Armv8-M Security Extension

Version 1.0

User Guide

Non-Confidential

Issue 01

107655_100_01_en

Armv8-M Security Extension
User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	22 January 2025	Non-Confidential	First release

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introducing Armv8-M Security Extension......9

1.1 Security concept......9

1.2 Privilege levels and security states......10

1.3 Real world examples......12

1.3.1 Bluetooth device......12

1.3.2 Firmware IP protection......13

2. Registers......15

2.1 General-purpose registers......15

2.2 Stack pointers......16

2.3 Stack limit registers......17

2.4 Special-purpose registers......18

2.5 System control registers......19

3. Memory configuration......20

3.1 Memory security attributes......20

3.2 SAUs......20

3.2.1 SAU......21

3.2.2 IDAU......21

3.2.3 Address lookup......22

3.2.4 Configuring SAU using CMSIS......23

3.3 Memory configuration with the MPU in Secure state......24

4. Function calls......26

4.1 Transition between security states......26

4.2 Implementing function calls across the Security boundary for C development......27

4.2.1 Non-Secure software calling a Secure API......27

4.2.2 Secure software calling a Non-Secure function......28

4.3 The assembly instructions for Security states transition by function call......30

4.3.1 Function call from Non-Secure to Secure state......31

4.3.2 Function return from Secure state......33

4.3.3 Function call from Secure to Non-Secure state......33

4.3.4 Function return from Non-Secure state......34

4.4 Software considerations in function calls across security boundary 35
4.4.1 Pointer passing across Security boundary 35
4.4.2 Non-pointer parameter passing 37
4.4.3 CMSE intrinsic functions 38
4.4.4 TT instruction 39
4.5 Floating-point context consistency and FPCXT payload 40

5. Armv8-M exception model with Security Extension 43
5.1 Prerequisites 43
5.2 Target states of exceptions 43
5.3 Exception prioritization 45
5.4 Stack frames 47
5.4.1 State context 49
5.4.2 Additional state context 49
5.5 EXC_RETURN 50
5.5.1 Scenario 1 51
5.5.2 Scenario 2 52
5.5.3 Scenario 3 53
5.6 SecureFault 53
5.7 External interrupts configuration and management 54
5.8 SVC and PendSV 56
5.9 SysTick timer 57
5.10 MemManage faults 57
5.10.1 Caution for Secure code developers 59

6. Developing software with Security Extension 60
6.1 Memory map partitioning 61
6.2 Add CMSIS startup and initialization code 61
6.3 Write the linker script or scatter file 62
6.4 Develop Secure software using Armv8-M Security Extensions 63
6.5 Build the Secure image 65
6.6 Build a Non-secure image that can call Secure APIs 67
6.7 Launch Non-secure images from Secure side 67
6.8 Preload and run the images on your device 68
6.9 Build a Secure image using a previously generated import library 68

7. Booting and initializations 70

7.1 Vector table, VTOR and reset behavior........70
7.2 FPU related Security settings for a device with FPU implemented........71
7.3 Stack pointer limit setup and stack sealing........71

8. RTOS and Secure software design considerations........73
8.1 RTOS configurations........73
8.1.1 Possible OS configurations........73
8.1.2 Extension of CMSIS-RTOS for Non-secure RTOS........76
8.2 Context-switching operations........77
8.2.1 RTOS design requirements........77
8.2.2 RTOS in the Non-secure state........78
8.2.3 RTOS in the Secure state........79
8.2.4 Supporting multiple Secure software libraries........79
8.3 Secure software development design considerations........80
8.3.1 Prevent Secure thread mode reentrancy........80
8.3.2 Security and privilege combination........80
8.3.3 AIRCR.BFHFNMINS considerations........81
8.3.4 EXC_RETURN.DCRS and EXC_RETURN.FType........82
8.3.5 Interrupt deprivileging........82
8.3.6 Non-reentrant exceptions........82
8.3.7 Secure floating-point contexts........82

9. Armv8-M Security Extension use case examples........83
9.1 Generic information........83
9.1.1 Tool versions........83
9.1.2 What does the program image contain?........84
9.1.3 Stack sealing........84
9.1.4 System memory map........85
9.1.5 SAU regions configuration in Secure project........87
9.1.6 Import library........88
9.2 hello-world-in-security-states........89
9.2.1 Secure project structure........89
9.2.2 Non-secure project structure........92
9.2.3 Output in Target Console........93
9.3 security-func-call-params-passing........94
9.3.1 Secure project structure........95
9.3.2 Non-secure project structure........98

9.3.3 Output in Target Console........ 99
9.4 basic-Non-secure-only-program........ 99
9.4.1 Secure project structure........ 100
9.4.2 Settings for minimal secure boot code........ 101
9.4.3 Launch Non-secure image........ 102
9.4.4 Non-secure project structure........ 102
9.4.5 Output in target console........ 103
9.5 exception-across-security-state........ 103
9.5.1 Secure project structure........ 104
9.5.2 Non-secure project structure........ 106
9.5.3 Output in Target Console........ 109

1. Introducing Armv8-M Security Extension

The Security Extension provides a foundation for improved system security in a wide range of embedded applications. This guide is for programmers intending to use the Security Extension in their systems.

This guide describes:

- Basic concepts that are used by the technology
- Toolchain requirements to create Secure software
- Concepts and best practices that developers must understand.

Developers must still perform their own security analysis in the context of their own threat model.

The Security Extension has also been known as TrustZone technology. This guide uses the name Security Extension.

1.1 Security concept

At a high level, the concepts of the Security Extension are similar to those in Armv8-A architecture. In both architectures, the processor has Secure and Non-secure states:

- Non-secure software can access Non-secure resources only
- Secure software can access both Non-secure and Secure resources

When the Security Extension is included in a processor, the existing Thread and Handler modes are duplicated to create Secure and Non-secure versions.

The Security Extension is designed for small energy-efficient systems. Code, assets, and data that belong to the more trusted software must be protected from access by or interference from the lesser trusted software. The trust model is asymmetrical and allows the more trusted software to access all code and data.

The Security Extension key functionalities are:

- The Security Extension supports multiple Secure function entry points. These can be directly called from the Non-secure state,without the overhead of calling into the operating system.
- The division of Secure and Non-secure worlds is memory map-based.
- Non-secure interrupts can still be serviced when executing a Secure function. Transitions between security states take place automatically during exception handling.
- There are dedicated resources, such as SysTick timers and fault handlers, for each security state.
- Protected exception priorities ensure that critical Secure operations cannot be blocked by Non-secure code.

- Separate debug authentication ensures that the Non-secure state can be debugged even if the Secure state is locked down.
- Security attributes are added to the bus so that protection can be applied to the whole system, not just the CPU.

The Security Extension enables the system and the software to be partitioned into Secure and Non-secure states. This enables software with two different levels of trust to co-exist in the same system, running on the same processor. This reduces the attack surface by enabling security critical software in the Secure state to be protected even if there is a security vulnerability in the Non-secure software.

One of the use cases for Security Extension technology is implementing a PSA Certified Root of Trust. The PSA Certified scheme is a common industry framework and methodology for built-in security, enabling silicon manufacturers, system software providers, and OEMs. This ensures the security of connected products using a proven security architecture and corresponding open-source implementations. For more information see PSA Certified.

Trusted Firmware-M (TF-M) leverages Security Extension technology to provide a Trusted Execution Environment (TEE). It is the reference implementation of platform security architecture aligning with PSA Certified guidelines. See TF-M for more details.

1.2 Privilege levels and security states

The Armv8-M architecture uses modes to partition software between privilege levels.

- Handler mode is always privileged and is used for all exception and interrupt handlers.
- Thread mode is used for all other code and can be privileged or unprivileged.

Privilege affects the ability to access memory, the control register, and some instructions.

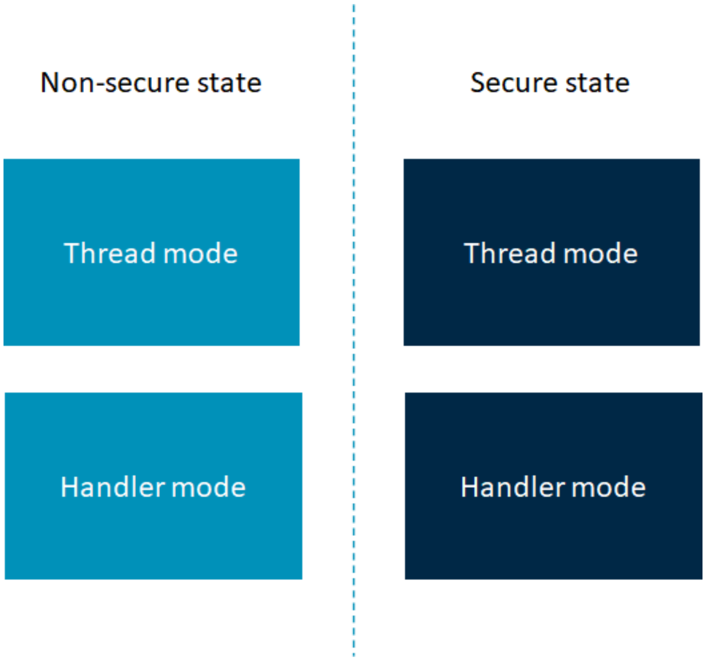
When the Security Extension is present two Security states are added:

- Secure software can access both Secure and Non-secure memories and resources
- Non-secure software can only access Non-secure memories and resources.

These security states are orthogonal to the existing Thread and Handler modes. This enables both a Thread and Handler mode in both Secure and Non-secure states.

The following figure shows Secure and Non-Secure processor states.

Figure 1-1: Processor states



This split enables you to allocate code to one of the four mode or state combinations depending on the level of trust and privilege that are necessary:

- Secure thread mode executes the initial boot code.
- You can configure each interrupt to run in either the Secure or Non-secure state.
- Application threads can run in a single security state, or transition between security states while running. The processor can switch between security states at a function call boundary or by taking an exception that targets a different security state.
- Code must be allocated to either the Secure or Non-secure states, but not both, because the security of the memory indicates which security state the code must execute in.
- In Armv8.0-M privileged access is unified across the security states. For example, an attacker might control Non-secure privileged code and Secure unprivileged code. This means that the attacker has both privileged and Secure access, and therefore can access the Secure privileged state. Enhancements in the Armv8.1-M architecture increase the level of protection and make privileged access independent in both security states.

Note

If the Security Extension is implemented, the system starts up in Secure state. If the Security Extension is not implemented, the system is always in Non-secure state.

The Secure software and Non-secure software are built as separate images. To understand how to build and link these images, see Booting and initializations.

1.3 Real world examples

Microcontrollers have a diverse set of use cases, ranging from simple bare metal systems to complex RTOS based environments. This section describes a few real-world use case examples that can be mapped with the Armv8-M Security Extensions.

Note

This guide does not describe all the steps that you must take to implement the mapping or ensure the security of the system. Information is here for clarity.

1.3.1 Bluetooth device

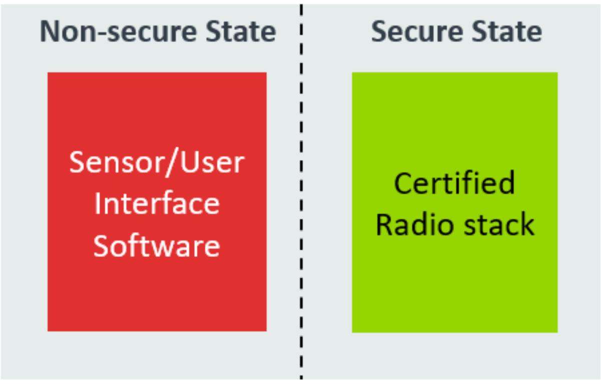
The Armv8-M Security Extensions enables a better foundation for system-level security. The simplest use case is where the thread modes in both the security states are privileged and only a single thread or firmware library needs protection.

One example of this is a radio communication system where:

- The certified radio stack, such as a Bluetooth stack, runs in Secure state
- The sensor software runs in Non-secure state

The following figure shows this kind of system.

Figure 1-2: IoT smart sensor



In this use case, the Security Extensions prevent:

- The application code from interfering with the operation of the certified radio stack
- Cloning and reverse engineering of the stack

If there are any bugs in the sensor software running in Non-secure state, they do not affect the radio stack because it is running in a Secure state. The low complexity of this use case and the ability to directly call across the security boundary makes it easy to retrofit protection to an existing software stack.

Note

In this example, a simple monolithic sensor code and a monolithic radio stack are used within a single thread. In this case, using a single stack pointer (MSP) in each security state, you can set:

- All the sensor software to run in Non-secure Privileged thread mode
- The certified radio stack to run in Secure Privileged thread mode

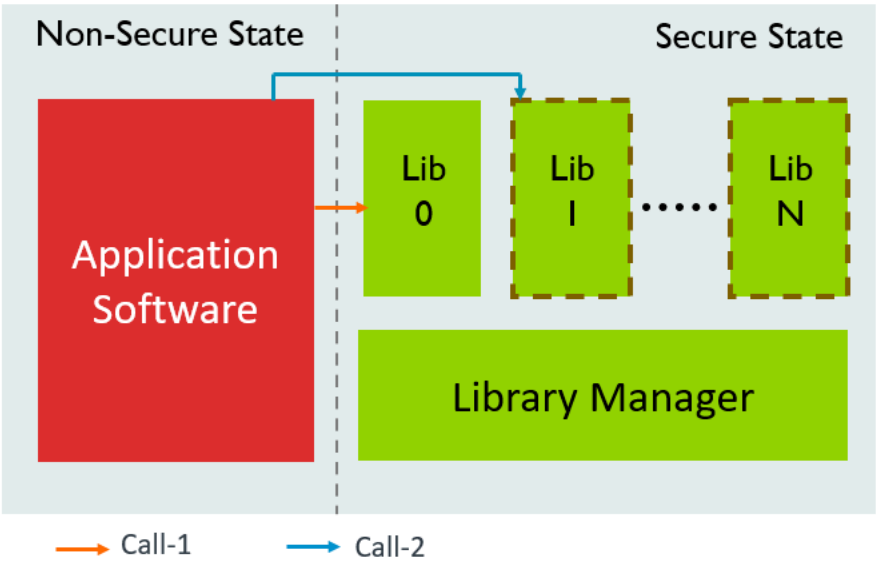
You can also use this setup to end system software of a safety critical systems like automotives or medical devices.

1.3.2 Firmware IP protection

In some systems, software libraries from several mutually distrustful vendors are run on a single processor. For example, an OEM might buy a microcontroller with multiple add-on libraries such as USB stacks, motor control code, and graphics libraries, from mutually distrustful 3rd party suppliers.

The following figure shows an example setup.

Figure 1-3: Firmware IP protection



In this system, a library manager running in the Secure privileged states uses the Secure MPU to make all except one of the Secure libraries inaccessible. If a function call is made to the active library that is currently accessible, then no fault is generated. The call has a very low calling overhead. See call 1 in the figure above. When a function call is made to an inactive library, an

MPU fault is generated. See call 2 to lib 1 in the figure above). This fault causes the library manager to swap the MPU configuration and Secure stack pointer to make the inactive library active.

This functionality has several advantages:

- Function calls are made directly to the destination library, which makes the software easier to write.
- The overhead of changing the MPU configuration only happens when a call is made to an inactive library. Subsequent calls to the same library do not cause a fault and have minimal overheads.
- Direct calls between Secure libraries are supported without a loss of protection.

2. Registers

Seeing data in registers and accessing control registers is crucial for the security of the system. The behavior of registers based on the current security states can include:

- Common access: The register or field is accessible from both security states.
- Secure access only: The register or field can only be accessed from the Secure state.
- Banked registers or fields: Both a Secure and Non-secure version of the register or field are implemented and can hold different values.

Armv8-M architecture banking behaves as follows:

- Sometimes only one of a banked pair of registers or fields is used at a time.
    - The bank to select is sometimes based on the state the processor is in at the moment, for example stack pointers.
    - The bank to select is sometimes based on the state of another control bit. For example, the bank of FPCCR.LSPACT bit to use is based on the FPCCR.S bit.
    - The bank to select is based on another condition, such as whether Secure debug is allowed.
- Sometimes, both the Secure and Non-secure banks of a register or field apply at the same time. You then get the combined behavior of both sides, for example priority boosting registers like BASEPRI.

For more information, see the architectural register description in Armv8-M Architecture Reference Manual that specifies the banking options used.

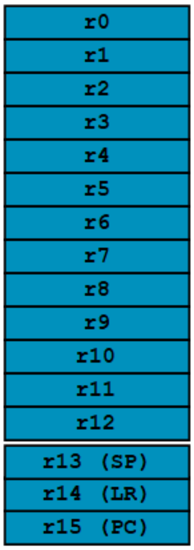
2.1 General-purpose registers

Armv8-M uses 16 General-Purpose Registers (GPRs), R0 - R15, for normal execution. Most instructions use these registers. Some registers have special uses:

- R13 accesses the current stack pointer.
- R14 is a link register to hold the return address of function calls.
- R15 represents the current program counter.

The following figure shows the GPRs.

Figure 2-1: GPRs



All of these registers, except R13 the stack pointer, are unbanked and always accessible to software. This means that a value, left in a register after a change in state, is visible to the other security state. Sometimes this is useful. For example, it allows data to be passed as part of a function call. It is crucial that Secure data is not made visible to non-trusted code in a unexpected way. For exceptions that change Security state, the registers are protected automatically by the hardware. For function calls, software must protect the registers. See Memory configuration.

2.2 Stack pointers

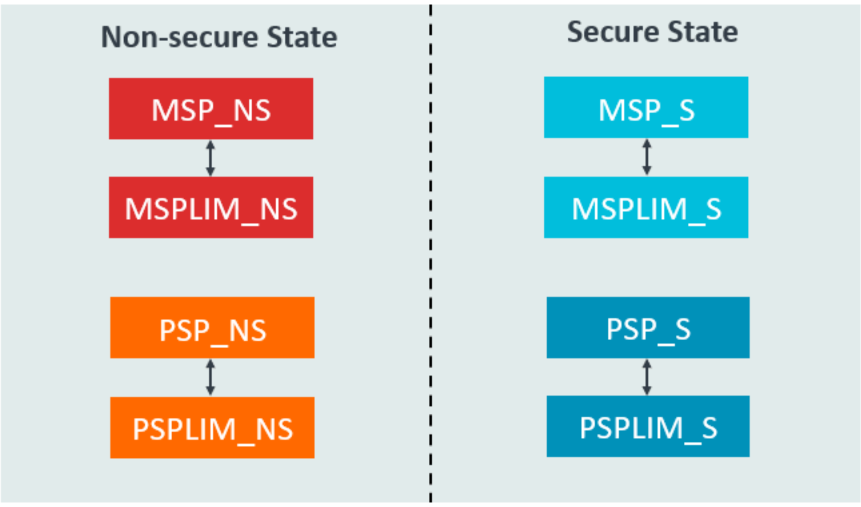
The Armv8-M architecture contains two stack pointers:

- The Main Stack Pointer (MSP): Software running in Thread mode can use the MSP. It is always used by Handler mode.
- The Process Stack Pointer (PSP): Thread mode software can use the PSP. The SPSel bit in the CONTROL register configures the Stack Pointer register to be used by Thread mode.

If the Armv8-M Security Extension is implemented, both the MSP and PSP are banked between security states. This provides four stack pointers: MSP_NS, PSP_NS, MSP_,S and PSP_S. The bank of the stack pointer is selected based on the current security state. The CONTROL.SPSel bit is also banked. This enables different stack configurations in Secure and Non-secure Thread modes.

The following figure shows the stack pointers and the stack limit registers.

Figure 2-2: Stack pointers and stack limit registers



Note

For stacks and stack limit register configurations:

- All Secure stack pointers must point to memory that is marked as Secure and so it is not accessible for Non-secure code.
- All the stacks, both Secure and Non-secure, must be placed in a memory with an eXecute Never (XN) attribute.
- If both privilege levels are used, then the Thread mode must always use the PSP stack.
- The top of all Secure stacks must be sealed with a value of `0xFEF5EDA5`. This concept is called stack sealing. See Armv8-M Architecture-Stack sealing and why it is needed in TrustZone for Armv8-M.
- Each stack pointer has an associated limit register which detects and prevents stack overflows. See Stack limit registers.

2.3 Stack limit registers

The stack limit registers in Armv8-M architecture can minimize stack overflow errors which are common in software. With stack limit registers, the privileged software can define the stack sizes for MSP and PSP in each security state as the [Stack Pointers and Stack Limit Registers] figure shows. If the value of the stack pointer goes below the stack limit registers, then it raises a synchronous stack limit violation (STKOF) UsageFault.

Armv8-M Architecture Reference Manual describes every instruction operation that operates with stack limit checks. Legal instructions that write to the stack pointer are subject to stack pointer limit checking, except where SP is the destination register for load instruction.

Table 2-1 shows how the stack limit registers can be accessed through CMSIS-CORE functions.

Function	Description
uint32_t __get_PSPLIM (void)	Get PSP limit in the current security state.
void __set_PSPLIM (uint32_t ProcStackPtrLimit)	Set PSP in the current security state.
uint32_t __get_MSPLIM (void)	Get MSP limit in the current security state
__set_MSPLIM (uint32_t MainStackPtrLimit)	Set MSP limit in the current security state.
void __set_MSPLIM(void)	Set MSP limit in the current security state
uint32_t __TZ_get_PSPLIM_NS (void)	Get PSP limit (Non-secure)
void __TZ_set_PSPLIM_NS (uint32_t ProcStackPtrLimit)	Set PSP (Non-secure)
uint32_t __TZ_get_MSPLIM_NS (void)	Get MSP limit (Non-secure)
void __TZ_set_MSPLIM_NS (uint32_t MainStackPtrLimit)	Set MSP limit (Non-secure)

Note

In Cortex-M processors that implement Armv8-M-Baseline, the stack limit registers are not implemented. If you use CMSIS-CORE functions shown in Table 2-1 in Cortex-M processors, reads return zero and writes are ignored.

2.4 Special-purpose registers

PRIMASK, FAULTMASK, and BASEPRI registers are Special-Purpose Registers (SPRs) used for exception masking. See Armv8-M Programmers Model User Guide. When Security Extension is implemented, these mask registers are banked between the security states. Both the Secure and Non-secure versions are applied at the same time, regardless of the current security state of the processor. These and other SPRs are accessible in privileged state only. To access these registers, use special move instructions such as MRS, MSR, VMSR, VMSR, and CPS. In addition to accessing the register associated with the current security state, the Secure state can also access the Non-secure version of the registers:

```
MRS R0, PRIMASK       // Copy PRIMASK value to R0
                      // in current security state
MRS R0, PRIMASK_NS    // Copy Non-secure PRIMASK value to R0. Only
                      // allowed when executed in the Secure state
```

CMSIS-CORE also provides functions access to exception masking registers as Table 2-2 shows.

Function	Usage
void __set_PRIMASK (uint32_t priMask)	Sets the PRIMASK register
uint32_t __get_PRIMASK (void)	Reads the PRIMASK register
void __set_FAULTMASK (uint32_t faultMask)	Sets the FAULTMASK register
uint32_t __get_FAULTMASK (void)	Reads the FAULTMASK register
void __set_BASEPRI(uint32_t basePri)	Sets the BASEPRI register
uint32_t __get_BASEPRI(void)	Reads the BASEPRI register

Function	Usage
void __set_BASEPRI_MAX (uint32_t basePri)	Sets the BASEPRI register using BASEPRI_MAX symbol
void __TZ_set_PRIMASK_NS (uint32_t priMask)	Sets the PRIMASK_NS register
uint32_t __TZ_get_PRIMASK_NS (void)	Reads the PRIMASK_NS register
void __TZ_set_FAULTMASK_NS (uint32_t faultMask)	Sets the FAULTMASK_NS register
uint32_t __TZ_get_FAULTMASK_NS (void)	Reads the FAULTMASK_NS register
void __TZ_set_BASEPRI_NS (uint32_t basePri)	Sets the BASEPRI_NS register
void __TZ_get_BASEPRI_NS (void)	Reads the BASEPRI_NS register

For more information, see Armv8-M Architecture Reference Manual

2.5 System control registers

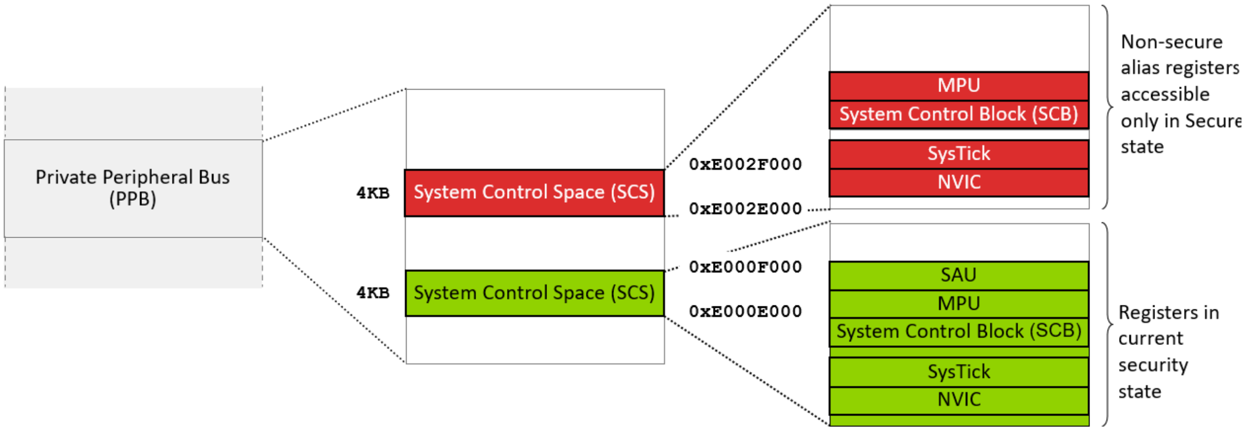
The System Control Space (SCS) provides registers for control, configuration, and status reporting of the processor. The SCS is in the Private Peripheral Bus (PPB) space and contains:

- The NVIC
- The MPU
- The System Control Block (SCB)
- Various peripherals

The SCS is at address `0xE000E000`, which Secure and Non-secure software can access. The SCS accesses the bank of the registers associated with the current security state. Secure software can access the Non-secure version of SCS registers at alias address `0xE002E000`.

The following figure shows the SCS in Security state.

Figure 2-3: SCS in Security State


Private Peripheral Bus
(PPB)
4KB
System Control Space (SCS)
4KB
System Control Space (SCS)
0xE002F000
0xE002E000
0xE000F000
0xE000E000
MPU
System Control Block (SCB)
SysTick
NVIC
Non-secure alias registers accessible only in Secure state
SAU
MPU
System Control Block (SCB)
SysTick
NVIC
Registers in current security state

3. Memory configuration

This chapter describes:

- Memory security attributes and Security Attribute Units (SAU)
- How to partition memory into Secure, Non-secure, and Secure, Non-secure Callable (NSC) regions using SAU
- Memory Protection Unit (MPU) interactions with the security state

3.1 Memory security attributes

When the Security Extension is implemented, the address space is partitioned into Secure and Non-secure memory regions. The Secure memory space is further divided into two types:

- Non-secure (NS): Non-secure memory regions are accessible by both Non-secure and Secure software. When executing software in Non-secure memories, the processor is in Non-secure state.
- Secure: Secure regions are used for memory and peripherals that are only accessible by Secure software. When executing software in Secure memories, the processor is in Secure state. If a data access is made from Non-secure state to an address marked as Secure, then the processor takes a SecureFault.
- Non-secure Callable (NSC): NSC region is a special Secure memory region, which is the only type that an Armv8-M processor permits to hold entry points for Secure APIs. These entry points enable software to transition from Non-secure state to Secure state. The combination of a Secure NSC region and the SG instruction provides a mechanism to prevent Non-secure software jumping into arbitrary Secure code and potentially bypassing security checks in entry points.

  If the Non-secure software does branch into Secure executable memory region where either the first instruction is not an SG instruction or the address does not have a Secure NSC attribute, then then the processor takes a SecureFault. Non-secure software cannot read or write to a Secure NSC memory, but can branch into it if the branch target is a SG instruction.

Exempted memory regions can be accessed by both Secure and Non-secure software. These regions are generally used by processor and debug registers.

3.2 SAUs

The Security attribution of a memory region is controlled by a combination of two attribution units:

- Security Attribution Unit (SAU): The SAU is programmable in Secure state and is controlled by CPU registers in a similar way to the MPU.

- Implementation Defined Attribution Unit (IDAU): The IDAU is external to the processor and depends on chip designer implementation.

Both the SAU and IDAU are optional. The memory Security attribution is determined by the unit which has the strictest Security attribution specified.

The device designer divides the memory spaces into Secure and Non-secure areas. Designers can use an IDAU to define a fixed memory map, and use a SAU to override the security attributes for some parts of the memory.

3.2.1 SAU

The SAU contains programmable registers within System Control Space (SCS). The SAU is programmable by Secure privileged software. The number of SAU regions depends on the implementation of the Armv8-M processor. It is common for processors to include 0, 4, or 8 SAU regions. Each region is defined using the base address register and the limit address register. These registers have a minimum granularity of 32 bytes.

The following table lists the SAU registers used for programming the regions.

Address	Register	Description	CMSIS-Core Symbol
0xE000EDD0	SAU_CTRL	SAU Control register	SAU->CTRL
0xE000EDD4	SAU_TYPE	SAU Type register	SAU->TYPE
0xE000EDD8	SAU_RNR	SAU Region Number register	SAU->RNR
0xE000EDDC	SAU_RBAR	SAU Region Base Address register	SAU->RBAR
0xE000EDE0	SAU_RLAR	SAU Region Limit Address register	SAU->RLAR

3.2.2 IDAU

Chip vendors design the IDAU. Usually, the IDAU:

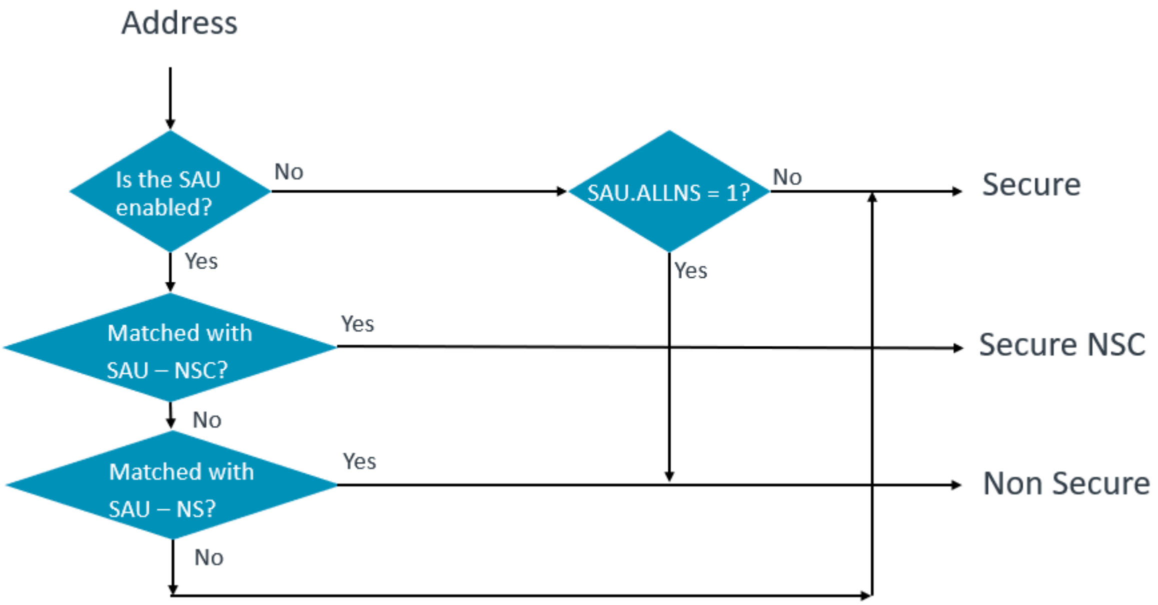
- Provides address lookups
- Generates Security attributes of the address being accessed
- Defines the memory regions as Secure, Non-secure, Non-secure Callable, or exempt from Security checking

For more details on a particular IDAU, read the materials provided by the relevant chip vendor. However, IDAUs often follow the guidelines in the TrustZone Technology Microcontroller System Hardware Design Concepts User Guide.

3.2.3 Address lookup

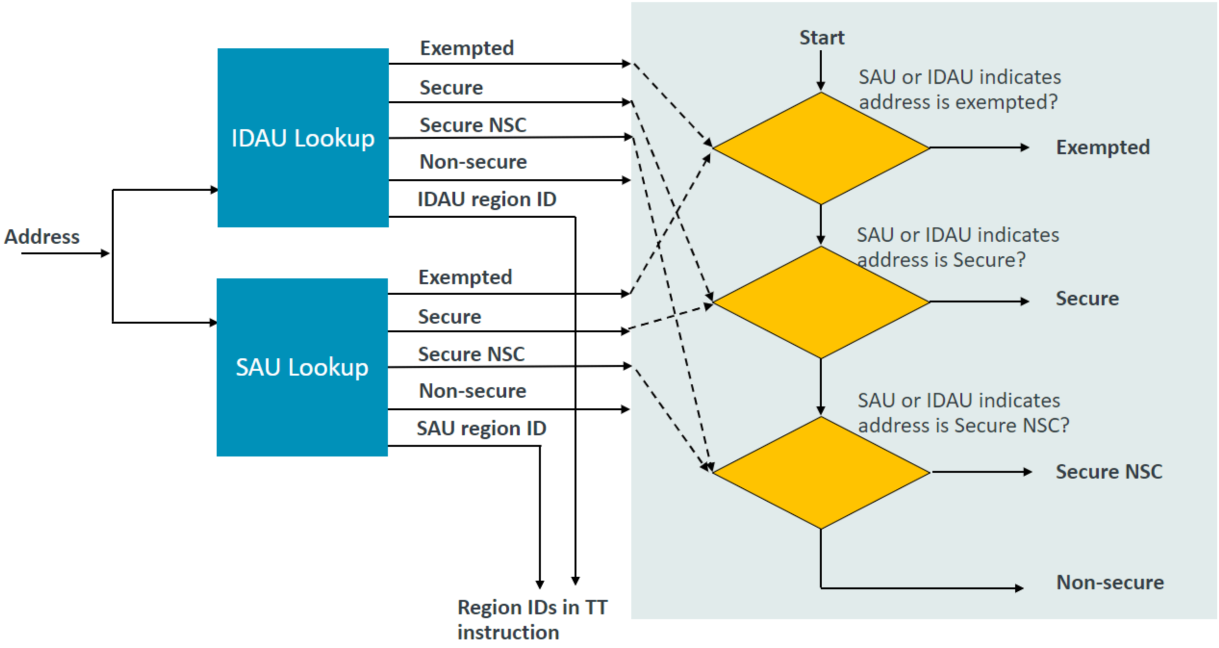
The following figure shows the SAU address lookup function.

Figure 3-1: SAU address lookup



When both SAU and IDAU are implemented in a system, then the address lookup occurs from both as the following figure shows.

Figure 3-2: Combined SAU and IDAU Address lookup



Note

The addresses for registers inside the CPU defined as exempt by the SAU are not programmable

3.2.4 Configuring SAU using CMSIS

To program the SAU, Secure privileged software performs the following steps:

1. Read the SAU_TYPE register to find the number of available regions for the Armv8-M processor you are working with.
2. Use the SAU_RNR register to select a region for configuration. For example, in a system that allows 8 SAU regions, the software can write a `0x3` to SAU_RNR to select region three.
3. Configure the selected SAU region:
   a. Write the region base address into the SAU_RBAR
   b. Write the region limit address into the SAU_RLAR. Bits [4:0] of the limit address are defined as `0x1F`.

      The SAU_RLAR register also contains two additional fields, the NSC bit and the ENABLE bit:

      - The NSC bit determines whether a region is Non-secure or Secure NSC.

- The ENABLE bit determines whether the region is currently enabled or disabled. Repeat steps 2 and 3 for the remaining SAU regions to be programmed.

4. Add DSB to ensure that the register accesses are finished.
5. Enable SAU by using the SAU_CTRL.ENABLE bit.

CMSIS-Core pack provides the `partition_<device>.h` file which contains the `TZ_SAU_Setup()` function and related settings. SAU registers are set up in the `TZ_SAU_Setup()` function.

Example SAU region settings in the `partition_<device>.h` file are as follows:

```
//  Initialize SAU Region 0 Setup SAU Region 0 memory attributes
    #define SAU_INIT_REGION0    1
    #define SAU_INIT_START0   0x101FFC00    // Start address
    #define SAU_INIT_END0     0x101FFFFF    // End address
    #define SAU_INIT_NSC0       1           //Region is 0: Non-secure 1:Secure, Non-
secure Callable

//  Initialize SAU Region 1 Setup SAU Region 1 memory attributes
    #define SAU_INIT_REGION1    1
    #define SAU_INIT_START1   0x00000000
    #define SAU_INIT_END1     0x001FFFFF
    #define SAU_INIT_NSC1       0
```

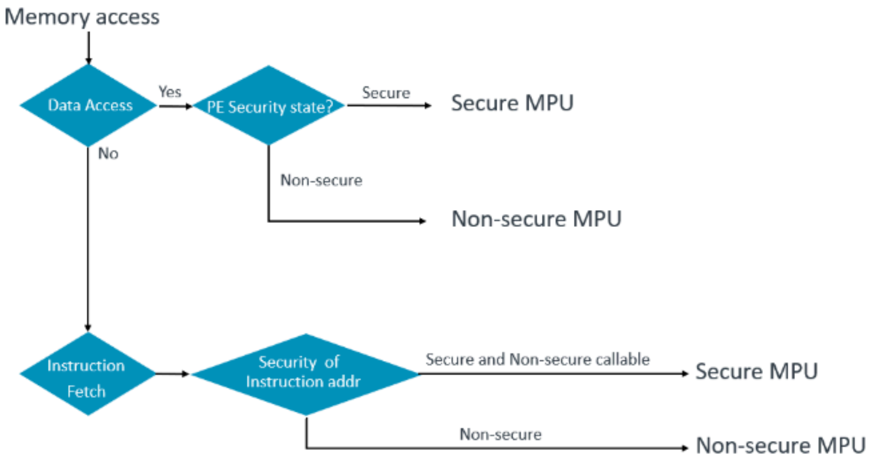
The `TZ_SAU_Setup()` function in `partition_<device>.h` uses these settings to configure SAU regions one by one. The combination of the IDAU and programmed SAU attributes must match the system memory map and the linker script, so that secret data is marked as Secure. For example, this ensures that a Secure peripheral is not accessible to the Non-secure state. In software programming, we recommend that you allocate a single software object, such as the stack, within one single SAU/IDAU region. This ensures that the TT instruction can be used to quickly check ranges of addresses.

3.3 Memory configuration with the MPU in Secure state

The processor contains a Secure MPU and a Non-secure MPU. The MPUs set memory attributes like cacheability, and provide several privileged and unprivileged permissions. See Armv8-M Memory Model and Memory Protection User Guide.

The following figure shows the address MPU lookup flow when both the MPU and SAU or IDAU are implemented in a system.

Figure 3-3: MPU in Security state



In software programming, we recommend that you allocate a single software object, such as the stack, within a single MPU region. This ensures that the TT instruction can be used to quickly check ranges of addresses.

4. Function calls

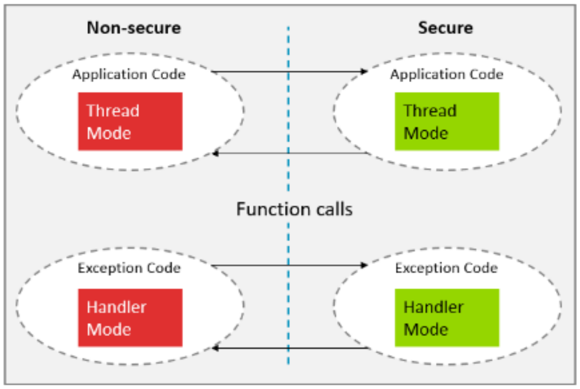
This chapter describes the following functionality:

- Security state transitions
- Function calls across the Security boundary
  - Parameters passing
  - Pointer passing
    - TT instruction and CMSE intrinsic functions
  - Floating point context consistency and FPCXT payload

4.1 Transition between security states

The Armv8-M Security Extension supports Security state transition by function call, as the following figure shows.

Figure 4-1: State transitions across security states



Function calls can cross the security boundary in both directions. That is, both Secure to Non-Secure calls and Non-Secure to Secure calls are supported. Although cross security boundary calls are supported in both thread and handler mode, they do not change the mode the CPU is in. For example, a function call in Secure Handler mode which targets the Non-Secure state, stays in Handler mode.

4.2 Implementing function calls across the Security boundary for C development

The Cortex-M Security Extension (CMSE) helps you to implement function calls across the Security boundary. CMSE is a toolchain extension which:

- Provides support for Secure software that is written in the C language
- Provides mechanisms to build and link the Secure and Non-Secure part of the software independently
- Generates Security Extension-related instructions such as SG, BXNS, `BXNS`
- Generates an import library that allows the Non-Secure image to be linked against the functions provided by the Secure image

The Non-Secure software build process does not need the CMSE toolchain support. The `<arm_cmse.h>` header must be included by the Secure software before using CMSE support. For information about CMSE, see CMSE.

The following section describes how to implement function calls across the Security boundary with the CMSE for C level development.

4.2.1 Non-Secure software calling a Secure API

To enable a Secure function to be called from the Non-Secure state follow these steps:

1. Include the `arm_cmse.h` file and add the CMSE function attribute `__attribute__(cmse_nonsecure_entry))` in a Secure function definition. The function can then be called from the Non-Secure state.

   The cmse_nonsecure_entry attribute causes the compiler to:

   - Automatically generate a veneer with the SG entry point gateway
   - Clear the registers of any secret data before returning
   - Cause the return operation to be performed with a BXNS instruction

   A simple Secure function is as follows:

```
#include <arm_cmse.h>
#include "secure_interface.h"
int __attribute__((cmse_nonsecure_entry)) entry1(int x)
{
    ...
}
```

2. Secure APIs are prototyped as normal in an interface header in a Secure project. For example:

```
Secure_interface.h:
int entry1(int x);
```

3. Group the entry point veneers into a section to define as Non-Secure callable (NSC). The following scatter file section shows how to define a NSC region and place the veneer section:

    ```
    LR_CMSE_VENEER 0x10100000 ALIGN 32 0x400
    {
        *(Veneer$$CMSE)
    }
    ```

    See Secure entry veneers.

4. Set up SAU regions so that veneer regions are marked as NSC. For more details on configuring SAU regions, see Configuring SAU using CMSIS.
5. Compile by using the command-line build option `-mcmse`. Add the linker command-line option to generate import library.
6. During the Non-Secure project build process, the linker must resolve the symbols and addresses of the Secure APIs entry point. The CMSE toolchain provides build options, such as `--import-cmse-lib-out` in armclang to generate the import library for a Non-Secure project.
7. In the Non-Secure project, perform the following steps without any CMSE support:
    a. Add and include the header file that declares the Secure functions that are callable from the Non-Secure state. In this example this is the `secure_interface.h` file.
    b. Link with the Secure import library generated in step above.
    c. Call the Secure APIs as a normal library function

    A simple example code is as follows:

    ```
    #include "secure_interface.h"
    int main(void)
    {
        int x = 100;
        entry1(x);
        ...
    }
    ```

4.2.2 Secure software calling a Non-Secure function

Secure software can call Non-Secure functions. The CMSE provides the function attribute `__attribute__((cmse_nonsecure_call))` to declare a function pointer that may cause a transition to the Non-Secure state. This attribute instructs the compiler to generate code including:

- Using the `BXNS` instruction for the function call
- Saving and clearing the registers that might contain secret data. To save time the floating point registers might be handled lazily.
- Restoring the registers after the Non-Secure function returns

Whether a function pointer with the cmse_nonsecure_call attributes actually causes a Secure to Non-Secure transition, or stays within the Secure state is determined an runtime by checking the

LSB of the pointer. This allows the same function in C to be used for both Secure and Non-Secure function pointers, causing a security state transition only when necessary.

During the Secure software development, the address location of the Non-Secure function is unknown. Function pointers are passed from Non-Secure software to Secure software via Secure APIs, when the Secure software receives the Non-Secure function pointer, Secure software can call the Non-Secure function when needed later.

The following is an example of a Secure software calling a Non-Secure function:

1. Define a function pointer type for Non-Secure function with the CMSE attribute `__attribute__((cmse_nonsecure_call))` in a Secure project
2. Define a Secure function that takes a pointer to the Non-Secure function to call. Use the `cmse_nsfptr_create()` intrinsic to mark the pointer as coming from the Non-Secure state.
3. The Non-Secure function will be called later in a Secure project.

   Example code is as follows:

   ```
   typedef int __attribute__((cmse_nonsecure_call)) nsfunc(int);
   nsfunc -ns_callback = 0;

   int __attribute__((cmse_nonsecure_entry)) ns_callable_fn(nsfunc- callback)
   {
      ns_callback = (nsfunc -)cmse_nsfptr_create(callback);
      ...
   }

   void Secure_fn(void)
   {
    int indata = 0;
    ...
      ns_callback(indata);
      ...
   }
   ```

   -cmse_nsfptr_create()-#

   This intrinsic function returns the value of the callback function pointer, with its LSB cleared. You can find more details about this in section CMSE intrinsic functions. In this example the Non-Secure function pointer is called from a different function, then cmse_nsfptr_create() should be called in the function where the Non-Secure pointer is passed into the Secure state, and not the one where it is called.

   The security_func_call_params_passing example in [Use case examples] shows you how to implement function call with function pointer as argument and call the Non-Secure function in Secure state.

4. The Non-Secure software implements Non-Secure functions as usual, and passes this function pointer to Secure side when Non-Secure software calls the Secure API. A simple example is as follows:

   ```
   #include "secure_interface.h"

   int func_ns (int x)
   {
   ```

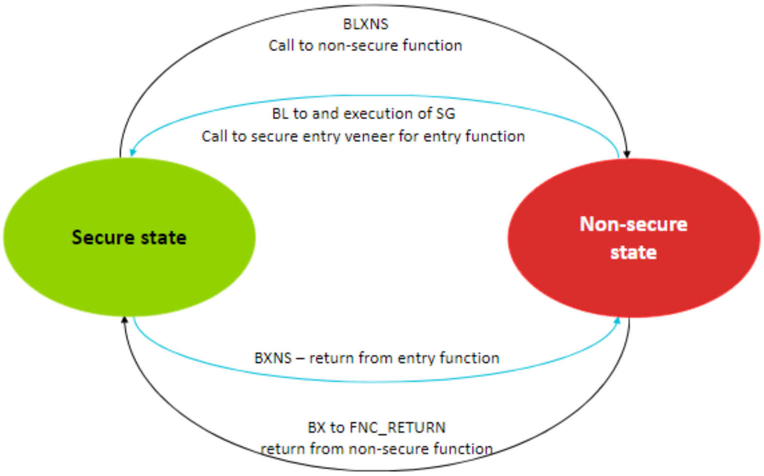
```
    ...
}

int main(void)
{
    ns_callable_fn(func_ns);
        ...
}
```

4.3 The assembly instructions for Security states transition by function call

The following figure shows the assembly instructions for Security states transition by function call.

Figure 4-2: Security switch with Function call


BLXNS
Call to non-secure function
BL to and execution of SG
Call to secure entry veneer for entry function
Secure state
Non-secure state
BXNS – return from entry function
BX to FNC_RETURN
return from non-secure function

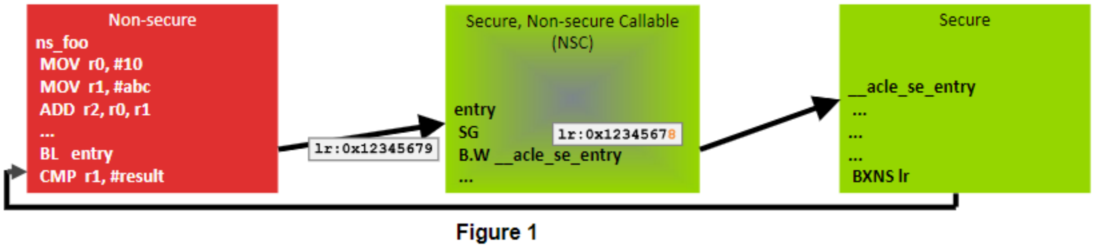
- `SG`: Secure gateway. Used for switching from Non-Secure to Secure state as the first instruction of Secure entry veneer.
- `BXNS <Rn>`: Branch with exchange to Non-Secure state if Rn[0] = 0. Used by Secure software to return from a Security entry point function.
- `BXNS <Rn>`: Branch with link and exchange to Non-Secure state if Rn[0] = 0. Used by Secure software to call Non-Secure functions.
- `FNC_RETURN`: When a `BXNS` instruction causes a transition to the Non-Secure state, it places a reserved address called FNC_RETURN in the link register. The Non-Secure function triggers the return to the Secure state by branching to this address, for example, with `BX LR`.

4.3.1 Function call from Non-Secure to Secure state

Function calls from Non-Secure to Secure state can be initiated by a Non-Secure software function branching to a Secure gateway.

Figure 4-3 shows how a Non-Secure program calls a Secure API.

Figure 4-3: Non-Secure call Secure



```
ns_foo
 MOV  r0, #10
 MOV  r1, #abc
 ADD  r2, r0, r1
 ...
 BL   entry
 CMP  r1, #result
```

```
entry
 SG
 B.W __acle_se_entry
 ...
```

```
__acle_se_entry
 ...
 ...
 ...
 BXNS lr
```

The process is as follows:

1. Non-Secure code calls a Secure API. The Secure address location through which the Non-Secure code branches into the Secure API is called Secure entry veneer. The Secure entry veneer must be in the Non-Secure Callable (NSC) memory region with the first instruction `SG`.
2. If `SG` is executed in the Non-Secure state, as the figure above shows:
    - Bit 0 of the return address in LR is set to 0.
    - Bit 0 of the return address can therefore be trusted to indicate which Security state the Secure function must return to when it is complete.
3. After `SG` instruction is executed from the NSC memory, the processor state switches to Secure.

Note

`SG` is treated as a No Operation (**NOP**) if it is fetched from Non-Secure memory.

If a Non-Secure program tries to branch or call a Secure program address without using a valid entry gateway, a SecureFault event is generated. On Cortex-M processors built with Armv8-M Baseline architecture, such as Cortex-M23, SecureFault is permanently disabled, so a Secure HardFault is generated instead.

4.3.1.1 Secure entry veneers

Secure entry veneers consist of an SG instruction followed by a B.W instruction that targets the entry function in Secure memory.

An example veneer code is as follows:

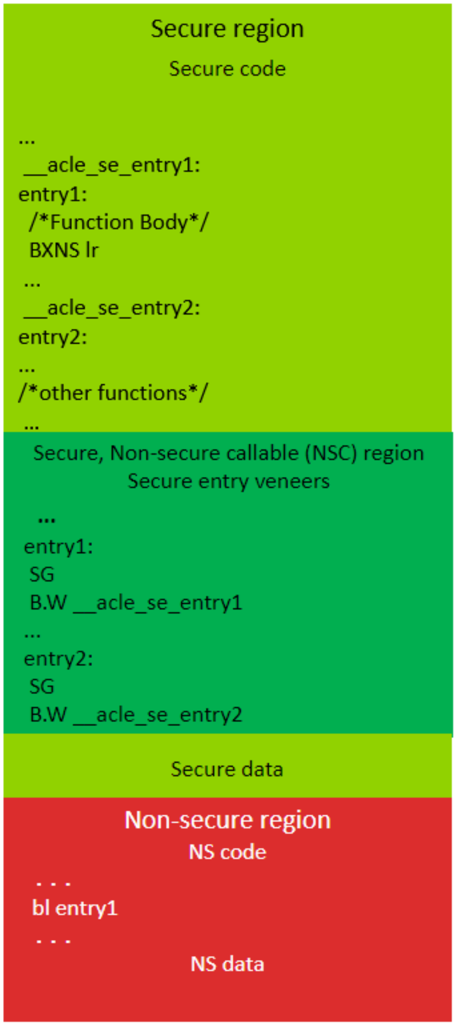
~~~
```
entry1
    0x10100000:    e97fe97f    ....    SG        ;
    0x10100004:    f702bae0    ....    B.W        __acle_se_entry1 ;
```
~~~

The Secure APIs are prefixed with __acle_se to indicate that the symbols point to the body of Secure entry functions:

~~~
```
__acle_se_entry1:
entry1:
    ...
    bxns lr;
```
~~~

Secure entry veneers allow Non-Secure code to call Secure APIs via SG instructions. The following figure shows the relationships between Non-Secure code, veneer code, and Secure code.

Figure 4-4: Secure entry veneers



```
...
 __acle_se_entry1:
entry1:
 /*Function Body*/
 BXNS lr
 ...
 __acle_se_entry2:
entry2:
...
/*other functions*/
 ...
```

```
  ...
entry1:
 SG
 B.W __acle_se_entry1
...
entry2:
 SG
 B.W __acle_se_entry2
```

```
 . . .
bl entry1
 . . .
```

4.3.2 Function return from Secure state

As the above figure shows, after Secure API completes, it uses a BXNS instruction to return back to the Security state it was called from, as shown by bit 0 of the return address. Before executing the BXNS instruction, all sensitive data must be cleared from the caller saved registers. For example, if the floating point registers are not used, these are R0-R3, R12, LR, and APSR.

4.3.3 Function call from Secure to Non-Secure state

A function call from Secure to Non-Secure state can be initiated by Secure software using the BXNS instruction that has the Least Significant Bit (LSB) of the target address set to 0.

The following figure shows how a Secure program calls a Non-Secure function.

Figure 4-5: Secure call Non-Secure

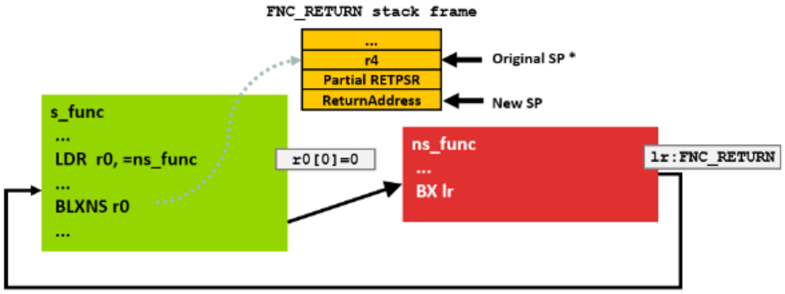


Figure 2

Steps for a Secure program to call a Non-Secure function are as follows:

1. Before executing the `BXNS` instruction to switch to Non-Secure state, Secure software must save and clear all callee and caller saved registers except the following registers:
   - Link Register (LR)
   - Registers that hold arguments for the Non-Secure function
   - Registers that do not hold secret information
2. The function call is performed using a `BXNS` instruction. If bit 0 of the function pointer is 0 the instruction transitions to the Non-Secure state. If a transition to the Non-Secure state is NOT requested, then all the operations below are skipped. Instead, a normal branch is performed.
3. The function return address is pushed into the FNR_RETURN stack frame on the Secure stack to hide it from Non-Secure code. This prevents Non-Secure code from modifying the Secure return address. The processor also pushes what is known architecturally as the Partial RETPSR into the Secure stack.
4. If the processor is in Secure Handler mode, IPSR is set to 1. Non-Secure code has no knowledge of which exception was run previously.
5. The processor stores the value called FNC_RETURN into the LR.

4.3.4 Function return from Non-Secure state

As the above figure shows, after the Non-Secure function finishes execution, it returns to caller function. The return operation, for example, `BX LR`, loads the FNC_RETURN value into the Program Counter (PC). 1. Integrity checks are performed. For example, one check is that the current mode of the processor matches the mode indicated by the RESPSR value saved in the FNC_RETURN stack frame. Any failed integrity check on function return generates Fault. 2. If integrity checks are successful, the real ReturnAddress is unstacking from the FNC_RETURN stack frame on the Secure stack. 3. The processor switches back to Secure state.

4.4 Software considerations in function calls across security boundary

This section describes key software considerations during function calls across security boundary.

4.4.1 Pointer passing across Security boundary

Pointer passing in function calls across the Security boundary is supported. Several Security issues can occur from passing pointers across Security state boundaries, including: 1. The Non-Secure state might be able to pre-empt Secure code and modify the data referenced by a pointer passed to a Secure function. This might break assumptions a programmer makes about the data not changing, for example between validating it and using it. 2. The Non-Secure state can pass a pointer to Secure memory to a Secure function. If the Secure code uses this pointer without validating it, it might be tricked into performing an operation on behalf or Non-Secure code that the Non-Secure code does not have permission to do directly. This is more commonly known as the confused deputy attack.

Note

This can happen when either:

```
- The Non-Secure software tries to trick the Secure code into accessing
 Secure data
- The unprivileged Non-Secure software tries to trick the Secure code
 into accessing privileged Non-Secure data
```

The Armv8-M Security Extension defines `TT` instructions and CMSE-compliant compiler toolchain support CMSE intrinsics. [Use Case Examples] describes how to use these intrinsics to check pointer arguments in Secure software.

4.4.1.1 Data Pointer validation

Secure APIs perform operations on behalf of Non-Secure software, such as cryptographic data processing. Non-Secure software passes data pointers to the Secure software to indicate where the data sources are and where to put the processing results. When a Secure API gets the data pointer from Non-Secure side, it must check that Non-Secure software can read or write this memory before accessing it.

Non-Secure memory can be changed asynchronously during the execution of a Secure API. When a Secure API is executed, the processor is interrupted to service Non-Secure interrupts. The Non-Secure data that the pointer points to can be accessed and modified by the Non-Secure interrupt handler. The following code shows the problem:

```
    int array[N];
    void __attribute__((cmse_nonsecure_entry)) SecureFunc1(int -p)
    {
        // No checks performed on Non-Secure pointer before it is dereferenced.
        if (-p >= 0 && -p < N)
```

```
       {
           /- Non-Secure memory (-p) may be changed at this point so that
               -p is outside the range of the array. For example,
               by a Non-Secure interrupt handler.
           -/
           ...
           array[-p] = 0;
       }
   }

   void __attribute__((cmse_nonsecure_entry)) copy(int -src, int -dest, int len)
   {
       for ( int i = 0; i < len; i++)
       {
       /- if Non-Secure code passed Secure addresses to either src or dest
              then this code would allow the Non-Secure state to bypass the memory
protection
              by getting the Secure code to copy secret data into Non-Secure
memory,
              or by overwriting Secure memory
       -/
           dest[i] = src[i];
       }
   }
```

The following code shows a safe example:

```
   int array[N];
   void __attribute__((cmse_nonsecure_entry)) SecureFunc1(int -p)
   {
       int index = 0;
       volatile int -psafe = NULL;
       /- The cmse_check_pointed_object() intrinsic checks that p points to
            Non-Secure memory and Non-Secure MPU indicates that it is readable in
              the current mode.
       -/

       psafe = cmse_check_pointed_object(p, CMSE_NONSECURE|CMSE_MPU_READ);
       if ( psafe != NULL)
       {
           /-  without the volatile keywork to declare psafe, the compiler may
choose
                   not to copy the value to index variable,
                   and could still access Non-Secure memory multiple times.
           -/
           index = -psafe;   // copy the value from Non-Secure to Secure memory
           ...
           /-  even if the data pointed to by psafe changes,
                   then all references to index here get the same value
                   because we have a local copy
           -/
           if ( index >= 0 && index < N )
           {
               array[index] = 0;
           }
       }
       ...
   }

   void __attribute__((cmse_nonsecure_entry)) copy(int -src, int -dest, int len)
   {
       int -srcsafe = NULL;
       int -destsafe = NULL;
       srcsafe = cmse_check_address_range(src, len, CMSE_NONSECURE |
CMSE_MPU_READ );
       destsafe = cmse_check_address_range(dest, len, CMSE_NONSECURE |
CMSE_MPU_READWRITE );
       if ((srcsafe != NULL) && (destsafe != NULL))
```

```
        {
            for ( int i = 0; i < len; i++ )
            {
                destsafe[i] = srcsafe[i];
            }
        }
        ...

    }
```

You must also consider other usage scenarios for data pointer validation across the security boundary:

- When passing a data pointer from the Non-Secure side, permission checks on the objects pointed to by the pointer can be performed. These checks use the address range checking intrinsic, cmse_check_address_range() and cmse_check_pointed_object().
- When dealing with complex software objects like linked lists, software developers must use the address range checking intrinsic to verify each linked list object individually.
- If the return values from Non-Secure to Secure functions involve complex software objects, the Secure function must use the address range checking intrinsic to verify the permissions.

4.4.1.2 Function pointer checking

For function pointer checking:

- No special checking is needed when a function pointer which remains within the Secure state, and always points to a Secure function. Call the function normally.
- When a function pointer passes to a Secure entry point that is always called from the Non-Secure state, software developers must use the cmse_nsfptr_create() function to clear the LSB of the Non-Secure function pointer.
- A function pointer can pass to a Secure entry point which can be called by both the Secure state and the Non-Secure state. We recommend that you use cmse_nonsecure_caller() to check whether the Secure entry point function has been called from the Non-Secure state or Secure state. Only use the cmse_nsfptr_create() function when the Secure entry point is called by the Non-Secure state.

4.4.2 Non-pointer parameter passing

Non-pointer parameter passing in function calls across the Security boundary is supported for Secure APIs or Non-Secure functions calls that have more arguments than can fit in registers. You can pass a pointer to a struct containing all the arguments. The security_func_call_params_passing example in Armv8-M Security Extension use case examples describes how to use a struct to pass more arguments during a function call.

4.4.3 CMSE intrinsic functions

CMSE provides intrinsics to check the memory attributes within C/C++ code:

```
    cmse_address_info_t cmse_TT(void -p)
```

The intrinsic cmse_TT returns the memory attributes for a single address in memory:

```
    void -cmse_check_address_range(void -p, size_t size, int flags)
```

The cmse_check_address_range intrinsic checks that the specified address range meets the access permissions outlined by the flags. It returns NULL on a failed check and `-p` on a successful check.

```
    void -cmse_check_pointed_object(void -p, int flags)
```

The cmse_check_pointed_object intrinsic checks that the specified object meets the access permissions outlined by the flags. Returns NULL on a failed check and `-p` on a successful check.

When using these intrinsic functions, the access permission condition needs to be specified by using the flags parameter. The flag values are defined in CMSE using C macros.

Macro	Description
CMSE_MPU_UNPRIV	Forces the check to be done using unprivileged permission. Without this flag, the current mode and CONTROL.nPRIV flag for the security state corresponding to the MPU selected are used to determine the privilege level.
CMSE_MPU_READWRITE	Checks if the readwrite_ok field is set in the permission of an address
CMSE_MPU_READ	Checks if the read_ok field is set in the permission of an address
CMSE_AU_NONSECURE	Checks if the Secure field is unset in the permission of an address
CMSE_MPU_NONSECURE	Checks the permission of an address using the Non_secure MPU
CMSE_NONSECURE	The combined semantics of CMSE_AU_NONSECURE and CMSE_MPU_NONSECURE

The address range checking intrinsics work assuming that the configurations of the SAU, IDAU, and MPU are constrained as follows:

- An object or an address range to be checked is allocated in a single region.
- A stack is allocated in a single region.
- A region does not overlap other regions.

The address range checking intrinsics use TT instruction to return an SAU, IDAU, and MPU region number. When the region numbers of the start and end of the address range match, the complete range is in one SAU, IDAU, and MPU region. See TT instruction for information on TT instruction checks across the region boundary.

CMSE also provides Non-Secure function pointer intrinsic functions listed as follows:

- cmse_nsfptr_create(p): Makes a function pointer as Non-Secure so it can only be used to call back to the Non-Secure state.

- cmse_is_nsfptr(p): Checks whether a given function pointer value should be interpreted as a Non-Secure function address.
- cmse_nonsecure_caller(): Returns non-zero if the Secure function is called from the Non-Secure state and zero otherwise.

For details about these functions, see TT instruction intrinsics.

4.4.4 TT instruction

The Armv8-M architecture defines the Test Target (TT) instruction. The TT instruction is used to query the Security state and access permission of a memory location.

TT instruction variant	Description
TT (Test Target)	Queries the Security state and access permissions of a memory location
TTT (Test Target Unprivileged)	Queries the Security state and access permissions of a memory location for unprivileged access to that location
TTA (Test Target Alternate domain)	Query the Security state and the Non-Secure MPU setting to get access permissions of a memory location for a Non-Secure access to that location. Only valid when executing in Secure state. **UNDEFINED** if they are used from Non-Secure state.
TTAT (Test Target Alternate domain Unprivileged)	Query the Security state and the Non-Secure MPU setting to get access permissions of a memory location for a Non-Secure unprivileged access to that location. Only valid when executing in Secure state. **UNDEFINED** if they are used from Non-Secure state.

TTA and TTAT instructions are only valid when executing in Secure state, the instructions are **UNDEFINED** if they are used from Non-Secure state.

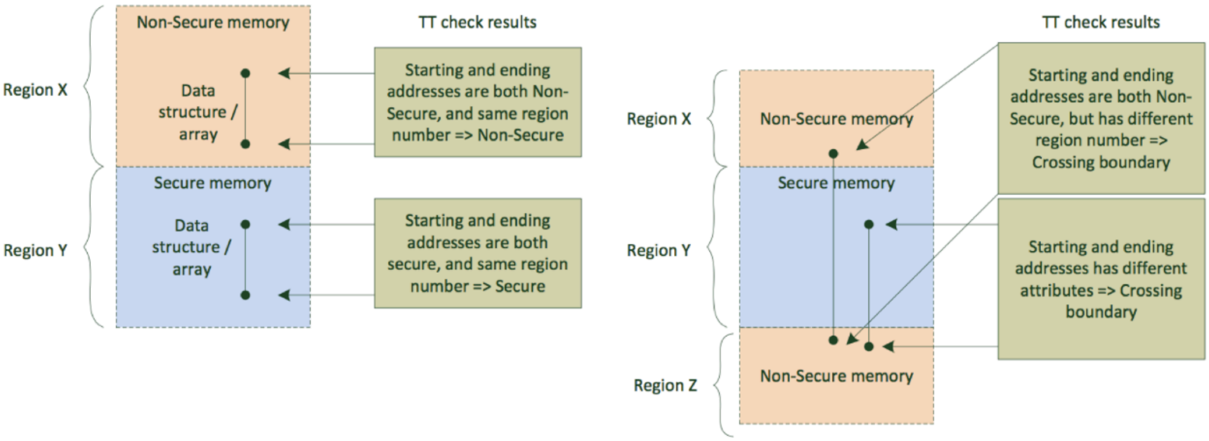
TT_RESP payload provides the response information from a TT, TTA, TTT, or TTAT instruction. *Armv8-M Architecture Reference Manual* defines the TT_RESP bitfields.

For each memory region defined by the SAU, IDAU, and MPU, there is an associated region number that is generated by the SAU, IDAU, or MPU. Software uses this region number to determine if a contiguous range of memory shares common security attributes. The TT instruction returns the attributes and region numbers for an address value. For data array or data structure, software can quickly determine that the memory range is located in the same SAU, IDAU, and MPU region by:

- Using a TT instruction on the start and end addresses of the memory range
- Identifying that both reside in the same region number

The following figure shows TT instruction checks across the region boundary.

Figure 4-6: TT instruction checks across the region boundary


Region X
Region Y
Non-Secure memory
Data structure / array
Secure memory
Data structure / array
TT check results
Starting and ending addresses are both Non-Secure, and same region number => Non-Secure
Starting and ending addresses are both secure, and same region number => Secure
Region X
Region Y
Region Z
Non-Secure memory
Secure memory
Non-Secure memory
TT check results
Starting and ending addresses are both Non-Secure, but has different region number => Crossing boundary
Starting and ending addresses has different attributes => Crossing boundary

Note

Software needs to check that memory addresses configured in SAU, IDAU, and MPU regions are not overlapping with each other because this affects the range checking mechanism implemented by TT instruction.

Using this mechanism, Secure code can determine if the memory referenced by a pointer from Non-Secure software has the appropriate security attribute. This prevents Non-Secure software from using Secure software to read out or corrupt Secure or privileged information it does not have access to.

4.5 Floating-point context consistency and FPCXT payload

The floating-point context, stored in FPSCR, is not banked between the security states. In Armv8.1-M, the floating-point context payload, FPCXT_S and FPCXT_NS, is added to enable floating-point context to be saved/restored during function calls with Security state switch.

Both FPCXT_NS and FPCXT_S can only be accessible in the Secure state. For more details on FPCXT bitfield, See *Armv8.1-M Architecture Reference Manual*.

FPCXT_S enables saving and restoring the floating-point context around a call to a Non-Secure function from the Secure state. It also initializes the floating-point context ready for the Non-Secure code. FPCXT_S only needs to be used if the Secure state has used floating point instructions. FPCXT_S saves and restores the context if this has not been performed by other instructions, for example, if VLSTM or VLLDM are not used because floating-point arguments are passed to the Non-Secure function.

FPCXT_NS enables saving and restoring the floating-point context and the start/end of a Secure entry point function. This only needs to be done if the Secure entry point function or the other functions it calls might use floating point instructions.

The following example code shows a Secure function secureFunc1() which calls nonSecureFuncPtr. nonSecureFuncPtr is a pointer to a Non-Secure function with a floating-point parameter.

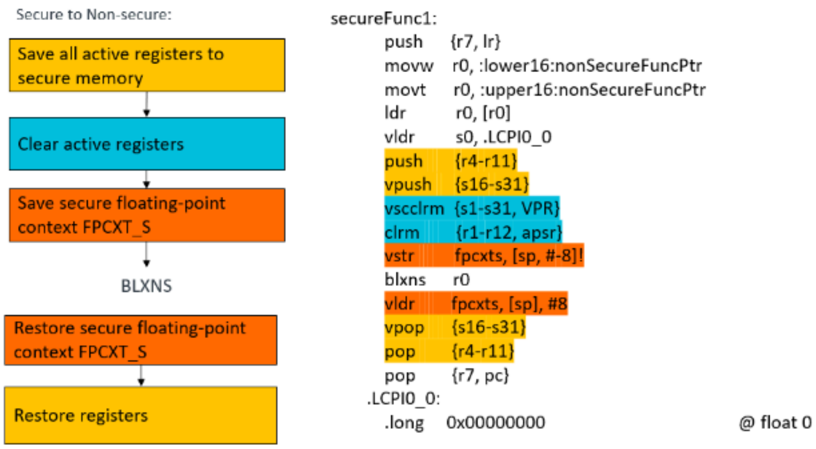
```
#include <arm_cmse.h>

void __attribute__((cmse_nonsecure_call)) (-nonSecureFuncPtr)(float a);
void SecureFunc1()
{
    nonSecureFuncPtr(0.0);
}
```

Build the example code with the `-march=armv8.1-m.main -mfpu=fpv5-sp-d16 -mfloat-abi=hard -mcmse` build options.

When –secureFunc1– calls –nonSecureFuncPtr–, it causes Security state switch. The following figure shows the actions needed before switching from Secure to Non-Secure state, and the generated assembly code.

Figure 4-7: Secure to Non-Secure save flow


Secure to Non-secure:
Save all active registers to secure memory
Clear active registers
Save secure floating-point context FPCXT_S
BLXNS
Restore secure floating-point context FPCXT_S
Restore registers

```
secureFunc1:
        push    {r7, lr}
        movw    r0, :lower16:nonSecureFuncPtr
        movt    r0, :upper16:nonSecureFuncPtr
        ldr     r0, [r0]
        vldr    s0, .LCPI0_0
        push    {r4-r11}
        vpush   {s16-s31}
        vscclrm {s1-s31, VPR}
        clrm    {r1-r12, apsr}
        vstr    fpcxts, [sp, #-8]!
        blxns   r0
        vldr    fpcxts, [sp], #8
        vpop    {s16-s31}
        pop     {r4-r11}
        pop     {r7, pc}
    .LCPI0_0:
        .long   0x00000000                  @ float 0
```

The following example code shows a Secure entry point function secureAPI() which calls a sub function secureFunc2(), which can use floating point.

```
#include <arm_cmse.h>

extern void SecureFunc2(void);

void __attribute__((cmse_nonsecure_entry)) SecureAPI()
{
    SecureFunc2();
}
```

FPCXT_NS is saved before other operations in Secure side and restores FPCXT_NS before returning back to the Non-Secure state. VSCCLRM zeros the specified floating-point registers if there is an active floating-point context, as the following generated assembly code shows.

```
__acle_se_secureAPI:
SecureAPI:
```

```
             vstr    fpcxtns, [sp, #-8]!
             push    {r7, lr}
             bl      SecureFunc2
             pop     {r7, lr}
             vscclrm {s0-s15, vpr}
             vldr    fpcxtns, [sp], #8
             clrm    {r0-r3, r12, apsr}
             bxns    lr
```

Caution

SoftFP ABI use with Security extension `-mfloat-abi=softfp` uses floating point instructions. However, there is no requirement to save and restore the callee floating-point registers, S16-S31. Therefore, these registers can leak secret values when transitioning from Secure to Non-Secure. This applies to both function returns to Non-Secure, and when a secure exception is returning to Non-Secure background code. We recommend that you do not use the SoftFP ABI with the Security Extensions.

5. Armv8-M exception model with Security Extension

This guide describes exception handling functionality when Security Extension is implemented in a system. See Armv8-M Exception Model User Guide for details on how exceptions and interrupts are handled in the Armv8-M architecture when Security Extension is not implemented in a system.

The Security Extensions modify the exception model so that exceptions can be taken directly to the required security state, without compromising the security of the system. Some exceptions are banked between the security states, for example, MemManage Fault. This enables each state to operate independently. Other exceptions have a single instance that can be configured to target either security state, for example, external interrupts. The exception model also allows the priority of critical Secure exceptions to be protected so they cannot be blocked by Non-secure exceptions. The subsequent section explains about the details of the exception model along with security extension.

5.1 Prerequisites

We recommend that you fully understand the concepts that the Armv8-M Exception Model User Guide describes.

5.2 Target states of exceptions

The following table summarizes the configurability and default target states for interrupts and exceptions available in an Armv8-M processor.

Exception number	Exception type	Type	Default target state	Notes
1	Reset	Secure only	Secure	-
2	NMI	Configurable	Secure	Configurable by AIRCR.BFHFMNINS bit. However, AIRCR.BFHFNMINS bit should be set to 1 only when Secure state is unused and calling Secure functions or Secure exceptions should be strictly blocked.
3	HardFault	Configurable	Secure	Configurable by AIRCR.BFHFMNINS bit. However, AIRCR.BFHFNMINS bit should be set to 1 only when Secure state is unused and calling Secure functions or Secure exceptions should be strictly blocked
4	MemManage Fault	Banked	N/a	-
5	BusFault	Configurable	Secure	Configurable by AIRCR.BFHFMNINS bit. However, AIRCR.BFHFNMINS bit should be set to 1 only when Secure state is unused and calling Secure functions or Secure exceptions should be strictly blocked

Exception number	Exception type	Type	Default target state	Notes
6	UsageFault	Banked	N/a	-
7	SecureFault	Secure only	Secure	-
11	SVC	Banked	N/a	-
12	Debug monitor	Configurable	Depends on External Debug authentication signals	If Secure debug is enabled in the debug authentication interface, it will target the Secure state, else Debug Monitor exception targets Non-secure state
14	PendSV	Banked	N/a	-
15	SysTick	Banked or Configurable	Banked if two SysTick timers are available. If there is only one SysTick timer, by default the SysTick exception targets Secure state.	When one SysTick timer is used, then ICSR.STTNS bit is used to configure the target state of SysTick. Two SysTick timers are always present on Mainline CPUs like Cortex-M33 and greater.
16 - 495	External interrupts	Configurable	Secure	Using NVIC_ITNS register the interrupts can be configured to either Secure or Non-secure state.

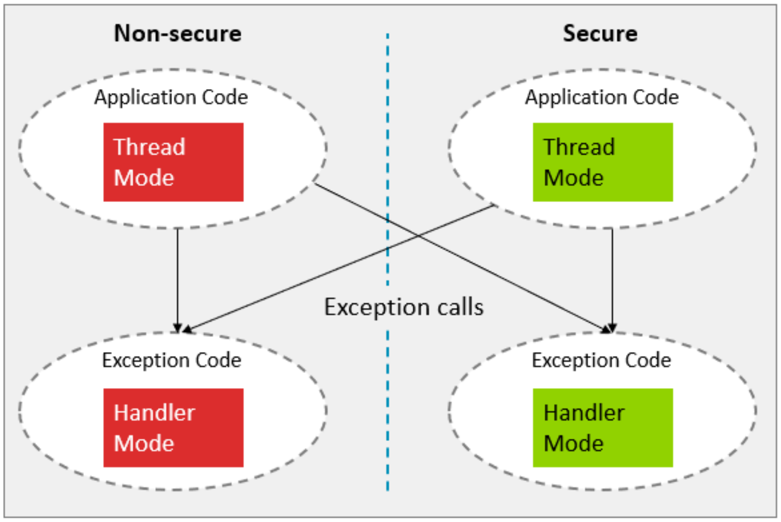
If you implement Security Extension in a system, then:

- Some of the system exceptions are banked. This means that there are both Secure and Non-secure versions of these system exceptions. You can trigger these banked system exceptions independently in a particular security state with different priority level settings.
- Each security state has its own vector table containing the addresses of the exception handlers associated with that security state.
- The exception handler is executed in the security state associated with the exception. As a result, handlers must be stored in the correct type of memory. For example, a SecureFault handler must be stored in Secure memory.
- Secure exception handlers do not need to start with an SG instruction because the processor automatically switches to the security state associated with an exception. The processor must not have the `cmse_nonsecure_entry` attribute.

You must be able to configure their interrupts to target either Secure or Non-secure states. This is because there can be end user peripherals in either Secure or Non-secure memory space. With the external interrupts, the system exceptions, such as SysTick and PendSV, must be handled correctly in either Secure or Non-secure state.

You can configure each external interrupt as Secure or Non-secure in the Interrupt Target Non-secure (NVIC_ITNS) register. This register is only programmable in the Secure world. When taking an exception, the CPU transitions directly to the security state associated with that exception. See the following figure.

Figure 5-1: Security switch through Exceptions



If the arriving exception or interrupt has the same security state as the current processor state, then the exception sequence is almost identical to the Armv7-M/Armv6-M based processors. This enables low interrupt latency. The main difference occurs when a Non-secure interrupt takes place, and is handled by the processor during execution of Secure code. In this case, the processor automatically pushes all Secure information onto the Secure stack and erases the contents of the registers. This avoids an information leak.

There is a slightly longer interrupt latency because all Secure contents must be pushed onto the stack before going to Non-secure exception from Secure code. All existing interrupt handling features, such as nesting of interrupts, tail-chaining, vectored interrupt handling, and vector table relocation, are supported. The Security Extension enhancement of the exception model also works with the lazy stacking of registers in the Floating-Point Unit (FPU), especially when saving the Secure Floating-point context.

5.3 Exception prioritization

Each exception in the Armv8-M architecture has an exception number and an exception priority associated with it. When a higher priority exception occurs, it preempts the lower priority code that is getting executed. For details on exception priority level definitions and its associated features like priority grouping, see Exception priority level definitions in Armv8-M Exception Model User Guide.

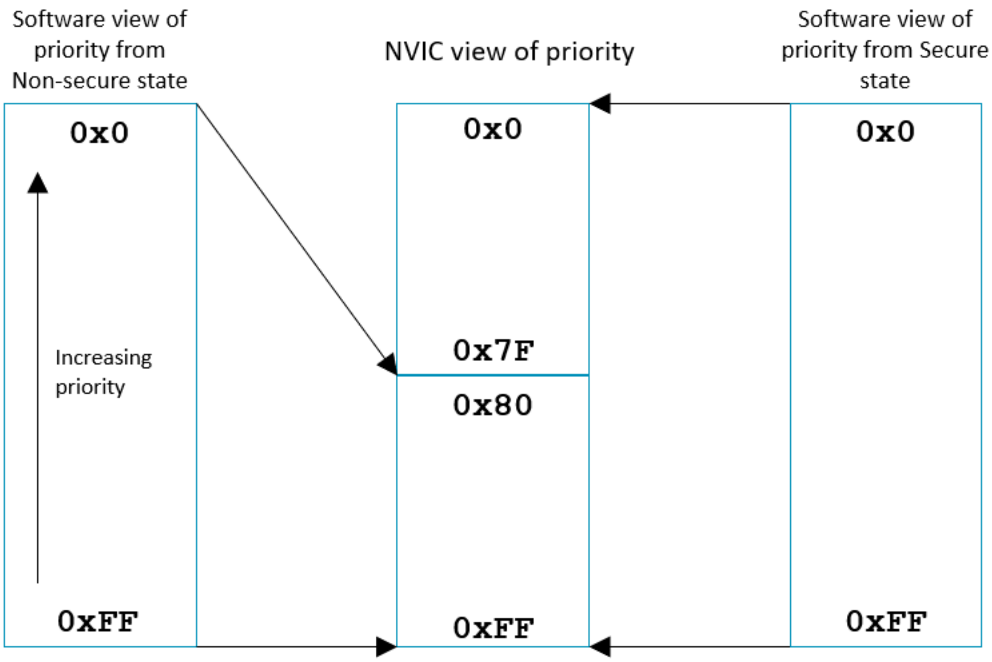
When Security extensions are included in a system, it can be important to be able to guarantee that critical Secure exceptions are always given higher priority than the Non-secure exceptions. However, you can still configure Secure exceptions to be in the same priority range as Non-secure exception. You can control the Secure exceptions prioritization using a programmable bit in the AIRCR register called PRIS. By default, AIRCR.PRIS bit is set to 0. This setting means that both Secure and Non-secure exceptions share the same configurable priority level, `0x0` to `0xFF`.

Note

A lower numerical value denotes a higher priority.

When AIRCR.PRIS bit is set to 1, then the effective priority of Non-secure exceptions is mapped to the lower half of the priority range. See the following figure. As a result, higher priority Secure exceptions, those with priority values in the range `0x0` to `0x7F`, cannot be blocked or pre-empted by Non-secure exceptions.

Figure 5-2: NVIC view of priority


Software view of priority from Non-secure state
NVIC view of priority
Software view of priority from Secure state
0x0
Increasing priority
0xFF
0x0
0x7F
0x80
0xFF
0x0
0xFF

Note

You must not allow Non-secure code to execute after Secure faults have been raised. Secure faults are Secure MemManage fault, Secure BusFault, SecureFault and Secure UsageFault. These faults can occur as the result of an attempted attack. Such an attack might be able to bypass the security measures built into the architecture if the faults are not handled correctly. For example, you must prevent further Non-secure code execution by restarting the system. Because of this reason, you must set the priorities of these faults so that even the highest priority Non-secure interrupt cannot preempt the Secure fault handlers. To do this, either:

- Set AIRCR.PRIS bit to 1 so that all Non-secure exceptions are in the priority range of `0x80->0xFE`. Also, make sure that all Secure faults are assigned priorities in the range 0->`0x7F`.

- If AIRCR.PRIS is set to 0, then ensure that no Secure faults are enabled. This is to make sure that all Secure faults escalate to a Secure HardFault at a negative execution priority and so Non-secure exception at priority 0 cannot preempt.

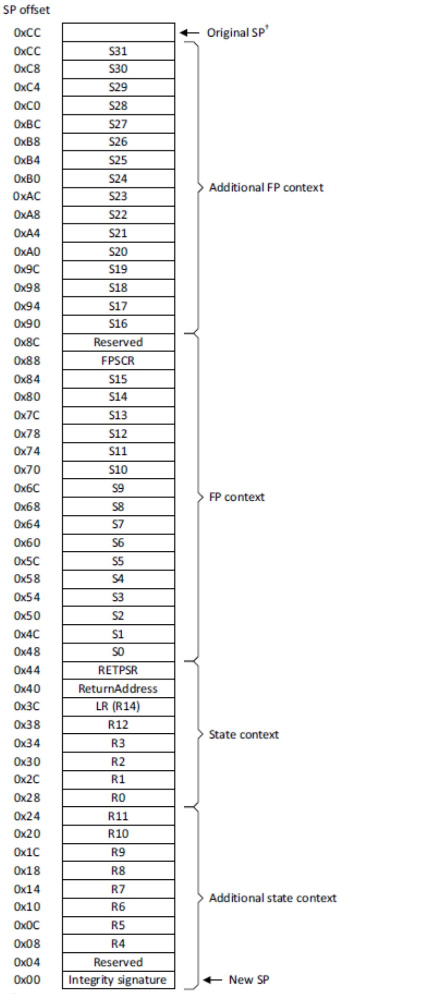
5.4 Stack frames

The exception stack frame formats are not relevant to most software. However, you need to know the formats of some parts of an operating system, such as the SVC handler, which get its arguments from the exception stack frame. Also, when debugging the cause of a fault it is useful to understand the format of the stack memory contents.

For processors implementing the Armv8-M architecture, exception handling methods are transparent to the end user software. For an exception, automatic stacking and unstacking processes are handled by the processor hardware. See Exception handling sequences for more details.

The format of the exception stack frame used by the hardware depends on what data needs to be preserved, and the type of security transition being performed. This ensures that Secure register values are not visible to Non-secure exception handlers. The following figure shows the stack frame format that is possible on various exception handling mechanisms.

Figure 5-3: Stack frame format



† Or at offset 0xD4 if at a word-aligned but not doubleword-aligned address.

The following table summarizes the stack frame type that is used on a security state transition through an exception.

Background execution state	Exception target state	FP context active	FPCCR_S.TS	State context: integer caller saved	Additional state context: integer callee saved	FP context: FP caller saved	Additional FP context: FP callee saved)
Non-secure	Non-secure	No	N/a	Yes	No	No	No
Non-secure	Non-secure	Yes	N/a	Yes	No	Yes	No
Secure	Secure	No	N/a	Yes	See EXC_RETURN.DCRS	No	No
Secure	Secure	Yes	0	Yes	See EXC_RETURN.DCRS	Yes	No
Secure	Secure	Yes	1	Yes	See EXC_RETURN.DCRS	Yes	Yes
Non-secure	Secure	No	N/a	Yes	No	No	No
Non-secure	Secure	Yes	N/a	Yes	No	Yes	No

Background execution state	Exception target state	FP context active	FPCCR_S.TS	State context: integer caller saved	Additional state context: integer callee saved	FP context: FP caller saved	Additional FP context: FP callee saved)
Secure	Non-secure	No	N/a	Yes	Yes	No	No
Secure	Non-secure	Yes	0	Yes	Yes	Yes	No
Secure	Non-secure	Yes	1	Yes	Yes	Yes	Yes

When taking a Secure exception from Secure background code, the stacking of the additional state context is not required to maintain the security of the system. However, in some cases the additional state context might still be stacked, for example if Secure exception has tail-chained from a Non-secure exception that required the extra stacking. The EXC_RETURN.DCRS flag indicates if additional state context has been stacked in this case

The stack frame gets pushed onto the stack belonging to the background state that was pre-empted. It can be MSP or PSP depending on software configuration. Also,the exception being entered uses the MSP associated with the security state of the exception.

5.4.1 State context

A state context or basic exception stack frame contains 8 words of data, known as “caller-saved registers”. These are in the integer register bank which is often known as the basic integer context or basic stack frame. This basic exception stack frame is identical to the stack frame used in the Armv6-M/Armv7-M architecture without floating-point extensions. For more details on basic integer context, see Stack frames in Armv8-M Exception Model User Guide.

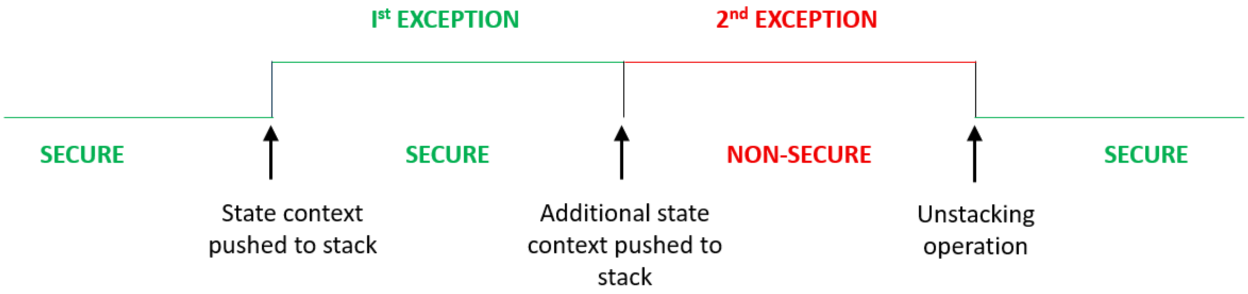
An extended stack frame contains data from both basic floating-point context and basic integer context. The basic floating-point context includes caller-saved registers”from the floating-point register bank, that is S0-S15,FPSCR,VPR. This basic floating-point context can either be saved automatically during exception stacking process or by the lazy-stacking process. For more details on extended stack frame, see [Stack frames in Armv8-M Exception Model User Guide] (https://developer.arm.com/documentation/107706/0100/Exceptions-and-interrupts-overview/Stack-frames?lang=en).

5.4.2 Additional state context

The additional state context is also known as “callee-saved registers”. When an exception occurs during the execution of a Secure program code when the target state of the exception is to a Non-secure state, then the additional state context is saved during the exception stacking process. Using late-arrival or tail-chaining optimizations, sometimes you can also have this stack frame format when a Secure exception is taken and the background code is also Secure.

In addition to the extra register state, the additional state context also contains an integrity signature that is stored during the exception stacking process by the processor hardware. The integrity signature value is defined by the Armv8-M architecture as either `0xFEFA125A` or `0xFEFA125B`. Here, the value of 0 in the LSB states whether stack frame contains a floating-point context or not. The following figure shows a basic exception stack frame with integer additional context.

Figure 5-4: Stacking process in multiple stages



The additional state context is below the basic stack frame. This enables the additional state context to be appended to an existing stack frame. For example, the processer is running in a Secure program and receives two interrupt requests, (1) Secure interrupt and (2) Non-secure interrupt with Secure interrupt at higher priority. Then stacking of state context occurs initially on a Secure interrupt handler entry followed by just pushing the additional context when tail-chaining to the Non-secure interrupt.

When floating-point state needs to be protected, when FPCCR_S.TS = 1, and if an exception occurs when executing Secure code, then the stack frame contains both the FP context and the additional FP context.

5.5 EXC_RETURN

Usually, C functions are called using a branch instruction, `BL <function_call>`. The function return is usually carried out by a `BX LR` instruction. This instruction loads the return address saved in Link Register (LR/R14) in the Program Counter (PC). Armv8-M processors are designed so that the exception handlers can be written as C functions. To differentiate between regular function return and an exception return, on exception entry the processor writes a special token called EXC_RETURN into the link register instead of a normal return address. When the exception handler is complete, the last step of the interrupt handler loads the EXC_RETURN into the PC. This triggers the exception return process. For more details see EXC_RETURN in Armv8-M Exception Model User Guide.

When security extensions are not implemented in a system, then the EXC_RETURN token identifies the background context such as type of mode as Thread or Handler, and the stack pointer used in background context as MSP or PSP.

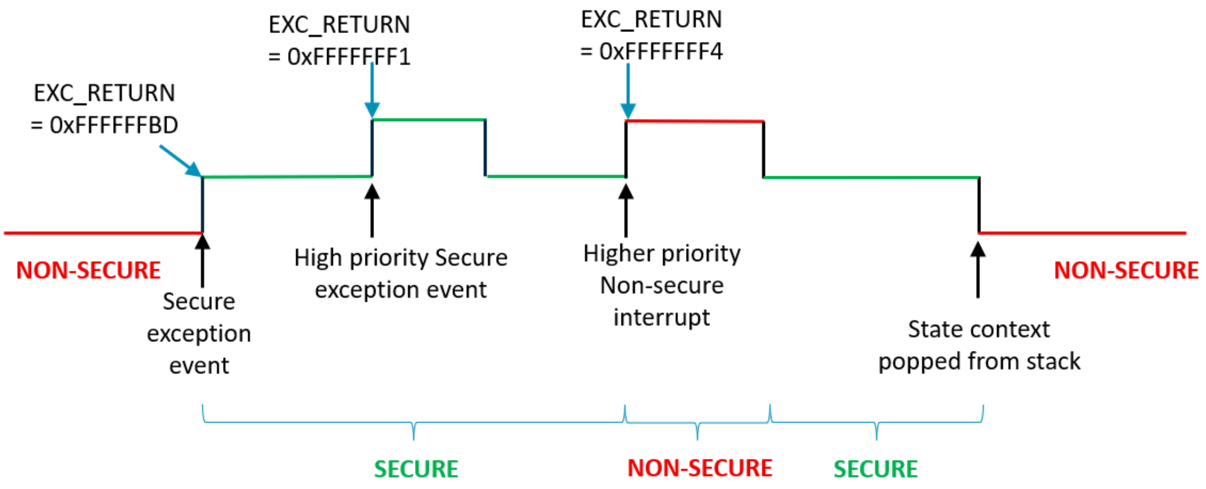
Also, when Security extensions are implemented the EXC_RETURN token is extended to include following information:

Name	Position	Description
S	6	Indicates whether the exception stack frame is on a Secure or a Non-secure stack. 0 = Use Non-secure stack frame. 1 = Use Secure stack frame. Always 0 if the security extension is not configured.
DCRS	5	Default callee register stacking. 0 = When addition context state has been stacking when the combination of background and exception security state indicate that the additional stacking is not required. 1 = When the default rules are used for callee register stacking is followed. This bit is always 1 if the security extension is not implemented.
ES	0	Exception Secure - Indicates the security state that the exception is taken to. 0 = Non-secure. 1 = Secure. Always 0 if the security extension is not configured.

On exception return, the processor performs various integrity checks to ensure the EXC_RETURN value is consistent with the rest of the system.

The following figure shows a simple exception transition and its corresponding EXC_RETURN values.

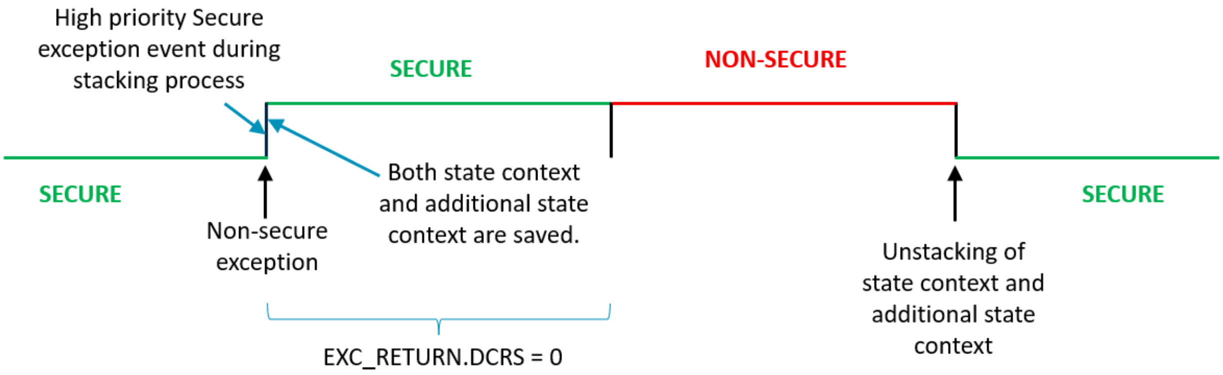
Figure 5-5: EXC_RETURN values



5.5.1 Scenario 1

In this scenario, when the background Secure thread is interrupted by a Non-secure exception, the integer additional context is saved on to the stack. However, just before entering the Non-secure exception handler, if a higher priority late-arriving Secure interrupt occur, then processor uses its late-arriving optimization technique and switches to execute Secure exception handler first before the Non-secure exception handler. To indicate that callee-saved registers are already saved in the stack frame, EXC_RETURN.DCRS bit is set to 0.

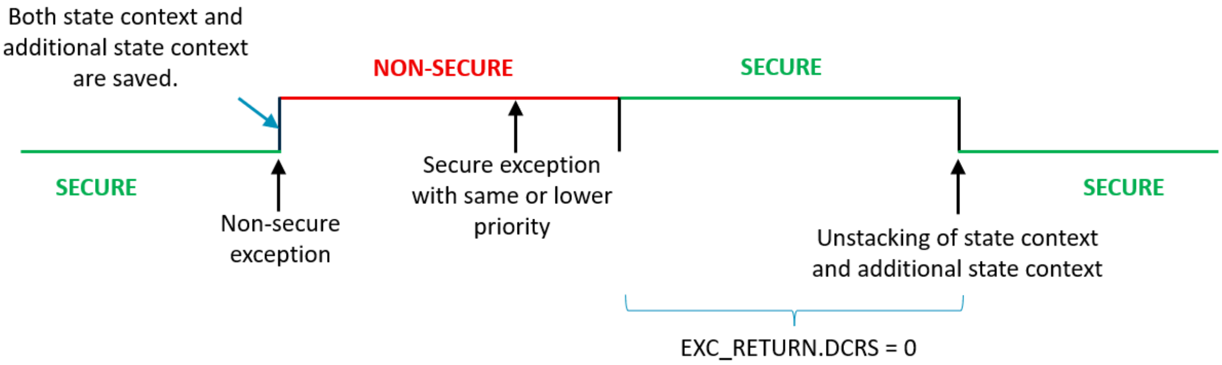
Figure 5-6: Scenario:1 for EXC_RETURN.DCRS=0


High priority Secure exception event during stacking process
SECURE
NON-SECURE
SECURE
Both state context and additional state context are saved.
SECURE
Non-secure exception
Unstacking of state context and additional state context
EXC_RETURN.DCRS = 0

5.5.2 Scenario 2

When the background is Secure, it is interrupted by a Non-secure interrupt and the additional state context information (i.e. callee-save registers) has been pushed to the stack. During the execution of the Non-secure exception handler a Secure interrupt can occur. If this has a lower priority than the executing Non-secure interrupt, then after the execution of the Non-secure interrupt handler, the processor switches to execute the pending Secure interrupt handler. To indicate that additional state context is already on the stack, EXC_RETURN.DCRS is set to 0 on entry to the Secure interrupt handler. The following figure shows this scenario.

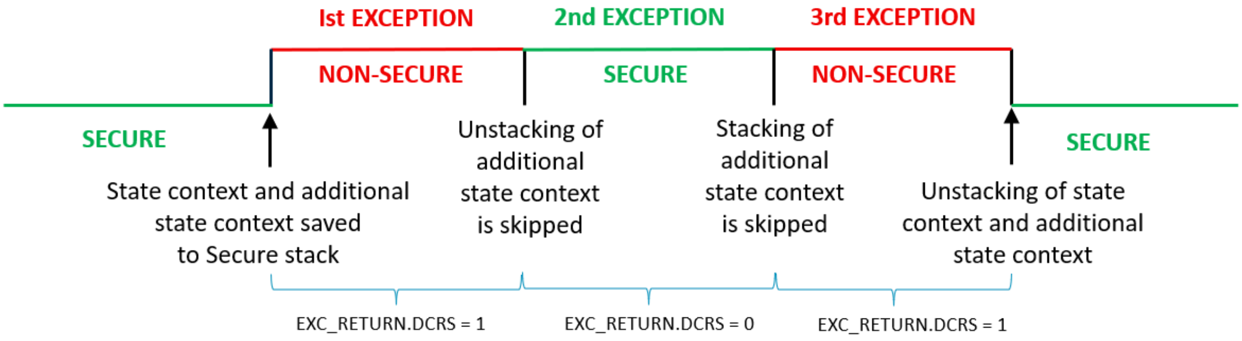
Figure 5-7: Scenario:2 for EXC_RETURN.DCRS=0


Both state context and additional state context are saved.
NON-SECURE
SECURE
SECURE
Secure exception with same or lower priority
SECURE
Non-secure exception
Unstacking of state context and additional state context
EXC_RETURN.DCRS = 0

5.5.3 Scenario 3

The following figure shows a simple example where three exceptions are tailchained with different security states.

Figure 5-8: Scenario:3


Ist EXCEPTION
2nd EXCEPTION
3rd EXCEPTION
NON-SECURE
SECURE
NON-SECURE
SECURE
SECURE
State context and additional state context saved to Secure stack
Unstacking of additional state context is skipped
Stacking of additional state context is skipped
Unstacking of state context and additional state context
EXC_RETURN.DCRS = 1
EXC_RETURN.DCRS = 0
EXC_RETURN.DCRS = 1

When multiple exceptions are tail-chaining, a Secure tail-chained exception after a Non-secure exception cannot rely on any registers containing the values they had when no exception was active. We recommened that FPCCR.CLRONRET is set to 1. This ensures that hardware automatically clears the Floating-point context registers to 0 on exception return. See Secure Software Development design considerations.

5.6 SecureFault

The SecureFault exception is triggered by violations of security rules defined in Armv8-M Security extension architecture. The SecureFault exception is not available in processors that do not include Security extension.

The following are some of the scenarios that can result in SecureFault:

1. Memory accesses from the Non-secure state that target Secure memory that violate security permission: The memory accesses can be a memory data read/write (by LDR/STR instructions) or exception stacking or unstacking process or a memory access from the Non-secure state that target Secure memory
2. An illegal transition between security states
3. When a security integrity check fails during an exception return sequence

On a SecureFault, one of the fault status bits in the SecureFault Status Register (SFSR) is set to 1 to indicate the cause of error. When the SFSR.SFARVALID bit is set, then the address of a memory access that attempts to violate the security permission is captured in the SecureFault Address Register (SFAR). This helps in debugging the cause of the SecureFault.

The following table shows the programmers model for the SFSR. Both SFSR and SFAR are available only when Security extension is included in the system.

SFSR bit	Name	Description
7	LSERR	Lazy state error flag. This flag indicates that an error occurred during lazy state activation or deactivation.
6	SFARVALID	Indicates that SFAR contains valid address.
5	LSPERR	Lazy state preservation error. A SAU/IDAU violation during a lazy stacking operation for floating-point registers.
4	INVTRAN	Invalid transition error flag. Indicates either a branch from Secure to Non-secure memory without using BXNS/BLXNS or a branch from Secure to Non-secure memory without the LSB of the branch target address indicated a branch to the Non-secure state is expected.
3	AUVIOL	Attribution unit violation. This bit is set when the Non-secure state attempts to access a Secure memory address.
2	INVER	Invalid exception return flag. This can be caused either by EXC_RETURN.DCRS being set to 0 when returning from an exception in Non-secure state or by EXC_RETURN.ES bit set to 1 when returning from an exception in the Non-secure state.
1	INVIS	Invalid integrity signature. When an exception return switches the processor from Non-secure to Secure state and the Secure stack being used for unstacking does not have a valid integrity signature.
0	INVEP	Invalid entry point. Either non-secure code tries to branch into a Secure address where the first instruction is not an SG instruction or the address is not marked as Non-secure Callable (NSC) by SAU/IDAU.

The SFSR and SFAR registers can be accessed by CMSIS-Core symbols via `SAU->SFSR` and `SAU->SFAR` respectively.

Address	Register	Description	CMSIS-Core symbol
`0xE000EDE4`	SAU_SFSR	Secure Fault Status Register	SAU->SFSR
`0xE000EDE8`	SAU_SFAR	Secure Fault Address Register	SAU->SFAR

When Halting debug mode is enabled, you can use the vector catch mechanism, by setting DEMCR.VC_SFERR to 1, to generate a debug event and enter Debug state on entry to a SecureFault exception.

Note

SecureFaults must be considered fatal and must prevent further execution of Non-secure code, including Non-secure exception handlers.

5.7 External interrupts configuration and management

Several registers in NVIC control external interrupts: Exception numbers 16 - 49. Configure and control these external interrupts in System Control Space (SCS). See Armv8-M Exception Model User Guide for NVIC registers for interrupt management, that is Interrupt Enable, Set Pending, Clear Pending and Active.

When Security extension is implemented in a system, then along with other interrupt configuration registers, there is an NVIC Interrupt Target Non-secure register, `NVIC->ITNS`. This NVIC_ITNS register enables the Secure software to configure an external interrupt to be either Secure or Non-secure. Each bit in NVIC->ITNS register corresponds to an external interrupt. By default, each bit in NVIC->ITNS is set to 0 for Secure state. To retarget an external interrupt to Non-secure, set the NVIC->ITNS bit to 1 corresponding to its exception number.

Management function	Register type	Address	CMSIS-Core symbols	Notes
Define the interrupt's target security state when the Security extension is implemented	Interrupt Target Nonsecure State Registers	`0xE000E380` to `0xE000E3BC`	NVIC->ITNS[0] to NVIC->ITNS[15]	Each of the NVIC_ITNS register controls the target state for 32 interrupts. For example bit 2 of NVIC_ITNS[1] controls the target state of interrupt 34.

The following table shows the CMSIS-Core functions that you can use to set up the interrupt target state as either Secure or Non-secure.

Function	Purpose
uint32_t NVIC_SetTargetState (IRQn_Type IRQn)	Programs the interrupt target state as Non-secure. This function returns the target state. 0 - Secure, 1 - Non-secure
uint32_t NVIC_ClearTargetState (IRQn_Type IRQn)	Programs the interrupt target state as Secure. This function returns the target state. 0 - Secure, 1 - Non-secure
uint32_t NVIC_GetTargetState (IRQn_Type IRQn)	Reads back the target security state of the interrupt

If an interrupt is configured as Secure, then from a Non-secure software, all the NVIC registers and control bits associated with that interrupt are read as 0 and writes are ignored.

The following table shows the CMSIS-Core NVIC interrupt management functions that can be accessed by Secure software to have NVIC Non-secure alias view of the external interrupts.

Function	Usage
void TZ_NVIC_EnableIRQ_NS (IRQn_Type IRQn)	Enable External Interrupt
void TZ_NVIC_DisableIRQ_NS (IRQn_Type IRQn)	Disable External Interrupt
uint32_t TZ_NVIC_GetEnableIRQ_NS (IRQn_Type IRQn)	Get Interrupt Enable status
void TZ_NVIC_SetPendingIRQ_NS (IRQn_Type IRQn)	Set Pending Interrupt
void TZ_NVIC_ClearPendingIRQ_NS (IRQn_Type IRQn)	Clear Pending Interrupt
uint32_t TZ_NVIC_GetPendingIRQ_NS (IRQn_Type IRQn)	Get Pending Interrupt
uint32_t TZ_NVIC_GetActive_NS (IRQn_Type IRQn)	Get Active Interrupt
void TZ_NVIC_SetPriority_NS (IRQn_Type IRQn, uint32_t priority)	Set Interrupt Priority
uint32_t TZ_NVIC_GetPriority_NS (IRQn_Type IRQn)	Get Interrupt Priority
void TZ_NVIC_SetPriorityGrouping_NS (uint32_t PriorityGroup)	Set Priority Grouping
uint32_t TZ_NVIC_GetPriorityGrouping_NS (void)	Get Priority Grouping

The NVIC->ITNS registers are accessible only from Secure Privileged state. If Security Extension is not implemented, then ITNS registers are not implemented.

5.8 SVC and PendSV

The SVCall and PendSV exceptions are banked between security states. When the processor is in Secure state, the SVC instruction causes a Secure SVCall exception. When the processor is in Non-secure state, the SVC instruction causes a Non-secure SVCall exception.

Similarly, the PENDSVSET and PENDSVCLR bit in the Interrupt Control and State Register (ICSR) is banked. Setting PENDSVSET for a particular security state raises a PendSV exception for that security state.

The priority level registers for SVC and PendSV are also banked between security states.

If a Secure exception is taken from a Secure context of execution, the processors can choose either:

- The additional state context stacking is not attempted, indicated by EXC_RETURN.DCRS = 1
- The additional state context stacking is attempted, indicated by EXC_RETURN.DCRS = 0

Whether the additional state context stacking is performed is important for an SVCall handler. additional state context stacking changes the format of the stack frame, and therefore where the arguments for the SVCall handler are on the stack. See SVC example project which gives steps on how to construct SVCall handler and EXC_RETURN.DCRS checks in the SVC handler code. It is important that you use both EXC_RETURN.Mode and EXC_RETURN.SPSel bits to determine the stack used by the background thread irrespective of security state.

```
// main.c
  __asm(
    "MOV R0,  #83     \n"
    "MOV R1,  #7      \n"
    "SVC      #3      \n"
  );

// SelectSVCNumber.c
extern void SVC_Handler(void){
  __asm(
    ".global SVC_Handler_Main        \n"
    "TST      LR, #0x4               \n"  /* Check for EXC_RETURN.SPSEL */
    "ITE      EQ                     \n"  /* Set R0 = Stack pointer of background
 code */
    "MRSEQ    R0, MSP                \n"
    "MRSNE    R0, PSP                \n"
    "TST      LR, #0x20              \n"  /* Check for EXC_RETURN.DCRS */
    "IT       EQ                     \n"  /* Adjust R0 such that it points to */
    "ADDEQ    R0, R0, #0x40          \n"  /* state context if additional state
                                          /* context is stacked */
    "MOV      R1, LR                 \n"  /* Move EXC_RETURN to R1 */
    "B        SVC_Handler_Main       \n"
  );
}
```

5.9 SysTick timer

Having two independent SysTick timers, one for each security state, is mandatory for Mainline processors such as Cortex-M33, and optional for Baseline processors such as Cortex-M23. Both timers can trigger SysTick exceptions in their corresponding security states. Secure software can also access the Non-Security SysTick registers using an alias address.

For the Armv8-M Baseline architecture, you have the option to implement just one SysTick. You can configure the SysTick timer that is implemented to be Secure or Non-secure using the STTNS, bit[24], field of the Interrupt Control and State Register (ICSR).

For bit[24] of this register, SysTick Targets Non-secure State (STTNS):

- If bit[24] = 0 SysTick targets the Secure state (default).
- If bit[24] = 1 SysTick targets the Non-secure state.

The STTNS bit is accessible only when the processor is in Secure state, and only if the design configuration of the processor implements one SysTick timer.

5.10 MemManage faults

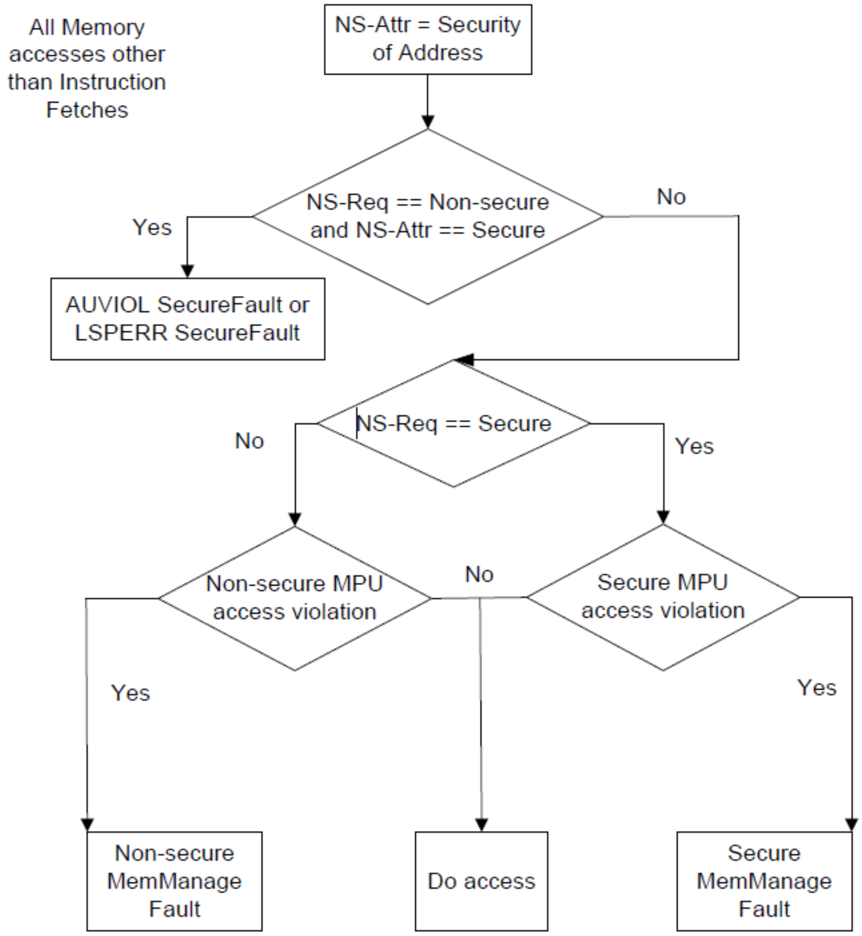
When Security Extension is implemented, you can have one of the following combinations:

- No MPU
- Secure MPU only
- Non-secure MPU only
- MPU present in both security states

Because the MPU can be banked across the security state, it can have a different number of MPU regions in each security state. The MPU defines the memory attribute and privilege permissions of memory. The Security Attribution Unit (SAU/IDAU) defines the security attribute of the memory. The MPU check sequence is executed on any load, store, or instruction fetch transaction with security attribution checks.

The following figure shows how the checks are done for accesses generated by load and store instructions. The current security state of the processor, shown by the NS-Req signal, determines which MPU performs the checks. For more details see Armv8-M Protected Memory System Architecture in Armv8-M Architecture Reference Manual.

Figure 5-9: Check sequence for Load/Store transactions


All Memory accesses other than Instruction Fetches
NS-Attr = Security of Address
NS-Req == Non-secure and NS-Attr == Secure
Yes
No
AUVIOL SecureFault or LSPERR SecureFault
NS-Req == Secure
No
Yes
Non-secure MPU access violation
No
Secure MPU access violation
Yes
Yes
Non-secure MemManage Fault
Do access
Secure MemManage Fault

For instruction fetches, the MPU used to perform the checks is determined by the security state of the memory being accessed, not the current security state of the processor. In most cases, this is the same as the current security state of the processor. However, there might be a branch from Non-secure to Non-secure callable memory, for example, as part of a Non-secure to Secure function call. In this case, the fetch of the instruction in Non-secure callable memory is checked by the Secure MPU. This is because Non-secure callable memory is a type of Secure memory. See Memory Security attributes. As a result, the Secure MemManage fault handler must be able to handle faults triggered by Non-secure code. If a Secure library manager is being used, see Firmware IP protection, then such a fault might not indicate an error. It might be an implicit request from the Non-secure side to activate the inactive library by calling a Secure function the Non-secure code is permitted to use, but is not currently accessible. In this case, to determine whether an error occurred, the Secure MemManage Fault handler must check:

- The type of access that generated the fault
- The address the access was targeting
- Whether the MPU must be reconfigured to allow the access

5.10.1 Caution for Secure code developers

The Non-secure application can trigger a MemManage Fault that targets Secure state. Developers of the Secure MemManage fault handler must take care.

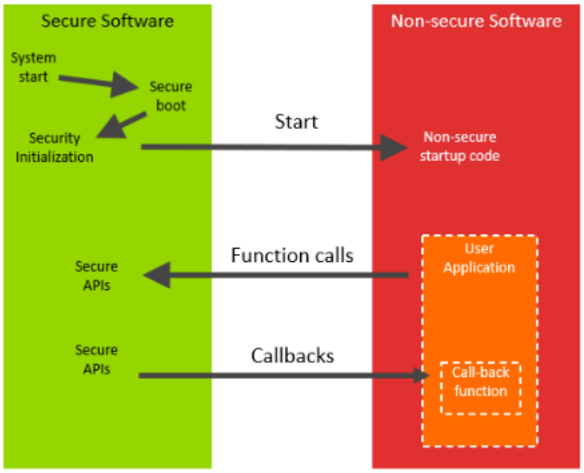
If there is malicious Non-secure code that attempts to perform a FNC_RETURN when an EXC_RETURN is expected, then the integrity signature at the bottom of the exception return stack frame is interpreted as the function return address. Because the integrity signature is in an area of the address space that is not executable this raises a Secure MemManage fault. Secure MemManage fault handlers must detect this case and act appropriately to prevent further execution of the malicious Non-secure code.

6. Developing software with Security Extension

This chapter describes how to develop software with Security extensions.

The following figure shows an embedded application based on TrustZone.

Figure 6-1: Secure/Non-secure embedded application



Secure/Non-secure embedded application

The embedded application is split into a Non-secure software and Secure software. After reset, the system boots up in Secure state and runs Secure firmware. When the Secure firmware has initialized the system appropriately, it branches to the entry point of the Non-secure application. The Non-secure software can request the Secure services via the Secure APIs when needed. The Secure APIs might call the Non-secure function as a call-back function, for example, to provide previously requested information when it is available.

Secure and Non-secure software images are usually developed separately, the Secure software sets up Secure stack pointers, performs system Security configuration, initializes Secure peripherals and Secure firmware framework, and implements Secure services. The Non-secure software has its own startup code and Non-secure system initialization routines, Non-secure software implements user application, performs function call to Secure APIs and implements Non-secure call-back functions.

In some contexts, especially in Arm Compiler, the Armv8-M Security Extensions is also referred to as Cortex-M Security Extensions or CMSE. As a part of Security Extensions inclusion in Armv8-M architecture, there is a need to enhance Arm C Language Extensions (ACLE) specifications and the Arm Architecture Procedural Call Standard (AAPCS). These enhancements and requirements are available in (Armv8-M Security Extensions Requirements on Development Tools Specification) [https://developer.arm.com/documentation/ecm0359818/latest/].

These steps are often required during development of Secure and Non-secure images:

1. Memory map partitioning
2. Add CMSIS startup and initialization code
3. Write the linker script or scatter file
4. Develop Secure software using Armv8-M Security Extensions
5. Build the Secure image
6. Develop and build a Non-secure image that can call Secure APIs
7. Launch Non-secure images from Secure side
8. Preload and run the images on your device
9. Build a updated Secure image using a previously generated import library

6.1 Memory map partitioning

In the initial phase of your project, you must define the overall system configuration. This includes partitioning memory, peripherals and related interrupts between Security states. The partitioning of the memory space is handled by the Security Attribution Unit (SAU) and Implementation Defined Attribution Units (IDAU), as Memory configuration describes. Peripherals are also partitioned between the Secure and Non-secure states, and how this is done depends of the microcontroller or SoC being used.

Consider Arm Microcontroller Prototyping System 2 (MPS2) Fixed Virtual Platform (FVP) models as an example platform. The following table shows part of the example system memory map.

Memory	Description
`0x10000000-0x101FFBFF`	Secure code SRAM
`0x101FFC00-0x101FFFFF`	Secure code, Non-secure Callable (NSC) SRAM
`0x00200000-0x003FFFFF`	Non-secure code SRAM
`0x30000000-0x301FFFFF`	Secure data SRAM
`0x20200000-0x203FFFFF`	Non-secure data SRAM
`0x40000000-0x40001FFF`	Non-secure peripheral
`0x50002000-0x50002FFF`	Secure peripheral

Many systems enable you to configure the amount of memory allocated to the different security states, or allocate peripherals to a specific security state. In this case, developers must decide which allocation best suits their application.

6.2 Add CMSIS startup and initialization code

For a CMSIS-based project, device-specific CMSIS-Core pack provides the following files to the embedded application:

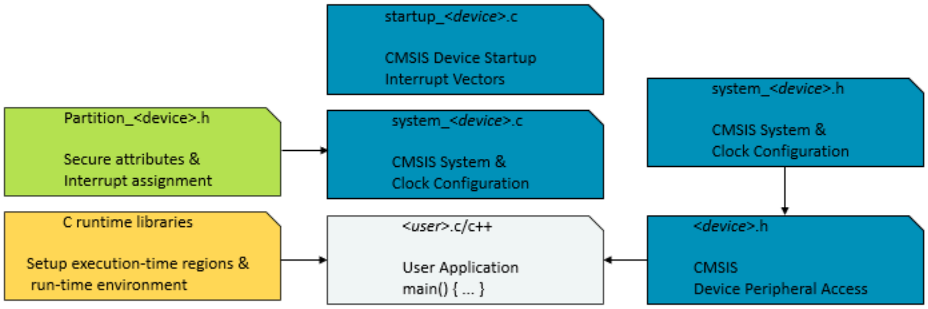
- Startup file `startup_<device>.c` with reset handler and exception vectors

- System configuration file `system_<device>.c` file with general device configuration
- Device head file `device.h` which gives access to processor core and all peripherals

See Using CMSIS in Embedded Applications for a general description of the CMSIS-Core(Cortex-M) files. You must modify these generated files according to your system design.

The following figure shows components in Cortex-M software project:

Figure 6-2: CMSIS software components



6.3 Write the linker script or scatter file

A linker script or scatter file is used in the linker phase, and it defines the following:

- All the memory devices with address range, size information based on the target hardware
- The placement of the load and execution regions for both code and data

Because a different compile and link flow is used for each security state, a different linker script or scatter file is required for each security state. Memory map partitioning is done in previous steps. Therefore, the linker script for each security state must align with that memory map partitioning.

To ensure that the system is Secure, you must make sure that resources used in Secure side are placed in the Secure memory region. This includes:

- Secure code
- Secure data, including stacks and heap memories
- the Secure vector table

For Secure APIs which provide services to Non-secure software, the linker generates Secure gateway veneers for these APIs, and gathers them into the veneer section. The veneer section is placed in a memory region specified in the memory description file. This memory region must be set with the Secure Non-secure Callable (NSC) attribute in SAU/IDAU.

Different toolchains use different ways to generate the CMSE veneer section and place it in Non-secure Callable (NSC) region. In Arm Compiler toolchain, this is indicated by the section "Veneer$$CMSE" in the scatter file. The following example shows you how to define one NSC region and place the veneer code in this region in Armclang scatter file.

```
#define __CV_BASE    0x101FFC00
#define __CV_SIZE    0x400

LR_CMSE_VENEER __CV_BASE ALIGN 32 __CV_SIZE
{
    *(Veneer$$CMSE)
}
```

In GCC, you need to create an output section ".gnu.sgstubs" with a specified runtime address in the GCC linker script `.ld` file. The following example shows you how to define one NSC region and place the veneer code in this region in GCC `.ld` file.

```
__ROM_BASE_NSC =  0x101FFC00;
__ROM_SIZE_NSC =  0x400;

MEMORY
{
    FLASH_NSC (rx)  : ORIGIN = __ROM_BASE_NSC, LENGTH = __ROM_SIZE_NSC
}

.gnu.sgstubs :
{
    . = ALIGN(32);
    *(.gnu.sgstubs*)
} > FLASH_NSC
```

6.4 Develop Secure software using Armv8-M Security Extensions

When creating a Secure software project, the system Security configuration is set up at the Secure software initialization. In addition to legacy boot up operations, such as initializing PSP, some security configuration operations are as follows:

- Configuring the security of the memory map using SAU region programming. For more details, see SAUs
- Interrupt Target Non-secure Register (NVIC->ITNS): for each interrupt, whether it should target the Secure or the Non-secure state. For more details, see Target states of exceptions
- System Handler Control and State Register (SHCSR): enable the SecureFault, MemManage, UsageFault, exception and other system exceptions
- The priority of critical Secure exceptions, such as SecureFalut, MemManage, UsageFault, BusFault, must be set so that they cannot be blocked or preempted by Non-secure code.
- Application Interrupt and Reset Control Register (AIRCR)
    - AIRCR.BFHFNMINS: This bit must be kept at 0, NMI, HardFault and BusFault exceptions remain in Secure state, when Security extension is used

  - AIRCR.PRIS: set this bit to prioritize Secure exceptions
  - AIRCR.SYSRESETREQS: set this bit to decide whether Non-secure software can trigger a self-reset
- Defining the features available to Non-secure software
  - Non-secure Access Control Register (NSACR) must be programmed to define whether the FPU, coprocessors and the Arm Custom Instructions features are accessible from the Non-secure state
  - Coprocessor Power Control Register (CPPWR) need configuring to prevent the Non-secure software from accessing the power control of the FPU and coprocessors
  - System Control Register: SCR.SLEEPDEEPS should be set properly to decide whether the Non-secure software can control the SLEEPDEEP feature
- Configuring Floating-Point Context Control Register (FPCCR). See FPU related Security settings for a device with FPU implemented for more details.
  - If Secure software uses the FPU/MVE for sensitive data, FPCCR.TS, FPCCR.CLRONRET,FPCCR.CLRONRETS must be set to 1
  - If the Secure software does not use the FPU/MVE registers for sensitive data, the Secure software can leave the FPCCR.TS and FPCCR.CLRONRETS as 0. Non-secure privileged software can then set the FPCCR.CLRONRET to 1 to prevent privileged data in the FPU from being visible to unprivileged software
- Sealing the stacks and stack limit: see Stack pointer limit setup and stack sealing for additional details)
- Configuration and Control Register (CCR): CCR.TRD must be set to enable reentrancy checking if the Secure software vulnerable to reentrancy attacks, and the Secure software does not check for reentrancy.
- Secure debug must be disabled before the processor at power-up to prevent pre-boot attacks. Processors with the security extensions provide external signals to allow debug to be disabled. Read the documentation for your device to determine how to control these signals.

CMSIS-Core pack provides `partition_<device>.h` file which contains “TZ_SAU_Setup()” function and related setting parameters. See Configuring SAU using CMSIS.

The Secure project implements Secure services that can be exported as Secure APIs and called by the Non-secure side. The CMSE toolchain supports to develop Secure software in C language. To develop a Secure API, follow these steps:

1. Add CMSE function attribute __**attribute__((cmse_nonsecure_entry))** to Secure API definition
2. Declare Secure APIs as normal in an interface header file
3. Include <arm_cmse.h> header

A simple example is as follows:

```
#include <arm_cmse.h>
#include "secure_interface.h"
void __attribute__((cmse_nonsecure_entry)) ns_callable_fn1(void)
```

```
    {
        ...
    }
```

The Secure API shown below is declared in an interface header secure_interface.h in Secure project.

```
    void ns_callable_fn1(int val);
```

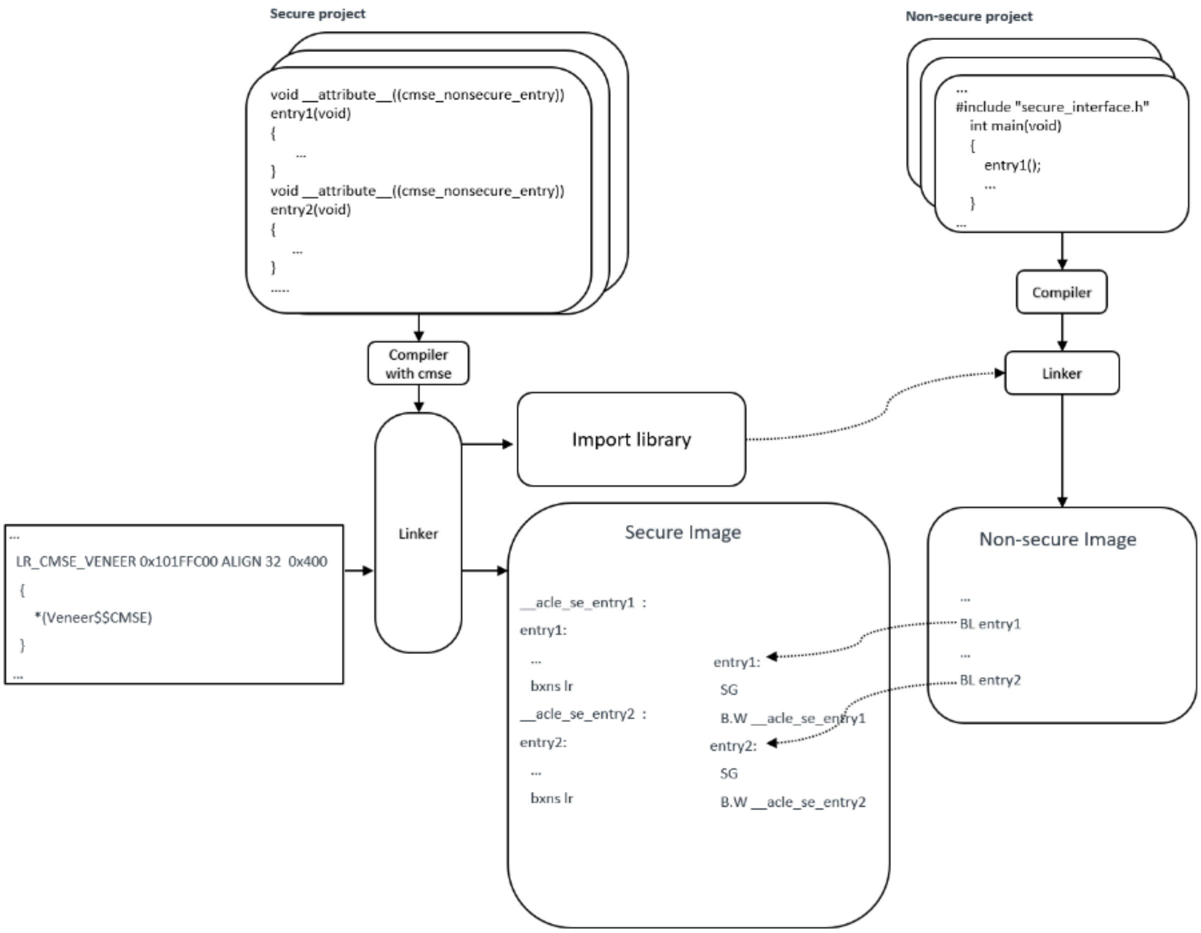
6.5 Build the Secure image

Arm Compiler for Embedded tools enables you to build images that run in the Secure state of the Armv8-M Security Extension. To build an image that runs in the Secure state you must:

- Compile with armclang command-line option -mcmse
- Add armlink command-line option –import-cmse-lib-out to generate import library for the Non-secure project

The following figure shows an overview building process for Secure and Non-secure.

Figure 6-3: Secure & Non-secure software build process



```
void __attribute__((cmse_nonsecure_entry))
entry1(void)
{
    ...
}
void __attribute__((cmse_nonsecure_entry))
entry2(void)
{
    ...
}
.....
```

```
...
#include "secure_interface.h"
  int main(void)
  {
    entry1();
    ...
  }
...
```

```
...
LR_CMSE_VENEER 0x101FFC00 ALIGN 32  0x400
 {
    *(Veneer$$CMSE)
 }
...
```

The import library consists of a relocatable file containing types, names, and addresses of the Secure APIs symbols. Non-secure projects can contain function calls to the Secure APIs. The import library provides information about the addresses of the Secure APIs when linking Non-secure projects.

An example build script in Arm compiler is as follows:

```
    armclang --target=arm-arm-none-eabi -march=Armv8-m.main -mcmse -c Secure.c
    armlink --import-cmse-lib-out=export\secure_library.o --scatter=secure.sct -o
  Secure.axf Secure.o system_Armv8-M.o startup_Armv8-M.o
```

GCC also provides command-line options –csme-implib and –out-implib. A similar build script is shown here:

```
    arm-none-eabi-gcc -march=armv8-m.main -mthumb -mcmse -c -o Secure.o Secure.c
    arm-none-eabi-ld -T Secure.ld -mthumb -march=armv8-m.main  --specs=nosys.specs -
 Xlinker --cmse-implib -Xlinker --out-implib=secure_library.o -o Secure.elf Secure.o
  system_Armv8-M.o startup_Armv8-M.o
```

6.6 Build a Non-secure image that can call Secure APIs

Add the interface header secure_interface.h and the import library secure_library.o to Non-secure project. Doing so enables Non-secure software to call these Secure APIs as a normal function call. Non-secure project does not require any additional CMSE toolchain support.

```
    #include "secure_interface.h"
    int main(void)
    {
        ns_callable_fn1();
        ...
    }
```

The following example build script builds the Non-secure image:

```
   armclang --target=arm-arm-none-eabi -march=Armv8-m.main nonsecure.c -c -o
nonsecure.o
   armlink  -o nonsecure.axf --scatter=nonsecure.sct nonsecure.o system_Armv8-M.o
startup_Armv8-M.o ..\export\secure_library.o
```

6.7 Launch Non-secure images from Secure side

For a device with the Security extension implemented, the device runs in Secure state after it is powered on. When the system Security initialization process completes, Secure software sets up the MSP_NS from the first item in the Non-secure vector table and then calls the Non-secure reset handler. An example code is as follows:

```
    #include <arm_cmse.h>

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

    int main(void) {

    ...
    funcptr_void NonSecure_ResetHandler;

    /* Perform all setup of the Secure state before booting Non-secure image */

    /* Set the Non-secure main stack (MSP_NS) */
    __TZ_set_MSP_NS((uint32_t)(VectorTable_NS[0]));

    /* Get Non-secure reset handler */
    NonSecure_ResetHandler = (funcptr_void)(VectorTable_NS[1]);

    /* Start the Non-secure state software application */
    /* WARNING: Assume that the Non-secure image doesn’t return */
    NonSecure_ResetHandler();

    }
```

6.8 Preload and run the images on your device

The final Secure and Non-secure image must be preloaded and run on the device.

The following table shows the Secure memory layout for Arm MPS2 FVP. The FVP boots from `0x10000000`.

Memory	Description
`0x00200000-0x003FFFFF`	Non-secure code SRAM
`0x10000000-0x101FFFFF`	Secure code SRAM

You can use the following command to load the binary into the FVP and run the images in the FVP.

```
   FVP_MPS2_AEMv8M.exe -C NSC_CFG_0=1 -C fvp_mps2.DISABLE_GATING=1 -C
cpu0.has_arm_v8-1m=1 -C cpu0.baseline=0 --data cpu0=secure.bin@0x10000000  --data
cpu0=nonsecure.bin@0x00200000
```

6.9 Build a Secure image using a previously generated import library

In some cases, Secure image is updated with new features or bug fixes. If a new Secure image needs to be used with an existing Non-secure image, you can use an import library from a previous build in the Secure build flow. This ensures that the Secure entry point addresses are kept constant.

To achieve this, you can specify command-line option `--import-cmse-lib-in=<old import library>` with `--import-cmse-lib-out=<import library>` in Armclang build. In GCC, you can use `--in-implib=<old import library> --cmse-implib --out-implib=<import library>` command-line option. Also, make sure that the input and output libraries have different names.

Consider the following Secure APIs as an example. Secure software defines two Secure APIs as follows, and generates the import library importlib_v1.o:

```
   int __attribute__((cmse_nonsecure_entry)) entry1(int x) { ... }
   int __attribute__((cmse_nonsecure_entry)) entry3(int x) { ... }
```

Detailed address information is as follows:

```
       entry1
           0x10100000:    e97fe97f    ....    SG        ; [0x100ffe08]
           0x10100004:    f702bae0    ....    B        __acle_se_entry1 ;
0x100025c8
       entry3
           0x10100008:    e97fe97f    ....    SG        ; [0x100ffe10]
           0x1010000c:    f702baec    ....    B        __acle_se_entry3 ;
0x100025e8
```

Later the Secure software upgrades with new Secure APIs:

```
    int __attribute__((cmse_nonsecure_entry)) entry1(int x) { ... }
    int __attribute__((cmse_nonsecure_entry)) entry2(int x) { ... }
    int __attribute__((cmse_nonsecure_entry)) entry3(int x) { ... }
```

Use the following command to generate the new import library importlib_v2.o:

```
    armlink --entry=Reset_Handler -o Secure.axf --import-cmse-lib-out=importlib_v2.o
 --import-cmse-lib-in=importlib_v1.o --scatter=secure.sct Secure.o system_Armv8-M.o
 startup_Armv8-M.o
```

Detailed address information is as follows. Previous Secure APIs entry point addresses remain unchanged.

```
    entry1
        0x10100000:    e97fe97f    ....    SG        ; [0x100ffe08]
        0x10100004:    f702bae0    ....    B        __acle_se_entry1 ; 0x100025c8
    entry3
        0x10100008:    e97fe97f    ....    SG        ; [0x100ffe10]
        0x1010000c:    f702bafc    ....    B        __acle_se_entry3 ; 0x10002608
    ...
    entry2
        0x10100020:    e97fe97f    ....    SG        ; [0x100ffe28]
        0x10100024:    f702bae0    ....    B        __acle_se_entry2 ; 0x100025e8
```

You can also see various examples available in Armv8-M Security Extension use case examples to understand the actual source code and how it can be launched using ArmDS.

7. Booting and initializations

This chapter describes the booting process.

7.1 Vector table, VTOR and reset behavior

When processors implement the Security Extension, then just out of reset, the processor enters Secure state, Secure Privileged Thread mode. After reset, the processor performs a data access and reads the first two values of the default vector table. Because the processor is in Secure state, these accesses are Secure reads. Therefore, the vector table must be located in Secure memory. The first value read is used as the default value for Secure Main stack pointer `MSP_S`. The second value read from the vector table is the address of the Secure reset vector. The processor branches to the address read from this Secure vector.

Because the processor comes out of reset in Secure Privileged Thread mode, this ensures that the *Secure boot code* can write to both Secure and Non-secure state's software configuration settings. Therefore:

- All exceptions can be configured to target either Secure or Non-secure state.
- The Secure boot code can choose to change priority values of the Secure exceptions to be of higher priority than Non-secure exceptions using AIRCR.PRIS bit.
- The Secure boot code canpProgram the value for `PSP_S` and the Non-secure stack pointers.

Wen the Secure boot code has completed its initialization of the Secure world, it needs to hand control to the Non-secure boot code provided by the system programmer of the Non-secure world. The address of the vector table at reset is not architecturally defined. Most processor implementations allow it to be configured by hardware signals that are sampled at reset, for example `INITSVTOR[31:7]` in Cortex-M55. This value is visible to software in the register `VTOR_S`. `VTOR_S` holds the base address of the vector table used by exceptions that target Secure state.

The Non-secure register `VTOR_NS` holds the base address of the vector table used by Non-secure exceptions. The value of `VTOR_NS` must be configured by the Secure software before booting the Non-secure software. However, implementations often allow a default value to be provided by reset signals, for example `INITNSVTOR[31:7]` on Cortex-M55.

See [no-secure-callback] which describes how to write a minimalistic Secure boot code and branch to a Non-secure world.

7.2 FPU related Security settings for a device with FPU implemented

If Secure software uses FPU, the Secure software must:

- Set CP10 and CP11 bits in SCB->CPACR to enable FPU
- Configure NSACR register to define whether Non-secure software can access the FPU
- Configure CPPWR register to set FPU power control is only accessible from the Secure state
- Configure Floating-Point Context Control Register (FPCCR)

If Secure software uses FPU for sensitive data, the Secure software must set TS, CLRONRET and CLRONRETS control bits to 1 in the FPU Floating-point Context Control Register (FPCCR) at initialization, and must not change the setting afterwards.

If the Secure software does not use the FPU registers for sensitive data, the Secure software can leave the TS and CLRONRETS control bits as zero. Non-secure privileged software can set the CLRONRET control bit to 1 to prevent privileged data in the FPU from being visible to unprivileged software on exception return.

- The Non-secure software needs to enable the FPU independently of the Secure software by setting `SCB->CPACR` at Non-secure initialization.

7.3 Stack pointer limit setup and stack sealing

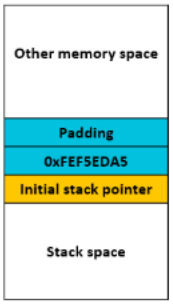
In Armv8-M, each stack pointer has a limit register, PSPLIM_NS, MSPLIM_NS, PSPLIM_S, MSPLIM_S, to trap usageFault exception triggered by stack overflows. In the initialization and stack switching phase, you must set these stack limit registers properly. For details, see TrustZone for Armv8-M/v8.1-M.

In normal execution on entry into Non-secure state, Secure stacks are topped with either the Integrity Signature or a valid Secure return address. The value in the Link Register can be modified by the Non-secure software during the execution of the Non-secure function or the Non-secure interrupt handler. This means that the Non-secure software can trigger illegal return operations, such as using an exception return instead of a function return or using a function return instead of an exception return. The hardware-generated stack frames are checked to prevent the malicious use of EXC_RETURN or FNC_RETURN in the Non-secure side.

However, the Non-secure world can manipulate the execution state. This manipulation can cause the return process to use a different stack pointer that is not the Secure stack previously used to switch from Secure to Non-secure state. This wrong Secure stack is not initialized before entering the Non-secure state, so the protection described previously might not be present.

The problem can be mitigated by sealing the stack. This involves placing a special value at the address above the real stack space. Arm recommends using the value `0xFEF5EDA5`. This value is an execute Never (XN) address, and also does not match the integrity signature.

Figure 7-1: secure stack sealing



Although you only need one word to detect an invalid return, you must push two words to keep the stack doubleword-aligned. This is a requirement of the Procedure Call Standard for the Arm Architecture (AAPCS).

Usually:

- Both MSP_S and PSP_S must be sealed
- You must seal a Secure stack when it is allocated
- You must seal the Secure stack that is not currently selected

CMSIS-Core for Cortex-M already provides function **__TZ_set_STACKSEAL_S (uint32_t* stackTop)** that implements the stack stealing operation. This function writes the stack seal values (2 x `0xFEF5EDA5U`) to the given address when in Secure state.

8. RTOS and Secure software design considerations

This chapter gives information about RTOS and bare-metal system configurations, and secure software design considerations.

8.1 RTOS configurations

Various operating system configurations are possible with the Armv8-M architecture. The following table lists some systems which include the Security Extensions and allow multiple OS configuration arrangements.

Configuration	RTOS design requirement
RTOS running in Secure state. Some threads are Secure, and some threads are Non-secure. Cross state APIs calls are also possible between security states	Both Secure and Non-secure stack space must be allocated for any threads that can call between states.
RTOS running in Non-secure state. Only a single Non-secure thread can call Secure API	In this case, only one Secure stack is needed. There is no need for the Secure state to support the RTOS API for context switching
RTOS running in Non-secure state. Multiple Non-secure threads can call into the Secure state.	In this case, each NS thread that can call across must have a corresponding Ssecure context and a Secure stack. Secure firmware must include CMSIS-RTOS API to support handling of context switching on the Secure side to manage creating and deleting contexts.
RTOS running in Non-secure state. Threads are Non-secure and there is no calling in to Secure state. (Secure state is not used by the application).	All threads have Non-secure stack only. The design is similar for the RTOS to chip designs without Security extension support.

8.1.1 Possible OS configurations

This section describes how you can have a bare-metal or interrupt-driven application in one security state and running an RTOS in the other security state.

8.1.1.1 Case 1: RTOS in Secure state

Running the RTOS only in the Secure state is an atypical configuration. It might be useful for simple systems where most of the software functionality is already implemented at delivery time by the chip developer. The software can be protected in the Secure state and the OEM or end-user can integrate a small amount of code in the Non-secure threads to customize the end device.

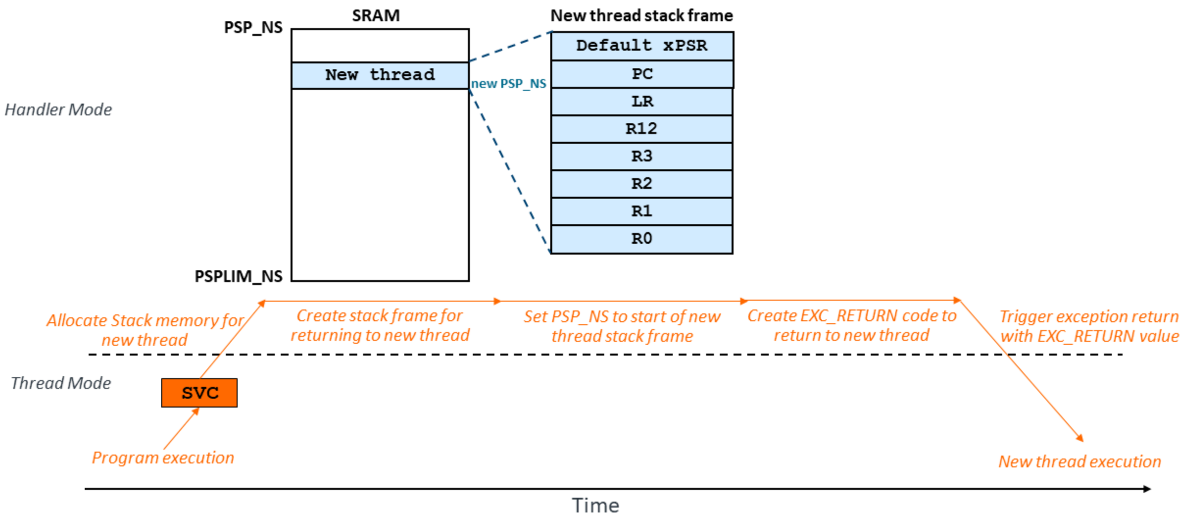
When an RTOS is running in Secure state, threads can be Secure or Non-secure. You can also have function calls between Secure and Non-secure software, which means that each thread can have both Secure and Non-secure stack allocation.

You can create a Non-secure thread from the Secure state. However, a simpler implementation is to call a function in the Non-secure state that is responsible for creating the Non-secure thread.

8.1.1.2 Case 2: RTOS in Non-secure state

The RTOS runs in the Non-secure state and threads are Non-secure by default. The following figure shows the process of a thread creation in Non-secure state.

Figure 8-1: Thread creation in non-secure state


SRAM
PSP_NS
New thread
PSPLIM_NS
new PSP_NS
New thread stack frame
Default xPSR
PC
LR
R12
R3
R2
R1
R0
Handler Mode
Thread Mode
SVC
Program execution
Allocate Stack memory for new thread
Create stack frame for returning to new thread
Set PSP_NS to start of new thread stack frame
Create EXC_RETURN code to return to new thread
Trigger exception return with EXC_RETURN value
New thread execution
Time

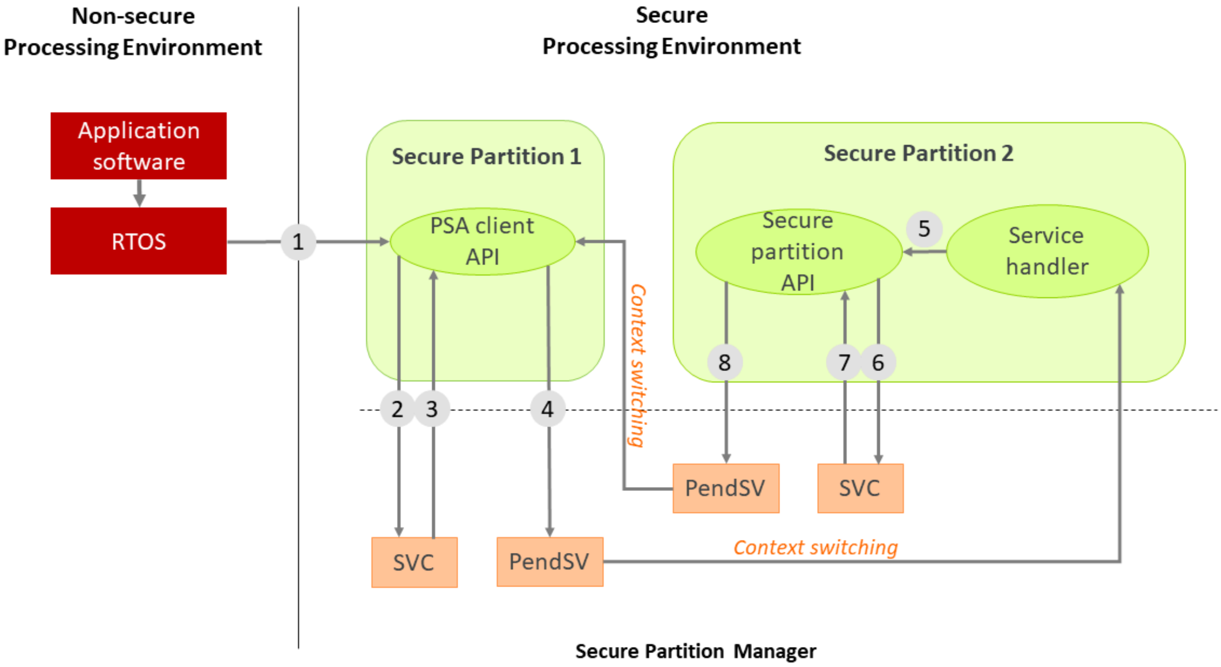
When creating a thread from Non-secure handler mode, the EXC_RETURN code must be `0xFFFFFFBC`. The thread is Non-secure and without FP extensions. The stack frame must be pointed to by PSP_NS. However, threads can call functions in Secure firmware, so must have have both Secure and Non-secure stack allocation. Because the RTOS is in a Non-secure state, it cannot directly access the Secure registers including the Stack Pointer and is therefore unable to directly handle context switching of Secure stacks.

Software in the Secure state needs to be aware of threads that might call Secure functions so that stack space can be allocated and destroyed at appropriate times.

To solve this problem, CMSIS-RTOS APIs are being developed which support Secure stack management. See Extension of CMSIS-RTOS for Non-secure RTOS.

TF-M's implementation is different but retains the APIs. See TF-M design docs. The following figure shows the process when a non-secure task requests the secure service and how the TF-M allocates/destructs the stack for secure service.

Figure 8-2: TF-M context switch flow



1. Non-secure task can call the Platform Security Architecture (PSA) API in RTOS to request the secure service. Then, RTOS calls the psa_connect() function in PSA client APIs to enter into secure environment. TF-M handles this request.
2. When TF-M receives the request, the PSA client API generates a SVC exception. This requests the Secure Partition Manager (SPM) help to switch to the secure service partition. At SVC handler, the SPM saves all infomation and pends a PendSV exception.
3. The SVC handler returns to the partition 1, but there has a pending interrupt (PendSV).
4. PendSV is tailchained by the PendSV handler. At this handler, the SPM switches the secure partition thread context including stack frame, from partition 1 to partition 2. The service call schedules to run.
5. When partition 2 handles the request, it uses the Secure Partition APIs to handle the request. These APIs use the SVC to request the data at SPM.
6. When SPM receives the secure partition request information, it acts differently according to the request. Some requests need other SP to fill data, so that needs to switch the SP.
7. If SP switching is needed, the PendSV is triggered and handled. SPM switches the target SP thread context from SP1.

8.1.1.3 Case 3: RTOS and application in Secure state only

A microcontroller vendor can ship a blank device without locking down the Secure memory. In this case, software developers can create an application with an RTOS which runs entirely in Secure state.

In Secure state only:

- EXC_RETURN for creating a thread must be `0xFFFFFFFD`. The thread is Secure and without FP extension.
- The stack frame must be pointed to by PSP_S.

8.1.2 Extension of CMSIS-RTOS for Non-secure RTOS

The concept behind the Security extension is to split the software into security critical and non-critical code. Most RTOSs do not have access to the Secure side. Therefore, the secure memory space cannot be modified. Using a standardized API for the required communication between the two sides helps with code portability, even if the software developer has access to both security states. Therefore, many embedded applications that are based on Armv8-M architecture might have an RTOS running from the Non-secure state.

In this configuration, Non-secure threads can call Secure APIs in the Secure firmware. This means that these threads need Secure stack space allocation. The context switching of the RTOS must also switch the Process Stack Pointer (PSP_S) and all other registers on the Secure side. To meet such requirements, the CMSIS-RTOS API is extended to support context switching of the Secure stack for Non-secure RTOS. The APIs are used by the Non-secure RTOS in initialization, thread creation, and context switching.

The CMSIS-RTOS API is standardized so that:

- The operations are identical across different chip products. There is compatibility between different RTOSs. Different secure software stacks, for example TF-M, work together.
- The API is open, so all RTOS designers can create RTOS running in the Non-secure states.

The CMSIS-RTOS API supplies functions to perform the following functions:

- Initialize Secure Process Stack management.
- Allocate Secure stack space for a thread. Because Non-secure software developers have no visibility of the Secure software details, this function does not have stack size requirement information. Typically, this API must allocate the maximum stack size required by API calls.
- Free Secure stack space for a thread. Free the allocated Secure stack space when a thread is removed or disabled.
- Store Secure content. If a context switch occurs when the current thread is in Secure state, the Non-secure RTOS calls this function to save the context of the thread before it is swapped out. Technically, the registers are in the Secure stack already, but the PSP_S value must be saved to the Task Control Block (TCB) in the Secure world.

- Load Secure content. If the OS has to switch to a context that has previously been saved, use this function to restore the context by setting PSP_S and all other registers.

CMSIS-RTOS APIs are defined in the CMSIS-Core header file tz_context.h as follows:

Function name	Description
TZ_InitContextSystem_S	Initialize the secure context memory system. This function is called during RTOS initialization.
TZ_AllocModuleContext_S	Allocate context memory for calling secure software modules. This function is called on the creation of a thread.
TZ_FreeModuleContext_S	Release previously allocated context memory. This function is called on the termination of a thread.
TZ_LoadContext_S	Load the secure context. This function is called on a thread context switch.
TZ_StoreContext_S	Store the secure context. This function is called on a thread context switch.

CMSIS contains a reference implementation tz_context.c. See here.

8.2 Context-switching operations

Implementations can be different depending on the RTOS design requirements.

8.2.1 RTOS design requirements

Several areas of the RTOS design requirements require attention. For example, to prevent Secure stack overflow, stacks for Secure software must be protected with stack limit registers.

To reduce stack size requirements, Process Stack Pointers (PSP) must be used for threads. The thread only needs to allocate stack for the thread and first level of the exception stack frame. This avoids the need to reserve stack space for multiple exception handlers.

Date structures called Task Control Blocks are often used to manage threads in RTOSs. They hold information about each thread's state and various attributes necessary for scheduling and execution, such as thread ID and execution priority.

Because the RTOS design uses PSP for thread stack management, the list of registers to be saved in the TCB includes:

- Callee saved registers R4-R11 in integer register banks. Caller saved registers R0-R3 and R12 are on the stack frame already.
- If FPU is used in threads, floating-point registers S16 to S31, S0 to S15, and FPSCR are saved in the stack frame with lazy stacking.
- PSP_NS and PSPLIM_NS if using ARMv8-M Mainline.
- CONTROL_NS and EXC_RETURN values. These hold the state information, privileged or unprivileged.
- MPU region configurations are optional.

The RTOS can also add additional information in the TCB, such as resources assigned to that task or if any mutexes held.

8.2.1.1 Impact of the AIRCR.PRIS bit

In the ARMv8-M architecture, Secure software can use the programmable Prioritize Secure Exceptions (PRIS) bit in the Application Interrupt and Reset Control Register (AIRCR) to shift the Non-secure exception priority by 1 bit so that the Non-secure exceptions are mapped to lower half of priority levels. See Exception prioritization for more details.

As a result of the PRIS bit, the Non-secure exceptions can have lower priority than the lowest priority Secure exceptions. For example, with Armv8-M Baseline, if Secure PendSV is set to the lowest priority, the priority level is `0xC0`. For Non-secure exceptions, the lowest priority level is `0xE0`. As a result, if Secure PendSV is used for context switching, for example triggered by Secure SysTick exception, the following sequence can occur:

1. Non-secure IRQ is running at lowest priority level.
2. Secure SysTick is triggered and executed.
3. Secure SysTick handler set Secure PendSV pending status.
4. Secure SysTick handler exit, tail chained into Secure PendSV.

In this situation, the Secure PendSV must not execute context switching because a Non-secure interrupt handler is still running. To solve this issue, there are several possible solutions:

- Make sure that AIRCR.PRIS is not used.
- Check EXC_RETURN and ICSR.RETTOBASE before context switches.
- Use Non-secure PendSV at lowest priority level to call a Secure API to handle context switching.

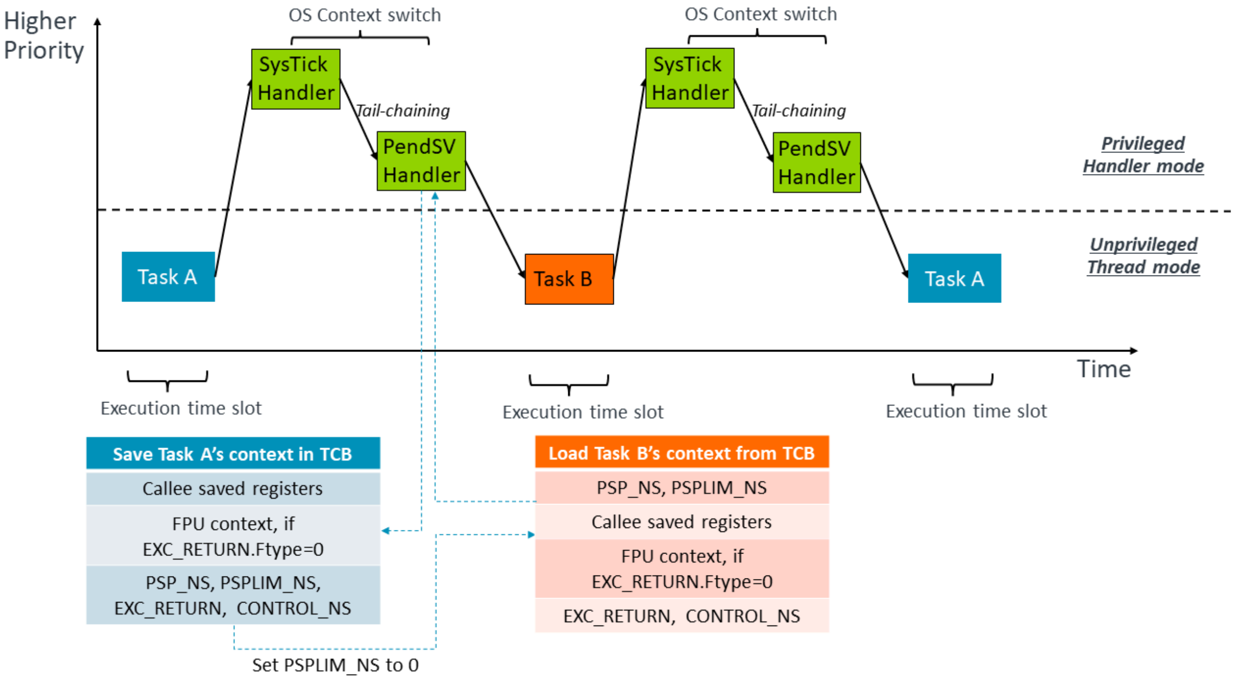
RTOSs running in Non-secure state do not have the same issue.

8.2.2 RTOS in the Non-secure state

When the RTOS is running from the Non-secure side, the RTOS kernel can only access the Non-secure Stack Pointers, stacks and other state, and must use the CMSIS-RTOS API v2 - Security Extension API to handle context switching.

If the background thread is Secure, then use TZ_Store_Context_S() to save the Secure contexts. Otherwise, it is usual to implement context switching in Non-secure PendSV at lowest Secure priority level, as the following figure shows.

Figure 8-3: General context switch flow with RTOS on Non-secure side



In this arrangement, there is no other active exception running when the Non-secure PendSV handler executes. This avoids the need for checking for running exception handlers before starting context switch.

8.2.3 RTOS in the Secure state

The RTOS kernel can access both Secure and Non-secure stack and Stack Pointers when the RTOS is running in the Secure state.

Like the figure in [RTOS in Non-secure state], the process of context switching is almost same when RTOS is running in the Secure world. In addition to the Non-secure information, the TCB might contain Secure information, including register contents and stack pointer in Secure state, such as PSP_S, PSPLIM_S and CONTROL_S. When thread switching, this Secure information also needs to be saved and reloaded in the same way as Non-secure information.

It is easier to place all TCBs in Secure memory because it can hold the register content of a thread, even if the thread is in a Non-secure state.

8.2.4 Supporting multiple Secure software libraries

Secure RTOS designers must also consider cases where Secure MPU is used for supporting multiple, mutually distrustful Secure software libraries. Here, the Secure MPU marks all except one of the libraries as inaccessible. If an attempt is made to call into an inaccessible library, the resulting MemManage fault can be used to trigger a context switching between different libraries on demand. In addition to the MPU blocking access to the code of an inaccessible library, it must also block access to data the library is using, including the stack. As a result, each library requires its own Secure stack.

8.3 Secure software development design considerations

Secure software must manage several design and development functions.

8.3.1 Prevent Secure thread mode reentrancy

The execution of a secure function can be preempted by a Non-secure exception raised. A Non-secure exception can schedule a second thread to run and call into the same or a related Secure function. This can lead to security issues or bugs if the secure function modifies the secure state in a way that is protected against reentrancy.

Armv8.1-M provides the hardware setting CCR_S.TRD to prevent the Secure thread mode reentrancy. Secure software must set `SCB->CCR.TRD` to 1 at system initialization. When a Secure thread mode reentrancy occurs, it is detected and a fault exception is triggered. Because this feature only protects against thread mode reentrancy, it must be used with the PXN MPU attribute to ensure that the code cannot be called reentrantly from handler mode.

8.3.2 Security and privilege combination

In the Armv8.0-M architecture, privilege and security were handled as unrelated and orthogonal concepts. An attacker might control a privileged mode such as Non-secure privileged mode and a Secure state such as Secure unprivileged mode. This means that they control both privilege and security. Also, by definition, they can access modes, such as Secure privileged mode, with that combination.

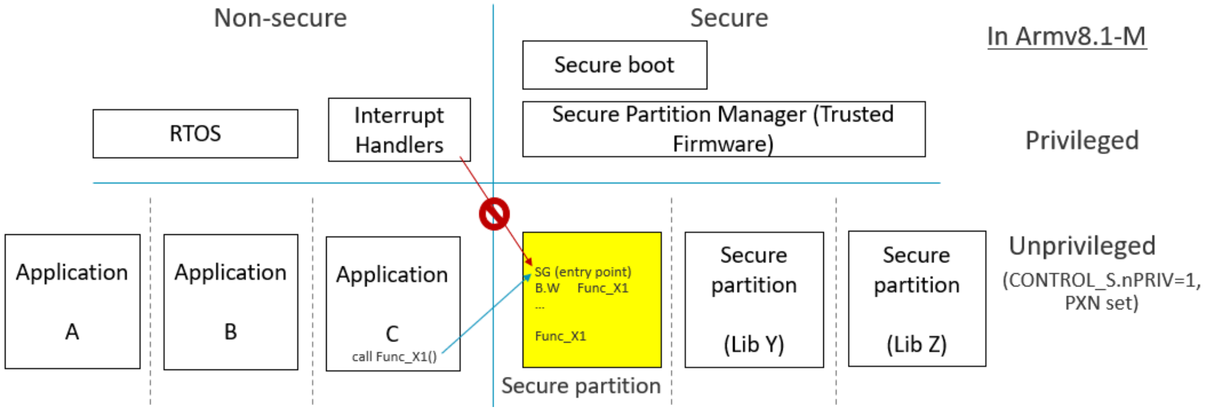
In the Armv8.1-M architecture, extra features are added so that the privileged mode is separated between the two security states. To use this, we recommend that software uses:

1. Privilege eXecute Never (PXN) bit configuration
2. Secure Process stack pointer (PSP_S) in thread mode

8.3.2.1 Using PXN bit

Armv8.1-M architecture introduces Privilege eXecute Never (PXN) bit to prevent secure privilege escalation attacks. We added the PXN bit to ensure that untrusted Secure code in a third-party unprivileged partition cannot be called from Non-secure handler mode and be executed in a privileged mode. However, it can have separate Secure entry points.

Figure 8-4: PXN bit to prevent Secure escalation attacks



An attempt to execute marked PXN in a privileged mode raises a MemManage fault.

See Significance of XN and PXN bits in *Armv8-M Memory model and Memory Protection User Guide*.

8.3.2.2 Using PSP_S in Secure thread mode

A secure escalation attack is possible if only a single stack is used for both secure privileged and secure unprivileged mode. To prevent such an attack, we recommend that you use secure process stack pointer (PSP_S) in thread mode and MSP_S in handler mode.

8.3.3 AIRCR.BFHFNMINS considerations

To support legacy code running on Armv8-M devices with Security extension, one of the common use-cases is to run all the application code in Non-secure state with only a small amount of boot code running in Secure state. One way to prevent the largely unused Secure state from being used by a malicious root kit, is to transfer critical exceptions, such as BusFault and NMI, to the Non-secure state before locking down the access to Secure state and preventing any further execution of Secure code. When you want software to lock down the Secure state in this way, set AIRCR.BFHFNMINS bit to 1 before switching to Non-secure state.

Caution

When AIRCR.BFHFNMINS bit is set to 1, some Non-secure exceptions, such as NMI, can preempt all Secure code execution, apart from the Secure HardFault handler. Therefore, an attacker can preempt the execution of critical Secure code, such as the SecureFault handler. They can potentially use this as a way to gain access to the Secure state, for example by regaining control after an initial attack has been detected. To prevent such attacks it is important that no Secure code is executed after AIRCR.BFHFNMINS has been set to 1. To achieve this, all Secure exceptions except the Secure HardFault must be disabled. This ensures that no

Secure functions can be called from the Non-secure state, for example by not marking any memory as Non-secure callable in the SAU.

8.3.4 EXC_RETURN.DCRS and EXC_RETURN.FType

In an operating system, when you need to pass arguments from threads to SVC handlers or when an exception needs to look at the exception stack frame stack frames are artificially created. This happens, for example, when determining the PC of the instruction that caused a fault. In these cases, it is important that EXC_RETURN.DCRS and EXC_RETURN.FType bits are checked correctly.

8.3.5 Interrupt deprivileging

To enable unprivileged code to services interrupts from peripherals, interrupts must be isolated by deprivileging interrupts to create a sandbox. See What is interrupt deprivileging. If Secure handler is deprivileging an exception, it must leave the Secure main stack (MSP_S) sealed when it transitions to Thread mode. The handler must ensure that the PSP_S stack being used by the sandbox is also sealed.

8.3.6 Non-reentrant exceptions

We recommend that the none re-entrant software that is callable from the non-secure thread mode be protected by setting the `CCR_S.TRD` (Secure Configuration Control Register - Thread Reentrancy Disable) to 1. When `CCR_S.TRD` bit is set to 1, then the processor checks to see if an exception stack frame is present on the stack when an SG instruction is executed in thread mode. If an exception stack frame is present it indicates that a partially executed call is already in progress and has been interrupted. If a reentrancy attempt is detected in thread mode when `CCR_S.TRD` is set, then the processor raises a INVEP SecureFault.

8.3.7 Secure floating-point contexts

`FPCCR.TS` is a configuration bit that determines whether the security of the floating-point values is important and required for the application. If `FPCCR.TS` bit is set, then the processor protects the Secure Floating point context set by the floating-point register values and ensures that it is not visible to Non-secure software.

We also recommend that you set `FPCCR.CLRONRET` and `FPCCR.CLRONRETS` to 1, to ensure that the processor automatically clears the floating-point context registers to 0 on exception return.

9. Armv8-M Security Extension use case examples

We describe the following use cases:

- hello-world-in-security-states: A basic example that shows the system boots from Secure state and the basic setup in the Secure state.
- security-func-call-params-passing: Describes function calls with parameter passing across a security boundary within an Armv8-M processor configured with the Security Extension.
- basic-Non-secure-only-program: A simple example that shows minimal boot code in Secure state, with the complete application run in Non-secure state without any function call backs from Non-secure state to Secure state.
- exception-across-security-state: Describes exceptions in different Secure modes, and how the system with Security Extension handles them.

The source code for these examples is in the GitHub repository.

9.1 Generic information

The embedded application based on Armv8-M Security Extension is split into a Non-secure software and Secure software. Secure and Non-secure projects are usually developed separately. Each Secure and Non-secure project can be completely written in C language. The C compiler compiles the C program code into object files and then generates the executable program image file using the linker.

9.1.1 Tool versions

This example project is created, built, and run using the following tool versions:

- Arm Development Studio 2022.c, available in Arm DS). It provides support for creating and debugging Secure and Non-secure applications for Armv8-M based devices.
- Arm Compiler for Embedded 6
- Fast Models Fixed Virtual Platforms (FVP) 11.19 (installed with Arm DS). This is the platform to run the Secure and Non-secure software image.
- CMSIS 5.9.0, available from the ARM-software/CMSIS_5.

9.1.2 What does the program image contain?

When a project is built using the toolchain, it generates a program image. Inside this program image, there is the application code that we want to run, and a range of software components. The program image contains the following:

- Vector table
- Reset handler or startup code
- C startup code and data
- C runtime library code and data
- Application code and data

See Armv8-M Memory Model and Memory Protection User Guide for information on the software components in the CMSIS M-profile project. The following sections describe the Armv8-M Security Extension software settings.

9.1.3 Stack sealing

In normal execution, on entry into Non-secure state from Secure state, the Non-secure software can trigger illegal return operations. This manipulation can cause the return process to use a different stack pointer that is not the Secure stack previously used to switch from Secure to Non-secure state. This incorrect Secure stack is not initialized before entering the Non-secure state, which leads to security vulnerability. This problem can be mitigated by stack sealing.

Secure software typically seals the stack in the Reset handler. It places a special value at the address above the real stack space. Arm recommends using the value `0xFEF5EDA5`. CMSIS-Core for Cortex-M already provides function **__TZ_set_STACKSEAL_S (uint32_t* stackTop)** that implements the stack stealing operation.

```
   __NO_RETURN void Reset_Handler(void)
   {
       __set_MSPLIM((uint32_t)(&__STACK_LIMIT));
       #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
       __TZ_set_STACKSEAL_S((uint32_t *)(&__STACK_SEAL));
       #endif

       SystemInit();                            /* CMSIS System Initialization */
       __PROGRAM_START();                       /* Enter PreMain (C library entry
point) */
   }
```

9.1.4 System memory map

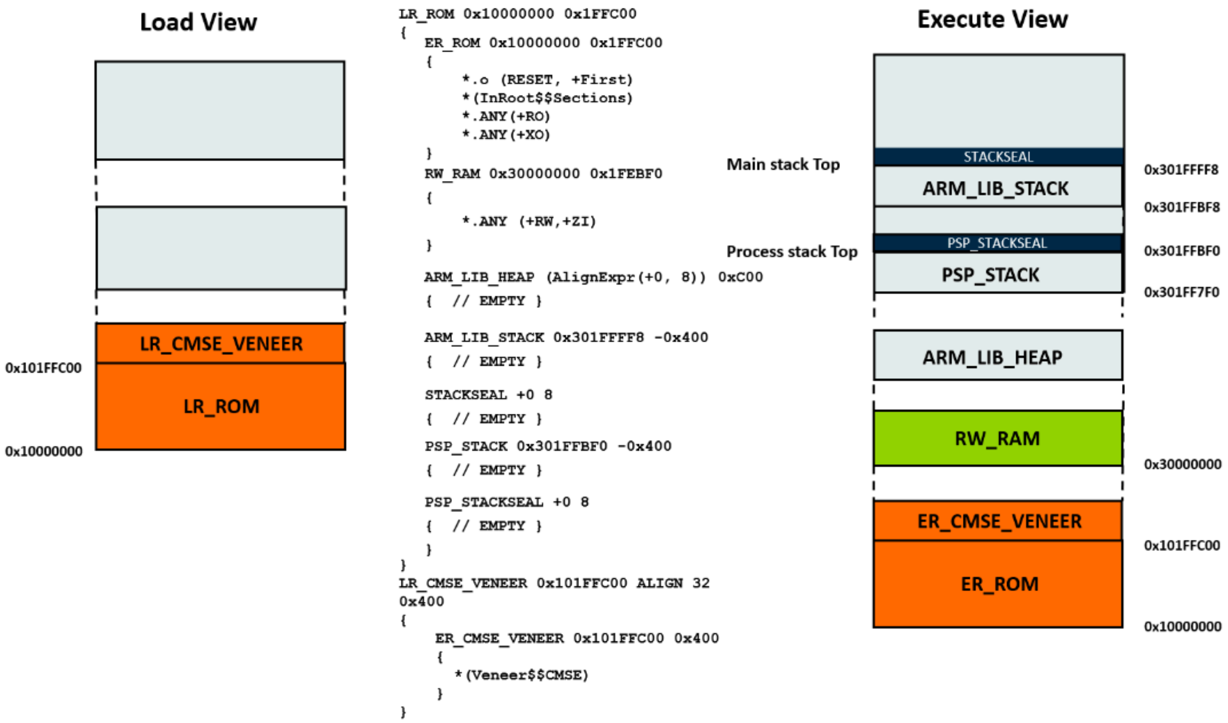
The example projects use the MPS2 Fixed Virtual Platform (FVP) to run the program. The memory map for the MPS2 FVP is in MPS2 - memory map for models with the Arm®v8 M additions.

The SAU regions are configured via Secure software. The final memory Security attribution is determined by the stricter Security attribution specified by the SAU and by the IDAU. The following table shows the system memory map used in the user case examples.

Memory	IDAU setting	SAU setting	Description
`0x10000000-0x101FFBFF`	Secure,NSC	Secure	Secure code SRAM
`0x101FFC00-0x101FFFFF`	Secure,NSC	Secure,NSC	Secure, NSC code SRAM
`0x00200000-0x003FFFFF`	Non-secure	Non-secure	Non-secure code SRAM
`0x30000000-0x301FFFFF`	Secure	Secure	Secure data SRAM
`0x20200000-0x203FFFFF`	Non-secure	Non-secure	Non-secure data SRAM

A scatter file enables you to control where the linker places different parts of your image for your target, including the location and size of various memory regions that are mapped to ROM, RAM, and FLASH. Considering the target memory map of MPS2 FVP model, the following figure shows the execution regions that are defined for the Secure project in the use case examples.

Figure 9-1: Scatter file layout in Secure project



```
LR_ROM 0x10000000 0x1FFC00
{
   ER_ROM 0x10000000 0x1FFC00
   {
       *.o (RESET, +First)
       *(InRoot$$Sections)
       *.ANY(+RO)
       *.ANY(+XO)
   }
   RW_RAM 0x30000000 0x1FEBF0
   {
       *.ANY (+RW,+ZI)
   }
   ARM_LIB_HEAP (AlignExpr(+0, 8)) 0xC00
   {  // EMPTY }

   ARM_LIB_STACK 0x301FFFF8 -0x400
   {  // EMPTY }

   STACKSEAL +0 8
   {  // EMPTY }
   PSP_STACK 0x301FFBF0 -0x400
   {  // EMPTY }

   PSP_STACKSEAL +0 8
   {  // EMPTY }
   }
}
LR_CMSE_VENEER 0x101FFC00 ALIGN 32
0x400
{
    ER_CMSE_VENEER 0x101FFC00 0x400
    {
      *(Veneer$$CMSE)
    }
}
```

See the scatter file on GitHub for the full definitions of the different regions in Secure project.

In Secure project, CMSE veneer section must be placed in the Non-secure Callable (NSC) region. In Arm Compiler toolchain, this is indicated by the section `Veneer$$CMSE` in the scatter file.

```
LR_CMSE_VENEER __CV_BASE ALIGN 32 __CV_SIZE
 {
     ER_CMSE_VENEER __CV_BASE __CV_SIZE
     {
         *(Veneer$$CMSE)
     }
 }
```

The NSC region start address and size information must be aligned with the system memory map and the SAU configuration.

In the Secure project use case example, another region is defined in the scatter file for the PSP stack. The MSP and PSP are initialized in different regions and sealed. In the application code, it switches to use the PSP stack.

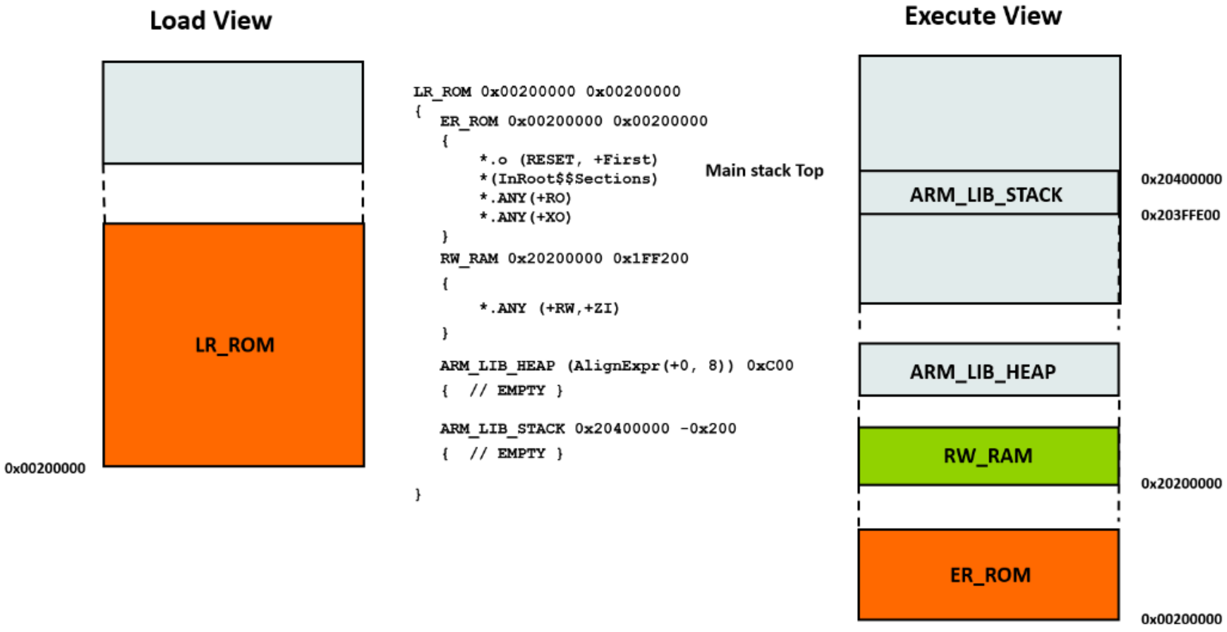
```
PSP_STACK __PSP_STACK_TOP EMPTY -__PSP_STACK_SIZE {
}

PSP_STACKSEAL +0 EMPTY __STACKSEAL_SIZE {
}
```

9.1.4.1 Scatter file definition in Non-secure project

The Non-secure project has a corresponding scatter file to define the load and execution regions in the Non-secure memory.

Figure 9-2: Scatter file layout in Non-secure project



```
LR_ROM 0x00200000 0x00200000
{
   ER_ROM 0x00200000 0x00200000
   {
        *.o (RESET, +First)
        *(InRoot$$Sections)
        *.ANY(+RO)
        *.ANY(+XO)
   }
   RW_RAM 0x20200000 0x1FF200
   {
        *.ANY (+RW,+ZI)
   }
   ARM_LIB_HEAP (AlignExpr(+0, 8)) 0xC00
   {  // EMPTY }
   ARM_LIB_STACK 0x20400000 -0x200
   {  // EMPTY }
}
```

In the Non-secure project use case example, PSP is not used. If the PSP is required, you must move it into another region by changing the scatter file.

See the scatter file on GitHub for the full definition of the different regions in Non-secure project.

9.1.5 SAU regions configuration in Secure project

The SAU registers are programmed via Secure software, `partition_ARMv81MML.h` from CMSIS-Core pack is used to set up SAU regions. The following snippet of `partition_ARMv81MML.h` file contains SAU regions settings. See the partition_ARMv81MML.h for the full configuration of the SAU regions.

```
    //  Initialize SAU Region 0 Setup SAU Region 0 memory attributes
    #define SAU_INIT_REGION0    1
    #define SAU_INIT_START0    &Image$$ER_CMSE_VENEER$$Base          // Start
address
    #define SAU_INIT_END0      &Image$$ER_CMSE_VENEER$$Limit - 1     // End address
    #define SAU_INIT_NSC0       1  //Region is 0: Non-secure 1:Secure, Non-secure
Callable

    //  Initialize SAU Region 1 Setup SAU Region 1 memory attributes
    #define SAU_INIT_REGION1    1
    #define SAU_INIT_START1    __ROM_BASE_NS
    #define SAU_INIT_END1      __ROM_BASE_NS + __ROM_SIZE_NS -1
    #define SAU_INIT_NSC1        0

    // Initialize SAU Region 2 Setup SAU Region 2 memory attributes
    #define SAU_INIT_REGION2     1
    #define SAU_INIT_START2     __RAM_BASE_NS
    #define SAU_INIT_END2       __RAM_BASE_NS + __RAM_SIZE_NS -1
    #define SAU_INIT_NSC2        0
```

- Image$$ER_CMSE_VENEER$$Base and Image$$ER_CMSE_VENEER$$Limit are the linker-generated symbols.
- Image$$ER_CMSE_VENEER$$Base is the execution address for ER_CMSE_VENEER section
- Image$$ER_CMSE_VENEER$$Limit is the address of the byte beyond the end of the ER_CMSE_VENEER section. ER_CMSE_VENEER is the section for CMSE veneer code.

Other address information must be aligned with the system memory map table.

The TZ_SAU_Setup() function uses these settings to configure SAU regions one by one. When SAU is enabled, the memory not covered by any enabled SAU region is Secure. The following code snippet shows you how to configure SAU regions.

```
    __STATIC_INLINE void TZ_SAU_Setup (void)
    {

     #if defined (SAU_INIT_REGION0) && (SAU_INIT_REGION0 == 1U)
      SAU_INIT_REGION(0);
     #endif

     #if defined (SAU_INIT_REGION1) && (SAU_INIT_REGION1 == 1U)
     SAU_INIT_REGION(1);
     #endif

     #if defined (SAU_INIT_REGION2) && (SAU_INIT_REGION2 == 1U)
     SAU_INIT_REGION(2);
     #endif

     SAU->CTRL = ((SAU_INIT_CTRL_ENABLE << SAU_CTRL_ENABLE_Pos) &
  SAU_CTRL_ENABLE_Msk) |
          ((SAU_INIT_CTRL_ALLNS  << SAU_CTRL_ALLNS_Pos)  & SAU_CTRL_ALLNS_Msk)
   ;

    }
```

9.1.6 Import library

The Secure project adds a linker option, such as –import-cmse-lib-out, into the Arm Compiler toolchain to generate import an library for the Non-secure project. When building the Secure project, it generates a Secure image and import library.

For more information about the Arm compiler toolchain, see the following resources:

- Arm Compiler for Embedded Reference Guide
- Arm Compiler for Embedded User Guide

9.2 hello-world-in-security-states

This example shows the details of memory security configuration and protection available in Cortex-M processors with Security Extension. This example project shows:

- How to build a Secure image
  - Program the SAU
  - Configure system control registers
  - Configure interrupts and exceptions for a security state
  - Enable SecureFault and writing a basic SecureFault handler
  - Create an import library to export to the Non-secure image
  - Configure Non-secure registers
  - Boot and call into a Non-secure image
- How to build a Non-secure image that can call a Secure image

The source code for this example is at security/hello-world-in-security-states.

9.2.1 Secure project structure

The file structure of Secure project is:

```
│   main_s.c
|   init.c
|   init.h
|   interface.h
|   region_defs.h
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6_s.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
                partition_ARMv81MML.h
```

The files in the example Secure project are as follows:

- main_s.c: The code in this file does the following:
  - Enable SecureFault exception and implement SecureFault handler
  - Implement a simple Secure API that can be called from Non-secure side
  - Call the Secure system initialization function
  - Launch Non-secure image
- init.c: The code in this file does the Secure system initialization, including
  - Configuring exceptions and interrupts in Security state
  - Configuring system control registers in Security state

    - Implementing system Secure initialization function, which calls the above functions
- init.h: Contains macro and function declarations.
- interface.h: The header file that declare the Secure API
- region_defs.h: Define the system memory address and size
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, initializes the MSP and PSP and seals the stack
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6_s.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/partition_ARMv81MML.h: The code in this file does the following:
    - Define SAU region address, size and Security attributes
    - Implement function to set up SAU regions

9.2.1.1 System security initialization

The system Security configuration is set up at the Secure software initialization. In the simple example, FPU/MVE is not used in Secure world, and can be used in Non-secure world, general security configurations are as follows:

- Configure memory region Security attribution using SAU region programming
- Configure Security states of interrupts via NVIC_ITNS registers
- Set up SecureFault priority and enable SecureFault
- Because this example enables the SecureFault handler, you must ensure that Non-secure code cannot pre-empt that handler. Therefore, Non-secure interrupts are deprioritized, by setting AIRCR.PRIS, and the priority of the SecureFault handler is set to 0.
- Target exception (BusFault, HardFault, NMI) to Secure state
- Configure Non-secure control points, including:
    - Setting up SLEEPDEEP, SYSRESETREQ are only configured from Secure state
    - Configuring Non-secure Access Control register (NSACR) to enable Non-secure access to FPU and MVE
- This example does not use FPU in Secure state. Set Floating-Point Context Control Register (FPCCR) as below:
    - Configure TS as 0 so the Floating-point registers are treated as Non-secure even when the core is in the Secure state
    - Configure CLRONRETS as 0 so CLRONRET is configurable from Non-secure state
    - Configure CLRONRET as 0 so floating-point caller saved registers are not cleared automatically on exception return

The following snippet of `init.c` file contains system control registers configuration. See the init.c for the full system Security configuration.

```
void System_Config()
{

  /* SCB->SLEEPDEEP bit is only configurable from the Secure state */
  SCB->SCR    = (SCB->SCR    & ~(SCB_SCR_SLEEPDEEPS_Msk    )) |
               ((SCB_CSR_DEEPSLEEPS_VAL     << SCB_SCR_SLEEPDEEPS_Pos)     &
 SCB_SCR_SLEEPDEEPS_Msk);

  /* Set Application Interrupt and Reset Control Register as below:
   * SYSRESETREQ accessible from Secure state
   * Non-secure exceptions are de-prioritized.
   * BusFault, HardFault, and NMI are Secure
    */
  SCB->AIRCR = (SCB->AIRCR & ~(SCB_AIRCR_VECTKEY_Msk   | SCB_AIRCR_SYSRESETREQS_Msk
 |
                SCB_AIRCR_BFHFNMINS_Msk | SCB_AIRCR_PRIS_Msk          ))
        |
               ((0x05FAU                    << SCB_AIRCR_VECTKEY_Pos)      &
 SCB_AIRCR_VECTKEY_Msk)      |
               ((SCB_AIRCR_SYSRESETREQS_VAL << SCB_AIRCR_SYSRESETREQS_Pos) &
 SCB_AIRCR_SYSRESETREQS_Msk) |
               ((SCB_AIRCR_PRIS_VAL         << SCB_AIRCR_PRIS_Pos)         &
 SCB_AIRCR_PRIS_Msk)         |
               ((SCB_AIRCR_BFHFNMINS_VAL    << SCB_AIRCR_BFHFNMINS_Pos)    &
 SCB_AIRCR_BFHFNMINS_Msk);


  FPU->FPCCR = (FPU->FPCCR & ~(FPU_FPCCR_TS_Msk | FPU_FPCCR_CLRONRETS_Msk |
 FPU_FPCCR_CLRONRET_Msk)) |
               ((FPU_FPCCR_TS_VAL        << FPU_FPCCR_TS_Pos       ) &
 FPU_FPCCR_TS_Msk       ) |
               ((FPU_FPCCR_CLRONRETS_VAL << FPU_FPCCR_CLRONRETS_Pos) &
 FPU_FPCCR_CLRONRETS_Msk) |
               ((FPU_FPCCR_CLRONRET_VAL  << FPU_FPCCR_CLRONRET_Pos ) &
 FPU_FPCCR_CLRONRET_Msk );


  /* Enable Non-secure access to Floating-point Extension and MVE */
  SCB->NSACR = (SCB->NSACR & ~(SCB_NSACR_CP10_Msk | SCB_NSACR_CP11_Msk)) |
               ((SCB_NSACR_CP10_11_VAL << SCB_NSACR_CP10_Pos) & (SCB_NSACR_CP10_Msk
 | SCB_NSACR_CP11_Msk));

}
```

9.2.1.2 Set up SecureFault exception

SecureFault exception must be set up correctly in the Secure software:

- Set priority to SecureFault exception
- Enable SecureFault
- In the SecureFault handler, print the SecureFault address Register (SFAR) and SecureFault status Register (SFSR) value for further analysis and clear the error status bits in SFSR before return from handler.
- Importantly the SecureFault handler must not allow Non-secure code to execute again as that may allow the attack to continue.

See the main_s.c for the SecureFault handler implementation.

9.2.1.3 Launch Non-secure image

After Secure software finishes Security initialization, Secure software sets up the MSP_NS from the first item in the Non-secure vector table and makes a call branch to the Non-secure reset handler.

```
typedef void (*funcptr_void) (void) __attribute__((cmse_nonsecure_call));

/* Secure main() */
int32_t main(void) {

    funcptr_void NonSecure_ResetHandler;
    ...
    Secure_System_Init();

    __TZ_set_MSP_NS((uint32_t)(VectorTable_NS[0]));
    SCB_NS->VTOR = (uint32_t)VectorTable_NS;

    /* Get Non-secure reset handler address from Non-secure vector table */
    NonSecure_ResetHandler = (funcptr_void)(VectorTable_NS[1]);

    /* Start Non-secure state software application */
    NonSecure_ResetHandler();
    ...
}
```

9.2.1.4 Implement a simple Secure API

Secure project implements a simple Secure API with CMSE function attribute __**attribute__((cmse_nonsecure_entry))**:

```
void __attribute__((cmse_nonsecure_entry)) simple_secure_lib_call_from_nonsecure()
{
  printf("S: Calling Secure function from Non-secure state\n");
}
```

9.2.2 Non-secure project structure

The file structure of Non-secure project is:

```
|   main_ns.c
|   interface.h
|   hello-world-in-security-states-CMSE-Lib.o
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
```

- main_ns.c: Call the simple Secure API
- interface.h: The header file for Secure API declaration. This file is imported from the Secure project and is required for a Non-secure image to call the Secure API.
- hello-world-in-security-states-CMSE-Lib.o: import library generated in Secure project, this file is imported from the Secure project and is required for a Non-secure image to call the Secure API.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, then initializes the MSP.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.

9.2.2.1 Add the import library to the Non-secure project

Add the interface header file `interface.h` and the import library hello-world-in-security-states-CMSE-Lib.o to the Non-secure project. Build and generate the Non-secure image.

9.2.2.2 Call Secure API in the Non-secure project

Call the Secure API as a normal function call:

```
int main(void)
{
  /* The Non-secure example call secure function  */

  printf("NS: Hello World in Non-secure State \n");

  /* call secure function */
  simple_secure_lib_call_from_nonsecure();

  printf("Example Project: hello-world-in-security-states End\n");

  return 0;
}
```

9.2.3 Output in Target Console

For this example, the output console shows the following:

```
Example Project: hello-world-in-security-states Start
S: Hello World in Secure State
NS: Hello World in Non-secure State
S: Calling Secure function from Non-secure state
Example Project: hello-world-in-security-states End
```

9.3 security-func-call-params-passing

This example project is based on the hello-world-in-security-states project framework and aims to demonstrate different function calls across the security boundary.

The secure project implements secure functions that can be called by the Non-secure side:

- ns_callable_fn1:
    - Support four parameters with integer and floating-point type
    - Add the received parameters
    - Return the result
- ns_callable_fn2:
    - Support a structure pointer as an input parameter that points to a structure with more than four elements
    - Call a CMSE intrinsic to check the structure read permission from Non-secure state
    - Sum up the structure elements
    - Return the result
- ns_callable_init:
    - Support a function pointer type parameter
    - Calls a CMSE intrinsic to create a Non-secure function pointer with the Least Significant Bit (LSB) cleared
    - Saves the Non-secure function pointer to a global callback function pointer (callback_NS)
- ns_callable_fn3:
    - Supports a data pointer type and length parameters
    - Call a CMSE intrinsic to check the address range has read permission from Non-secure state
    - Process data block after checking
    - Call the Non-secure callback function to pass the processing result to Non-secure side

The Non-secure project:

- Calls ns_callable_fn1 with four arguments and print the function return value
- Calls ns_callable_fn2 with a structure pointer argument that points to a structure with more than four elements. Then print value returned from the secure function
- Implements func_ns as a Non-secure function that is passed one integer argument, and print this value
- Calls ns_callable_init with the Non-secure function pointer (func_ns) as an input argument
- Calls ns_callable_fn3 with a data pointer and length arguments

The source code for this example is at security/security-func-call-params-passing

9.3.1 Secure project structure

The file structure of Secure project is:

```
│   main_s.c
|   init.c
|   init.h
|   interface.c
|   interface.h
|   region_defs.h
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6_s.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
                partition_ARMv81MML.h
```

- main_s.c: The code in this file does the following:
  - Enable SecureFault exception and implement SecureFault handler
  - Call the Secure system initialization function
  - Launch Non-secure image
- init.c: The code in this file does the Secure system initialization, including
  - Configuring exceptions and interrupts in Security state
  - Configuring system control registers in Security state
  - Implementing system Secure initialization function, which calls the above functions
- init.h: Contains macro and function declarations.
- interface.c: The code in this file implements four secure APIs that can be called by Non-secure side
- interface.h: The header file that declare the Secure APIs
- region_defs.h: Define the system memory address and size
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, initializes the MSP and PSP and seals the stack
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6_s.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/partition_ARMv81MML.h: The code in this file does the following:
  - Define SAU region address, size and Security attributes
  - Implement function to set up SAU regions

The Secure software configures the FPU settings correctly as follows:

- Set up CPACR to enable all access from the Secure state
- Set up CPPWR so that power to FPU can be controlled by the Secure state

- Configure NSACR to enable Non-secure access to FPU and MVE
- Set FPCCR as below:
    - Protect Floating-point values in the Secure state
    - CLRONRET is read-only from Non-secure state
    - Clear floating-point caller saved registers on exception return is enabled

The hello-world-in-security-states example describes the other steps to initialize system Security configuration, set up SecureFault exception, and launch the Non-secure image.

```
void System_Config()
{
  ...
  /* Enable Non-secure access to Floating-point Extension and MVE */
  SCB->NSACR = (SCB->NSACR & ~(SCB_NSACR_CP10_Msk | SCB_NSACR_CP11_Msk)) |
               ((SCB_NSACR_CP10_11_VAL << SCB_NSACR_CP10_Pos) & (SCB_NSACR_CP10_Msk
 | SCB_NSACR_CP11_Msk));

  /* SU10 & SU11 fields(FPU power control) are accessible from the Secure state*/
  SCnSCB->CPPWR = SCnSCB_CPPWR_SUS10_Msk | SCnSCB_CPPWR_SUS11_Msk;

  /* Set Floating-Point Context Control Register as below:
   * Treat Floating-point registers as Secure
   * CLRONRET is read-only from Non-secure state
   * Clear floating-point caller saved registers on exception return is enabled
   */
  FPU->FPCCR = (FPU->FPCCR & ~(FPU_FPCCR_TS_Msk | FPU_FPCCR_CLRONRETS_Msk |
 FPU_FPCCR_CLRONRET_Msk)) |
               ((FPU_FPCCR_TS_VAL        << FPU_FPCCR_TS_Pos       ) &
 FPU_FPCCR_TS_Msk       ) |
               ((FPU_FPCCR_CLRONRETS_VAL << FPU_FPCCR_CLRONRETS_Pos) &
 FPU_FPCCR_CLRONRETS_Msk) |
               ((FPU_FPCCR_CLRONRET_VAL  << FPU_FPCCR_CLRONRET_Pos ) &
 FPU_FPCCR_CLRONRET_Msk );
 ...
}
```

9.3.1.1 Implement Secure APIs

interface.c implements four Secure APIs as follows:

- ns_callable_fn1 supports four input parameters with integer and floating-point type, one return value.

```
float __attribute__((cmse_nonsecure_entry)) ns_callable_fn1(int32_t a, int32_t b,
 int32_t c, float d)
{
  float ret = 0;
  ret  = a + b + c + d;
  return ret;
}
```

- ns_callable_fn2 supports a structure pointer as an input parameter that points to a structure with more than four elements. It calls a CMSE intrinsic to check the structure read permission from Non-secure state, sums up the structure elements and return the result.

```
int32_t __attribute__((cmse_nonsecure_entry)) ns_callable_fn2(S* ptr)
{

  int32_t ret = 0;
  S*  in_check = NULL;

  /* check the memory pointed to by the pointer support read permission from Non-
secure state   */

  in_check = cmse_check_pointed_object(ptr,CMSE_NONSECURE|CMSE_MPU_READ);
  if (in_check == NULL) {
    printf("S: Non-secure access to the data structure is not permitted \n\r");
    exit(0);
  }
  else {
    ret = in_check->a + in_check->b + in_check->c + in_check->d + in_check->e;
    return ret;
  }
}
```

- ns_callable_init supports a function pointer type parameter. It calls a CMSE intrinsic to create a Non-secure function pointer with the Least Significant Bit (LSB) cleared and saves the Non-secure function pointer to a global callback function pointer (callback_NS).

```
void __attribute__((cmse_nonsecure_entry)) ns_callable_init(funcptr callback)
{
  callback_NS = (funcptr_NS)cmse_nsfptr_create(callback);
}
```

- callback_NS is a global function pointer. It saves the Non-secure function pointer passed from Non-secure side. It is initialized to `0xFFFFFFFF` which is an address with Execute Never memory attribute so a fault is raised if the callback function pointer is used before it is set.

```
funcptr_NS callback_NS = (funcptr_NS)0xFFFFFFFF;
```

- ns_callable_fn3 supports a data pointer type and length parameters. It calls a CMSE intrinsic to check the address range has read permission from Non-secure state, processes data block after checking, and calls the Non-secure callback function to pass the processing result to Non-secure side.

```
void  __attribute__((cmse_nonsecure_entry)) ns_callable_fn3(volatile uint32_t* ptr,
 uint32_t size)
{
  uint32_t* in_check = NULL;
  uint32_t i = 0;
  int32_t ret = 0;

  printf("S: check Non-secure permission to read the data region \n\r");

  /* check the address range has read permission from Non-secure state */
  in_check = cmse_check_address_range((void*)ptr,size,CMSE_NONSECURE|CMSE_MPU_READ);

  if (in_check == NULL) {
    printf("S: Non-secure read access to the data region 0x%x - 0x%x is not
 permitted \n\r",(unsigned int)ptr, ((unsigned int)(ptr + size))-1);
  exit(0);
  }
  else{
    printf("S: process Non-secure data in Secure side\n\r");
    for(i = 0 ; i < size; i++)
    {
        ret += *in_check;
        in_check++;
    }

    /* call the callback function to return the result back to Non-secure side */
    callback_NS(ret);
  }
}
```

9.3.2 Non-secure project structure

The file structure of Non-secure project is:

```
│   main_ns.c
│   callback_ns.c
│   callback_ns.h
│   interface.h
│   security-func-call-params-passing-CMSE-Lib.o
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
```

- main_ns.c: Calls the secure APIs.
- callback_ns.c: Implements Non-secure callback function.
- callback_ns.h: Header file for Non-secure callback function declaration.
- interface.h: The header file for Secure APIs declaration. This file is imported from the Secure project and is required for a Non-secure image to call the Secure APIs.
- security-func-call-params-passing-CMSE-Lib.o: import library generated in Secure project, this file is imported from the Secure project and is required for a Non-secure image to call the Secure APIs.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, then initializes the MSP.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.

Non-secure software must configure SCB->CPACR to enable Non-secure access to FPU/MVE during system initialization.

9.3.2.1 Implement Non-secure callback function

func_ns() is a Non-secure function. It implements a Non-secure function that has one integer parameter, and prints this value:

```
void func_ns (int32_t arg1)
{
  printf("NS: Non-secure callback function get Secure processing result = %d\n
\r",arg1);
}
```

9.3.2.2 Add the import library to the Non-secure project

Follow the same steps outlined in the hello-world-in-security-states example to add the linker option for security-func-call-params-passing-CMSE-Lib.o while building the Non-secure project in Arm DS.

9.3.2.3 Call the Secure APIs

Non-secure project calls the Secure APIs as a normal function call. You can check the example code in the Non-secure project. See the main_s.c for the SecureFault handler implementation.

9.3.3 Output in Target Console

For this example, the output console shows the following:

```
Example Project: security-func-call-params-passing Start
S: Hello World in Secure State
NS: Hello World in Non-secure State
NS: call Secure function
NS: get add result from Secure side: 1 + 2 + 3 + 5.800000 = 11.800000
NS: call Secure function with more input parameters
NS: get add result from Secure side: 1 + 2 + 3 + 4 + 5 = 15
NS: call Secure function with Non-secure function pointer as input parameter
NS: call Secure function with Non-secure data pointer as input parameter
S: check Non-secure permission to read the data region
S: process Non-secure data in Secure side
NS: Non-secure callback function get Secure processing result = 15
Example Project: security-func-call-params-passing End
```

9.4 basic-Non-secure-only-program

When migrating from processors based of Armv6/v7-M architecture to processors Armv8-M with the Security Extension, the processor will be in Secure state out of reset. This example code shows that only a small portion of secure boot code is required to switch the device to the Non-secure state, with minimal secure boot code.

This secure boot code contains basic initialization and secure settings that is required to be performed before calling any C library functions. Once the secure settings and Secure C library initializations are completed, the processor will hand-over its control to Non-secure state. On entering to Non-secure state, the Non-secure application program gets executed. It is expected that the Non-secure program gets executed without callback functions to Secure state.

The source code for this example is available at security/basic-Non-secure-only-program.

9.4.1 Secure project structure

The file structure of the Secure project is as follows:

```
│   main_s.c
│   region_defs.h
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6_s.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
                partition_ARMv81MML.h
```

The files in the example Secure project are as follows:

- main_s.c: The code in this file does the following:
  - Call the Secure system initialization function
  - Launch Non-secure image
- region_defs.h: Define the system memory address and size
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, initializes the MSP and PSP and seals the stack
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6_s.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/partition_ARMv81MML.h: The code in this file does the following:
  - Define SAU region address, size and Security attributes
  - Implement function to set up SAU regions

9.4.2 Settings for minimal secure boot code

This example shows how to initialize the processor from the Secure state, with minimal boot code, when Security Extension is included, before handing over to the Non-secure state.

These initializations are done in `Reset_Handler()` in startup_ARMv8MML.c before initializing the C-library using following steps:

- Boot from the Secure Reset Handler pointed by VTOR setup statically at compile time.
- Set NVIC_ITNS[...] to set all IRQs to Non-secure
- Set SCB_NSACR to allow Non-secure world to access to the FPU, coprocessor and Arm Custom instructions
- Set AIRCR.BFHFNMINS so that BusFault, HardFault and NMI are handled by the Non-secure world
- Set MSPLIM_S, and make sure the top of the stack pointers are sealed with a value of `0xFEF5EDA5`
- Configure Non-secure MSP register
- Program SAU regions to mark the NS table and NS application code as Non-secure, and the boot code, Secure table and MSP_S as Secure.
- Populate the VTOR_NS register

```
__NO_RETURN void Reset_Handler(void)
{

  /* Set NVIC_ITNS[] to set all IRQs to Non-secure */
  for (uint8_t i=0; i<sizeof(NVIC->ITNS)/sizeof(NVIC->ITNS[0]); i++) {
       NVIC->ITNS[i] = 0xFFFFFFFF;
  }

  /* Set SCB_NSACR to allow Non-secure world to access to the FPU, coprocessor
   * and Arm Custom Instructions. Enable Non-secure access to the Floating-point
   *  Extension and MVE. */

  uint32_t value_NSACR;
  value_NSACR = SCB->NSACR;
  SCB->NSACR = value_NSACR | 0x00000CFF;
  __ISB();

  /* Set AIRCR.BFHFNMINS so that BusFault, HardFault and NMI are handled by
   * the Non-secure world */


  uint32_t value_AIRCR;
  value_AIRCR = SCB->AIRCR;
  SCB->AIRCR = value_AIRCR | 0x2000;
  __ISB();


  /* Set MSPLIM_S, PSPLIM_S, and PSP_S */

  __set_PSP((uint32_t)(&__INITIAL_SP));

  __set_MSPLIM((uint32_t)(&__STACK_LIMIT));
  __set_PSPLIM((uint32_t)(&__STACK_LIMIT));

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  /* Make sure top of the stack pointers are sealed with a value of 0xFEF5EDA5 */
  __TZ_set_STACKSEAL_S((uint32_t *)(&__STACK_SEAL));
#endif


  /* Configure Non-secure Stack pointers and Stack limit registers */

  __TZ_set_MSP_NS((uint32_t)(0x50000)); /* Stack pointer for non-secure mode */


#if defined (__VTOR_PRESENT) && (__VTOR_PRESENT == 1U)
  SCB->VTOR = (uint32_t)(&__VECTOR_TABLE[0]);
  SCB_NS->VTOR = (uint32_t)TZ_START_NS;
#endif

  __DSB();
  __ISB();

  SystemInit();                                /* CMSIS System Initialization */
  __PROGRAM_START();                           /* Enter PreMain (C library entry point)
 */
}
```

9.4.3 Launch Non-secure image

Post completing the above minimal secure initializations,the processor can branch to the Non-Secure Vector Table address pointed by VTOR_NS.

```
/* VectorTable_NS: array of non-secure vector table */
uint32_t *VectorTable_NS = (uint32_t *)0x00200000;

...

/* Secure main() */
int32_t main(void) {

  funcptr_void NonSecure_ResetHandler;

  printf("S: hello from secure world \n");

  __TZ_set_MSP_NS((uint32_t)(VectorTable_NS[0]));

  /* Get non-secure reset handler from non-secure vector table */
  NonSecure_ResetHandler = (funcptr_void)(VectorTable_NS[1]);

  /* Start non-secure state software application */
  NonSecure_ResetHandler();

  return 0;
}
```

9.4.4 Non-secure project structure

The file structure of Non-secure project is:

```
│   main_ns.c
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6_s.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
                partition_ARMv81MML.h
```

The files in the example Non-secure project are as follows:

- main_ns.c: Call the simple Non-secure API
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/startup_ARMv81MML.c: Configures the vector table, then initializes the MSP.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/ARMv81MML_ac6.sct: Scatter file.
- RTE/Device/ARMv81MML_DSP_DP_MVE_FP/system_ARMv81MML.c: Target definitions.

9.4.5 Output in target console

For this example, the output console shows the following:

```
S: Hello from Secure world
NS: Hello from Non-secure world
```

The output shows that the processor enters the secure `main()`, in Secure state, and it prints a “Hello World” from the Secure state. Once the initializations are complete, the processor enters a Non-secure state and prints a “Hello World” from Non-secure main().

9.5 exception-across-security-state

This example demonstrates some use cases on how the Arm v8-m system with Security Extension handles exceptions, when the execution is in secure state or Non-secure state. The specific configurations of this example project are:

- Build a Secure image by:
    - Configuring interrupts and exceptions for a security state
    - Enabling SecureFault and writing a basic SecureFault handler
    - Implementing secure functions that can be called by Non-secure side:
        - ns_callable_pend_IRQs:
            - pend interrupts in specific order
        - Print_PendAndActiveStatus:
            - gets the status of IRQs by reading the ICSR register and print the active and pending status of exceptions
- Build a Non-secure image by:
    - Setting priority and enable Non-secure interrupt
    - Non-secure interrupt handlers
    - Calling ns_callable_pend_IRQs

- When handling the Non-secure interrupt 4, try to access the secure data and trigger SecureFault

The source code for this example is at security/exception-across-security-state.

9.5.1 Secure project structure

The file structure of Secure project is:

```
|   main_s.c
|   init.c
|   init.h
|   interface.h
|   interface.c
|   region_defs.h
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6_s.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
                partition_ARMv81MML.h
```

The files in the example Secure project are as follows:

- main_s.c: The code in this file does the following:
    - Implement a simple Secure API that can be called from Non-secure side
    - Call the Secure system initialization function
    - Launch Non-secure image
- init.c: The code in this file does the Secure system initialization, including
    - Configure secure exceptions and interrupts in Security state
    - Configure system control registers in Security state
    - Implement system Secure initialization function, which calls the above functions
    - Enable SecureFault exception and implement secure exception handler
- init.h: Contains macro and function declarations.
- interface.h: The header file that declare the Secure API
- interface.c: The code in this file includes two secure APIs with `cmse_nonsecure_entry` attribute, implementing:
    - Pend IRQs with different order.
    - Print the pending and active status of IRQs.

9.5.1.1 System Security initialization

The steps for System Security initialization and launching of the Non-secure image are almost the same as above. See the `hello-world-in-security-states` example.

As this program shows the behavior of exception handling with Security Extension, the specific exception configuration needs to be set at `Exception_Config()`, which includes the following:

- Configure security states of interrupts via NVIC_ITNS registers. Set IRQ0, IRQ2, and IRQ4 as Non-secure interrupts and IRQ1 and IRQ3 as Secure interrupts.
- Set up priority to exceptions. The priorities of BusFault, SecureFault, and MemoryManage Fault are less than `0x80`. This ensures that the Non-secure code cannot block critical secure exceptions. Set the BusFault and MemoryManage Fault as `0x00`, with SecureFault as `0x01`. The rest interrupts are set according to different test cases, seen at `ns_callable_pend_IRQs()`.
- Enable the secure exceptions via SCB.SHCSR registers and enable the secure interrupts via NVIC_ISER registers. Before enabling the secure exceptions, make sure the AIRCR.PRIS is set.

```
void Exception_Config()
{
  /* Set IRQ0, IRQ2 and IRQ4 as Non-secure interrupt */
  NVIC->ITNS[0] = 0x15;

  /* Set priority to exceptions  */
  NVIC_SetPriority(Interrupt1_IRQn, 0x40);
  NVIC_SetPriority(Interrupt3_IRQn, 0x85);
  NVIC_SetPriority(SecureFault_IRQn, 0x00);
  NVIC_SetPriority(MemoryManagement_IRQn, 0x00);
  NVIC_SetPriority(BusFault_IRQn, 0x00);

  /* Enable secure exceptions */
  NVIC->ISER[0] |= 1 << (uint32_t)Interrupt1_IRQn|
                   1 << (uint32_t)Interrupt3_IRQn;

  /* Enable MemManage Fault, SecureFault and BusFault*/
  SCB->SHCSR = SCB_SHCSR_SECUREFAULTENA_Msk | SCB_SHCSR_MEMFAULTENA_Msk |
 SCB_SHCSR_BUSFAULTENA_Msk;
}
```

9.5.1.2 Implement a simple Secure API

Secure project implements a simple Secure API with CMSE function attribute __**attribute__((cmse_nonsecure_entry))**. For this example, create two APIs `ns_callable_pend_IRQs()` and `Print_PendAndActiveStatus()` at [interface.c].

9.5.1.3 Build and generate the Secure image and import library

The link option –import-cmse-lib-out is also added to generate import library for the Non-secure project at Arm DS project properties setting.

When building the Secure project in Arm DS, it generates Secure image and import library `exception-across-security-state-CMSE-Lib.o`.

9.5.2 Non-secure project structure

The file structure of Non-secure project is:

```
|   main_ns.c
|   interface.h
|   IRQconfig_ns.c
|   IRQconfig_ns.h
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
                ARMv81MML_ac6.sct
                startup_ARMv81MML.c
                system_ARMv81MML.c
```

- main_ns.c: Initialize the Non-secure IRQs and call the simple Secure API
- interface.h: The header file for Secure API declaration. This file is imported from the Secure project and is required for a Non-secure image to call the Secure API.
- IRQconfig_ns.c: The file includes functions executed only at Non-secure world, such as Non-secure IRQs handler, initialization of Non-secure exceptions and functions to read secure data at Non-secure world.
- IRQconfig_ns.h: The file declares the functions at `IRQconfig_ns.c`.

9.5.2.1 Non-secure exception configuration

At `IRQconfig_ns.c`, the following functions are defined and executed only at Non-secure world:

1. `NS_Exception_Config()`: To set priority of Non-secure IRQs and deprioritize Non-secure exceptions, because the AIRCR.PRIS is set. With the de-priority, the actual priority of Non-secure exceptions is shifted `0x80` based on requested priorities as the following table shows. Enable the Non-secure IRQs.

Exceptions	Requested priority	NVIC_PRIO_BITS	Deprioritize	Actual priority
Non-secure IRQ0	0x12	3	yes	0xC0
Secure IRQ1	0x40	3	no	0x00
Non-secure IRQ2	0x00	3	yes	0x80
Secure IRQ3	0x85	3	no	0xA0
Non-secure IRQ4	0x63	3	yes	0xE0
SecureFault	0x00	3	no	0x00
MemoryManage Fault	0x00	3	no	0x00
BusFault	0x00	3	no	0x00

1. `read_secure_data()`: To read the Secure vector table at Non-secure world. Because this Non-secure program attempts to call a Secure program address without using a valid entry point, a SecureFault event is generated.

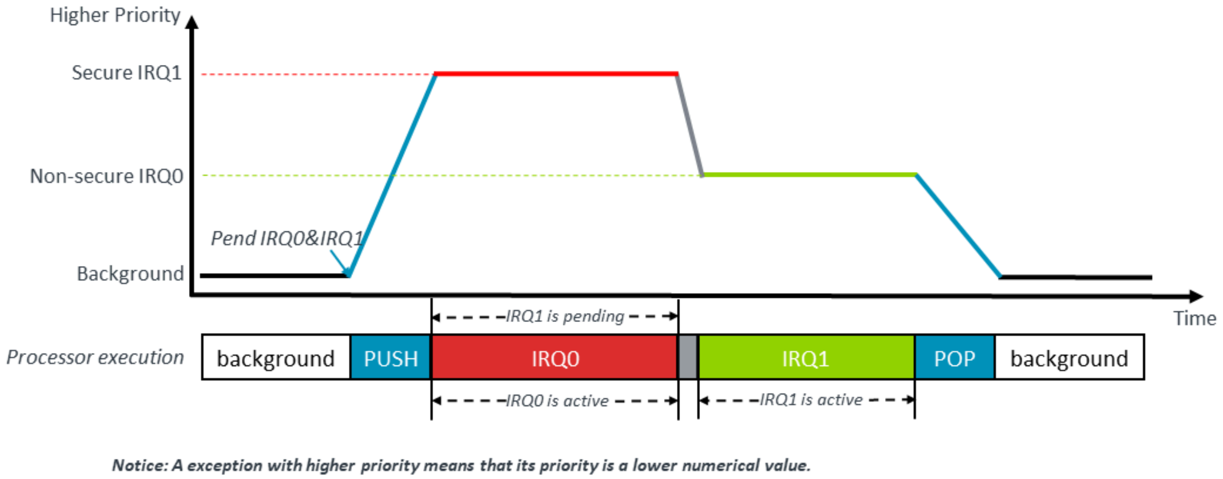
2. `Interrupt[...]_Handler()`: To overwrite the Non-secure IRQ handler and print the status of IRQs.

9.5.2.2 Call Secure API in the Non-secure project

At `main.c()`, after the completion of Non-secure exceptions, call the Secure API as a normal function call. The first one `ns_callable_pend_IRQs()` is to trigger IRQs according different cases.

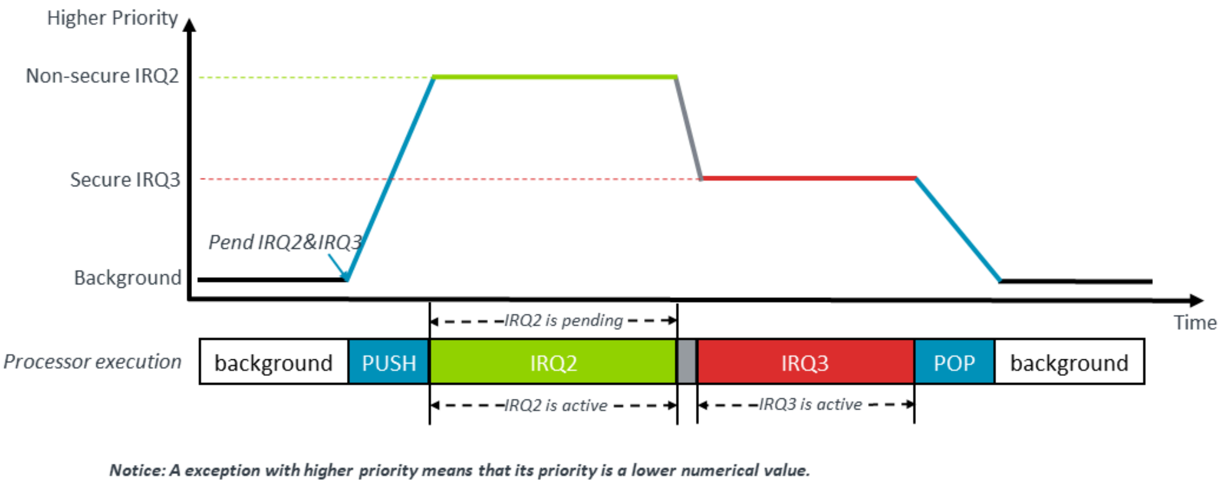
For case 1, the IRQ0 priority is `0xC0`, which is a higher numeric value than IRQ1 `0x00`. We trigger them at once, so the secure IRQ1 is handled first and the Non-secure IRQ0 is tailchained.

Figure 9-3: execution_flow_case1



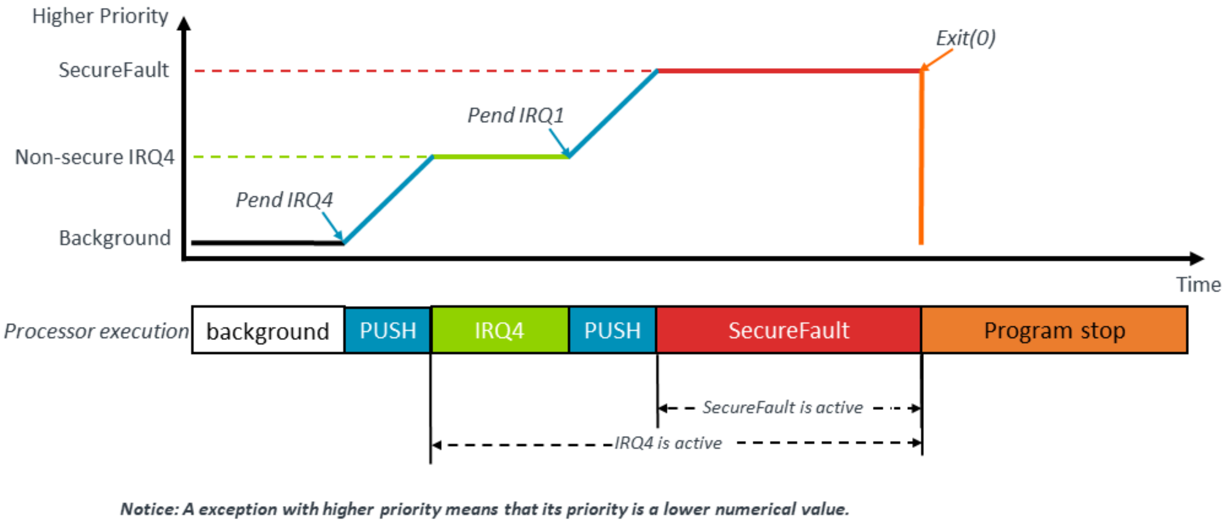
For case 2, as above, the IRQ2 priority is `0x80`, which is a lower numeric value than IRQ3 `0xA0`. The Non-secure IRQ2 is handled first and secure IRQ3 is tailchained.

Figure 9-4: execution_flow_case2



For case 3, trigger the Non-secure IRQ4 first and try to read the secure data at its handler. This pends SecureFault. Because the SecureFault priority is `0x00`, which is a lower numeric value than IRQ4 `0xE0`, the SecureFault preempts the IRQ4 and the program handles SecureFault first.

Figure 9-5: execution_flow_case3



```
void __attribute__((cmse_nonsecure_entry)) ns_callable_pend_IRQs(){
  printf("\n%s: Case1: start !     \n", get_state);
  NVIC->ISPR[0] |= 1 << (uint32_t)Interrupt0_IRQn|
                   1 << (uint32_t)Interrupt1_IRQn;

  printf("\n%s: Case2: start !     \n", get_state);
  NVIC->ISPR[0] |= 1 << (uint32_t)Interrupt2_IRQn|
                   1 << (uint32_t)Interrupt3_IRQn;

  printf("\n%s: Case3: start !     \n", get_state);
  NVIC->ISPR[0] |= 1 << (uint32_t)Interrupt4_IRQn;
}
```

As `Print_PendAndActiveStatus()` is called at each handler, when the program handles one exception, it prints the current active and pending status of exceptions.

```
void __attribute__((cmse_nonsecure_entry)) Print_PendAndActiveStatus(void){
  /* Read status of IRQs with exception number from ICSR register. */
  uint32_t ExcepNumPend    = (SCB->ICSR & SCB_ICSR_VECTPENDING_Msk)
                              >> SCB_ICSR_VECTPENDING_Pos;
  uint32_t NumsActive      = (SCB->ICSR & SCB_ICSR_RETTOBASE_Msk)
                              >> SCB_ICSR_RETTOBASE_Pos;
  uint32_t ExcepNumActive  = SCB->ICSR & SCB_ICSR_VECTACTIVE_Msk;

  if(ExcepNumPend > 0)
    printf("%s: The number of the highest priority pending exception is %d \n",
          get_state, ExcepNumPend);

  if(ExcepNumActive > 0){
    if(NumsActive == 0)
      printf("%s: There is more than one active exception. \n", get_state);
    else
      printf("%s: There is only one active exception. \n", get_state);

    printf("%s: The number of the highest priority active exception is %d \n",
 get_state,
          ExcepNumActive);
  }
}
```

9.5.3 Output in Target Console

The text output in the Target Console view is:

```
Example Project: exception-across-security-state Start
S: Hello World in Secure State

NS: Case1: start !
S: The number of the highest priority pending exception is 16
S: There is only one active exception.
S: The number of the highest priority active exception is 17
NS: There is only one active exception.
NS: The number of the highest priority active exception is 16

NS: Case2: start !
NS: The number of the highest priority pending exception is 19
NS: There is only one active exception.
NS: The number of the highest priority active exception is 18
S: There is only one active exception.
S: The number of the highest priority active exception is 19

NS: Case3: start !
We are accessing secure memory
S: There is more than one active exception.
S: The number of the highest priority active exception is 7
S: SecureFault occurred
S: SecureFault address Register:
  SCB->SFAR = 0x10000000
S: SecureFault status Register:
  SCB->SFSR = 0x00000048
-------------------------------------------------------
```