RTOS design considerations

Version 2.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
08 July 2016	0100_00	Confidential	First release
23 August 2016	0101_00	Confidential	Second release
28 February 2017	0200_00	Non-Confidential	Third release

Product Status

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

Web Address

http://www.arm.com

Contents

1 RTOS design for the ARMv8-M architecture

Several processor features have been extended in the ARMv8-M architecture. This can affect RTOS designs in several ways. Usually, the RTOS code must be updated to run on ARMv8-M architecture.

Some of the changes that are required are generic to RTOS designs:

- The change of Memory Protection Unit (MPU) programmers' model to Protected Memory System Architecture (PMSA) v8 means that an RTOS with MPU support must update MPU support code.
- The extension of the EXC_RETURN (Exception Return) value definition means that if the OS creates a thread by exception return with an artificially created stack frame, the value of EXC_RETURN used is likely to change.

Additional changes are required to take advantage of the new capabilities in the ARMv8-M architecture. For example:

- Stack limit features are available for both Secure and Non-secure states in the ARMv8-M architecture with Main Extension, and are available in Secure state for the ARMv8-M architecture. The stack limit feature enables stack overflow scenarios to be detected. Stack overflows are one of the most common sources of software errors in embedded systems.
- Depending on the configuration of the system, an RTOS can support both Secure and Non-secure threads. For such cases, you must update the RTOS to support extra stacks.
- For RTOS previous written for the ARMv6-M architecture, moving to the ARMv8-M architecture enables the OS to use exclusive access instructions for semaphore variable updates. This architecture enhancement also enables the ARMv8-M architecture with Main Extension and the ARMv8-M architecture versions of the OS to share semaphore code.
- If the RTOS is delivered in compiled library form, recompilation of the RTOS code enables the software to be optimized for ARMv8-M processors.

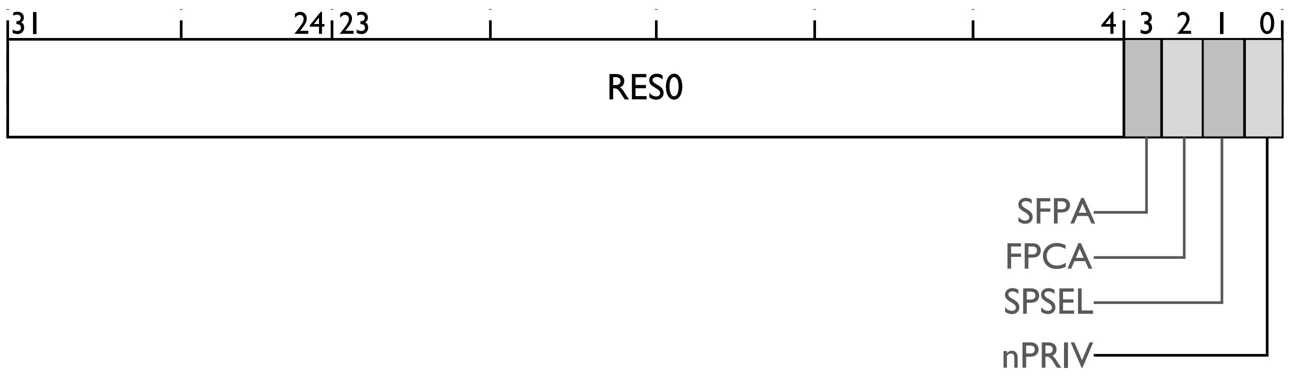
Depending on the system-level design around the ARMv8-M processor, the Secure software and associated resources might be locked down. This means that software developers can only update the Non-secure program address space and access to Non-secure hardware resources. Secure resources can only be accessed using APIs in the Secure firmware. Such restrictions affect RTOS designs and the Secure firmware. Multiple design configurations are possible.

There are several different possible systems and RTOS configurations. RTOS vendors might need to implement different configurations of their RTOS to address different product requirements.

To assist RTOS operations in devices with locked down Secure memories, CMSIS-RTOS has been extended with additional APIs. These APIs enable RTOS running in Non-secure domain to handle context switching.

2 Using the ARMv8-M architectural features

An RTOS can use the two stack pointers that the ARMv8-M architecture provides within each security state. The Main Stack Pointer (MSP) can be used by software running in Thread mode and is always used by Handler mode. A second register, the *Process Stack Pointer* (PSP) can be used by Thread mode software. The SPSel bit in the CONTROL register configures the stack pointer register that is used by Thread mode.



Bits	Field	Reset	Description
[31:4]	RES0	-	Reserved
[3]	SPFA	0	Secure floating-point active.
[2]	FPCA	0	Floating-point context active.
[1]	SPSEL	0	Sets the stack pointer to be used.
[0]	nPRIV	0	Defines the execution privilege in Thread mode.

Using both registers allows an RTOS to separate the stack that is used by a user application from the OS stack, and protects the OS stack from errors in a user thread that might cause stack corruption and affect the stack pointed to by PSP.

Note

Simple applications without an OS do not have to use this configuration and instead can configure Thread mode to use the MSP. This use of MSP is the default stack configuration out of reset.

If the Security Extension is implemented, MSP and PSP are banked between security states. The notations MSP_NS, PSP_NS, MSP_S and PSP_S are used to indicate which banked version is being referred to, either Secure (S) or Non-secure (NS). In addition, the CONTROL.SPSel bit is also banked, allowing different stack configurations in Secure and Non-secure Thread modes.

2.1 Stack and stack limit

If the ARMv8-M architecture Security Extension is implemented, a stack limit feature is provided using stack limit registers accessible using MSR and MRS instructions.

```
MRS RN, <x>                ; copy the value of <x> into RN
```

```
MSR <x>, RN                ; write the value in RN into <x>.
```

Where <x> is one of: MSP, PSP, MSPLIM, or PSPLIM.

For example:

```
MRS R0, MSPLIM             ; copy MSPLIM value into R0
```

Secure software can also use MSP_NS, PSP_NS, MSPLIM_NS and PSPLIM_NS to access the Non-secure version of those registers.

```
; Execute in Secure state
MSR MSP, R1                ; set Secure MSP
MSR MSP_NS, R2             ; set Non-secure MSP
```

The following table lists the available stack pointers and Stack limit register.

Stack	Stack pointers	Corresponding stack limit register
Secure Main Stack Used by Secure handlers, and Secure thread when bit[1] of the Secure CONTROL register is 0.	MSP_S	MSPLIM_S
Secure Process Stack Used by Secure threads when bit[1] of the Secure CONTROL register is 1.	PSP_S	PSPLIM_S
Non-secure Main Stack Used by Non-secure handlers, and Non-secure thread when bit[1] of the Non-secure CONTROL register is 0.	MSP_NS	MSPLIM_NS (available on ARMv8-M architecture with Main Extension only)
Non-secure Process Stack Used by Non-secure threads when bit[1] of the Non-secure CONTROL register is 1.	PSP_NS	PSPLIM_NS (available on ARMv8-M with Main extension only)

If, during a stack operation (for example, PUSH) the stack address is less than the corresponding stack limit register, a fault event is raised.

2.2 SVCall and PendSV exceptions

The SVCall and PendSV exceptions are banked between Secure and Non-secure states.

When the processor is in Secure state, the SVC exception handling sequence fetches the exception vector from the Secure vector table and executes the SVCall handler in Secure state. When the processor is in

Non-secure state, the SVC exception handling sequence fetches the exception vector from the Secure vector table and executes the SVCall handler in Non-secure state.

Similarly, the PendSVSet and PendSVClr bit in the Interrupt Control and State Register (ICSR) is banked. Secure software can also trigger Non-secure PendSV using the Non-secure alias of the ICSR(`0xE002ED04`).

The priority level registers for SVC and PendSV are also banked between Secure and Non-secure states.

2.3 SysTick timer

A timer is a requirement of most operating systems. ARMv8-M processors provide a basic countdown timer called SysTick that can be used to generate a periodic interrupt. SysTick reduces the need for software porting, because the SysTick programmers' model is the same on all Cortex-M devices. In ARMv8-M processors, the SysTick timer is an optional feature and designers can decide if the chip must include this timer or not.

In the ARMv8-M architecture, the programmers' model of the SysTick timer remains unchanged. However, the SysTick timer is banked, so there can be two physical SysTick timers in the design:

- When the processor is in Secure state, the Secure SysTick timer is accessed.
- When the processor is in Non-secure state, the Non-secure SysTick timer is accessed.

Each of these 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 architecture with Main Extension, there is the option of implementing just one SysTick. In this case, the SysTick timer that is implemented can be configured to be Secure or Non-secure using bit [24] of the Interrupt Control and State Register (ICSR).

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

- If this bit is 0 SysTick targets the Secure state (default).
- If this bit is 1 SysTick targets the Non-secure state.

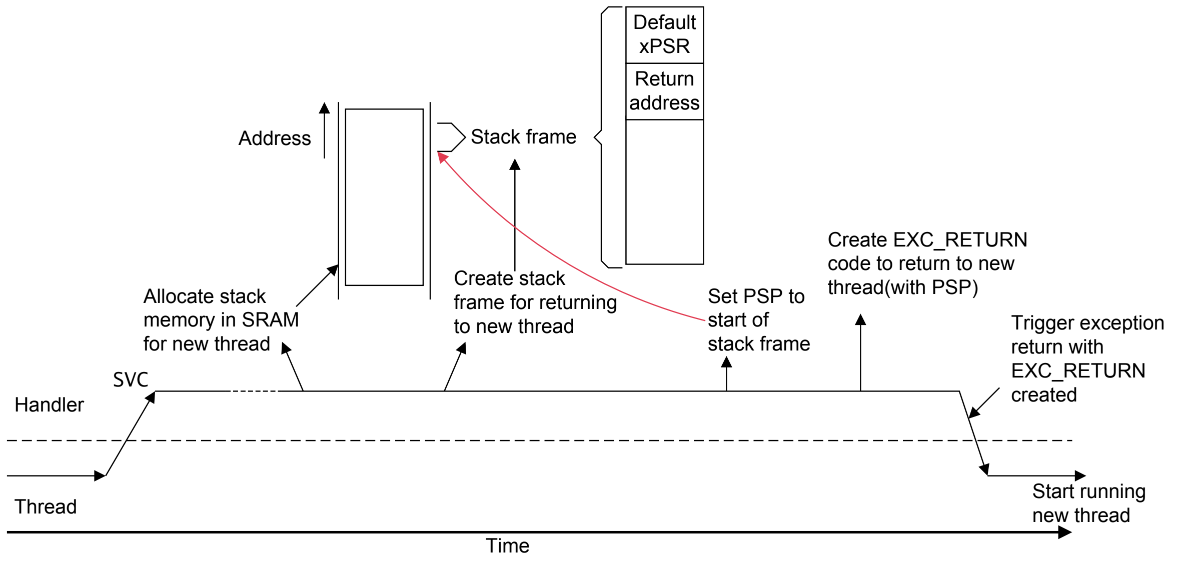
The STTNS bit is accessible only when the processor is in Secure state, and only if the design configuration of the processor selects one SysTick implementation.

An RTOS can also use device-specific timer peripherals for time keeping and task scheduling. In such case, the security domain of the timer must match the RTOS configuration. For example, if the RTOS is running in a Secure domain, the timer that is used by the RTOS must be configured to be Secure accesses only, and the corresponding interrupt must also be configured as Secure.

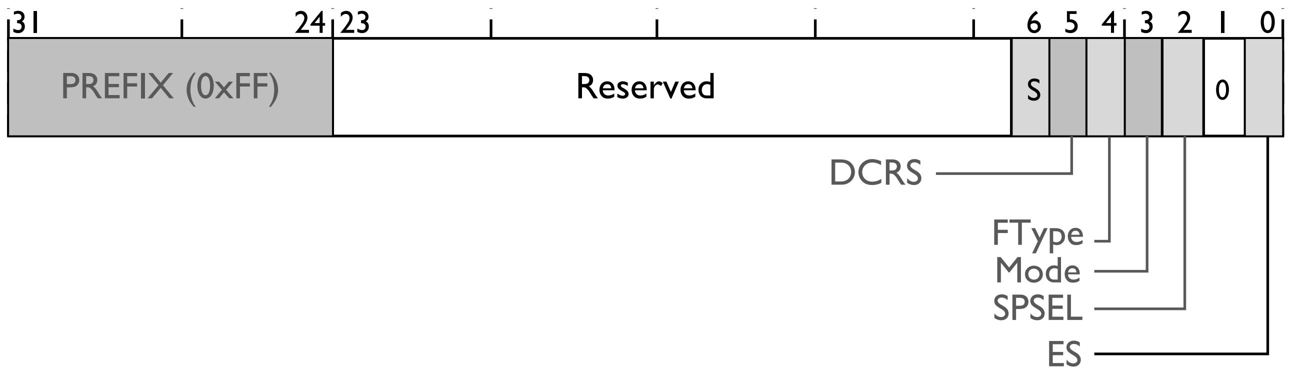
2.4 EXC_RETURN

EXC_RETURN is an exception return mechanism.

Many real-time operating systems create threads using an exception return with an artificially created stack frame, as the following figure shows.


Default xPSR
Return address
Address
Stack frame
Create EXC_RETURN code to return to new thread(with PSP)
Allocate stack memory in SRAM for new thread
Create stack frame for returning to new thread
Set PSP to start of stack frame
Trigger exception return with EXC_RETURN created
SVC
Handler
Thread
Start running new thread
Time

In the ARMv8-M architecture, EXC_RETURN has been updated as follows:



Bits	Field	Changes	Description
31:24	PREFIX	-	Must be 0xFF to be a valid EXC_RETURN code.
23:7	Reserved (fix to 1)	-	IMPLEMENTATION DEFINED
6	S	New	Secure or Non-secure stack. This bit indicates whether a Secure or Non-secure stack is used. 0 Non-secure stack used. 1 Secure stack used.
5	DCRA	New	Default callee register stacking.

			This bit indicates whether the default stacking rules apply, or whether the callee registers, for example, R4-R11, are already on the stack. 0 Stacking of the callee saved registers skipped. 1 Default rules for stacking the callee registers are followed.
4	FType	-	Stack Frame Type. This bit indicates whether the stack frame is a standard integer only stack frame, or an extended stack frame with floating-point register contents. 0 Extended stack frame 1 Standard stack frame.
3	Mode	-	Mode. This bit indicates the Mode that was stacked from. 0 Handler mode. 1 Thread mode.
2	SPSEL	-	Stack pointer selection. This bit indicates which stack point the exception frame resides on. 0 Main stack pointer. 1 Process stack pointer.
1	Reserved (fix to 0)	-	Reserved
0	ES	New	Exception Secure. The security domain the exception was taken to. 0 Non-secure. 1 Secure.

3 OS configurations

Various operating system configurations are possible with the ARMv8-M architecture. Since the ARMv8-M architecture Security Extension is optional, a processor design implementing the ARMv8-M architecture might have no TrustZone technology support.

Without TrustZone technology support, the processor:

- Is always in Non-secure state.
- Has no Security Attribution Unit (SAU).
- Has no Secure MPU, Secure SysTick, or SCB registers.

Designs which include the Security Extension allow multiple configuration arrangements. The following table describes several.

Configuration	RTOS Design requirement
RTOS running in Secure state. Some threads are Secure, and some threads are Non-secure. Cross domain APIs calls are also possible between domains.	Each thread might have Secure and Non-secure stack space allocation.
RTOS running in Non-secure state. Threads are Non-secure, but potentially some of the threads can call Secure API.	All threads have a Non-secure stack, and some threads that call Secure API also must have Secure stack allocation. Secure firmware must include CMSIS-RTOS API (version 2) to support handling of stack switching on the Secure side.
RTOS running in Non-secure state. Threads are Non-secure and there is no calling to Secure APIs. (Secure state is not used by the application).	All threads have Non-secure stack only. The design is identical to RTOS for chip designs without TrustZone technology support.
RTOS and all applications running in Secure state. Non-secure state is not used.	All threads have Secure stack only.
RTOS on Secure domain and use the idle thread of that OS to run a second OS (not real time) in the Non-secure domain.	Secure RTOS Each thread might have Secure and Non-secure stack space allocation. Non-secure OS All threads have a Non-secure stack, and some threads that call Secure API also must have Secure stack allocation. Secure firmware must include CMSIS-RTOS API (v2) to support handling of stack switching on the Secure side.

More configurations are possible, for example, by having a bare-metal or interrupt driven application in one security domain and running an RTOS in the other domain.

Case 1 – RTOS in Secure state

When a real-time operating system is running in Secure state, threads can be Secure or Non-secure. It is also possible to have cross-domain API calls between Secure and Non-secure software, which means each thread can have both Secure and Non-secure stack allocation.

When creating a thread from Secure handler mode, the EXC_RETURN code can be:

- `0xFFFFFFFD` if the new thread is Secure. The stack frame is PSP_S.
- `0xFFFFFFBD` if the new thread is Non-secure. The stack frame is PSP_NS.

Case 2 – RTOS in Non-secure state

In Non-secure state threads are Non-secure by default. However, threads can call APIs in Secure firmware memory, so threads can have both Secure and Non-secure stack allocation.

Because the RTOS is on a Non-secure domain, it cannot directly access the Secure stack pointers and is therefore unable to handle context switching of Secure stacks. To solve this problem, new versions of the CMSIS-RTOS APIs are being developed which support Secure stack management.

When creating a thread from Non-secure handler mode, the EXC_RETURN code must be `0xFFFFFFBC` (thread is Non-secure). The stack frame must be pointed to by PSP_NS.

Case 3 – RTOS and application all in Non-secure state only

Threads are always Non-secure when the RTOS and application are in Non-secure state.

In Non-secure state only, the RTOS can be based on existing code that was written for ARMv8-M processors. However, the following modifications are still required:

- EXC_RETURN for creating a thread must be `0xFFFFFFBC` (thread is Non-secure). The stack frame must be pointed to by PSP_NS.
- MPU support code must be updated to support the new programmers' model.

Case 4 – RTOS and application all in Secure state only

It is possible for a microcontroller vendor to 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 the Secure domain.

In Secure state:

- EXC_RETURN for creating a thread must be `0xFFFFFFFD` (thread is Secure).
- The stack frame must be pointed to by PSP_S.

4 Extension of CMSIS-RTOS for Non-secure RTOS

In some microcontroller devices, the Secure memory space can be locked down and therefore cannot be modified. As a consequence, many embedded applications that are based on ARMv8-M architecture might have an RTOS running from the Non-secure side.

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

The API is standardized so that:

- The operations are identical across different processors (allowing RTOS products to work on a range of ARMv8-M-based processors from different microcontroller vendors).
- The API is open, so all RTOS designers can create RTOS running in the Non-secure domains.

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

- Initialize Secure Process Stack management.
- Allocate Secure stack space for a thread. Since 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 that is required by API calls.
- Free Secure stack space for a thread.

  This frees 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 Trace 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.

5 RTOS design requirements

Several areas of RTOS design requirements require particular 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 the first level of the exception stack frame, which avoids having to reserve stack for exceptions and interrupt handlers.

Since Secure software can force Secure exceptions to have higher priority than Non-secure exceptions using AIRCR.PRIS, the Secure PendSV exception handler might be able to preempt a Non-secure interrupt. Therefore, the Secure RTOS scheduler might have to check EXC_RETURN bit[3], or then ICSR.RETTOBASE (Return to base) status bit in handling context switching.

5.1 RTOS running in the Secure world

The RTOS kernel can access both Secure and Non-secure stack and stack pointers when the RTOS is running in the Secure world. The *Task Control Blocks* (TCBs) might contain Secure information, including register contents in Secure state and the PSP_S values.

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

Assuming that the RTOS design uses PSP for thread stack management, the list of registers that must be saved in the TCB include:

- Callee saved registers (R4 to R11) in integer register banks.

  Note

  Caller saved registers R0-R3, R12 are on the stack frame already.

- If CONTROL.FPCA is 1 and EXC_RETURN bit [4] is 0, then floating-point registers S16 to S31. (S0 to S15, and the floating-point Status and Control register (FPSCR) are saved in the stack frame with lazy stacking.)
- PSP_S and PSP_NS, PSPLIM_S and PSPLIM_NS (if using ARMv8-M architecture with Main Extension).
- CONTROL_S (Secure), CONTROL_NS(Non-secure) and EXC_RETURN values. These registers hold the state information, which can be privileged or unprivileged, and Secure or Non-secure.
- MPU region configurations (optional).

If EXC_RETURN.SPSel is `0b0`, an exception handler other than PendSV is active. Here context switching cannot be performed and the RTOS must schedule context switch operations in the next tick.

If EXC_RETURN.FType is `0b0`, floating-point context is active and context switching can be implemented in Secure PendSV at lowest Secure priority level, as follows:

1. Store callee saved registers (R4 to R11) in TCB.

2. Save FPU S16-S31 to the TCB. S0-S15 and FPSCR would be saved automatically with lazy stacking.
3. Save PSP_S, PSP_NS, PSPLIM_S, PSPLIM_NS, EXC_RETURN, CONTROL_S, and CONTROL_NS in the TCB.
4. Set PSPLIM_S and PSPLIM_NS to 0 to disable the limit check.
5. Load PSP_S, PSP_NS, then PSPLIM_S and PSPLIM_NS for new thread.
6. Load callee save registers and FPU registers from the new TCB.
7. Load EXC_RETURN and CONTROL from the new TCB.

At the end of this sequence, the exception returns to the new thread.

5.2 RTOS running in the Non-secure world.

When the RTOS is running in the Non-secure world, the RTOS kernel can only access the Non-secure stacks and stack pointers, and must use the CMSIS-RTOS v2 Security Extension API to handle context switching. In this arrangement, there is no other active exception running when the Non-secure PendSV handler executes, and this avoids the need for checking for running exception handlers before starting context switch.

Note

The detail is based on the CMSIS-RTOS API v2 and the APIs has not been finalized.

When the RTOS design uses PSP for thread stack management the list of registers to be saved in the TCB include:

- Callee saved registers (R4 to R11) in integer register banks.

  Note

  Caller saved registers R0-R3, and R12 are on the stack frame already.

- If FPCA is 1 and EXC_RETURN bit [4] is 0, then the 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 architecture with Main Extension).
- CONTROL_NS and EXC_RETURN values. These hold the state information, privileged or unprivileged.
- MPU region configurations (optional).

The RTOS can also add additional information in the TCB, if necessary.

Depending on the value of EXC_RETURN.S, the thread is running a Secure API (`0b1`) or a Non-secure API (0b0).

If running a Secure API, call `TZ_Store_Context_S()` to save the Secure contexts. Otherwise, it is common practice to implement context switching in Non-secure PendSV at the lowest Secure priority level, as follows:

1. Store callee saved registers (R4 to R11) in the TCB.

   Note

   Caller saved registers R0-R3, and R12 are on the stack frame already.

2. If EXC_RETURN.FType is 0, the floating-point context is active. Save FPU S16-S31 to the TCB. S0-S15 and FPSCR are saved automatically with lazy stacking.
3. Save PSP_NS, PSPLIM_NS, EXC_RETURN and CONTROL_NS in the TCB.
4. Set PSPLIM_NS to 0 to disable the limit check.
5. Load PSP_NS, then PSPLIM_NS for new thread.
6. If EXC_RETURN.S of the new thread is 1, call `TZ_Load_Context_S()` to update PSP_S and PSPLIM_S. Otherwise, load the following register from TCB:
   a. Load callee save registers.
   b. FPU registers S16-S31 if the EXC_RETURN bit 4 of the new thread is 0.
7. Load EXC_RETURN and CONTROL_NS from new TCB.

At the end of this sequence, the exception returns to the new thread.

5.3 The 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.

As a result of the PRIS bit, the Non-secure exceptions can have lower priority than the lowest priority Secure exceptions. For example, if Secure PendSV is set to 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:

- Non-secure IRQ is running at lowest priority level.
- Secure SysTick is triggered and executed.
- Secure SysTick handler set Secure PendSV pending status.
- 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:

- Option 1: Make sure that AIRCR.PRIS is not used.
- Option 2: Check EXC_RETURN and ICSR.RETTOBASE before context switches.

- Option 3: Use Non-secure PendSV at lowest priority level to call a Secure API to handle context switching.

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

5.4 Supporting multiple Secure software libraries.

Secure RTOS designers must also consider cases where the Secure MPU is used for supporting multiple Secure software libraries. Here the Secure MemManage fault is used to control context switching between different libraries. A Secure RTOS must be aware of such MPU context changes.