Stack sealing on Armv8-M devices with CMSIS

MDK Tutorial

AN335, Winter 2021, v1.0

keil-feedback@arm.com

Abstract

This application note explains how to implement stack sealing for a CMSIS-based project. It also includes an example project for an Arm Cortex-M33 using a Fixed Virtual Platform (FVP) model.

The stack sealing technique is introduced in **Armv8-M Secure Stack Sealing advisory notice** that explains how to properly manage the secure stacks on Armv8-M architecture devices, such as Cortex-M23, Cortex-M33, Cortex-M35P and Cortex-M55. This ensures that non-secure software cannot potentially manipulate the stack and influence the secure control flow.

Contents

Prerequisites

To reproduce the example described in this application note the following tools are required:

Components from Arm:

- **Arm Keil MDK**: IDE and debugger used for project development and debug.
  MDK-Professional edition is required to run the reference example on Arm Fixed Virtual Platform (FVP). You can obtain a 30-day evaluation license of MDK-Professional as described at keil.com/support/man/docs/license/license_eval.htm. MDK v5.33 was used to verify the examples.
- **Arm CMSIS**: version 5.7.0 without provisions for stack sealing is used in the example.

- **Arm Fixed Virtual Model** for Cortex-M33 (FVP_MPS2_Cortex-M33_MDK.exe): included with Keil MDK-Professional edition. Version 11.12 is used to create the example.

Example project

A ZIP file is available for download at **keil.com/appnotes/docs/apnt_335.asp**. It contains projects implementing stack sealing as well as modified CMSIS-Core files.

Introduction

The **Armv8-M Secure Stack Sealing advisory notice** analyzes a vulnerability of secure software executed on Armv8-M processors to attacks from the non-secure state when the secure stacks are not properly managed. It also describes a mechanism called stack sealing that mitigates this vulnerability.

CMSIS-Core (M) v5.0.0 through v5.4.0 (delivered as part of CMSIS v5.0.0 through v5.7.0) provide support for devices based on Armv8-M architecture, and include templates for startup code and system files that are commonly used by device vendors to implement CMSIS compliant device support. However, in these versions of CMSIS-Core(M) there is no support for the stack sealing mechanism and additional considerations and modifications are required from users to add that functionality to a project.

The idea behind the stack sealing is to add a so called "seal" on top of the secure stacks. This seal consists of two words with the special value *0xFEF5EDA5* which are placed just above the real stack space.

This application notes uses a Keil MDK project and explains the modifications required to implement the stack sealing mechanism for the main secure stack. Process stack is typically managed by the application or an RTOS and is out of scope of this application note that targets a generic CMSIS-Core application (bare-metal).

The descriptions provided in the application note and related examples are based on an Arm Fixed Virtual Platform (FVP) model for Cortex-M33 delivered with MDK, but similar steps can be followed to add stack sealing on real target hardware.

Modifications in CMSIS-Core Files

CMSIS-Core for Cortex-M already implements functions for managing stack pointers and supports handling of secure and non-secure states on Armv8-M. So, it is logical to extend CMSIS-Core with a function that implements the stack sealing operation and thus can be universally used in device startup C code.

Note: if startup file is implemented in Assembler then the changes described below are not relevant. Proceed directly to section **Modifications in the Startup File**.

Following the CMSIS naming convention we introduce a new function:

```
__TZ_set_STACKSEAL_S (uint32_t* stackTop)
```

that adds the recommended values on top of the memory provided by `stackTop` pointer. This function becomes available from CMSIS version 5.8.0.

CMSIS version 5.7.0 and earlier do not contain this function. When running an application on this CMSIS versions it is recommended to make some additional modifications in the startup file and so avoid modifications in the CMSIS Core files. This is explained in section **C Startup file for use with CMSIS 5.7.0 or below** and **Assembler Startup file for use with CMSIS 5.7.0 or below**.

Subsections below explain modifications in the CMSIS-Core files for CMSIS 5.8.0 for Arm Compiler 6.x and GCC Compiler. The .zip archive supplied with the application note contains the modified CMSIS-Core files for the reference.

Arm Compiler 6.x support

For use with Arm Compiler 6.x, `__TZ_set_STACKSEAL_S` function could be added in *cmsis_armclang.h* file as follows:

```
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
#ifndef __STACK_SEAL
#define __STACK_SEAL              Image$$STACKSEAL$$ZI$$Base
#endif

__STATIC_FORCEINLINE void __TZ_set_STACKSEAL_S (uint32_t* stackTop) {
  *(stackTop    ) = 0xFEF5EDA5;
  *(stackTop + 1) = 0xFEF5EDA5;
}
#endif
```

GCC Compiler support

For use with GCC Compiler, similar code can be added to *cmsis_gcc.h* file, with the *__StackSeal* value to be defined in a scatter file.

```
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
#ifndef __STACK_SEAL
#define __STACK_SEAL              __StackSeal
#endif

__STATIC_FORCEINLINE void __TZ_set_STACKSEAL_S (uint32_t* stackTop) {
  *(stackTop    ) = 0xFEF5EDA5;
  *(stackTop + 1) = 0xFEF5EDA5;
}
#endif
```

Modifications in the Startup File

We recommend performing the stack sealing operation for the main secure stack in the *Reset_Handler* function of the secure software before the *SystemInit()* function is called. This ensures that the stack sealing is already performed before any non-secure software is executed.

Additionally, even if not used by an application, the values for the process stack pointer and limit register need to be assigned.

The startup file can be implemented in C or assembler languages. Sections below explain modifications for both cases. The application note examples demonstrate both implementation variants as explained in section **Example applications**.

Implementation in C language

The C language implementation of the *Reset_Handler* with stack sealing is done in the file *startup_ARMCM33.c* as described below. Code lines added for stack sealing are highlighted as New! in the comments.

Note that if CMSIS v5.7.0 or earlier is used then both the define *__STACK_SEAL* and the function *__TZ_set_STACKSEAL_S* are not present in the CMSIS-Core files and the code-snippet below should be different.

The examples provided with the application note enable by default the code for use with CMSIS 5.7.0 as explained in section **C Startup file for use with CMSIS 5.7.0 or below.**

For CMSIS versions 5.8.0 or above where the necessary provisions are present, only the following reference to the *__STACK_SEAL* is required in the secure startup file, and is same for Arm Compiler 6 or GCC Compiler:

```
/* Use #if defined block below with CMSIS v5.8.0 or higher, otherwise comment it. */
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U) // New!
extern uint32_t __STACK_SEAL;                                   // New! Defined in CMSIS-Core
#endif                                                          // New!
```

Later in the file, the *Reset_Handler* function is updated as follows:

```
__NO_RETURN void Reset_Handler(void)
{
  __set_PSP((uint32_t)(&__INITIAL_SP));     // New! Process Stack Pointer shall be set
  __set_MSPLIM((uint32_t)(&__STACK_LIMIT));
  __set_PSPLIM((uint32_t)(&__STACK_LIMIT)); // New! Process Stack Pointer Limit shall be set

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U) // New!
  __TZ_set_STACKSEAL_S((uint32_t *)(&__STACK_SEAL));           // New! Perform Stack Sealing
#endif                                                         // New!

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

C Startup file for use with CMSIS 5.7.0 or below

CMSIS 5.7.0 and earlier versions do not contain definitions for *__STACK_SEAL* and function *__TZ_set_STACKSEAL_S* used by the C startup file described above. To avoid modifications of the CMSIS Pack files, it is possible to implement them fully in the C startup file.

For that the code snippet defining the *__STACK_SEAL* in the startup C file (*startup_ARMCM33.c* in our example) shall be commented, and the following code shall be added instead.

For Arm Compiler 6.x:

```
/* Use #if defined block below with CMSIS v5.7.0 or lower (CMSIS-Core 5.4.0 or lower),
   otherwise comment it. */
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  #define __STACK_SEAL                  Image$$STACKSEAL$$ZI$$Base
  extern uint32_t __STACK_SEAL;
  __STATIC_FORCEINLINE void __TZ_set_STACKSEAL_S (uint32_t* stackTop) {
    *(stackTop    ) = 0xFEF5EDA5;
    *(stackTop + 1) = 0xFEF5EDA5;
  }
#endif
```

-Wno-dollar-in-identifier-extension need to be added to the compiler options string to avoid warnings related to the use of $ sign.

Implementation of the *Reset_Handler* function is the same as explained above in the section **Implementation in C language**.

For GNU Compiler the code snippet from above looks very similar with only *__STACK_SEAL* being defined differently:

```
/* Use #if defined block below with CMSIS v5.7.0 or lower (CMSIS-Core 5.4.0 or lower),
  otherwise comment it. */
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  #define __STACK_SEAL                  __StackSeal
  extern uint32_t __STACK_SEAL;
  __STATIC_FORCEINLINE void __TZ_set_STACKSEAL_S (uint32_t* stackTop) {
    *(stackTop    ) = 0xFEF5EDA5;
    *(stackTop + 1) = 0xFEF5EDA5;
  }
#endif
```

The *__StackSeal* value need to be defined in the scatter file as explained in corresponding section **Modifications in Scatter File**.

Implementation in Assembler

It is strongly recommended to use C startup files with CMSIS projects because templates for assembler startup files are deprecated and not maintained. But for legacy projects it is also possible to implement stack sealing mechanism in an assembler startup file.

Below is the description of the assembler implementation for Arm Compiler in armclang GNU syntax with pre-processing as also shown in the reference examples (CM33_AC6.S_NS\App_s\RTE\Device\ARMCM33_DSP_FP_TZ\startup_ARMCM33.S).

Note that template files and reference Armv8-M devices implemented in CMSIS v5.7.0 use an armasm syntax and need to be modified as explained in section **Assembler Startup file for use with CMSIS 5.7.0 or below**.

First *__STACK_SEAL* address is defined, as well as *__INITIAL_SP* and *__STACK_LIMIT*:

```
                #define __INITIAL_SP     Image$$ARM_LIB_STACK$$ZI$$Limit
                #define __STACK_LIMIT    Image$$ARM_LIB_STACK$$ZI$$Base
                #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
                #define __STACK_SEAL     Image$$STACKSEAL$$ZI$$Base
                #endif
```

Then, the *Reset_Handler* is implemented with values assigned to the stack pointers and limits and stack sealing performed:

```
Reset_Handler:
                ldr      r0, =__INITIAL_SP
                msr      psp, r0

                ldr      r0, =__STACK_LIMIT
                msr      msplim, r0
                msr      psplim, r0

                #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
                ldr      r0, =__STACK_SEAL
                ldr      r1, =0xFEF5EDA5
                strd     r1,r1,[r0,#0]
                #endif

                bl       SystemInit

                bl       __main

                .fnend
                .size    Reset_Handler, . - Reset_Handler
```

An implementation for the GNU Compiler is very similar. It defines *__INITIAL_SP*, *__STACK_LIMIT*, and *__STACK_SEAL* equal to the corresponding parameters that need to be defined in the scatter file.

```
                #define __INITIAL_SP     __StackTop
                #define __STACK_LIMIT    __StackLimit
                #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
                #define __STACK_SEAL     __StackSeal
                #endif
```

The *Reset_Handler* implementation can be kept the same as for the Arm Compiler, but up to the *__main* call, that should be replaced with the necessary initializations, because such pre-main function does not exist in GNU compiler. For details, please refer to the examples provided with the application notes.

Assembler Startup file for use with CMSIS 5.7.0 or below

Assembler startup files for the reference Armv8-M devices implemented in CMSIS v5.7.0 and lower use armasm syntax. However, because the stack sealing relies on definitions in the scatter file, it is convenient to update the whole startup file to armclang format with GNU syntax as used in the previous section.

So, in a project based on CMSIS 5.7.0 or lower update the assembler startup file with the startup file provided in the **Example applications**. For Arm Compiler this is CM33_AC6.S_NS\App_s\RTE\Device\ARMCM33_DSP_FP_TZ\startup_ARMCM33.S. Additionally, when using Arm Compiler, add following options to the Asm control *-x assembler-with-cpp.* In a Keil MDK project this can be done via **Options for Target** dialog - **Asm** tab and then the options need to be added in the **Misc. Controls** field.

For GCC Compiler use file CM33_GCC.S_NS\App_s\RTE\Device\ARMCM33_DSP_FP_TZ\startup_ARMCM33.S. as a reference.

Modifications in Scatter File

The stack sealing mechanism requires to use a scatter file (linker script) to reserve 8 bytes for the stack seal on top of secure main stack. This section explains the required changes in the scatter file. Other memory configuration options using the scatter file (common for Armv8-M devices such as memory split in secure and non-secure areas) is out of scope for this document.

Arm Compiler 6.x support

The scatter file for Arm Linker can be found in the reference example in *ARMCM33_ac6_s.sct* file. This is done as follows.

First, we just define the size of the stack seal as 8 bytes for secure code and 0 bytes for non-secure:

```
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
#define __STACKSEAL_SIZE     8
#else
#define __STACKSEAL_SIZE     0
#endif
```

Later, we shift the stack top value from the RAM end by the size of the stack seal:

```
#define __STACK_TOP    (__RAM_BASE + __RAM_SIZE - __STACKSEAL_SIZE) /* starts at end of RAM -
stack seal */
```

Finally, for secure project we reserve the memory for the stack seal right after the stack memory:

```
…
  ARM_LIB_STACK __STACK_TOP EMPTY -__STACK_SIZE { ; Reserve empty region for stack
  }
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  /* stack seal immediately after stack */
  STACKSEAL +0 EMPTY __STACKSEAL_SIZE {
  }
#endif
```

GCC Compiler support

The GNU Linker script *gcc_arm.ld* requires the following additions for the secure project to support stack sealing feature.

First, define the stack seal size:

```
  /* ARMv8-M stack sealing:
     to use ARMv8-M stack sealing uncomment set __STACKSEAL_SIZE to 8 otherwise keep 0
   */
__STACKSEAL_SIZE = 8;
```

Then, allocate memory for the stack with the size decremented by the stack seal size. Here the parameters *__StackLimit* and *__StackTop* get defined and later are used in the startup file.

```
.stack (ORIGIN(RAM) + LENGTH(RAM) - __STACK_SIZE - __STACKSEAL_SIZE )(COPY) :
  {
    . = ALIGN(8);
    __StackLimit = .;
    . = . + __STACK_SIZE;
    . = ALIGN(8);
    __StackTop = .;
  } > RAM
  PROVIDE(__stack = __StackTop);
```

Finally allocate the memory for the stack seal itself.

```
  .stackseal (ORIGIN(RAM) + LENGTH(RAM) - __STACKSEAL_SIZE) (COPY) :
  {
    . = ALIGN(8);
    __StackSeal = .;
    . = . + 8;
    . = ALIGN(8);
} > RAM
```

Example applications

This application note comes with a ZIP file ([keil.com/appnotes/docs/apnt_335.asp](https://www.keil.com/appnotes/docs/apnt_335.asp)) that contains simple µVision examples implementing stack sealing mechanism as described in the previous chapters.

Separate project workspaces are provided for Arm Compiler support and for GNU Compiler. Both workspaces contain secure and non-secure projects implementing corresponding parts of the application. And each project has two target variants: *FVP C-Start* target uses startup file in C language, while *FVP A-Start* uses startup file in Assembler.

By default, the *FVP C-Start* targets work with CMSIS 5.7.0 or lower. To use them with later CMSIS releases (that already contain provisions for stack sealing) the C startup files need to be modified as explained in section **Modifications in the Startup File**.

The examples use a Cortex-M33 FVP that allows code execution without target hardware. For that an MDK-Professional edition is required. You can obtain a 30-day evaluation license of MDK-Professional as explained at [keil.com/support/man/docs/license/license_eval.htm](https://www.keil.com/support/man/docs/license/license_eval.htm).

Make sure that the in the **Options for Target..** dialog, **Debug** tab, **Models ARMv8-M Debugger** is selected as the target debugger and after clicking **Settings** button, verify that the **Command** field contains the correct path to the target FVP file. By default, MDK has it at *<MDK path>ARM\FVP\MPS2_Cortex-M\FVP_MPS2_Cortex-M33_MDK.exe*.

In the project the secure RAM is is configured as:

```
__RAM_BASE       0x20000000
__RAM_SIZE       0x00020000
```

The secure main stack is placed at the end of that RAM area and the stack sealing bytes are located at address *0x2001FFF7*.

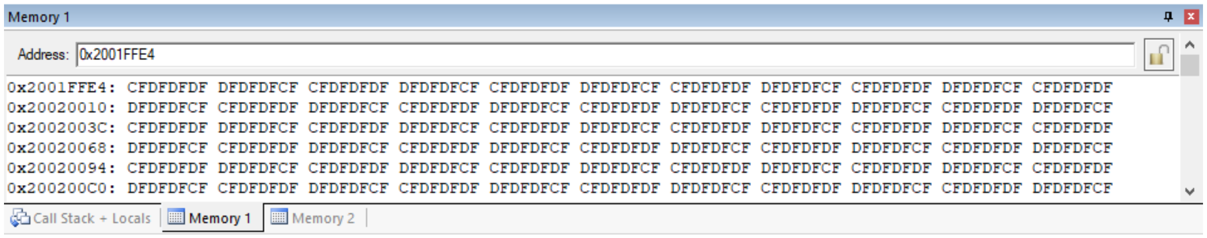
Arm Compiler 6.x support

The directory *CM33_AC6.S_NS* contains an example implementing stack sealing for Arm Compiler 6.x. The steps below explain how to run the program and observe the stack seal in the memory.

- Open the *S_NS.uvmpw* multi-project workspace in µVision. It contains two projects: secure (*App_s*) and non-secure (*App_ns*). By default, the *FVP C-Start* target is selected in both projects.
- Batch-build both projects. The projects should build without errors and warnings.
- Start a debug session (CTRL+F5).

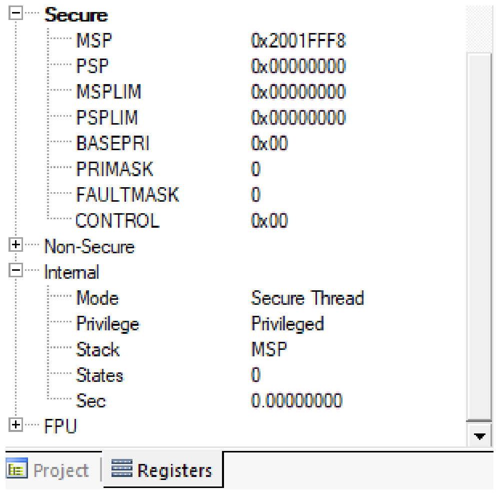
  The program stops at the beginning of the *Reset_Handler.* Note that if a C startup file is used, this may be optimized and debugger could stop at the beginning of *__set_PSP* function called first within the *Reset_Handler*.

  The *Memory 1* window shows that the FVP memory is filled with default values *0xCFDFDFDFDFDFDFCF* and as expected the stack seal is not yet applied:



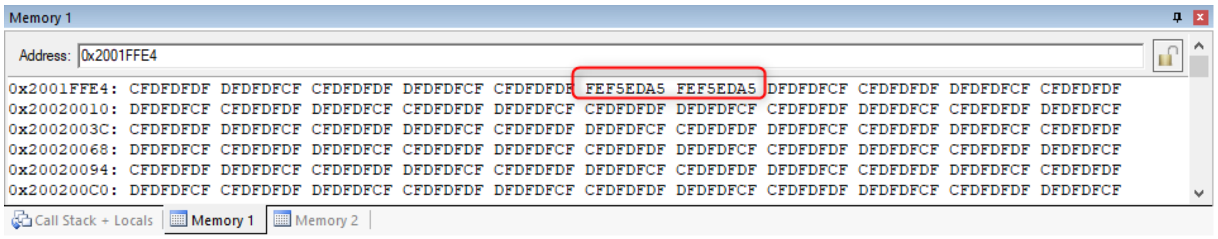
```
Address: 0x2001FFE4
0x2001FFE4: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020010: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x2002003C: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020068: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x20020094: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x200200C0: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
```

You can open the Registers view and see that among the secure stack registers only MSP has a value assigned:



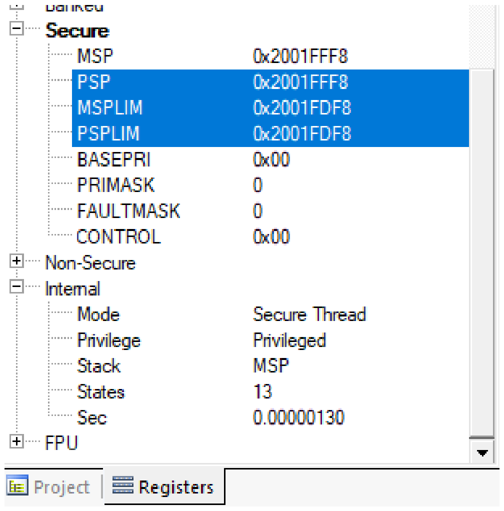
```
Secure
    MSP          0x2001FFF8
    PSP          0x00000000
    MSPLIM       0x00000000
    PSPLIM       0x00000000
    BASEPRI      0x00
    PRIMASK      0
    FAULTMASK    0
    CONTROL      0x00
Non-Secure
Internal
    Mode         Secure Thread
    Privilege    Privileged
    Stack        MSP
    States       0
    Sec          0.00000000
FPU
```

- Run the program (F5). Execution stops at a breakpoint in the beginning of *SystemInit* function and the stack sealing is already applied. Corresponding values *0xFEF5EDA5* can be observed in the *Memory 1* window as well:



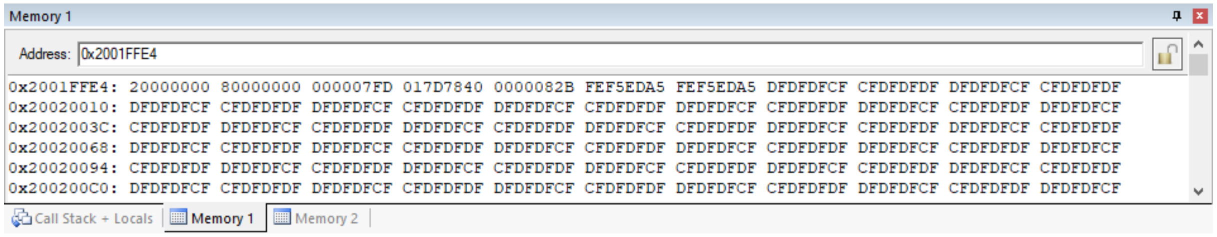
```
Address: 0x2001FFE4
0x2001FFE4: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF FEF5EDA5 FEF5EDA5 DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020010: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x2002003C: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020068: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x20020094: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x200200C0: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
```

All secure stack register values get values assigned to them as well.



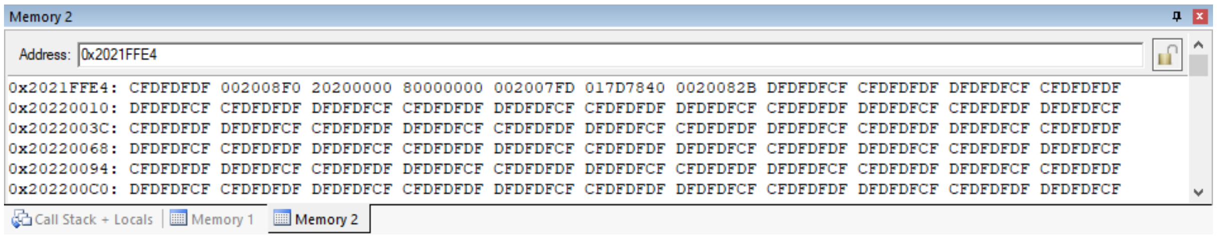
```
Banked
Secure
    MSP          0x2001FFF8
    PSP          0x2001FFF8
    MSPLIM       0x2001FDF8
    PSPLIM       0x2001FDF8
    BASEPRI      0x00
    PRIMASK      0
    FAULTMASK    0
    CONTROL      0x00
Non-Secure
Internal
    Mode         Secure Thread
    Privilege    Privileged
    Stack        MSP
    States       13
    Sec          0.00000130
FPU
```

- Continue program execution (F5). The program stops at a breakpoint in the *main* function of the secure code. You can observe that the main secure stack is being used now with default values overwritten up to the stack seal that stays untouched:



```
0x2001FFE4: 20000000 80000000 000007FD 017D7840 0000082B FEF5EDA5 FEF5EDA5 DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020010: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x2002003C: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20020068: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x20020094: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x200200C0: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
```

- Continue program execution further (F5). The program switches to execute the non-secure code. Using the *Memory 2* window you can observe the non-secure stack and see that there is no stack seal present there.



```
0x2021FFE4: CFDFDFDF 002008F0 20200000 80000000 002007FD 017D7840 0020082B DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20220010: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x2022003C: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x20220068: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
0x20220094: CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF
0x202200C0: DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF CFDFDFDF DFDFDFCF
```

GCC Compiler support

The directory *CM33_GCC.S_NS* contains a project demonstrating stack sealing implementation for a GCC Compiler. To run the program follow the steps as described in the previous section for Arm Compiler, but use the workspace *S_NS.uvmpw* in *CM33_GCC.S_NS* folder.

Summary

In this application note, we have explained how to implement a stack sealing mechanism in CMSIS-based Armv8-M devices. Example applications for Arm Compiler 6.x and GCC Compiler are provided with the application note.

References and Useful Links

[1] Armv8-M Secure Stack Sealing advisory notice
[2] Application Note 291: Using TrustZone on ARMv8-M
[3] Secure software guidelines for Armv8-M