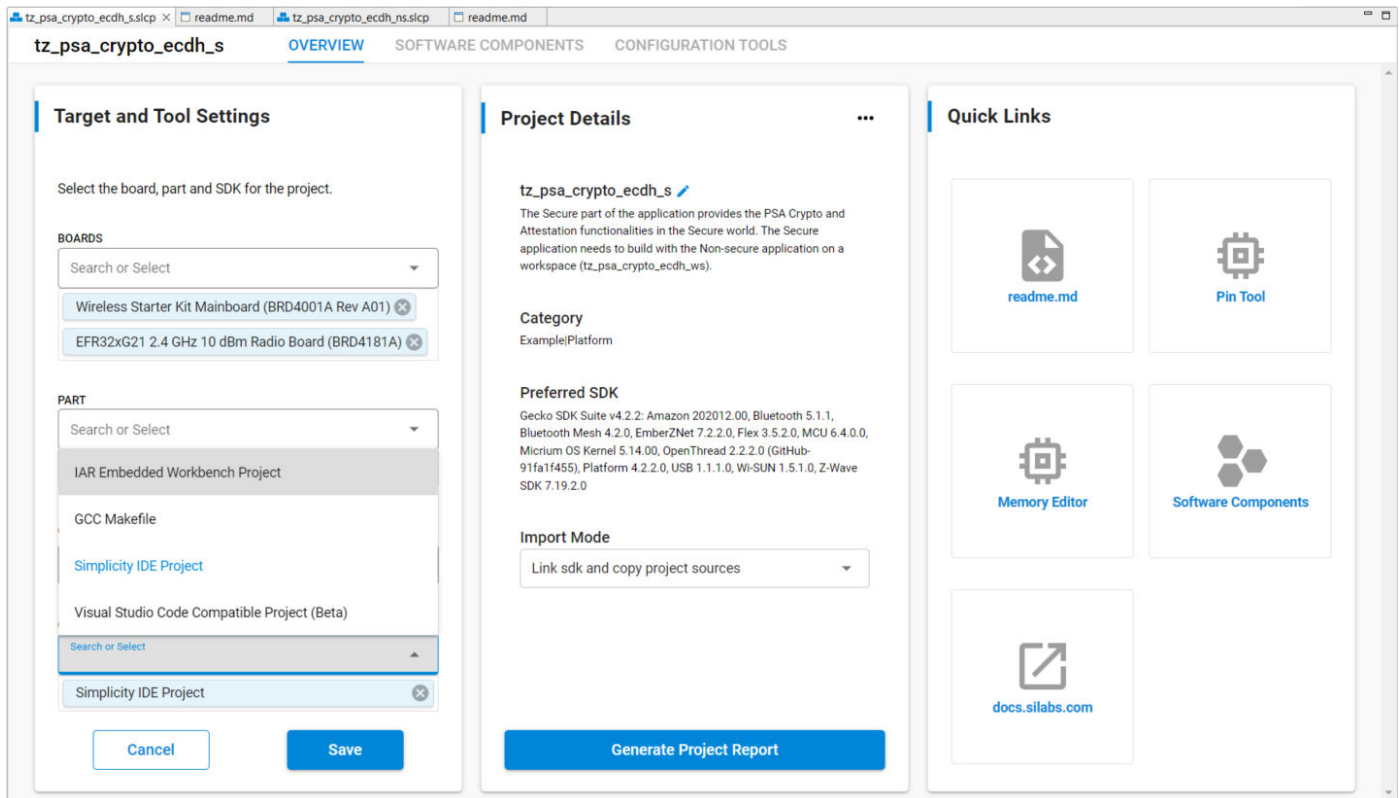
The following procedures are based on the **TrustZone PSA Crypto ECDH** example on BRD4181A Radio Board (EFR32MG21A010F1024IM32).

1. Follow steps 1 to 3 in [7.4.1 Simplicity IDE](#) to generate the solution for the `tz_psa_crypto_ws`. Select the `tz_psa_crypto_ecdh_s.slcip` file.
2. The **Overview** tab shows the **Target and Tool Settings** card on the left side. Scroll down if necessary and click **[ChangeTarget/SDK/Generators]**.

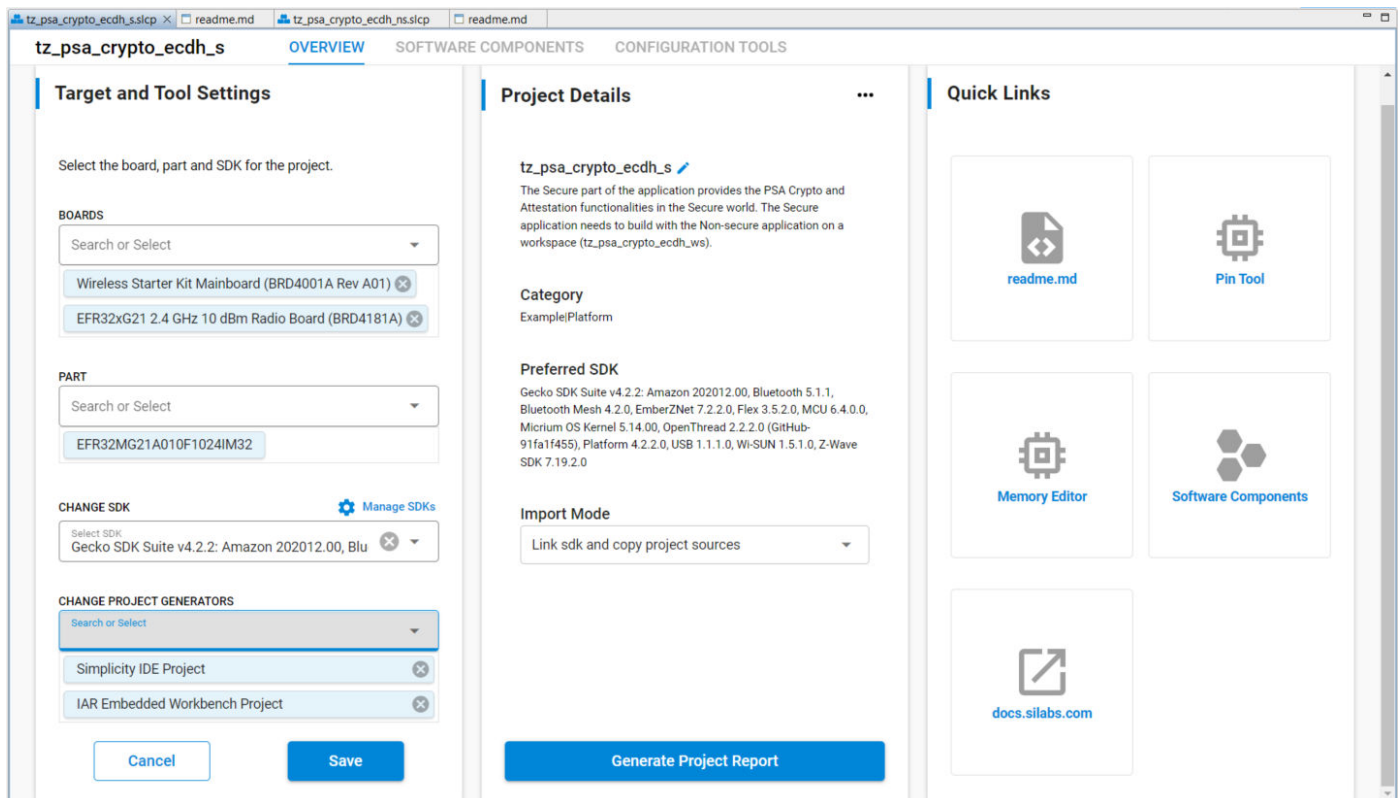
The screenshot shows the Simplicity IDE interface for the project `tz_psa_crypto_ecdh_s`. The **OVERVIEW** tab is active, displaying three main sections:

- Target and Tool Settings:** Features a card for the **Wireless Gecko** target (EFR32MG21A010F1024IM32). Below the card, it lists the target as "Wireless Starter Kit Mainboard (BRD4001A Rev A01)" and "EFR32xG21 2.4 GHz 10 dBm Radio Board (BRD4181A)". It also shows the "Selected SDK" as "Gecko SDK Suite v4.2.2" and "Project Generators" as "Simplicity IDE Project". A button labeled "Change Target/SDK/Generators" is visible at the bottom of this section.
- Project Details:** Shows the project name `tz_psa_crypto_ecdh_s` and a description: "The Secure part of the application provides the PSA Crypto and Attestation functionalities in the Secure world. The Secure application needs to build with the Non-secure application on a workspace (tz_psa_crypto_ecdh_ws)". It lists the "Category" as "ExamplePlatform" and the "Preferred SDK" as "Gecko SDK Suite v4.2.2" with various components. An "Import Mode" dropdown is set to "Link sdk and copy project sources". A button labeled "Generate Project Report" is at the bottom.
- Quick Links:** A grid of six links: "readme.md", "Pin Tool", "Memory Editor", "Software Components", and "docs.silabs.com".

3. Drop down the **CHANGE PROJECT GENERATORS** list and select **IAR Embedded Workbench Project**.



4. Click **[Save]** to generate an IAR **Secure** project (tz_psa_crypto_ecdh_s.ewp).



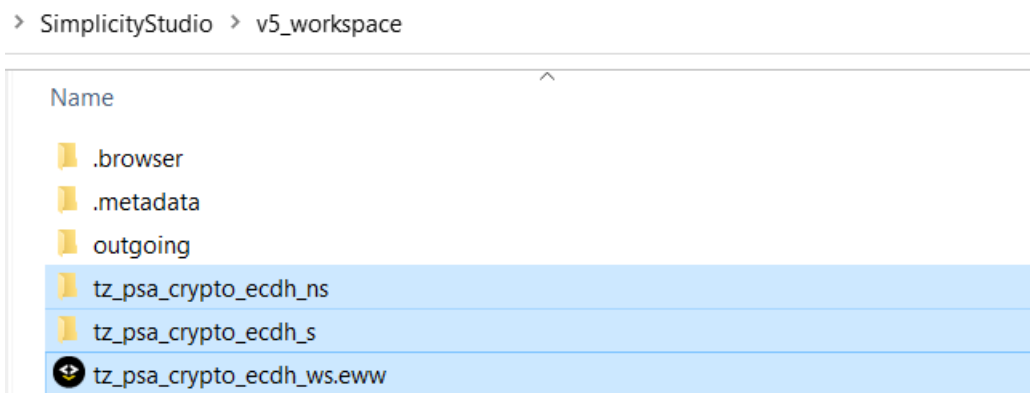
5. Select the tz_psa_crypto_ecdh_ns.s1cp file. Repeat steps 2 to 4 to generate an IAR **Non-secure** project (tz_psa_crypto_ecdh_ns.ewp).

6. Use a text editor to create an IAR `tz_psa_crypto_ecdh_ws.eww` file (shown below) to house the projects (`tz_psa_crypto_ecdh_s.ewp` and `tz_psa_crypto_ecdh_ns.ewp`) generated in steps 4 and 5. The location of the `tz_psa_crypto_ecdh_ws.eww` is the directory for `WS_DIR`.

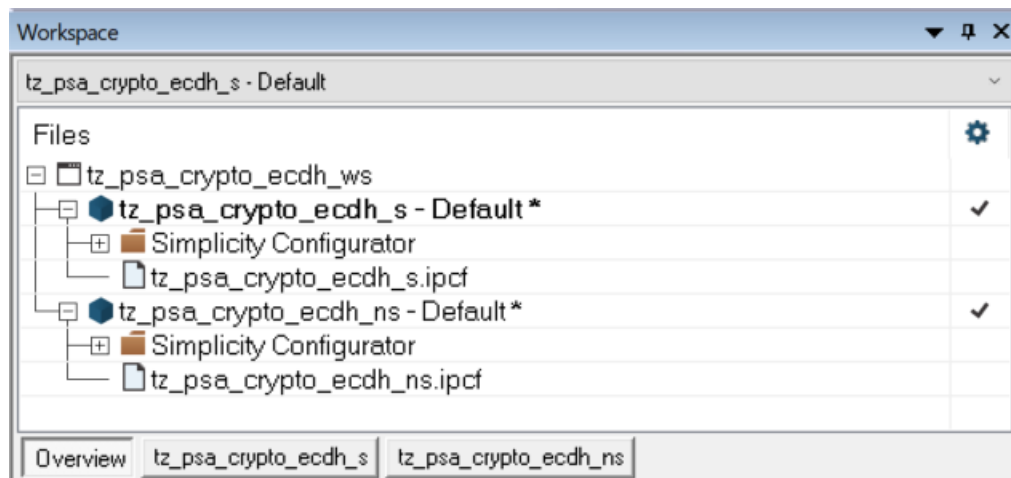
```
<?xml version = "1.0" encoding="iso-8859-1"?>


<workspace>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_s\tz_psa_crypto_ecdh_s.ewp</path>
  </project>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_ns\tz_psa_crypto_ecdh_ns.ewp</path>
  </project>

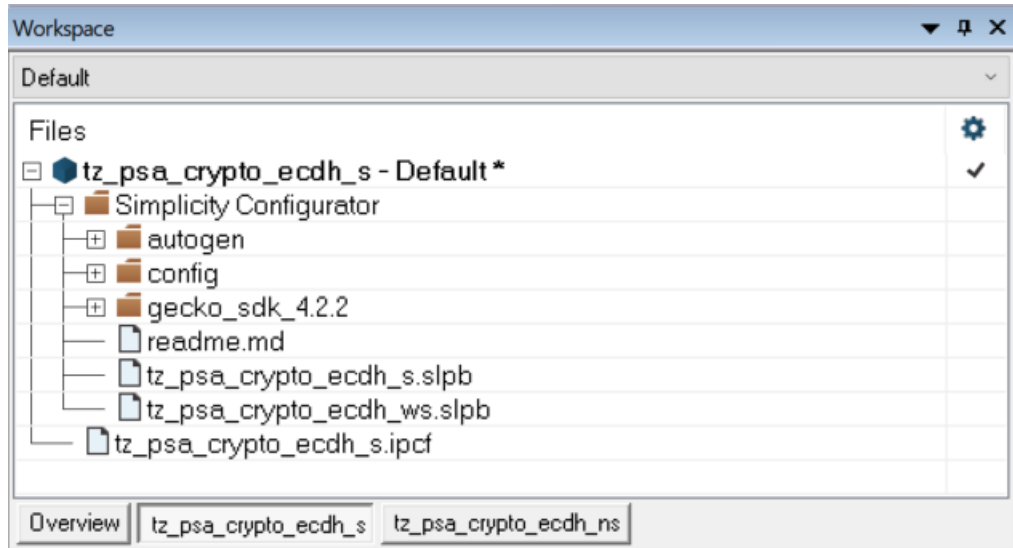
  <batchBuild/>
</workspace>
```



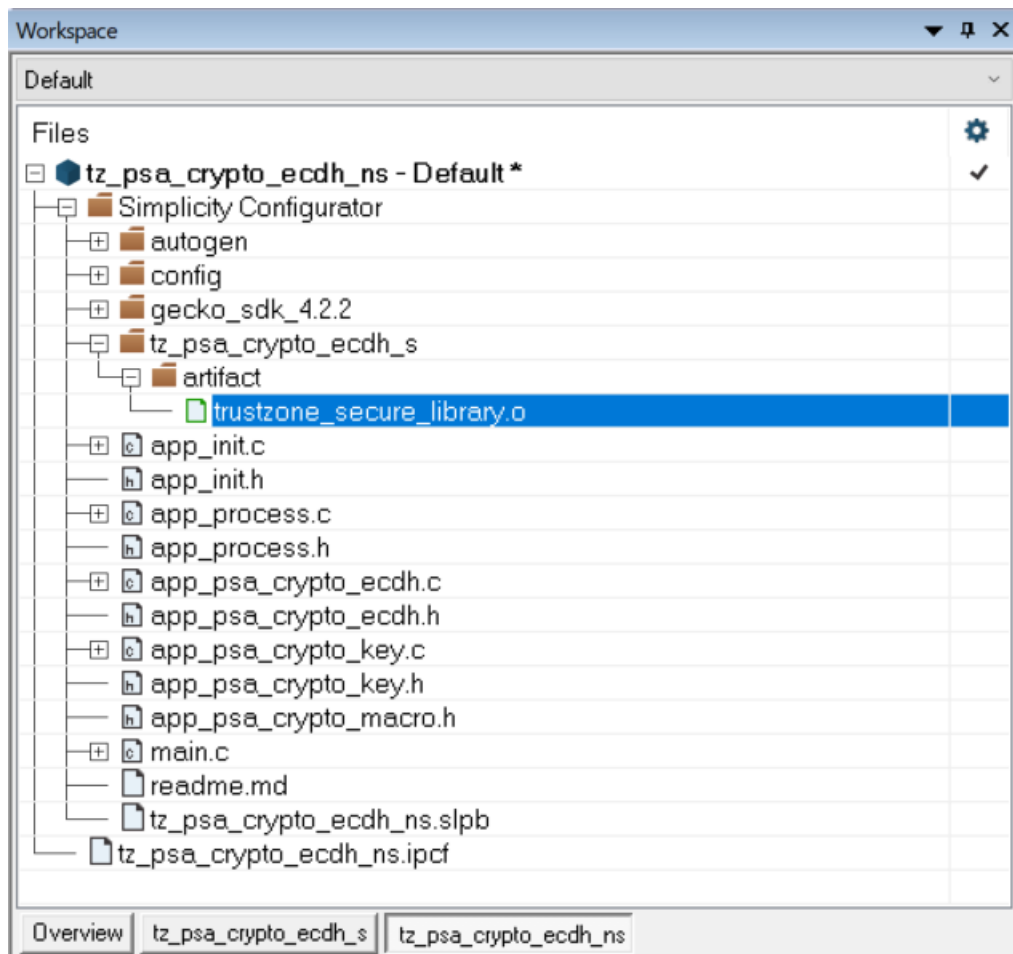
7. Double-click the `tz_psa_crypto_ecdh_ws.eww` file to open the workspace that includes Secure and Non-secure projects.



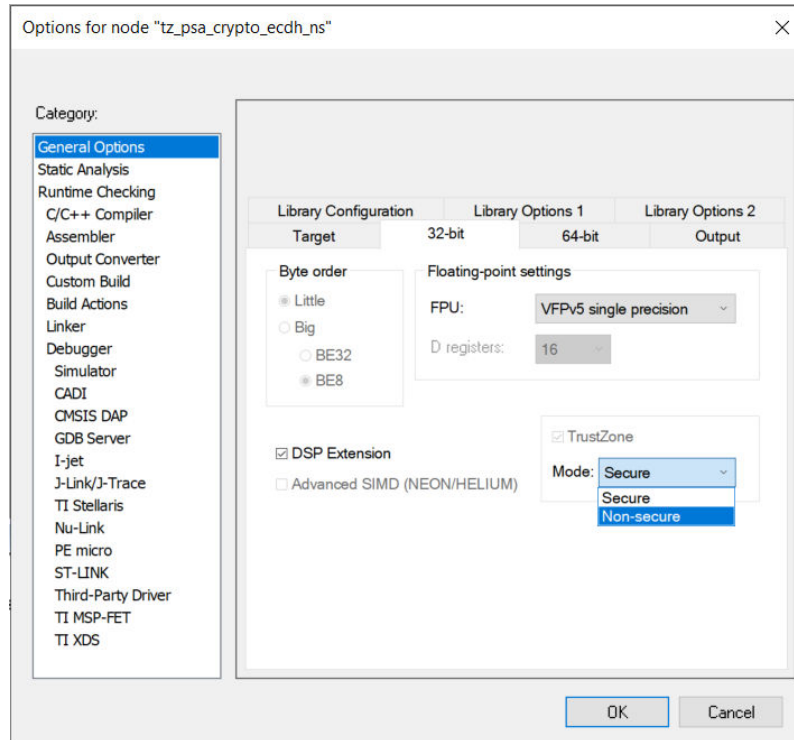
8. Click the `tz_psa_crypto_ecdh_s` tab to open the Secure project. Click  (Make) to build. It exports the Secure object library (`trustzone_secure_library.o`) for function entries that will be used by the Non-secure project.




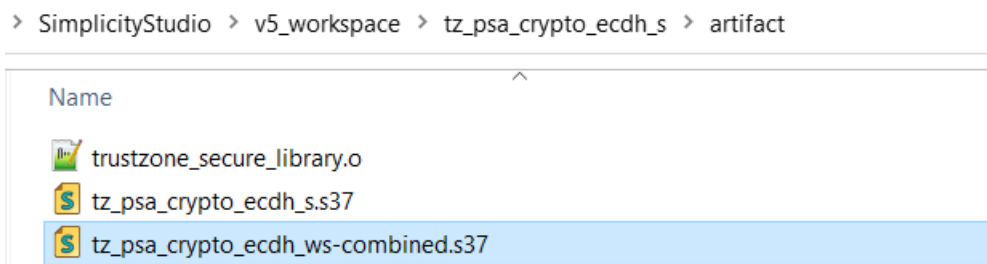
9. Click the `tz_psa_crypto_ecdh_ns` tab to open the Non-secure project.



10. The `SL_TRUSTZONE_NONSECURE` defined in the Non-secure project disables the [CMSE compiler option](#) (`--cmse`) regardless of whether the `Project` → `Options...` → `General Options` → `32-bit` → `TrustZone` → `Mode`: setting is Secure or Non-secure. So changing this configuration from Secure to Non-secure is optional. Click **[OK]** to exit.



11. Click  (Make) to build the Non-secure project. The post-build actions of the workspace (`tz_psa_crypto_ecdh_ws.slpb`) will be triggered in IAR to combine the Secure and Non-secure images (`tz_psa_crypto_ecdh_ws-combined.s37`) to the artifact folder of `tz_psa_crypto_ecdh_s` for programming the device.



12. Use [Memory Editor](#) to finalize the memory layouts of Secure and Non-secure applications and rebuild the Secure and Non-secure projects to update the memory configurations.

Note: The IAR EWARM can only apply the workspace post-build action to a particular project if multiple Secure or Non-secure projects exist in the workspace.

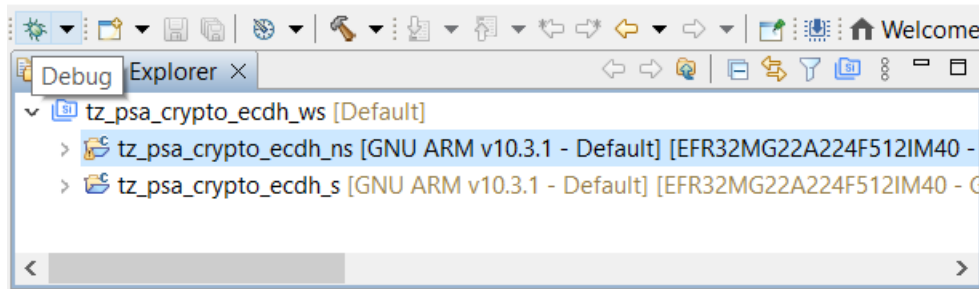
7.5 Debugging

Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to debug the TrustZone platform examples. Building the projects with Optimization Level None (`-O0`) is recommended for debugging.

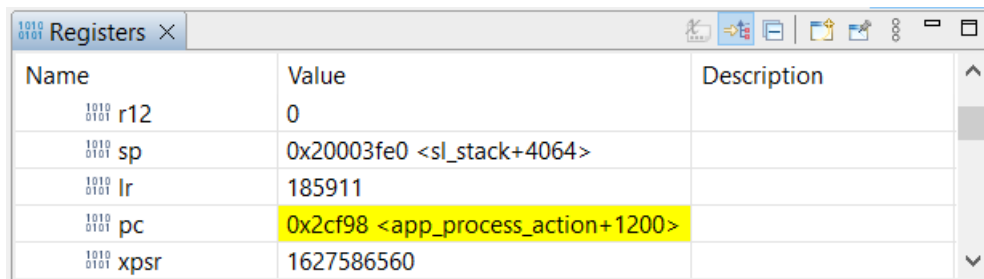
7.5.1 Simplicity IDE

The TrustZone debugging process on Simplicity IDE is similar to the existing sample projects in Simplicity Studio.

1. [GNU Debugger \(GDB\)](#) is recommended to debug TrustZone applications.
2. Flash the combined image (tz_psa_crypto_ecdh_ws-combined.s37) generated in [7.4.1 Simplicity IDE](#) to the device.
3. Select the Secure or Non-secure project and use the **Debug** icon to launch a debug session.



4. Follow the instructions in the [Using the Debugger](#) section in Simplicity Studio 5 User's Guide to debug the Secure or Non-secure application.
5. The debugger cannot step into the function in a Non-secure application when debugging the Secure application and vice versa. Use the **Program Counter** (PC in Secure or Non-secure address) in the **Registers** window to determine the program status.

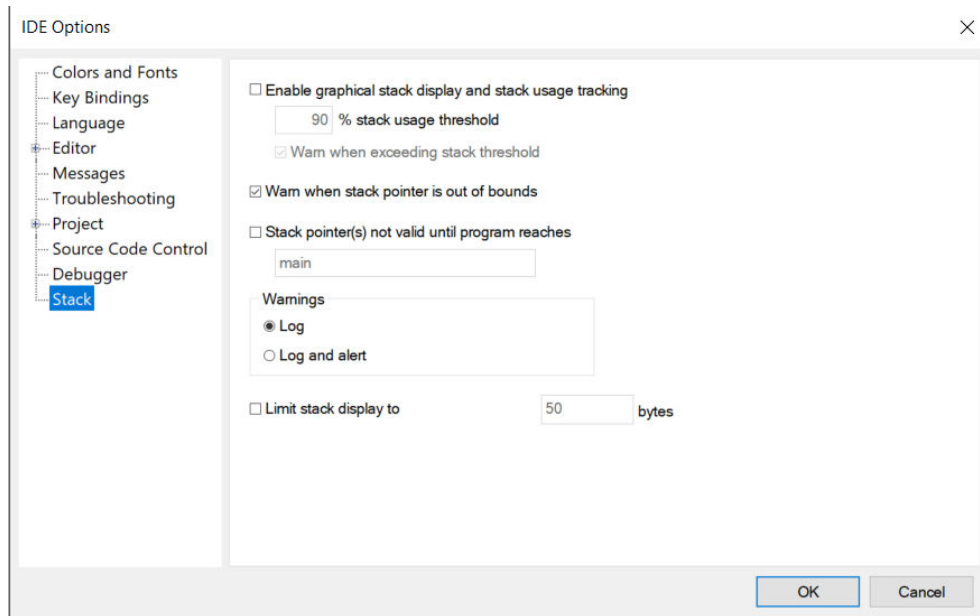
A screenshot of the Registers window in the debugger. The window title is 'Registers x'. It contains a table with three columns: 'Name', 'Value', and 'Description'. The registers listed are r12, sp, lr, pc, and xpsr. The 'pc' register value '0x2cf98 <app_process_action+1200>' is highlighted in yellow.

Name	Value	Description
r12	0	
sp	0x20003fe0 <sl_stack+4064>	
lr	185911	
pc	0x2cf98 <app_process_action+1200>	
xpsr	1627586560	

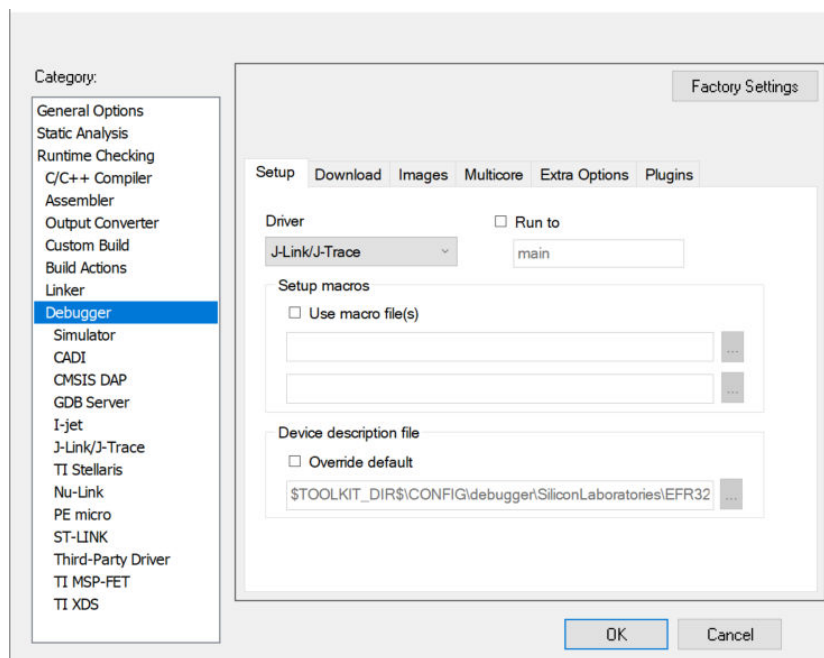
7.5.2 IAR EWARM

Use the `tz_psa_crypto_ecdh_ws.eww` workspace created in 7.4.2 IAR EWARM for the debugger settings. Except for a minor difference in step 3, the following steps are the same as those to set up the Secure (`tz_psa_crypto_ecdh_s`) and Non-secure (`tz_psa_crypto_ecdh_ns`) projects for debugging.

1. Select **Options...** in the **J-Link** **Tools** **Window** context menu of the Secure or Non-secure project and open the **IDE options** → **stack** dialog. Uncheck the **Stack pointer(s) not valid until program reaches** checkbox. Click **[OK]** to exit.

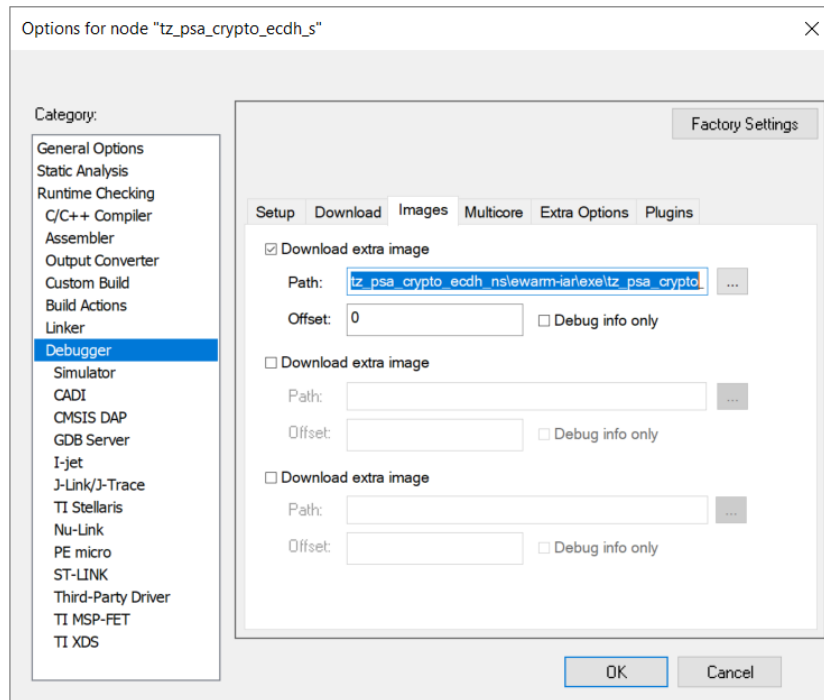


2. Select **Options...** in the **View** **Project** **J-Link** context menu of the Secure or Non-secure project and open the window for **Debugger** options. Click the **Setup** tab to open a dialog, and uncheck the **Run to** → **main** checkbox. Click the **Images** tab to set up another option.

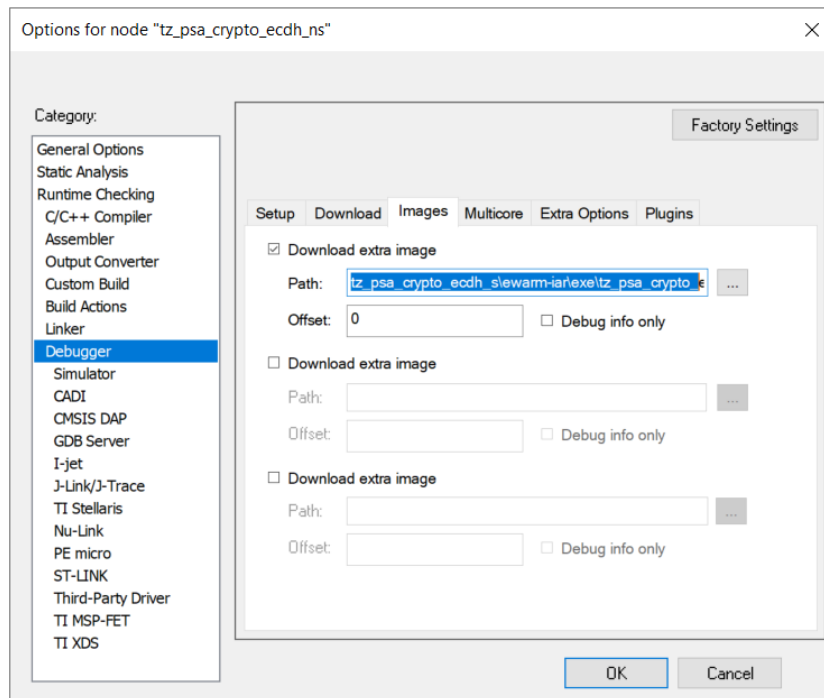


3. Check the **Download extra image** option. Enter the location of the .out file to **Path:** with **Offset:** set to 0. All project relative paths are resolved from the directory location of the tz_psa_crypto_ecdh_ws.eww workspace file.

Location of Non-secure .out file for Secure project: tz_psa_crypto_ecdh_ns\ewarm-iar\exe\tz_psa_crypto_ecdh_ns.out

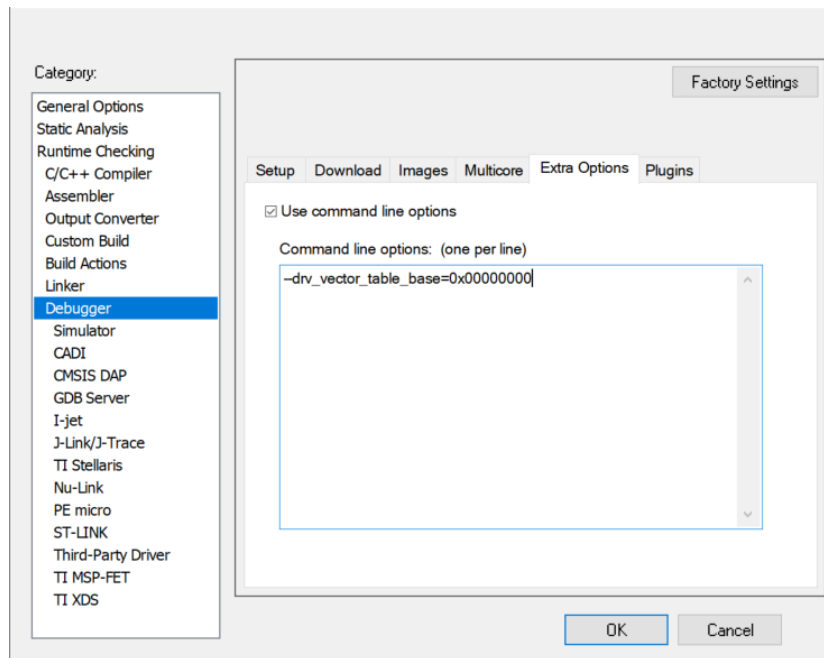




Location of Secure .out file for Non-secure project: tz_psa_crypto_ecdh_s\ewarm-iar\exe\tz_psa_crypto_ecdh_s.out

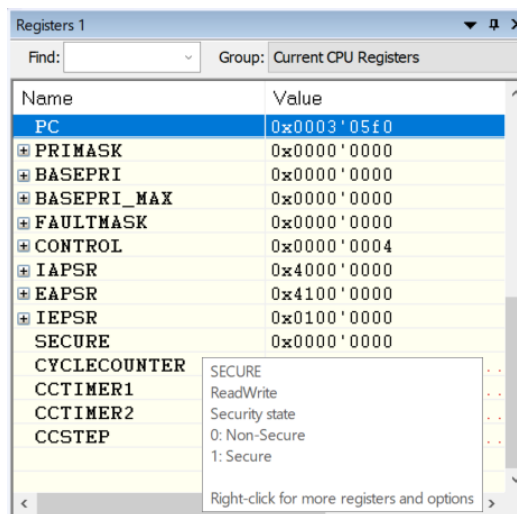



4. Click the **Extra Options** tab to set up another option.

5. Check the **Use command line options**. Enter `--drv_vector_table_base=0x00000000` to **Command line options: (one per line)** window. Click **[OK]** to exit.



6. Finish the debug settings in Secure and Non-secure projects, and click  (Download and Debug) in the Secure or Non-secure project to download the Secure and Non-secure images for debugging (assume both projects had successfully **built** before). Click  (Go) to start running the code in a Secure or Non-secure project.
7. The debugger will automatically switch between Secure and Non-secure projects when stepping into a function or hitting a breakpoint in a Secure or Non-secure project. Use the **Program Counter** (PC in Secure or Non-secure address) or **SECURE** (0 or 1) in the **Registers** window to determine the program status.



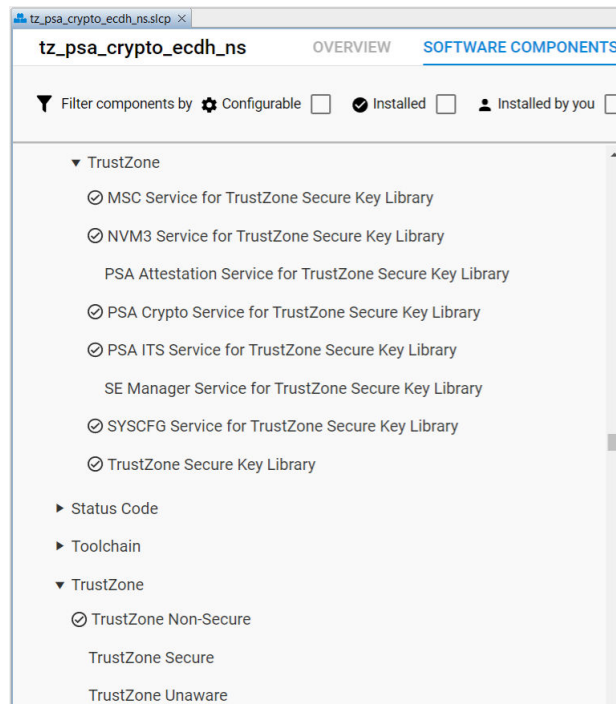
8. Click  (Stop Debugging) to end the debug session.

7.6 Benchmark

The TrustZone implementation will affect the memory footprint and performance of cryptographic operations. The following comparisons are based on the **TrustZone PSA Crypto ECDH** example on BRD4182A Radio Board (EFR32MG22C224F512IM40) with SE firmware v1.2.14.

7.6.1 Memory Footprint

The memory footprint of a TrustZone project depends on which services (software components in the figure below) provided by the [Secure Library](#) are used in the Non-secure application (`tz_psa_crypto_ecdh_ns` project).



The following tables compare the memory footprint of the [TrustZone-unaware](#) (Platform - PSA Crypto ECDH) and [TrustZone-aware](#) projects (`tz_psa_crypto_ecdh_ws`) based on the following conditions.

- The `tz_psa_crypto_ecdh_ns` reuses the source code from the Platform - PSA Crypto ECDH example without any changes.
- The total size in `tz_psa_crypto_ecdh_ns` does not consider the [4 kB alignment](#) on the Secure and Non-secure flash and RAM. The 4 kB alignment requirement will increase the actual usage of flash and RAM.
- All source code is compiled with Optimize for size (`-Os`) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.

Table 7.1. Flash Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	64688 B	—	—	64688 B
<code>tz_psa_crypto_ecdh_ws</code>	79172 B	288 B	29264 B	108724 B

Note: The NSC is part of the Secure code, and the total size does not include the flash for NVM3 storage.

Table 7.2. RAM Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	3784 B	—	—	3764 B
<code>tz_psa_crypto_ecdh_ws</code>	2156 B	—	1200 B	3356 B

Note: The total size does not include the RAM for the stack and heap. The Secure and Non-secure applications have their independent stack and heap.

7.6.2 PSA Crypto Performance

The following sections compare the PSA Crypto performance of the [TrustZone-unaware](#) (Platform - PSA Crypto ECDH) and [TrustZone-aware](#) projects (tz_psa_crypto_ecdh_ws) based on the following conditions.

- The tz_psa_crypto_ecdh_ns reuses the source code from the Platform - PSA Crypto ECDH example without any changes.
- All source code is compiled with Optimize most (-O3) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.
- Use ECC curve SECP256R1 on volatile and persistent keys.
- The EFR32MG22C224 runs at 38 MHz HFRCODPLL.

Volatile key ECDH operation on Platform - PSA Crypto ECDH

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 2928 time: 77 us)
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 2960 time: 77 us)
+ Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 332134 time: 8740 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 336860 time: 8864 us)
```

Volatile key ECDH operation on tz_psa_crypto_ecdh_ws

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 5047 time: 132 us)
+ Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 5067 time: 133 us)
+ Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 333956 time: 8788 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338470 time: 8907 us)
```

Persistent key ECDH operation on Platform - PSA Crypto ECDH

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 27489 time: 723 us)
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 27587 time: 725 us)
+ Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 332949 time: 8761 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 337803 time: 8889 us)
```

Persistent key ECDH operation on tz_psa_crypto_ecdh_ws

```
. ECDH Client
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 46998 time: 1236 us)
+ Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 45962 time: 1209 us)
+ Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 334127 time: 8792 us)
+ Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338321 time: 8903 us)
```

The overheads on the TrustZone-aware project (tz_psa_crypto_ecdh_ws) are due to the following operations of [Secure Library](#) implementation.

- Packages the list of input arguments in the appropriate format before calling into the NSC function.
- Switches from a Non-secure to a Secure state.
- Validates all input arguments before calling into the function in SPE.
- Encrypts PSA ITS if using a persistent key.
- Returns to a Non-secure state.

8. Revision History

Revision 0.3

March 2023

- Updated Root Key to TrustZone Root Key.
- Updated [5.1 System Configuration](#) and its sub-sections.
- Updated [5.3 Secure Library](#) for PSA Attestation.
- Updated [5.4 TrustZone Secure Key Storage](#).
- Added [5.5 PSA Attestation](#).
- Updated [6. Upgrade Existing Application to TrustZone](#).
- Added [7. TrustZone Platform Examples](#).

Revision 0.2

August 2022

- Various copy edits and copyright updates (Beta revision).

Revision 0.1

June 2022

- Beta revision.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!



IoT Portfolio
www.silabs.com/IoT



SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality



Support & Community
www.silabs.com/community

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

Note: This content may contain offensive terminology that is now obsolete. Silicon Labs is replacing these terms with inclusive language wherever possible. For more information, visit www.silabs.com/about-us/inclusive-lexicon-project

Trademark Information

Silicon Laboratories Inc.[®], Silicon Laboratories[®], Silicon Labs[®], SiLabs[®] and the Silicon Labs logo[®], Bluegiga[®], Bluegiga Logo[®], EFM[®], EFM32[®], EFR, Ember[®], Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Redpine Signals[®], WiSeConnect, n-Link, ThreadArch[®], EZLink[®], EZRadio[®], EZRadioPRO[®], Gecko[®], Gecko OS, Gecko OS Studio, Precision32[®], Simplicity Studio[®], Telegesis, the Telegesis Logo[®], USBXpress[®], Zentri, the Zentri logo and Zentri DMS, Z-Wave[®], and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.



Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com