

How should an RTOS work in a TrustZone for Armv8-M environment?

arm

Joseph Yiu, Senior Principal Engineer, Arm Ltd

As presented at Embedded World 2019

White Paper

Abstract—TrustZone for Armv8-M is designed to be very flexible, but such flexibility can also lead to some confusion. For example, in the case of RTOS design, should the RTOS be running in the Secure world or Non-secure world? This is a technically challenging question, with pros and cons for both approaches, and this article will discuss those aspects in detail.

This paper will also cover how Arm helps software developers and ecosystem partners in this area - Arm will provide a reference implementation of software platform (Trusted Firmware-M or TF-M) as a part of the Arm Platform Security Architecture, and this paper will explain how the RTOS is integrated under TF-M.

Introduction

With the introduction of Armv8-M based microcontrollers from Nordic Semiconductor, Microchip, Nuvoton, NXP, ST, Renesas and many more, Real-Time Operating System (RTOS) vendors need to update their RTOS to support [TrustZone for Armv8-M](#). While on the application side the migration is relatively straightforward, RTOS updates require more work due to the following architectural differences:

-
- ✦ Addition of the TrustZone security extension (optional to chip implementations)
 - ✦ Addition of a stack limit checking feature (RTOS support for this is optional)
 - ✦ New programmer's model for the Memory Protection Unit (MPU)
 - ✦ Extension of the EXC_RETURN values (exception return)

When we first introduced TrustZone to RTOS vendors, there were many questions about how an RTOS should work in new Armv8-M devices. The situation could be a bit confusing because TrustZone for Armv8-M is designed to be very flexible, and potentially there are different ways that an RTOS could work in a TrustZone based system. However, each of these methods has pros and cons, and not all of these methods make sense in real-world applications, primarily due to the need to ensure the software solutions are secure and satisfy requirements for many applications.

Many Arm activities have since been launched to address the challenge of enabling market adoptions. These include the [Platform Security Architecture \(PSA\)](#) initiative, announced in 2017, which allows the ecosystem to create security management solutions and more natural collaborations between various parties. The key goals for PSA include:

- ✦ Provide guidelines and reference implementations to allow product designers, including SoC designers, to develop secure solutions
- ✦ Increase awareness that security is no longer optional in the market
- ✦ Provide the industry with a common foundation and understanding of the security requirements
- ✦ Standardize a range of APIs to enable better compatibility between Secure firmware, RTOS, and applications

As a part of the PSA activities, a reference implementation of Secure software framework (Trusted Firmware-M) is available as open source project. Also, Arm provides:

- ✦ A reference RTOS with example integration with TF-M
- ✦ Reference hardware with reference PSA firmware

These resources enable RTOS vendors to understand the design concepts, and port their RTOS quickly and correctly.

There are other activities that Arm is implementing to enable market adoption, which include the development of IoT test chips and boards such as [Musca](#), a dual-core Cortex-M33 system, updates to the [Cortex Microcontroller Software Interface Standard \(CMSIS\)](#) to support TrustZone, as well as various technical resources available on [developer.arm.com](#).

OS support in the Armv8-M architecture

Before we get into the technical details of how RTOS should work in the Armv8-M architecture, first we will cover the architectural features in Armv8-M for OS support.

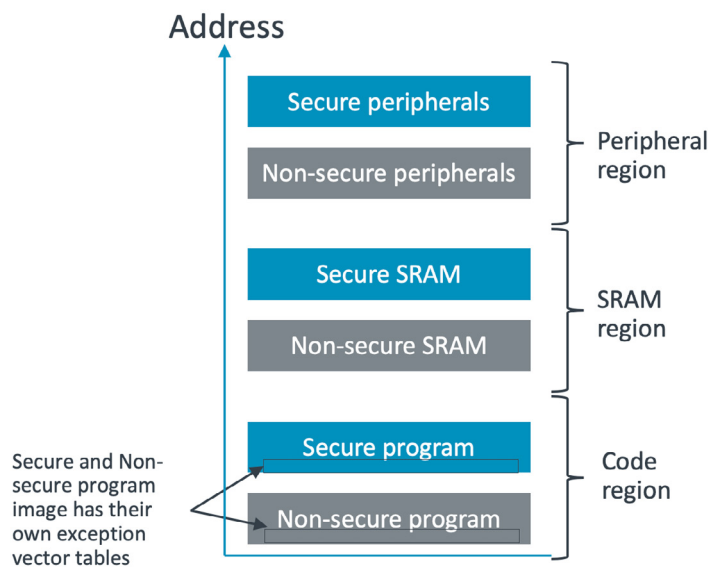
A. Secure and Non-secure memory partitioning and security states support

The most significant enhancement in the Armv8-M architecture is the TrustZone security extension. If a processor system implements TrustZone security, its memory system is partitioned into Secure and Non-secure address spaces. In a simplified view:

- ✚ When the processor is running software in Secure memory, it is in a Secure state and can access both Secure and Non-secure memory spaces
- ✚ When the processor is running software in Non-secure memory, it is in a Non-secure state and can only access Non-secure memory spaces

A typical TrustZone based system contains Secure and Non-secure memories and peripherals.

Fig. 1:
Memory partitioning into
Secure and Non-secure
regions

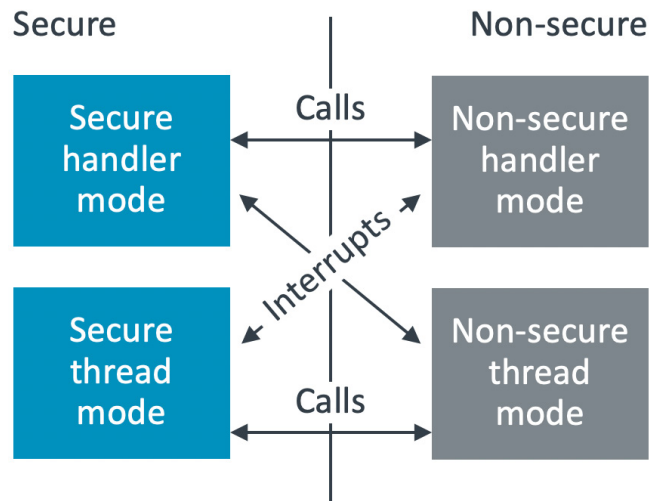


The exact partitioning arrangement of memory space is device specific.

The security state can change by:

- ✚ Direct function calls and returns of software APIs
- ✚ Exception/Interrupt entry and exit

Fig. 2:
Processor states in
TrustZone for Armv8-M

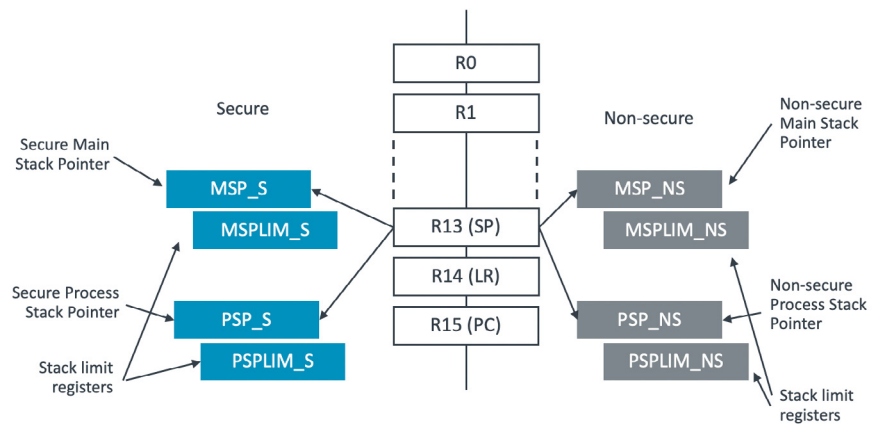


Additional protection mechanisms are defined in the architecture to ensure that software in the Non-secure state can branch into software in the Secure state only via valid entry points or via a valid return. It also ensures Secure information is not leaked into Non-secure worlds when a Non-secure interrupt takes place.

B. Banked stack pointers and stack limit registers

To minimize area and power, Secure and Non-secure state software share the same register bank, except for stack-related registers like stack pointers and stack limit registers.

Fig. 3:
Processor's integer register
bank and banked register
in TrustZone for Armv8-M



The banked stack registers allow Secure and Non-secure stacks to be separated. To improve security further, stack limit registers are added to detect and prevent stack overflow attacks. Due to the limitation of power and area budget for the [Arm Cortex-M23 processor](#) (Armv8-M baseline), only Secure stack limit registers are implemented, and Non-secure stacks need to rely on Non-secure MPU for stack overflow protection (if implemented).

C. Banked SysTick timers

The SysTick timer in Cortex-M processors provide low area hardware for periodic OS interrupt and can be used for other timekeeping purposes. In the Armv8-M architecture, the SysTick timer hardware is banked into Secure and Non-secure SysTick, allowing completely independent operations. For the Cortex-M23 processor (which is based on Armv8-M baseline), there is also additional implementation options to remove the SysTick timer or to instantiate just one SysTick timer and let Secure software to determine if the SysTick is allocated for Secure or Non-secure software.

D. Banked OS related exceptions

To aid OS operations, Cortex-M processors support:

- + SuperVisor Call (SVC),
- + PendSV (pendable SuperVisor service)
- + SysTick (system tick timer) exceptions

In the Armv8-M architecture, TrustZone security banks these exceptions into Secure and Non-secure exceptions. Each of them has their programmable priority levels and can operate individually. For example, when in Secure state, execution of an SVC instruction triggers the Secure SVC exception. Whereas in the Non-secure state, execution of an SVC instruction triggers the Non-secure SVC exception.

E. Banked MPU (Memory Protection Unit)

With TrustZone security extension, the MPU is also banked into Secure and Non-secure MPUs.

- + Secure MPU is used during operation of Secure software, including fetching of instructions in Secure memories while switching into the Secure state
- + Non-secure MPU is used during operation of Non-secure software, including fetching of instructions in Non-secure memories while switching into the Non-secure state.

The Secure and Non-secure MPU are both optional and can be independent configurations. For example, the number of MPU regions support can be different between Secure and Non-secure MPU.

The enhancements of MPU also include a new programmer's model, which defines MPU regions based on a starting and ending address with a granularity of 32 bytes. This is much more flexible than the previous MPU programmer's model, in previous versions of the architecture, which requires MPU region sizes to be to the power of two, where the starting address must be an integer multiple of the region size.

Requirements of IoT devices that affect hardware & software designs

When looking into hardware support for OS in Armv8-M, it seems that the available resources are almost symmetric across Secure and Non-secure worlds except:

- + The Non-secure world cannot access Secure resources directly
- + Details of protection mechanism (e.g. use of SG instructions to indicate valid Secure entry points)
- + Stack limit registers are only available in the Secure side of the baseline processors, for example, the Cortex-M23 processor

With the banked resources, it almost seems that both the Secure world and the Non-secure world could host an RTOS. However, in IoT applications, many requirements affect how software operations should look like, for example:

- + Security – as the deployment of IoT devices increase, security is becoming more important. For IoT, the security requirements include, but are not limited to:
 - Protection of crucial information such as personal details, passwords and certificates
 - Prevention of devices being compromised and turned into zombies for launching other attacks such as DDoS (Distributed Denial of Services)
 - Firmware asset protection
 - Preventing end users from bypassing security checks, for example, to enable features that they haven't paid for
- + Firmware updates – there is a strong demand to enable secure firmware update capability in a range of IoT devices, not just for bug fixes, but also to allow new features to be added. At the same time, the firmware update mechanisms need to be secure to ensure the program image being used is genuine and has not been tampered with. In some cases, it is also essential to ensure that users of the products cannot roll back to older firmware releases that might have known security vulnerabilities.
- + The ecosystem for software developers – for many software developers, freedom of choosing software components (e.g. RTOS, middleware) is essential. In many cases, software developers would like to use specific RTOS/middleware because of:
 - Specific software features that are unique to certain RTOS or personal preferences
 - Specific application requirements such as functional safety certification
 - Ease of migration for existing codes
 - Cost (e.g. the developers might want to use a royalty-free software solution, or already have licensed specific software components and would like to use that)
- + Customization of RTOS and middleware – in some cases, software developers also need to customize the RTOS and middleware to match specific hardware configuration.
- + Debug visibility of the whole application (including RTOS and middleware) – as there is a need to customize RTOS and middleware, software developers need to have good debug visibility of these software components.

These requirements have shaped the way that microcontrollers are designed, and the way that RTOS works under PSA. For microcontroller design aspects, a number of features can often be found in the new Armv8-M based microcontrollers:

- ✦ TrustZone security extensions
- ✦ Memory Protection Units (MPU)
- ✦ Support for secure boot
- ✦ Support for secure firmware updates
- ✦ Support for secure storage for crypto keys, certificates, etc.
- ✦ Crypto engines and accelerators
- ✦ True Random Number Generator (typically needed for session key generation in a Secure network connection)
- ✦ Support for lifecycle state management
- ✦ Built-in firmware for accessing security features
- ✦ Support for debug authentication
- ✦ Device level firmware readout protection

In many cases, microcontroller device manufacturers would like to install value-added firmware and install certain certificates or secure ID information in the Secure memories before the device is released. Some devices could have additional protection features such as anti-tampering.

With these requirements, there is a need to have a strong protection mechanism to protect a range of critical security resources, and TrustZone security extension is a natural solution.

Where should the RTOS run from?

When the Arm product management team introduced the Armv8-M architecture to RTOS vendors, there was quite a lot of discussions on how RTOS should work with the TrustZone environment. Traditionally, security features in microcontroller systems are used to protect privileged software including RTOS:

- ✦ Separation of privileged and unprivileged code
- ✦ Memory Protection Unit (MPU) to prevent unprivileged code to access memories used by privileged software

Therefore, many microcontroller software developers would consider TrustZone as a new protection mechanism that will be used to protect an OS. With such arrangement, an RTOS could

- + Use TrustZone to prevent untrusted applications to access its memories
- + Allow some of the applications to have a higher privilege level by running them in the Secure state

However, when considering security requirements described earlier, such an approach is unsuitable for secure IoT devices.

Since software developers would like to use their own choice of RTOS and be able to customize and debug the applications, it means that the Secure world of the microcontroller device has to be accessible for debugging and flash programming. As a result, microcontroller vendors cannot put manufacturer's certificate or secure ID information in the chip with such arrangement, as it will be exposed to software developers (don't forget that a hacker could also purchase the chips and reverse engineer them).

If a hacker managed to hack into an application in the Secure privileged world, the device could be fully compromised, and that will also allow the device's flash memory to be modified and Secure boot code to be disabled or bypassed. The device cannot be recovered unless completely erased and reprogrammed.

Typically, the Secure side of the device needs to be heavily protected due to the presence of high-value information and the update of Secure firmware should have very strong security protection in place. It means the update of Secure firmware should be avoided in general operation scenarios. This conflicts with the need to update RTOS and middleware regularly for introducing bug fixes and new features.

It is already possible to prevent untrusted application to access privileged memories used by OS with existing privilege levels and MPU-based method. Therefore, using TrustZone to protect OS kernels does not bring many additional advantages. From these observations, the RTOS should run from the Non-secure side:

- + Non-secure flash can be updated easily
- + Secure information is protected
- + If Non-secure applications or privileged code are compromised, hackers would not be able to modify flash memories and disable the secure boot
- + Software developers can use their own RTOS and middleware, and be able to customize these software components

This arrangement aligns with the “least privilege” concept that is commonly adopted in secure software development. By reducing the amount of software on the Secure side, it reduces the risk of hacking into the Secure side of the device (i.e. reduction of attack surface). In fact, this arrangement is not new to Arm’s software development – with Cortex-A processors, the rich OS (e.g. Linux) executes in Non-secure world, and the Secure world is used only for security critical software(s) (e.g. secure boot, payment application, digital right management). In a Cortex-M based IoT device, it is expected that the following software components should be placed in the Secure memory:

- + Secure boot
- + Secure storage APIs
- + Crypto APIs
- + Debug authentication support
- + Lifecycle state management support
- + Firmware update
- + Secure services (e.g. cloud service clients)
- + Secure partition management agent (e.g. Secure Partition Manager in Trusted Firmware-M) for Secure services.
- + RTOS helper APIs (this will be explained in the next section).

While this is not new, in some way it is a change of paradigm for some microcontroller software developers that protection of the OS is no longer the key goal of security features (and there is something new for them to take into consideration for security).

Operations of OS in a TrustZone for Armv8-M environment

While the decision to run RTOS on the Non-secure side is relatively easy to settle, the exact operations of RTOS on the Non-secure side is a bit more complicated because the context-switching operations need to be carried out on both sides.

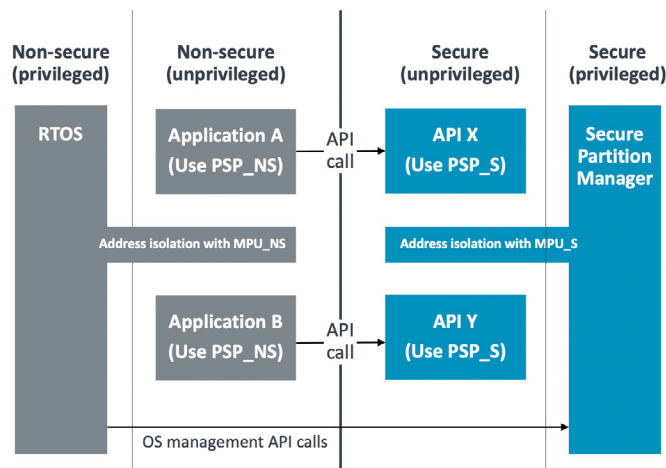
Consider the following case:

- + A processor system has RTOS on the Non-secure side, and has application A and B
- + Application A in Non-secure side calls a Secure API X
- + During execution of Secure API X, a Non-secure system tick exception takes place and context switch the Non-secure side to Application B
- + During execution of Application B, the application code calls a Secure API Y

For each of these software components, they have their stack allocation:

- ✦ Application A has Non-secure stack allocation, using PSP_NS (Non-secure Process Stack Pointer) for stack operations
- ✦ Instance of API X called by Application A has Secure stack allocation, using PSP_S (Secure Process Stack Pointer) for stack operations
- ✦ Application B has Non-secure stack allocation, using PSP_NS (Non-secure Process Stack Pointer) for stack operations
- ✦ Instance of API Y called by Application B has Secure stack allocation, using PSP_S (Secure Process Stack Pointer) for stack operations

Fig. 4:
In TrustZone for Armv8-M,
each thread calling Secure
API(s) need to have Secure
stack allocation



As a result of this arrangement, a context switching on the RTOS (running in the Non-secure side) must also trigger a context switch of the PSP on the Secure side. Optionally, the MPU on both sides also needs to be reconfigured if it is used to isolate the memory spaces allocated for each application or the library code.

Some additional OS helper APIs are needed. For example, when an application is created and if it is going to call a Secure API, the firmware on the Secure side needs to be aware of the new context so that it can allocate stack space for it. Similarly, an API is could be introduced to allow stack allocation to be deleted if an application is to be removed from the Non-secure OS queue. Alternatively, some forms of stack memory management can be used to remove Secure stack allocation for a context when it is inactive.

As discussed earlier, there is a need to support different types of RTOSes, and therefore the OS helper APIs need to be an open standard. These APIs were originally defined in the CMSIS-5 (Cortex Microcontroller Software Interface Standard), under RTOS Management in TrustZone for Armv8-M support code. The APIs are listed below in Table 1 and their documentation can be found [here](#).

Functions	Descriptions based on original CMSIS-5 implementation. TF-M's implementation is a bit different but retained the APIs.
uint32_t TZ_InitContextSystem_S(void)	<p>Initializes the memory allocation management for the Secure memory regions. As a minimum, the Secure thread mode stack will be provided.</p> <p>Call by Non-secure RTOS kernel in RTOS initialization. Returns execution status (1: success, 0: error)</p>
TZ_MemoryId_t TZ_AllocModuleContext_S (TZ_ModuleId_t module)	<p>Allocates the Secure memory regions for thread execution. The parameter module describes the set of Secure functions that are called by the Non-secure thread. Set module to zero if no Secure calls are used/allowed. This leads to no Secure memory to be assigned which results in zero being returned as memory id as well. An RTOS kernel should call this function at the start of a thread.</p> <p>Returns value != 0 if TrustZone memory slot identify. If value==0, no memory available or internal error.</p> <p>Note: Unlike original CMSIS implementation, module ID parameter is unused in TF-M.</p>
uint32_t TZ_FreeModuleContext_S (TZ_MemoryId_t id)	<p>De-allocates the Secure memory regions. The parameter id refers to a TrustZone memory slot that has been obtained with TZ_AllocModuleContext_S. An RTOS kernel should call this function at the termination of a thread.</p> <p>Returns execution status (1: success, 0: error)</p>
uint32_t TZ_LoadContext_S (TZ_MemoryId_t id)	<p>Prepare the Secure context for execution so that a thread in the Non-secure state can call Secure library modules. The parameter id refers to a TrustZone memory slot that has been obtained with TZ_AllocModuleContext_S which might be zero if not used. An RTOS kernel should call this function at thread context switch before running a thread.</p> <p>Returns execution status (1: success, 0: error)</p>
uint32_t TZ_StoreContext_S (TZ_MemoryId_t id)	<p>Free the Secure context that has been previously loaded with TZ_LoadContext_S. The parameter id refers to a TrustZone memory slot that has been obtained with TZ_AllocModuleContext_S which might be zero if not used. An RTOS kernel should call this function at thread context switch after running a thread.</p> <p>Returns execution status (1: success, 0: error)</p>

To modify an existing RTOS for Cortex-M processor to be used with TrustZone with these APIs, an RTOS software developer needs to insert these API calls in:

- + RTOS initialization
- + Context creation and deletion
- + Context switching (in both swapping in and out tasks)

Examples of the function call needed are shown in figure 5:

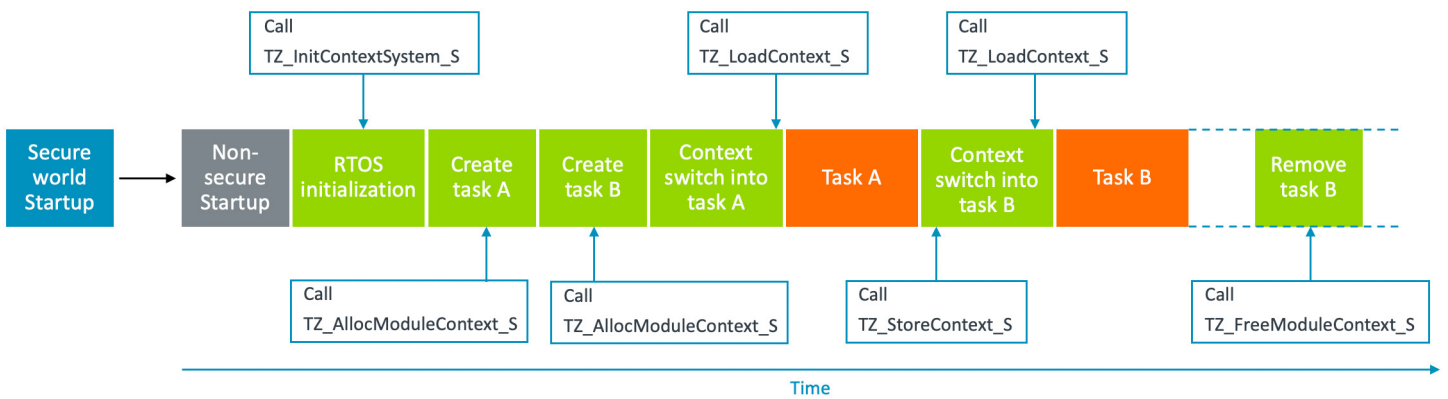


Fig. 5:
OS support APIs being
used in an RTOS running
in the Non-secure state

In addition to the API calls, the task control block (TCB) for the Non-secure RTOS needs to add a new element to store the return value from `TZ_AllocModuleContext_S()`. This data type (struct `TZ_MemoryId_t`) is used as a set of parameters for the subsequent API calls.

To link the Non-secure project which contains these APIs, the Non-secure software developers should obtain an export library (containing only symbols of Secure APIs but not the content) from chip vendors or Secure software developers and add this to their projects. This should allow the linker to generate correct function calls.

While the OS helper's APIs created in the CMSIS-5 project was reused, the design of the API implementations in TF-M were changed: In the original CMSIS-5 implementation, the helper API calls handle the Secure stack allocation. For example, to allow the Secure firmware to know how much stack space is to be allocated for the context of the Secure API calls, the "module" parameter of `TZ_AllocModuleContext_S()` is used in CMSIS-5. In contrast, for TF-M the same API ignores the "module" parameter and handles the stack allocation internally (as a part of the Secure Partition Manager, SPM).

Similarly, TF-M implementation of `TZ_LoadContext_S()` and `TZ_StoreContext_S()` does not directly switch the Secure Process Stack Pointer (`PSP_S`). Instead, these APIs make the SPM aware of the changes in Non-secure context and the SPM handles the Secure context changes. By handling memory allocation dynamically in SPM, Secure SRAM footprint can be reduced as application threads that are not calling Secure APIs does not consume Secure stack space.

While there are lots of differences between the CMSIS-5 OS helper APIs and the TF-M version, from Non-secure software development point of view, the API calls are the same.

Please note that the OS helper APIs `TZ_LoadContext_S` and `TZ_StoreContext_S` does not handle the Non-secure side context switching operations. The context switching code for the Non-secure side (register context saving and restoring, and Non-secure MPU configuration switching) is largely unaffected by these APIs, and therefore the required changes in the Non-secure RTOS when porting to Armv8-M can be minimized. However, some changes are needed – in addition to TCB change mentioned earlier (adding of a new element), the Non-secure RTOS code is likely to need additional updated due to:

- + New values for `EXC_RETURN` code
- + Optionally - adding of stack limit checking support the new programmer's model for the MPU

Other RTOS functionalities, such as task scheduling algorithms and semaphore handling are not affected.

At this point, it is useful to mention that the reference Secure firmware is available from the Trusted Firmware-M (TF-M). This is open-source code and can be [downloaded here](#).

There is still a range of on-going development on the TF-M, including additional Secure services APIs. Further information about the Trusted Firmware project, including ongoing development and plans can be found in the Trusted Firmware web pages (see the 'Getting started' section).

Getting started

Here is a list of resources to help you get started:

- + Microcontroller devices based on Armv8-M are being released by Arm's partners today. At the time of writing this paper, many microcontroller vendors have already announced their products including. Visit our [TrustZone for Armv8-M Community to find out more.](#)
- + More hardware available today is the [Musca boards](#), Arm-developed test chips that include a dual-core Cortex-M33 system.

Fig. 6:
Arm Musca-A
development board



- + Examples of the TF-M project based on the Musca-A board can be found in the Keil software pack for [Musca-A here](#). (Musca-B1 software coming soon)
- + More information about the software pack and the examples can be [found here](#)
- + Recently, Arm announced a new FPGA platform to allow software developers to test drive a dual-core Cortex-M33 based system on FPGA. This platform, named DesignStart FPGA on Cloud, is hosted in the [AWS cloud service here](#).
- + [Trusted Firmware information](#) (note: Trusted Firmware is an open-source, open governance [Linaro](#) Community Division Project)
- + [Development Status dashboard](#)
- + [Development roadmap](#)

Using these hardware/platforms/resources, RTOS vendors can start working on RTOS porting for Armv8-M processor-based products. If you have a question on RTOS porting for Trusted Firmware-M, please feel free to [submit questions here](#).

If you need other assistance, you are welcome to submit your questions on <https://community.arm.com>, or contact the Arm partner management team directly [here](#).

Conclusion

In a world heading for one trillion connected devices, security is no longer optional. Arm TrustZone has now been adopted in new chip families from the largest MCU vendors, so RTOS developers must understand how to update the RTOS in order to take advantage of the security benefits. While TrustZone for Armv8-M is designed to be very flexible, with an RTOS able to run from either the Secure or Non-secure world, such flexibility can also lead to some confusions in the case of RTOS operations.

This paper covered the security features of Armv8-M processors that are related to RTOS operations, and then demonstrated that due to the security and application requirements, a RTOS is best run from the Non-secure side with the help of Arm Trusted Firmware-M (TF-M) on the Secure side for typical microcontroller systems.

With better understanding of the interaction of Non-secure RTOS and Secure firmware, how the APIs are used and the range of resources available, RTOS developers can ensure a successful, seamless migration to TrustZone-enabled microcontrollers. At the same time, application developers can still enjoy the freedom of using their own choices of RTOS, and update the RTOS with secure firmware update easily.

In addition, Arm helps software developers and ecosystem partners with security - Arm is working on a reference implementation of software platform (Trusted Firmware-M or TF-M) as a part of the Arm Platform Security Architecture (PSA), and examples are available to show how a RTOS is integrated under TF-M.

With a common foundation of security principles through PSA and the system-wide microcontroller security offered by TrustZone for Armv8-M, developers can now secure the next generation of trusted Embedded and IoT devices easier than ever before.

Acknowledgment

I want to thank Trusted Firmware team, and Arm Embedded marketing team for reviewing this paper.



All brand names or product names are the property of their respective holders. Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given in good faith. All warranties implied or expressed, including but not limited to implied warranties of satisfactory quality or fitness for purpose are excluded. This document is intended only to provide information to the reader about the product. To the extent permitted by local laws Arm shall not be liable for any loss or damage arising from the use of any information in this document or any error or omission in such information.

© Arm Ltd. 2019