

The latest version of this document is here: www.keil.com/appnotes/docs/apnt_291.asp

Abstract

This application note explains the features that are available in CMSIS and MDK to utilize the secure and non-secure domains in the Armv8-M architecture. It contains several programming examples, including an RTOS application that shows the interaction of non-secure thread execution with libraries that are provided by the secure domain of an Armv8-M system.

Prerequisites

MDK v5.22 provides support for creating and debugging secure and non-secure applications for Armv8-M based devices, especially for Arm Cortex-M23 and Arm Cortex-M33. To be able to use the examples provided in this application note, you need to have a valid license for [MDK-Professional](#) and have the following software packs installed: ARM.CMSIS.5.0.11.pack (or higher). Note that there is an evaluation version available for [MDK-Professional](#).

Contents

Abstract	1
Prerequisites.....	1
Introduction	3
Documentation.....	3
Example projects.....	3
Armv8-M programmer’s model.....	4
CMSIS-Core extensions	5
Secure and non-secure domains.....	6
Writing secure software.....	7
Return from the secure to the non-secure state	7
Obtain trusted data from non-secure code	7
TT instruction.....	7
Address range check intrinsic	8
Asynchronous modifications to currently processed data.....	9
CMSIS-RTOS v2 for Armv8-M.....	9
RTOS thread context management	10
Armv8-M debug.....	11
Simulation model.....	11
Secure debug access.....	11
Non-secure debug access	11
Armv8-M debug components	12
Create Armv8-M software projects	13
System and Memory configuration.....	13

Setup secure and non-secure projects	13
Step 1: Secure project setup	15
Step 2: Non-secure project setup.....	15
Step 3: Multi-project workspace	16
Debugger configuration.....	16
Example: TrustZone for Armv8-M No RTOS	17
Multi-project workspace setup.....	17
Program Code	17
Call sequence	19
Project Build	19
Debug non-secure to secure state switches	20
Example: TrustZone for Armv8-M RTOS	21
Call sequence	21
Example: TrustZone for Armv8-M RTOS Security Tests	22
Appendix	23
MDK – Microcontroller Development Kit	23
ULINK – Debug/trace adapter series.....	23
CMSIS – Cortex Microcontroller Software Interface Standard.....	23
Books	24
Application notes	24
Useful Arm websites.....	24

Introduction

Embedded system programmers face demanding product requirements that include cost sensitive hardware, deterministic real time behavior, low-power operation, and secure asset protection. As time-to-market is critical, Arm provides a set of development tools and software components to accelerate the overall system design.

Modern applications have a strong need for security. Assets that may require protection are:

- device communication (using cryptography and authentication methods)
- secret data (such as keys and personal information)
- firmware (against IP theft and reverse engineering)
- operation (to maintain service and revenue)

Arm® TrustZone® technology is a System on Chip (SoC) and CPU system-wide approach to security. The TrustZone for Armv8-M security extension is optimized for ultra-low power embedded applications. It enables multiple software security domains that restrict access to secure memory and I/O to trusted software only.

TrustZone for Armv8-M:

- preserves low interrupt latencies for both secure and non-secure domains.
- does not impose code overhead, cycle overhead or the complexity of a virtualization based solution.
- introduces efficient instructions for calls to the secure domain with minimal overhead.

Documentation

This application note focusses on how to use TrustZone for Armv8-M in Keil MDK. If you want to learn more about the technology behind it, there are several documents that go into further detail:

- [Armv8-M Security Extensions: Requirements on Development Tools](#) explains concepts implemented in compiler toolchains to support the Armv8-M architecture.
- [Secure software guidelines for Armv8-M based platforms](#) lists the requirements when creating secure software for an Armv8-M based platform.
- The [Arm C Language Extensions \(ACLE\) for Armv8-M](#) enables the Armv8-M Security Extension to build a secure image, and to enable a non-secure image to call a secure image. This document includes details of a possible compiler implementation.
- The [Armv8-M Architecture Reference Manual](#) gives a complete overview of the Armv8-M architecture. The following sections put a spotlight on some of the aspects that are of special interest to the software developer.

Example projects

The ARM:CMSIS software pack contains the following example projects that show TrustZone programming. Use the **Pack Installer** to locate and copy the TrustZone project examples. Refer to the page for further details.

Example	Description	Page
TrustZone for Armv8-M No RTOS	bare-metal secure/non-secure example without RTOS	17
TrustZone for Armv8-M RTOS	secure/non-secure RTOS example with thread context management	21
TrustZone for Armv8-M RTOS Security Tests	secure/non-secure example that utilizes security faults to restart a system	22

Armv8-M programmer's model

Figure 1 shows the memory view for the secure state. In the secure state, all memory and peripherals can be accessed. The **system control and debug** area provides access to secure peripherals and non-secure peripherals that are mirrored at a memory alias.

Code that is executed from a secure region (secure code) is executed in secure state and can access memory in both secure and non-secure regions.

The **secure peripherals** are only accessible during program execution in secure state. The **Security Attribution Unit (SAU)** configures the non-secure memory, peripheral, and interrupt access. A secure MPU (memory protection unit), secure SCB (system control block), and secure SysTick timer are available as well.

The system supports two separate interrupt vector tables for secure and non-secure code execution. This interrupt assignment is controlled during secure state code execution via the NVIC (nested vector interrupt controller).

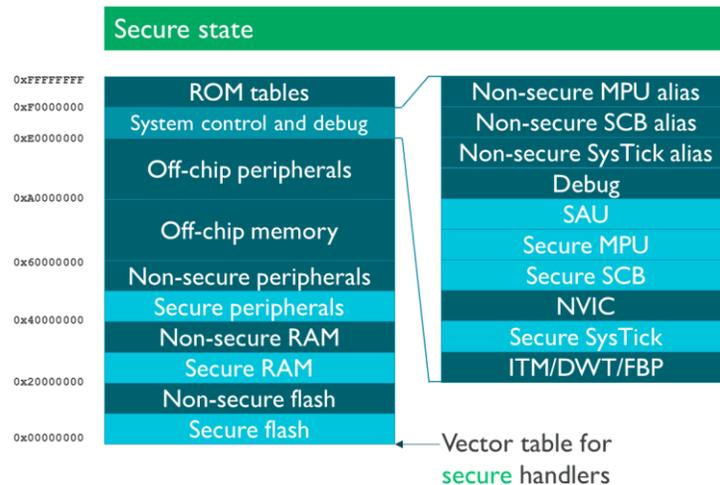


Figure 1 Secure memory map

Note: CMSIS-Core defines an additional file (`partition_<device>.h`) that is used to setup the SAU. Please refer to CMSIS-Core extensions on page 5.

Figure 2 shows the memory view for the non-secure state. This memory view is similar to the classic Cortex-M memory map. Access to any secure memory or peripheral space triggers a security exception that executes a handler in secure state.

Code that is executed from a non-secure region (non-secure code) is executed in non-secure state and can only access memory in non-secure regions.

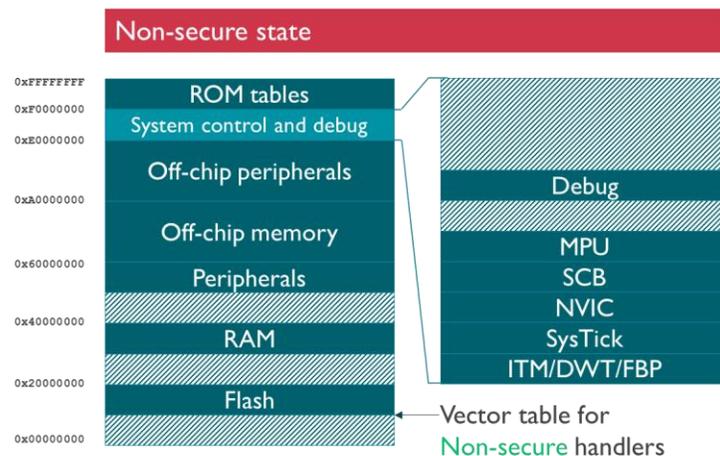


Figure 2 Non-secure memory map

Attempts to access secure regions from non-secure code or a mismatch between the (secure or non-secure) code that is executed and the security state of the system results in a fault exception.

Figure 3 shows the register view of an Armv8-M system with TrustZone. As the general purpose registers can be accessed from any state, function calls between the states use these registers for parameter and return values.

The register R13 is the stack pointer alias, and the actual stack pointer (PSP_NS, MSP_NS, PSP_S, MSP_S) accessed depends on the state (secure/non-secure) and the mode (handler=exception/interrupt execution or thread=normal code execution).

Each stack pointer has an optional limit register (PSPLIM_NS, MSPLIM_NS, PSPLIM_S, MSPLIM_S) used to trap stack overflows triggering a UsageFault exception.

An Armv8-M system with TrustZone has an independent CONTROL register for each state (secure or non-secure). The interrupt/exception control registers (PRIMASK, FAULTMASK, BASEPRI) are banked between the states, however the interrupt priority for the non-secure state can be lowered so that secure interrupts have always a higher priority.

The core registers of the current state are accessed using the standard core register access functions. In secure state all non-secure registers are accessible.

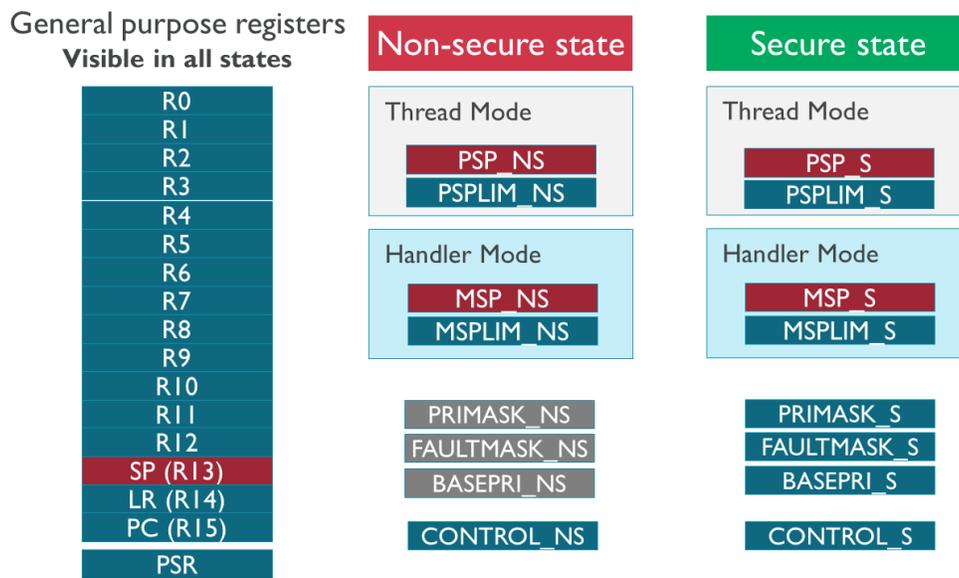


Figure 3 Registers

CMSIS-Core extensions

CMSIS-Core implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals. The CMSIS-Core files are extended by the system partition header file `partition_<device>.h` which defines the initial setup of the non-secure memory map during system start in the secure state.

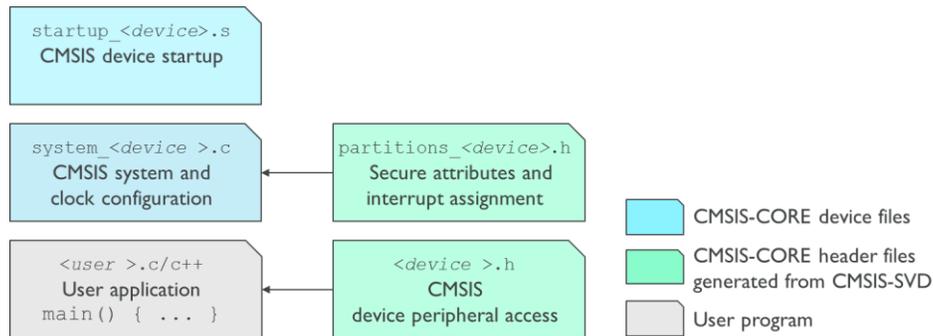


Figure 4 CMSIS-Core files

This file contains the initial setup of the TrustZone hardware in an Armv8-M system. SystemInit calls the function TZ_SAU_Setup, which uses the settings in this file to initialize the Security Attribution Unit (SAU) and to define non-secure interrupts (register NVIC_INIT_ITNS). It performs the following initialization tasks:

- Provide settings for the SAU CTRL register.
- Configure the SAU Address Regions.
- Configure device-specific deep sleep and exception settings.
- Configure device-specific interrupt target settings.

Secure and non-secure domains

Figure 5 shows an embedded application that is split into a **User project** (executed in non-secure state) and a **Firmware project** (executed in secure state).

System Start: after power-on or reset, an Armv8-M system starts code execution in the secure state. The access rights in the SAU for the non-secure state are configured.

User Application: control can be transferred to the non-secure state to execute user code. This code can only call functions in the secure state, which are marked for execution with the SG (secure gate) instruction and additional memory attributes. Any other attempt to access memory or peripherals that are assigned to the secure state triggers a security exception.

Firmware callbacks: code running in the secure state can execute code in the non-secure state using call-back function pointers. For example, a communication stack (protected firmware) could use an I/O driver that is configured in user space.

Program execution in the secure state is further protected by TrustZone hardware from software failures. For example, an Armv8-M system may implement two independent SYSTICK timers which allow stopping code execution in non-secure state in case of timing violations. Also, function pointer callbacks from secure state to non-secure state are protected by a special CPU instruction and the address LSB 0 which prevents executing code in non-secure state accidentally.

For a real use-case refer to “Example: TrustZone for Armv8-M No RTOS” on page 17.

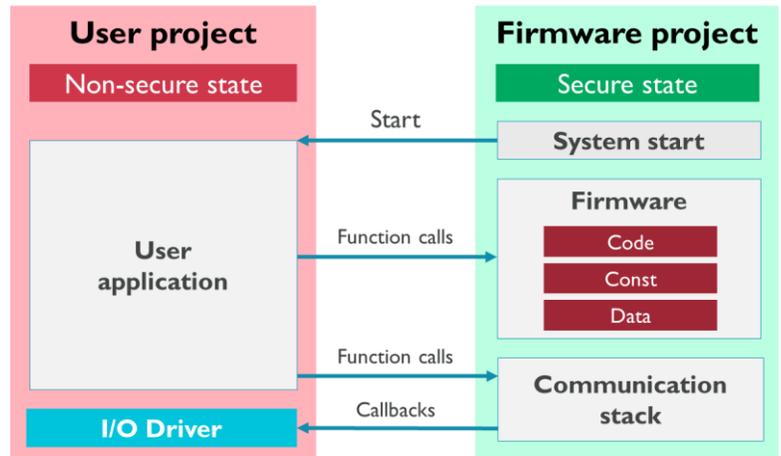


Figure 5 Secure/non-secure embedded application

Writing secure software

Even though the Armv8-M architecture offers specific features and instructions designed to improve the security of the device, if the software running contains bugs or design flaws, the whole system security is put at risk. As always, the entire system security depends on the security of the weakest link of the chain. Refer to the document [Secure software guidelines for Armv8-M based platforms](#) for more information.

There are three main potential attacks to an Armv8-M system:

- Leaking secret information in registers when switching from a secure to a non-secure state
- Not checking pointers passed from the non-secure state could potentially give access to secure memory
- Changing non-secure memory while in secure state

Return from the secure to the non-secure state

The compiler uses:

- Registers R0 to R3 to pass parameters and return values.
- Registers R4 to R12 during function execution. The called function restores these registers.

As registers R0 to R3 are not normally cleared because they are used for parameters and return values, secure information might leak to the callee function when returning to non-secure state.

Example:

Secure state	Non-secure state
<pre>void decrypt(int32_t *data) { key = SECRET; // do the work }</pre>	<pre>void spy_function() { decrypt(NULL); print_content_of_registers(); }</pre>

To avoid such an exploit, you need to add the `cmse_nonsecure_entry` attribute to the `decrypt` function:

```
void decrypt(int32_t *) __attribute__((cmse_nonsecure_entry));
```

When this attribute is applied, Arm Compiler 6 clears registers R0 to R3 and the status flag when used. This ensures that information cannot leak via the CPU registers to the non-secure state.

Obtain trusted data from non-secure code

When secure code has to access non-secure memory using an address calculated by the non-secure state, it cannot trust that the address lies in a non-secure memory region.

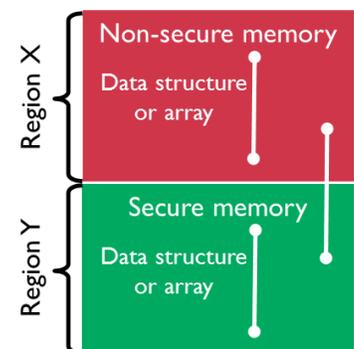
TT instruction

To allow software to determine the security attribute of a memory location, the Test Target (TT) instruction is used. TT is used to check the access permissions, different security states and privilege levels that a pointer might have for a specific target address.

When executed in the secure state, the result of this instruction is extended to return the Security Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU) configurations at the specific address.

For each memory region defined by the SAU and IDAU, there is an associated region number that is generated by the SAU or by the IDAU. This region number is used by software to determine if a contiguous range of memory shares common security attributes.

The TT instruction returns the security attributes and region number, and the MPU region number, from an address value. By using a TT instruction on the start and end addresses of the memory range, and identifying that both reside in the same region number, software can quickly determine that the memory range, for example, for data array or data structure, is located entirely in non-secure space.



The `TT` instruction is useful for determining the security state of the MPU at that address. Although the instruction cannot be accessed in C/C++ code there are several intrinsic functions which make this functionality available to the developer.

Intrinsic	Semantics
<code>cmse_address_info_t cmse_TT(void *p)</code>	Generates a <code>TT</code> instruction.
<code>cmse_address_info_t cmse_TT_fptr(p)</code>	Generates a <code>TT</code> instruction. The argument <code>p</code> can be any function pointer type.
<code>cmse_address_info_t cmse_TTT(void *p)</code>	Generates a <code>TT</code> instruction with the <code>T</code> flag.
<code>cmse_address_info_t cmse_TTT_fptr(p)</code>	Generates a <code>TT</code> instruction with the <code>T</code> flag. The argument <code>p</code> can be any function pointer type.

The result of the `TT` instruction is described by a C type containing bit-fields. This type is used as the return type of the `TT` intrinsics.

Address range check intrinsic

Checking the result of the `TT` instruction on an address range is essential for programming in C. It is used to check permissions on objects larger than a byte. The address range check intrinsic can be used to perform permission checks on C objects.

Intrinsic	Semantics
<code>*cmse_check_address_range (void *p, size_t size, int flags)</code>	Address range check intrinsic. It checks the address range from <code>p</code> to <code>p + size - 1</code> .
<code>*cmse_check_pointed_object (void *p, int flags)</code>	Returns the same value as <code>cmse_check_address_range(p, sizeof(*p), f)</code>

The `cmse_check_address_range` intrinsic operates under the assumption that the configuration of the SAU, IDAU, and MPU is constrained as follows:

- An object is allocated in a single region.
- A stack is allocated in a single region.

These points imply that a region does not overlap other regions. The `TT` instruction returns 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 contained in one SAU, IDAU, and MPU region. In this case two `TT` instructions are executed to check the address range.

Example:

```
void GetIncidentLog_s (struct IncidentLog_t *IncidentLog_p) __attribute__((cmse_nonsecure_entry));
void GetIncidentLog_s (IncidentLog_t *IncidentLog_p) {
    memcpy (IncidentLog_p_ok, &IncidentLog, sizeof (IncidentLog_t));
}
```

You can verify that the target address is within the non-secure memory region using the `cmse_check_address_range` intrinsic:

```
void GetIncidentLog_s (IncidentLog_t *IncidentLog_p) {
    struct IncidentLog_t *IncidentLog_p_ok;

    IncidentLog_p_ok = cmse_check_address_range (IncidentLog_p, sizeof(IncidentLog_t), CMSE_NONSECURE);
    if (IncidentLog_p_ok != NULL) {
        /* requested copy range is completely in non-secure memory */
        memcpy (IncidentLog_p_ok, &IncidentLog, sizeof (IncidentLog_t));
    }
    else
    {
        //do something else
    }
}
```

Asynchronous modifications to currently processed data

Secure code should **never** trust non-secure data, as it may be modified by interrupt handlers. High priority interrupts in non-secure state can interrupt the secure code execution and change non-secure data.

Example:

```
void setup_entry (struct config *c, int value) {
    if (c->index > 0 && c->index < sizeof (array)) {
        array[c->index] = value;
    }
}
```

To overcome this situation, copy non-secure data before validation and use the `volatile` attribute to disable potential compiler access optimizations:

```
void setup_entry (volatile struct config *c, int value) {
    int index_s = c->index;
    if (index_s > 0 && index_s < sizeof (array)) {
        array[index_s] = value;
    }
}
```

CMSIS-RTOS v2 for Armv8-M

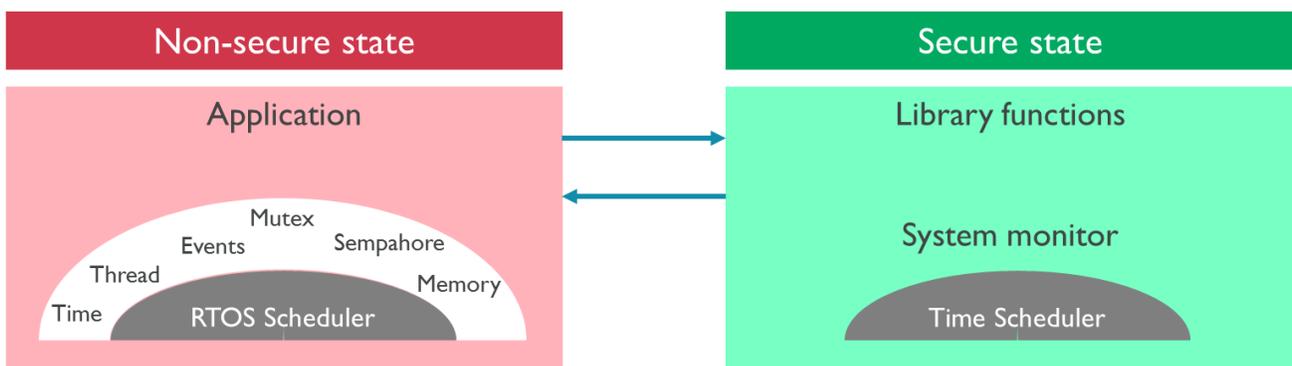
A Real-Time Operating System (RTOS) is required frequently for applications that perform multiple tasks simultaneously. These tasks are executed by threads that operate in a quasi-parallel fashion.

It is certainly possible to create real-time applications without an RTOS (by executing one or more tasks in a loop) but usually there are numerous scheduling, maintenance, and timing issues that can be easily solved by using an RTOS. For example, an RTOS enables flexible scheduling of system resources like CPU and memory, and offers methods to communicate between threads.

CMSIS-RTOS v2 is a full-featured RTOS for non-secure applications and supports function calls to the secure state and callback events from the secure state. It manages microcontroller resources and implements the concept of parallel threads that run concurrently.

To reduce the potential surface for hackers and to avoid unnecessary standardization, CMSIS-RTOS is running in the non-secure state and its functionality is also only available to non-secure software. This is acceptable as most security related software components do not require RTOS functionality.

To utilize the TrustZone for Armv8-M, it is necessary to split the RTOS into portions that execute in the non-secure and in the secure state of the processor:

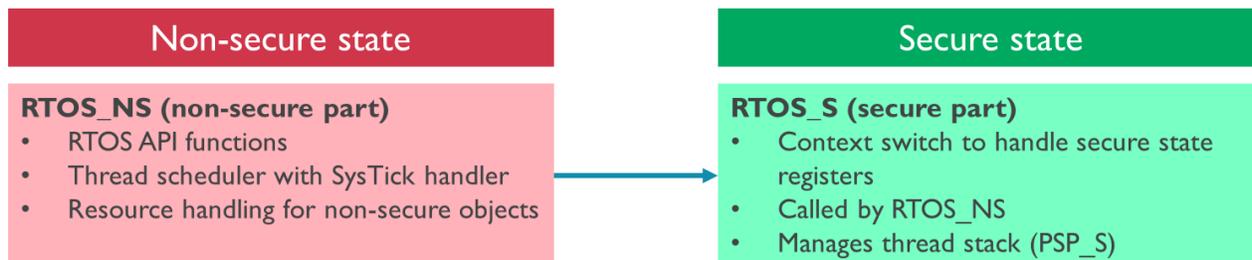


The secure state provides data and firmware protection and a system monitor for operation protection. This system monitor includes an additional time scheduler using the secure SysTick timer and runs on the secure Main Stack Pointer (MSP_S) at the highest interrupt priority.

RTOS thread context management

To provide a consistent RTOS thread context management for TrustZone for Armv8-M across the various real-time operating systems (RTOS), CMSIS-CORE includes the header file `tz_context.h` with API definitions. A non-secure RTOS application calling secure library modules requires the management of the secure stack space. Since secure registers cannot be accessed by an RTOS running in a non-secure state, secure functions implement the thread context switch.

As the non-secure and secure parts of an application are separated, the API for managing the secure stack space should be standardized. Otherwise the secure library modules would force the non-secure application to use a matching RTOS implementation.



In non-secure state, the task scheduler (RTOS_NS) is executed and all threads are started. Thread execution starts in non-secure state. The complete OS function API is available to code running in the non-secure state.

To allocate the context memory for threads, a non-secure RTOS kernel calls the interface functions defined by the header file `tz_context.h`. The interface functions themselves are part of the secure application:

- **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.

A minimum implementation should handle the secure stack for the thread execution. However, it is also possible to implement the context memory management system with additional features such as access control to secure state memory regions using an MPU. CMSIS contains a reference implementation (user code template `tz_context.c`) as part of CMSIS-RTOS v2.

Armv8-M debug

The MDK debugger offers connection to:

- simulation model of a virtual system
- target hardware using debug adapters

For application verification and optimization the MDK debugger offers two access modes to Armv8-M:

- Secure: with access to the complete system including secure and non-secure programmers views.
- Non-secure: with access to the non-secure programmers view only. In this mode, there is no way to analyze code that is executed in the secure state of the device.

Simulation model

Fixed Virtual Platforms (FVPs) enable development of software without requiring access to physical hardware and allow software verification prior to silicon availability. The functional behavior of a FVP is equivalent to real hardware but sacrifices absolute timing accuracy to achieve fast simulation speed.

MDK-Professional includes an FVP for Arm Cortex-M23 and Cortex-M33 that simulates a complete system including peripherals.

Secure debug access

Secure access offers full visibility to all instruction execution, memory regions, and device peripherals. It allows to debug and trace the secure and the non-secure software running on the target. Debugging of secure firmware is only available in this mode.

Secure access can be controlled via debug authentication methods. These methods are defined with the element `<sequence name="DebugDeviceUnlock">` in the PDSC file of the related Device Family Pack. For more information refer to the CMSIS documentation under “CMSIS-Pack – Pack Description (*.PDSC) Format – Debug Access Sequences”.

Non-secure debug access

The non-secure debug view protects the secure memory and peripherals. These are invisible to the debugger in non-secure mode. Debug and trace capabilities are limited to non-secure system resources. The following limitations can occur:

- Non-secure debuggers cannot read the register AIRCR.SYSRESETREQS. This means that a SYSRESETREQ via the reset button cannot be blocked up front. A fail of SYSRESETREQ can only be detected after trying to do so.
- Depending on the security measures of the target system, further restrictions in terms of secure/non-secure memory visibility may apply.
- Single-steps from non-secure state to secure state can lead to a longer "run" phase if a lot of secure code is executed before returning to non-secure state.
- Pushing the stop button while executing secure code may not immediately take effect. The CPU continues to run until entering non-secure state.
- Memory access breakpoints are never hit, if defined for secure memory.
- Setting SW breakpoints may fail due to memory access restrictions
- Setting HW breakpoints has no effect if defined for secure code
- Resets with reset vector catch can lead to a longer "run" phase if a lot of secure code is executed before entering non-secure state for the first time
- Depending on the target system, limited debug accesses to memory can show as errors ("Cannot access memory") or as RAZ/WI
- The secure MPU and the SAU setup cannot be read by a non-secure debugger
- CPU peripherals have limited visibility to sensitive information
- Secure variants of banked CPU registers are not readable
- Trace of secure resources (secure instructions/secure memory) is skipped

- Depending on DWT settings, the DWT cycle counter may be halted throughout secure code execution. This can falsify trace timestamps and the states value in the μ Vision register window, as well as the stop watches.

Armv8-M debug components

The new Arm Cortex-M23 and Cortex-M33 processor cores use the Arm CoreSight™ technology which introduces powerful debug and trace capabilities. The following components are available (depending on the implementation):

Component	Use
Instrumentation Trace Macrocell (ITM)	Provides information for annotated trace output; also used for simple printf-style debugging
Data Watchpoint and Trace unit (DWT)	Provides PC (Program Counter) sampling and event counters that show CPU cycle statistics, exception and interrupt execution with timing statistics, and trace data (data reads and writes used for timing analysis)
Embedded Trace Macrocell (ETM)	Provides high bandwidth instruction trace via the TPIU
Trace Port Interface Unit (TPIU)	Provides an output path for trace data from the DWT, ITM, and ETM
Flash Patch and Breakpoint unit (FPB)	Supports setting breakpoints on instruction fetches

These components are optionally available in Arm Cortex-M23 implementations. Please refer to your device's reference manual.

Create Armv8-M software projects

The steps to create a new Armv8-M software project in MDK are:

- Define the overall system and memory configuration. This has impact on:
 - Setup secure and non-secure projects, optional with multi-project workspace
 - Add startup code and ‘main’ module to secure and non-secure projects.
 - Reflect memory configuration in the CMSIS-Core file `partition_<device>.h`
- Define the API of the secure software part in a header file to allow usage from the non-secure part
- Create the application software for the secure and the non-secure part

System and Memory configuration

The definition of the memory layout is typically the first step of the system design. While the memory ranges may be reconfigured during the development cycle you should try to get a good initial setup. In the initial project phase you should also define the usage of peripherals along with the related interrupts in the secure and non-secure domain.

Memory	Description
0x00000000 .. 0x001F0000	ROM for secure program part
0x00200000 .. 0x003F0000	ROM for non-secure program part
0x20000000 .. 0x201F0000	RAM for secure program part
0x20200000 .. 0x203F0000	RAM for non-secure program part
0x40000000 .. 0x40040000	Peripherals accessible by non-secure program part

Startup Code

To add the startup code to your project select the software component **:Device:Startup** available in the dialog Manage Run-Time Environment. The startup code provides the file `partition_<device>.h` which configures the system and memory setup in the secure project.

‘Main’ module

The user code template ‘main’ module for Armv8-M can be used to add the main function for the secure project. The define `TZ_START_NS` specifies the address of the vector table in the non-secure part.

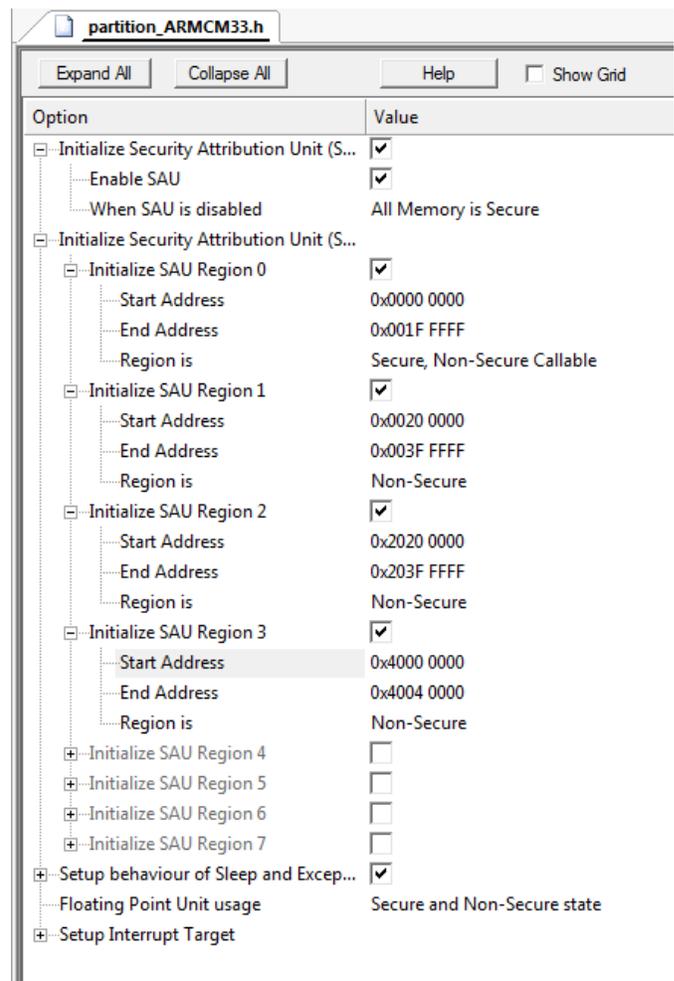


Figure 6 Configuration settings in `partition_<device>.h`

Setup secure and non-secure projects

The following explains the project setup of the secure and non-secure parts. Both projects must use the same processor configuration. First, create the secure project as it contains the interface for the non-secure project. It is recommended to create both projects in the same base directory (i.e. `Test`) and use sub-directories for the secure and non-secure part.

Step 1: Secure project setup

Create new project: `Test\Test_s\Test_s.uvprojx`

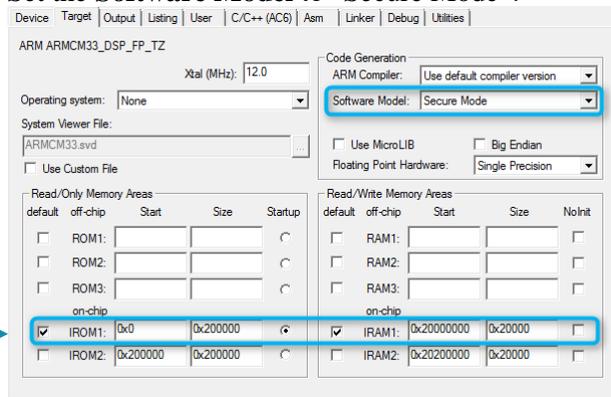
Add the software components **CMSIS:CORE** and **Device:Startup**.

The `partition_<device>.h` file is added which allows memory layout configuration.

Initialize SAU Region 0	<input checked="" type="checkbox"/>
Start Address	0x0000 0000
End Address	0x001F FFFF
Region is	Secure, Non-Secure Callable
Initialize SAU Region 1	<input checked="" type="checkbox"/>
Start Address	0x0020 0000
End Address	0x003F FFFF
Region is	Non-Secure

Memory settings under **Options for target** should reflect these memory settings.

Set the **Software Model** to “Secure Mode”:



Add the source code for the secure project. This should include an interface header file with the API of the non-secure function entries.

These entries are available for the non-secure project via an import library with the name

`.\Objects\Test_s_CMSE_Lib.o`

The object file name is always created according to this rule: `<projectname>_CMSE_Lib.o` in the `Objects` directory.

Step 2: Non-secure project setup

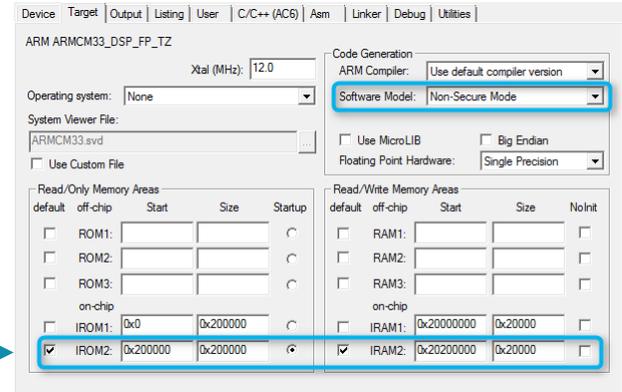
Create new project: `Test\Test_ns\Test_ns.uvprojx`

Add the software components **CMSIS:CORE** and **Device:Startup**.

The `partition_<device>.h` file is included for reference only and ignored in the non-secure project.

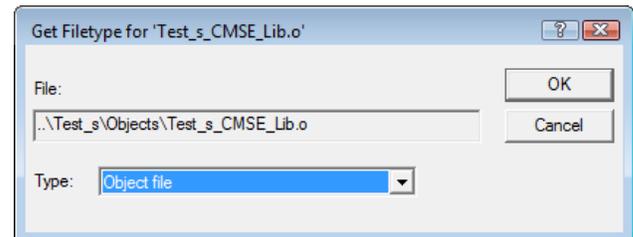
Memory settings under **Options for target** should reflect the memory settings from the secure project.

Set the **Software Model** to “Non-Secure Mode”:



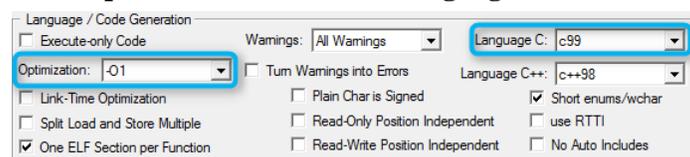
Add the source code for the non-secure project. The source code should use the same interface header file as the secure project.

Add also the import library (as an “Object file”) from the secure project:



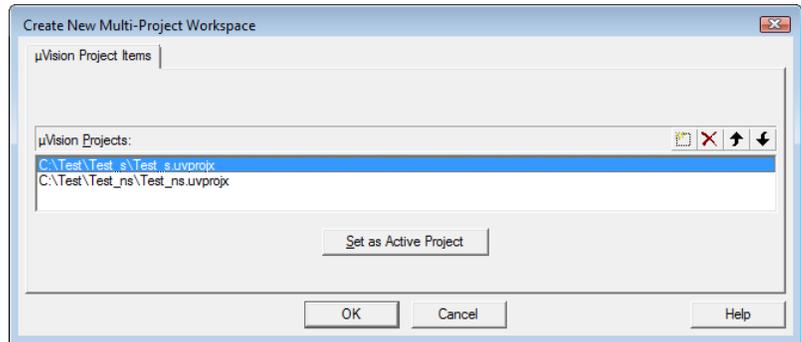
For good debug experience and CMSIS compatibility set the Compiler options on the `C/C++ (AC6)` tab to:

Optimization: “-O1” and Language C: “c99”:

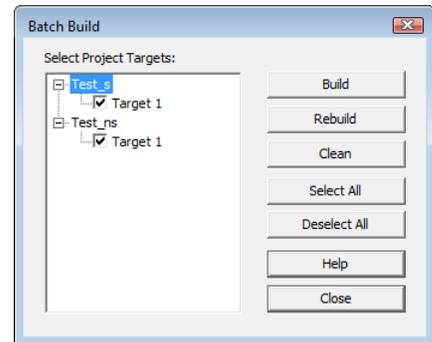


Step 3: Multi-project workspace

A multi-project workspace allows you to work on both projects at the same time. Create a multi-project workspace project in the `Test` directory and add the project files for secure and non-secure projects. The secure project must be first in order to create the import library for the non-secure project.



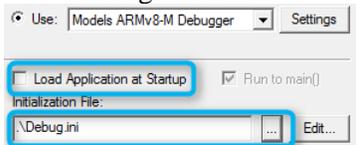
Now you can use **Project** → **Batch Build...** and click **Build** to create both projects in sequence.



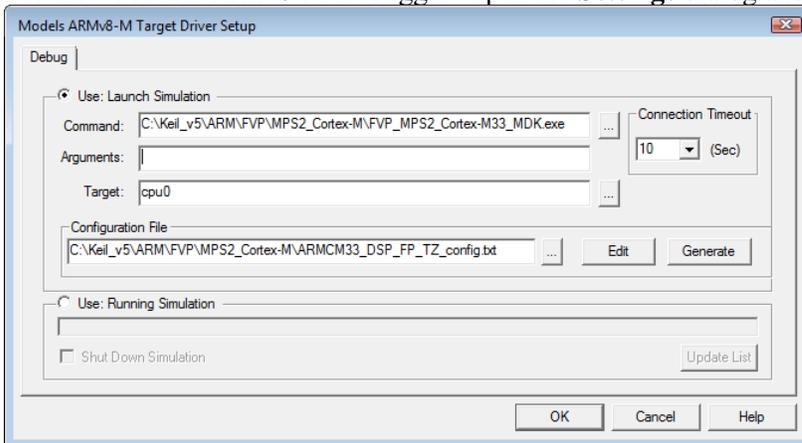
Debugger configuration

Setup µVision debugger in the dialog **Options for Target - Debug**:

- Use: “Models Armv8-M Debugger”
- Disable **Load Application at Startup**
- Add a debug initialization file:



- For the “Models Armv8-M Debugger” open the **Settings** dialog and enter:



- The `Debug.ini` file is executed at the debugger start. Enter the following commands:

```
LOAD "\\CM33_ns\\Objects\\CM33_ns.axf" incremental // load non-secure part
LOAD "\\CM33_s\\Objects\\CM33_s.axf" incremental // load secure part
RESET // reset device
g, \\CM33_s\\main_s\\main // run to 'main' in secure part
```

Example: TrustZone for Armv8-M No RTOS

This example project shows a basic TrustZone for Armv8-M setup. The application uses CMSIS and can be executed on a Fixed Virtual Platform (FVP) simulation model. It demonstrates function calls between secure and non-secure state. The secure application sets the system up and starts non-secure application which calls a secure function from the non-secure state. It also calls a secure function that calls back to a non-secure function. All variables used in this application can be viewed in the μ Vision Debugger Watch window.

Multi-project workspace setup

Once you have opened the project, you will see two projects in the *Project* window. The secure project is called CM33_s and the non-secure project is CM33_ns.

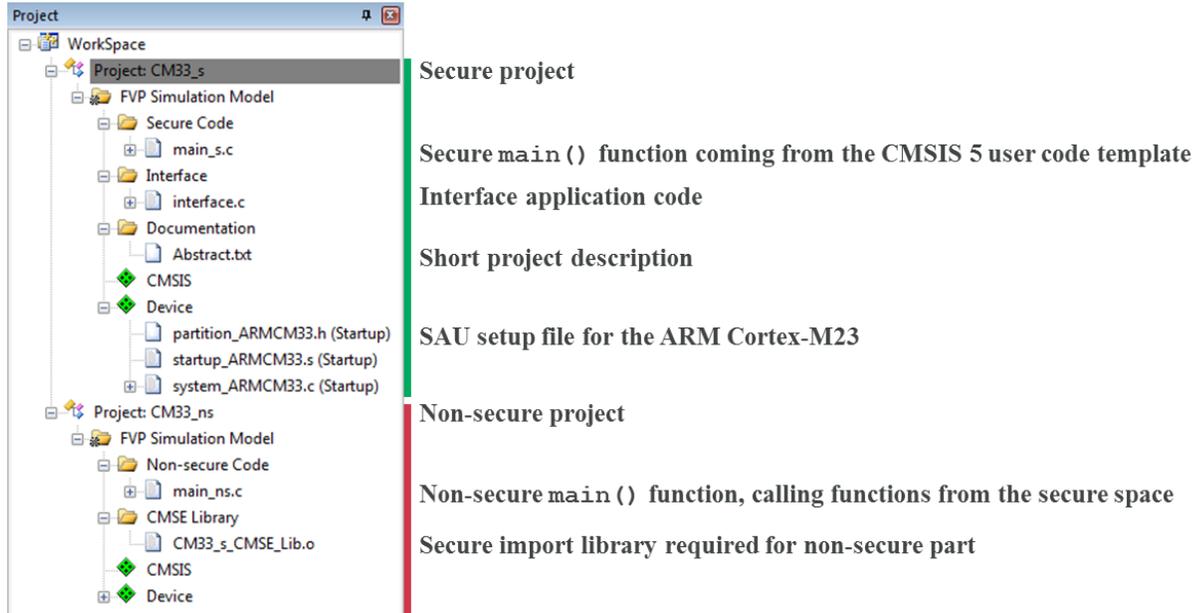


Figure 7 Secure/non-secure multi-project workspace

Program Code

The secure main_s.c file is available as a user code template from the CMSIS pack. No modifications were made to this file. Basically, it sets up the non-secure process and main stack pointers, as well as the reset handler and then switches to the non-secure state.

The functions that are available to the non-secure state are declared in interface.h and implemented in interface.c:

```
#include <arm_cmse.h> // CMSE definitions
#include "interface.h" // Header file with secure interface API

/* typedef for non-secure callback functions */
typedef funcptr funcptr_NS __attribute__((cmse_nonsecure_call));

/* Non-secure callable (entry) function */
int func1(int x) __attribute__((cmse_nonsecure_entry)) {
    return x+3;
}

/* Non-secure callable (entry) function, calling a non-secure callback function */
int func2(funcptr callback, int x) __attribute__((cmse_nonsecure_entry)) {
    funcptr_NS callback_NS; // non-secure callback function pointer
    int y;

    /* return function pointer with cleared LSB */
    callback_NS = (funcptr_NS)cmse_nsfptr_create(callback);

    y = callback_NS (x+1);

    return (y+2);
}
```

By definition, a non-secure function call must use function pointers. This is a consequence of separating secure and non-secure code into separate executable files. So we first define a function pointer for non-secure callbacks with the CMSE attribute `cmse_nonsecure_call`. This function attribute also tells the compiler to generate register-bank saving and clearing code.

`func1` is a non-secure callable or entry function (marked with the CMSE attribute `cmse_nonsecure_entry`). This makes the function callable from the secure or the non-secure state, but while executing, it does not change its context. It will be executed in the secure state.

`func2` is different. It is also an entry function, but it uses a function pointer for a callback function. Within the function, this function pointer is used as a non-secure callback function pointer. The CMSE intrinsic `cmse_nsfp_create` returns the value of the callback function pointer, only with its LSB cleared which marks it as non-secure. The state switches from secure to non-secure, the function executes, and the result is returned to the secure function which then returns it to the non-secure state.

The `main_ns.c` file mainly consists of the following code:

```
...
volatile int val1, val2;

/* Non-secure function */
int func3 (int x);

int func3 (int x) {
    return (x+4);
}

/* Non-secure main() */
int main(void) {

    /* Call non-secure callable function func1 */
    val1 = func1 (1);

    /* Call non-secure callable function func2
       with callback to non-secure function func3 */
    val2 = func2 (func3, 2);

    while (1);
}
```

`func3` is a non-secure function. In `main()`, the non-secure code calls the non-secure callable function `func1` and the non-secure callable function `func2`. For `func2` it uses `func3` as a parameter and this is why in `main_s.c` the function pointer to this function is marked as being non-secure.

Call sequence

Figure 8 shows the call sequence of the **No RTOS** example including the secure/non-secure state transitions.

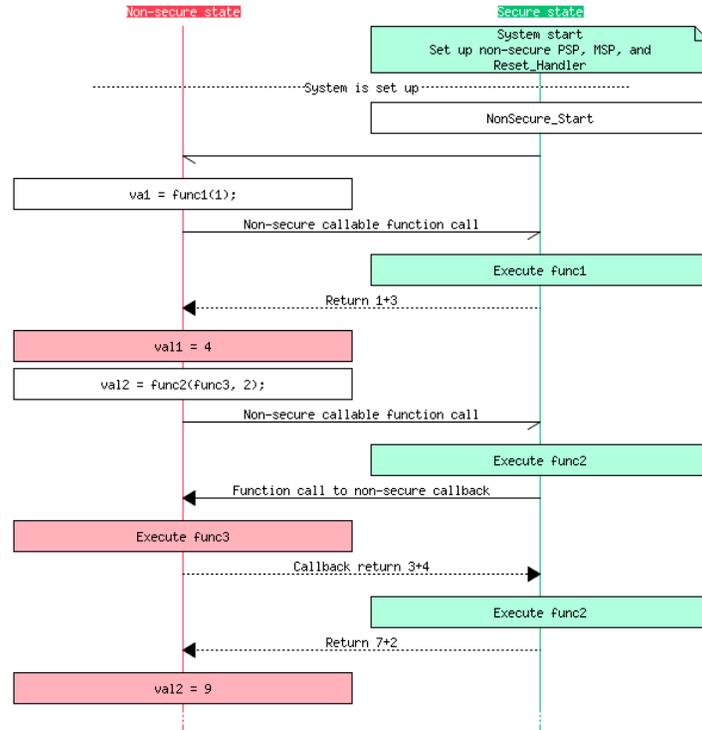
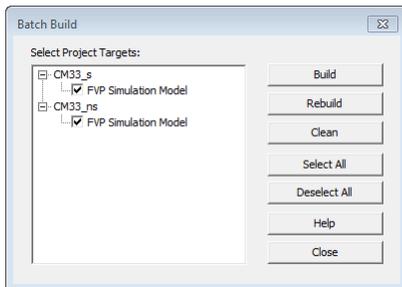


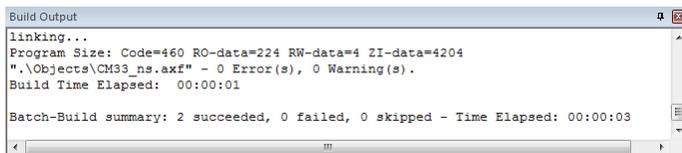
Figure 8 No RTOS example secure/non-secure state switches

Project Build

To build both projects at once, go to **Project** → **Batch Build...** or use the batch build icon . Build both projects in one step using the **Build** button:



The **Build Output** window will show a successful build without any errors or warnings:



Debug non-secure to secure state switches

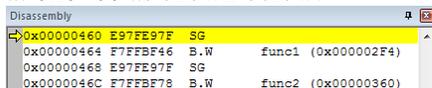
The following describes how a non-secure program calls a function in the secure mode.

1. Set a breakpoint in main_ns.c at line 42. This is the call from non-secure mode to the function func1 in the secure memory area.
2. Click Run  or F5 and the program stops as shown here:
3. Note the address in the Disassembly window (in this case) is 0x000200256. This is in the non-secure memory as specified in the partition_ARMCM33.h header file. At the bottom right this is also displayed as CPU: Non-Secure
4. This will also be displayed in the Registers window.
5. Click on the Disassembly window to bring it in focus.

TIP: It is important that the Disassembly window remains in focus. This ensures steps are by assembly instruction. Otherwise the steps will be by source code line and you won't be able to see the details of the secure state switch.

Enter secure mode:

6. Click Step  or F11 once to reach the BL instruction.
7. This is the Branch with Link to the func1 function in the secure memory area.
8. Click Step  or F11 to execute this BL.
9. The program will jump to the veneer SG (Secure Gate) instruction at 0x0460 as shown below:



```
Disassembly
0x00000460 E97FE97F SG
0x00000464 F7FFB46 B.W func1 (0x000002F4)
0x00000468 E97FE97F SG
0x0000046C F7FFB78 B.W func2 (0x00000360)
```

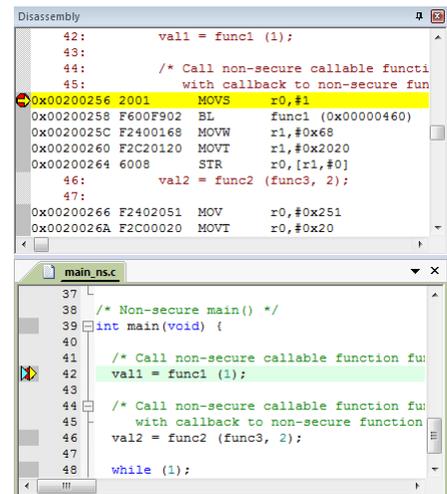
Note that this is in the secure memory area.

10. Click Step  or F11 to execute SG.
11. The CPU will now change to Secure mode: CPU: Secure
12. Click Step  or F11 to execute B.W.
13. The program counter will now be at the beginning of the func1 function.
14. Note the memory address is 0x02F4 which is in the secure area.

Exit secure state and switch back to non-secure state:

15. Find the instruction BXNS near line 0x035C at the end of the func1 function.
16. Set a breakpoint on this line and click Run  or F5.
17. Unselect the breakpoint as we do not need it.
18. BXNS is a new Armv8-M instruction that is used when returning from an entry function.
19. Click Step  or F11 twice.
20. val1 will show the new value '4'. Note that the CPU is back in non-secure mode: CPU: Non-Secure

The cycle is now complete. Go to **Debug** → **Breakpoints** (or press Ctrl-B) to unselect the breakpoint you set in main_ns.c.



Example: TrustZone for Armv8-M RTOS

The example project **TrustZone for Armv8-M RTOS** is based on the **No RTOS** example (explained on page 6) but adds an RTOS layer. It differs from the No RTOS example as follows:

- It uses the **Source_NS** variant of the software component **CMSIS:RTOS2 (API):Keil RTX5** which adds RTX in source code format
- The user code template **tz_context.c** is added to provide the implementation of the context management API

The only code changes are in the `main_ns.c` file. Instead of calling the non-secure callable functions, it is running RTX with threads that contain the function calls.

Call sequence

Figure 9 shows the call sequence of the **RTOS** example including the secure/non-secure state transitions.

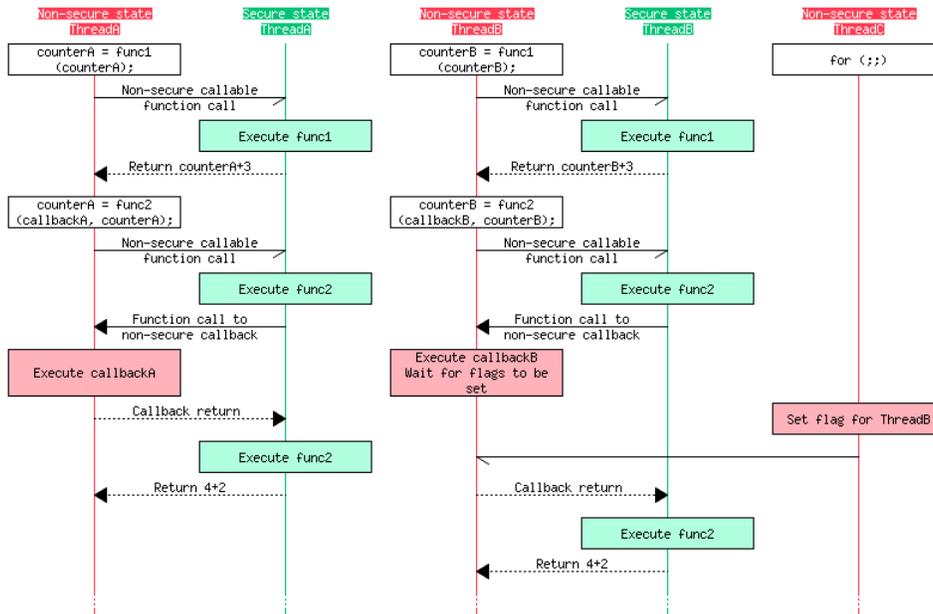


Figure 9 RTOS example secure/non-secure state switches

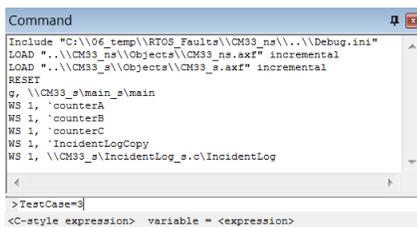
Example: TrustZone for Armv8-M RTOS Security Tests

The secure/non-secure RTOS example with security test cases and system recovery shows how the Armv8-M architecture reacts to potential security attacks. Each attack gets recorded in a log file and the application is automatically reset.

As for the previous project example, the secure project is used to boot the device and then hands-over to the non-secure project. Here, the user can trigger various security attacks/faults:

- **Illegal secure call (TestCase=0):** this attack tries to call directly into secure memory.
- **Stack overflow (TestCase=1):** this fault tries to allocate more memory on the stack than available, by creating a local variable that is too large. This will cause a PSPLIM error.
- **Division by zero (TestCase=2):** this example fault is caused by a division by zero. The fault is only generated, when SCB_CCR_DIV_0_TRP is set.
- **Data attack (TestCase=3):** tries to let secure domain overwrite secure memory by providing a pointer outside of the non-secure memory area.
- **Play dead (TestCase=4):** this attack simulates a broken non-secure application which is not returning. A secure SysTick watchdog is used to detect this inactivity.

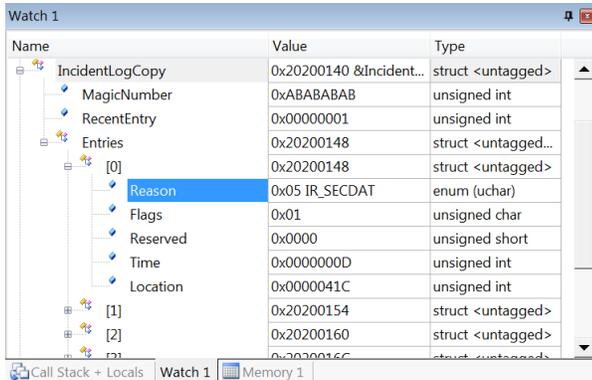
You can run the example in the μ Vision debugger without hardware. Build the projects, enter debug mode  (Ctrl+F5) and run  (F5) the project. In the **Command** window, enter `TestCase=n` (see test case numbers above):



```
Command
Include "C:\106_temp\RTOS_Faults\CM33_ns\...\Debug.ini"
LOAD "...CM33_ns\Objects\CM33_ns.axf" incremental
LOAD "...CM33_ns\Objects\CM33_s.axf" incremental
RESET
g, \CM33_s\main_s\main
WS 1, 'counterA
WS 1, 'counterB
WS 1, 'counterC
WS 1, 'IncidentLogCopy
WS 1, \CM33_s\IncidentLog_s.c\IncidentLog

>TestCase=3
<C-style expression> variable = <expression>
```

The last four security incidents will be logged and can be observed in the **Watch** window:



Name	Value	Type
IncidentLogCopy	0x20200140 &Incident...	struct <untagged>
MagicNumber	0xABABABAB	unsigned int
RecentEntry	0x00000001	unsigned int
Entries	0x20200148	struct <untagged...>
[0]	0x20200148	struct <untagged>
Reason	0x05_IR_SECDAT	enum (uchar)
Flags	0x01	unsigned char
Reserved	0x0000	unsigned short
Time	0x0000000D	unsigned int
Location	0x0000041C	unsigned int
[1]	0x20200154	struct <untagged>
[2]	0x20200160	struct <untagged>
[3]	0x2020016C	struct <untagged>

Appendix

MDK – Microcontroller Development Kit

Arm Keil MDK is a comprehensive software development solution for Arm-based microcontrollers and includes all components that you need to create, build, and debug embedded applications.

MDK includes the Arm Compiler 6 that combines LLVM technology with highly optimized Arm C libraries. Arm Compiler 6 improves compatibility with GNU GCC, covers the latest C language standards including C++11 and C++14, and the Armv8-M Security Extensions for secure application programming.

Software Packs add device support and software components which are used as application building blocks. MDK includes CMSIS, RTOS, and royalty-free middleware designed for microcontrollers. Third-party software packs provide components for IoT, security, encryption, and networking applications.

The MDK debugger offers access modes for secure and non-secure application verification and optimization. It connects to both simulation models and target hardware using debug adapters. Debugging is accelerated with meaningful peripherals dialogs and even while the program is running at full speed, variables can be inspected and breakpoints may be altered. Trace capabilities include variable tracking, code coverage, and performance analysis.

Visit www.keil.com/mdk for more information.

ULINK – Debug/trace adapter series

A ULINK debug adapter connects the MDK debugger to the target system and allows to program, debug, and analyze applications. Serial-wire trace offers event and timing information on interrupt execution, RTOS thread scheduling, code annotations, and variable access. Streaming trace utilizes the ETM technology for non-intrusive performance analysis and complete code coverage during system validation with real-time code execution.

Visit www.keil.com/ulink for more information.

CMSIS – Cortex Microcontroller Software Interface Standard

CMSIS provides industry standard software support for the Cortex-M series and includes an open source software framework with processor HAL, DSP library, and RTOS kernel. CMSIS-Pack defines the distribution of device support and software components and is widely adopted in the industry.

CMSIS Version 5 is extended for the Armv8-M architecture including access to TrustZone hardware security extensions. The RTOS API standardizes access to the secure domain which ensures software compatibility across compliant real-time operating systems. The RTX reference implementation is a full featured real-time operating system for non-secure applications that interfaces to the secure domain for data and firmware protection.

Visit www.keil.com/cmsis for more information.

Books

- **Free! Getting Started with MDK 5:** www.keil.com/mdk5/install.
- A list of books on Arm processors is found at: www.arm.com and search for *books*
- μ Vision[®] contains a window titled Books. Many documents including data sheets are located there.
- The Definitive Guide to the Arm Cortex-M0/M0+ ISBN 978-0123854773
- The Definitive Guide to the Arm Cortex-M3/M4 ISBN 978-0124080829
- Embedded Systems: Introduction to Arm Cortex-M Microcontrollers (3 volumes) by Jonathan Valvano.

Application notes

- Arm Compiler Qualification Kit: www.keil.com/safety
- Using Cortex-M3 and Cortex-M4 Fault Exceptions www.keil.com/appnotes/files/apnt209.pdf
- CAN Primer using NXP LPC1700: www.keil.com/appnotes/files/apnt_247.pdf
- CAN Primer using the STM32F Discovery Kit www.keil.com/appnotes/docs/apnt_236.asp
- Segger emWin GUIBuilder with μ Vision www.keil.com/appnotes/files/apnt_234.pdf
- Porting an mbed project to Keil MDK www.keil.com/appnotes/docs/apnt_207.asp
- MDK Compiler Optimizations www.keil.com/appnotes/docs/apnt_202.asp
- CMSIS-RTOS RTX in MDK: C:\Keil_v5\ARM\Pack\ARM\CMSIS\
Search for *DAI0321A* on www.arm.com
- Barrier Instructions Search for *DAI0298A* on www.arm.com
- Lazy Stacking on the Cortex-M4 <http://community.arm.com/docs/DOC-8587>
- Cortex-M Processors for Beginners: <http://community.arm.com/docs/DOC-8587>
- Arm CoreSight: www.keil.com/coresight
- Sending ITM printf to external Windows applications: www.keil.com/appnotes/docs/apnt_240.asp
- Migrating from Cortex-M4 to Cortex-M7 Processors: www.keil.com/appnotes/docs/apnt_270.asp
- ROM Self-Test in MDK: www.keil.com/appnotes/docs/apnt_277.asp
- Using ST-Link/V2 and MDK: www.keil.com/appnotes/docs/apnt_286.asp

Useful Arm websites

- CMSIS Standards: www.arm.com/cmsis/ www.keil.com/cmsis/
- Keil Forums: www.keil.com/forum
- Arm Developer: <https://developer.arm.com/embedded>
- Arm University Program: www.arm.com/university
- Arm mbed: www.mbed.com

Keil Direct Sales In USA: sales.us@keil.com or 800-348-8051. **Outside the US:** sales.intl@keil.com

Keil Distributors: See www.keil.com/distis/

Keil Technical Support in USA: support.us@keil.com or 800-348-8051. **Outside the US:** support.intl@keil.com.