embeddedworld2017
Exhibition&Conference
... it's a smarter world

Software Development in ARMv8-M Architecture

A technical overview of software development on Cortex-M23 and Cortex-M33 processors

Joseph Yiu
Senior Embedded Technology Manager,
CPU Product Group, ARM
Cambridge, UK

***Abstract* — The next generation of ARM® Cortex®-M processors, Cortex-M23 and Cortex-M33, based on the ARMv8-M architecture, introduce optimized TrustZone® security features into microcontrollers. These new features make it easier to create ultra-low power secure solutions, whether for IoT or the wider embedded market. This paper will outline the new features from a software developer's point of view and provide guidance on how developers can use the new security features as a foundation for secure embedded products.**

The paper will look at the software needed to make use of the new features of the processors, including topics such as memory protection and chip start-up. The paper will then explain how developers can ensure that the majority of software running on the new processors can work without change whilst still benefiting from security services protected by the isolation offered by TrustZone for ARMv8-M.

Keywords*— *Cortex-M; Architecture; Security Extension; TrustZone; Internet of Things (IoT); ACLE; CMSE; Coprocessor

I. INTRODUCTION

The ARM® Cortex®-M23 and the Cortex-M33 processors were announced in October 2016, enhancing the successful Cortex-M processor family with the new ARMv8-M Architecture. This new architecture brings in a number of architecture changes compared to the previous version of the architecture which are visible from a software developer point of view. In addition, there are a number of additional features in these processors and this paper highlights some of the areas that software developers need to be aware of when porting their applications to the new processors so that they can utilize the new features.

First, let us look at how the Cortex-M23 and Cortex-M33 processors are related to the previous generation of Cortex-M processors. The previous Cortex-M processors are based on the ARMv6-M and ARMv7-M architectures:

- The ARMv6-M architecture supports a small instruction set and system features that are optimized for ultra-low power designs. The Cortex-M0 and Cortex-M0+ processors are both based on the ARMv6-M architecture and have a tiny gate count starting from just 12K gates. These processors are suitable for most general data processing and I/O control tasks and are upward-compatible with processors based on ARMv7-M.
- The ARMv7-M architecture supports a much richer instruction set, with optional instructions for digital signal processing (DSP) and floating point support. The Cortex-M3, Cortex-M4 and Cortex-M7 processors are in turn upward-compatible, with each processor utilizing further instructions within the ARMv7-M instruction set. These processors are best for applications that demand more complex data processing and for systems with higher performance requirements.

The ARMv8-M architecture is divided into two sub-profiles and their partitioning is similar to the separation between ARMv6-M and ARMv7-M. The "Baseline" sub-profile is optimized for ultra-low power designs and the "Mainline" sub-profile is targeting at more powerful systems where energy efficiency is still crucial.

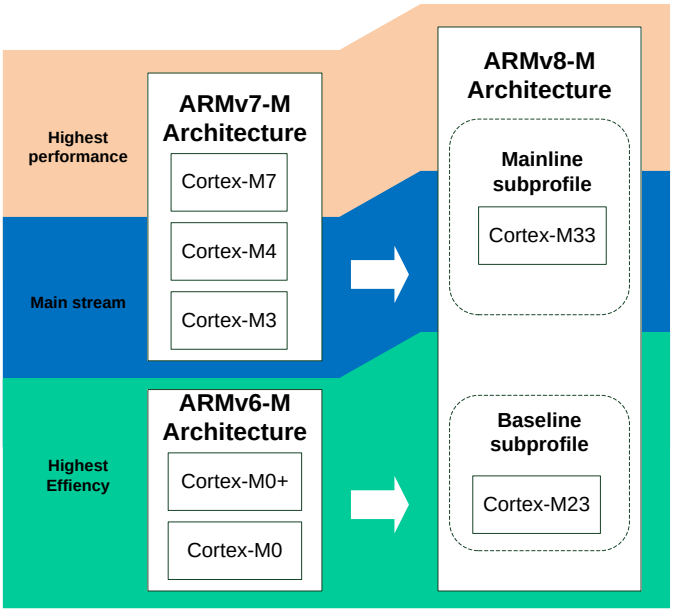

Fig. 1. Evolution of processor architecture for Cortex-M processors

The Cortex-M23 processor addresses the requirements for a range of constrained applications. It is based on ARMv8-M Baseline and supports a superset of features compared to the Cortex-M0+ processor. It has similar energy profile to the Cortex-M0+ processor.

The Cortex-M33 processor addresses the requirements for more capable applications. It is based on ARMv8-M Mainline and focuses on mainstream microcontrollers/SoC designs. It has slightly higher performance then existing Cortex-M3 and Cortex-M4 processors and has better energy efficiency in similar configurations. It also supports a range of features previously not available on Cortex-M3 and Cortex-M4 designs.

One of the key enhancements of the ARMv8-M architecture is the introduction of a security extension called TrustZone®. This technology partitions the execution environments into two security states and adds an extra level of security barrier to the more secure state (the trusted zone). This arrangement helps defend the system from a range of attacks. A range of details about how TrustZone protects embedded systems were covered in Embedded World 2016 paper "The Next Steps in the Evolution of Embedded Processors for the Smart Connected Era" (https://community.arm.com/docs/DOC-11532). There is no need to repeat these details here and this article will only cover some of the aspects of software development.

Both Cortex-M23 and Cortex-M33 processors are highly configurable. Silicon designers can select the features needed in the chip. For example, the number of interrupts available in the Nested Vectored Interrupt Controller (NVIC) is configurable and some system resources, such as the Memory Protection Unit (MPU), are optional. For the ARM Cortex-M33 processor, some subsets of the instruction set are also optional including:

- DSP extension (SIMD, saturation arithmetic)
- Single precision floating point unit (FPU)

As a result, the Cortex-M33 processor can be used in a similar configuration to a Cortex-M3 (no DSP extension, no FPU), a Cortex-M4, or indeed any other combination.

II. SOFTWARE CHANGES

A. *Overview*

The Cortex-M23 and Cortex-M33 are both 32-bit processors and use the same architecturally defined memory map concept as previous generations of Cortex-M processors. In addition, the use of a Nested Vectored Interrupt Controller (NVIC) for interrupt control and architectural defined sleep modes remain unchanged. So in most cases porting of software from previous designs to the new processors will be relatively straight forward. Of course, at the system level the memory map and peripheral programmer's model could change.

TrustZone technology is part of the ARMv8-M architecture. In a TrustZone-enabled system, the software is divided into Secure software and Non-secure software.

- Secure software executes from Secure address space and can access both Secure and Non-secure addresses. Typically, Secure software contains components that are critical to the security of the devices such as Secure boot, device provisioning, crypto libraries, firmware update, etc. The processor is in Secure state when executing Secure software.
- Non-secure software executes from Non-secure address spaces, and can access Non-secure addresses only. Most application code is likely to be executed in Non-secure state. The processor is in Non-secure state when executing Non-secure software.

The partition of the address space is handled by the processor using a combination of a programmable unit called the Security Attribution Unit (SAU) and a device specific logic block called the Implementation Defined Attribution Unit (IDAU). The memory space configuration can only be controlled by Secure software.

Secure software and Non-secure software are compiled separately as separated program images. However:

- Secure and Non-secure software can interact with each other using direct function calls;
- Exception handling sequences can switch the processor states automatically.

Since the TrustZone security extension is optional, some devices based on Cortex-M23 and Cortex-M33 processors may not have TrustZone support. If the TrustZone security extension is omitted, the processor is always in Non-secure state.

B. Non-secure software development

For most software developers, their applications will run in Non-secure state. The application scenarios include:

1. devices without TrustZone;
2. devices with TrustZone, but whose TrustZone features are not used;
3. devices with TrustZone and with software access to APIs in the Secure side.

For these cases, the operations of the processor in Non-secure state are mostly similar to the operations as in previous Cortex-M processors without TrustZone. However, there are some areas that software developers need to be aware of due to architecture enhancements.

1. The programmer's model of the MPU has changed.
2. The EXC_RETURN value (a special value used to trigger exception returns) is extended. This can have impact on an RTOS.
3. The vector table initial location is not necessary at address 0x00000000 (unlike previous Cortex-M0/M0+/M3/M4 processors where the initial location is fixed at 0x00000000).
4. For users of Cortex-M33 devices, when compared to Cortex-M3/M4:
    a. Unlike Cortex-M3 and Cortex-M4, there is no bit band feature on the Cortex-M33;
    b. The VECTRESET bit in SCB->AIRCR (Application Interrupt and Reset Control Register) has been removed. Software should use System Reset Request (SYSRESETREQ) bit to generate self-reset.
5. For users of Cortex-M23 devices, C/C++ code can be recompiled to make the most of the additional instructions previously not available in Cortex-M0/Cortex-M0+.

In the third scenario outlined above (where Non-secure software calls APIs from the Secure side), the software developer should receive header files that include the function prototypes and an export library (or other form of information) so that the linker can correctly generate the function calls to Secure software. In a C/C++ context, those APIs are just normal "extern" functions, so there is no need for special knowledge of how TrustZone works in order to utilize the features provided by the Secure APIs.

C. Secure software development

For devices with TrustZone, and when the software being developed is on the Secure side, typically the software developers will have to develop both Secure and Non-secure program images at the same time so that they can test the interactions between the two. Many software development environments support multi-project work space to make this easier, and the process is likely to include a number of steps. For example:

1. The Secure project is compiled and linked, and an export library generated for the Secure APIs;
2. The Non-secure project is then compiled, with the export library imported into the project, and used during linking stage;
3. Both Secure and Non-secure images are then loaded into the test platform (e.g. microcontroller board) for testing.

All of the architecture enhancements that affect Non-secure software (as listed in previous section) also apply to Secure software development. The programmer's model for Secure software project is mostly the same as Non-secure environment, with the following differences:

- Some of the resources, such as SysTick timer and Memory Protection Unit (MPU), are banked and Secure software can access the Non-secure MPU, SysTick and System Control Block registers via a Non-secure alias address.
- Secure software can configure which resources are available to the Non-secure software. For example, it can program the Security Attribution Unit (SAU) and any system level protection controllers which partition the system resources (such as memories and peripherals).
- Secure software running on Cortex-M23 can utilize stack limit registers for stack overflow detection, whereas Non-secure domain does not have stack limit registers. The Cortex-M33 processor supports stack limit registers for both Secure and Non-secure domains.
- Secure software can decide which of the processor's internal resources can be controlled by Non-secure software. For example:
    - Secure software can program the NVIC_ITNS[n] register to define which interrupts are Secure and which are Non-secure.
    - For some devices based on Cortex-M23 with just one SysTick timer, the Secure software can decide if the SysTick timer is for Secure usage or Non-secure usage.
    - Secure software can decide if the deep sleep and system reset request features are accessible from the Non-secure world.
    - Secure software can decide if HardFault, BusFault and NMI exceptions should be handled by Secure or Non-secure software.
- Secure software developers need to utilize a range of C language extension features defined in the

ARM C Language Extension (ACLE) to enable correct interactions between Secure and Non-secure software. This is covered later in section VI.B.

- Secure APIs that can be called by Non-secure software need to be written and reviewed carefully to avoid vulnerabilities. More information on this topic will be covered in section VI.

D. *Updates to development tools*

In addition to updates in software code, the enhancements in the architecture also require updates in software development tools.

Compiler tool chains – A number of C/C++ tool chains have already been updated to support the ARMv8-M architecture. The updates include:

- Support of the new instructions as defined in ARMv8-M architecture (ARMv8-M Baseline contains a number of additional instructions compared to ARMv6-M). In addition, new instructions are added to both ARMv8-M Mainline and Baseline sub-profiles to support TrustZone and C11 features.
- Support of ACLE features – to develop Secure software, software developers need to add macros, attributes and intrinsic functions defined in ACLE for TrustZone software development.

Debug tools - The debug tools (often part of a development suite) also need to be updated for ARMv8-M support. The changes in the debuggers include:

- Support for new registers in the processors, such as the stack limit registers, debug authentication support registers, etc.
- Support for the new programmer's model in a number of debug components.
- Enhanced instruction trace support for Cortex-M23 and Cortex-M33 (both of these processors can support MTB or ETM instruction trace).
- TrustZone also entails debug authentication support requirements for debug tools.

The JTAG and Serial Wire debug protocol remains unchanged, hence in most cases the debug probe/adaptor hardware can be reused with ARMv8-M processors. A firmware update to the debug probes may however be needed.

III. TRUSTZONE EXAMPLE AND DEBUG AUTHENTICATION

TrustZone technology can be utilized in a number of ways. In microcontrollers designed for a range of IoT applications, the chip manufacturers could design microcontroller chips such that:

- Secure boot, crypto libraries and related resources are placed in Secure domain and are protected by TrustZone;
- High value firmware, such as a communication protocol stack, can be added to Secure memory and be protected;
- Microcontroller software developers can access security features using validated APIs but cannot access secure hardware resources directly and cannot reverse engineer high-value protected firmware;
- Additional chip level protection features can be available to prevent firmware in a completed product being read out.

The basic system start-up operations can be illustrated in the following diagram:

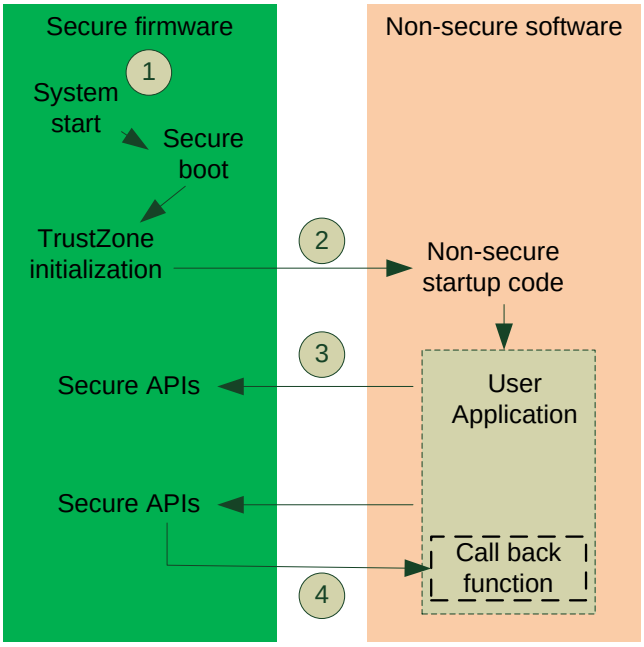

Fig. 2. Example software flow in a TrustZone-enabled Cortex-M based microcontroller

1. The system boots up in Secure state, optionally carrying out security checking in Secure boot, before performing various initialization tasks such as C startup and SAU programming.
2. The Secure firmware branches to the entry point of the Non-secure application. The Non-secure software has its own reset handler, C startup code and hardware initialization (e.g. peripherals). This does not conflict with initialization from the Secure code as the stack and heap spaces of Secure and Non-secure code are separated.
3. During execution of Non-secure applications, the application could call Secure APIs.

4. In some cases, Secure APIs might need to call Non-secure call-back functions (e.g. a hardware driver).

The debug access permissions need to be changed at different stages of the product development and this is managed by the debug authentication feature of the processor. For example, when the microcontroller chip is purchased by end product developers, the Secure program memory may already be programmed and locked down and the MCU software developer will not be able to use debug tools to see what is inside the Secure code or access Secure peripherals (including Secure storages, crypto accelerators) directly.

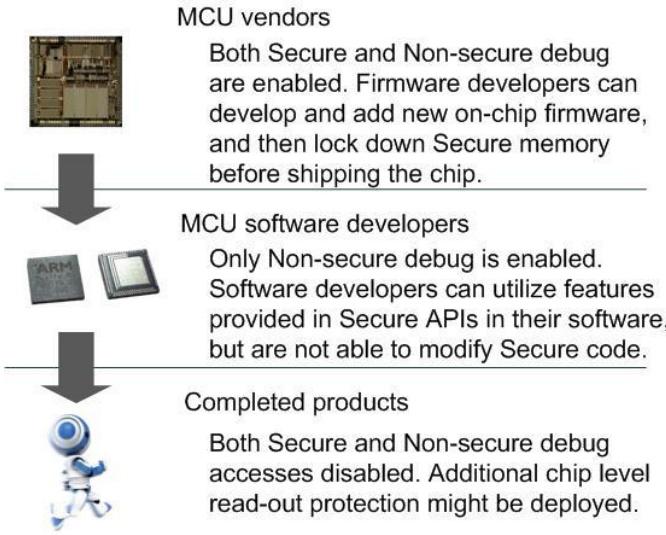

Fig. 3. Example debug authentication handling in product life cycle

The debug authentication at the processor level is controlled by a number of configuration signals enabling chip manufacturers to decide how they want to control the debug authentication by adding specific control hardware.

As a result of the debug authentication feature, debug tools supporting ARMv8-M need to support different scenarios. For example, when only Non-secure debug is enabled:

- The processor cannot be halted while executing Secure code. If a halt request is made during this period, the halt request is pended and will stop the processor when it switches to the Non-secure state.
- The debug software cannot single step into Secure code. If a software developer uses single stepping when debugging a piece of Non-secure code, the single stepping will be blocked if the code calls a Secure API. The debugger will halt at the SG instruction at the beginning of the Secure API and, when another step is made, it will continue to execute to the completion of the Secure API and halt when returning to Non-secure code.
- Tracing operations (e.g. instruction trace via ETM/MTB) is disabled during Secure code operation to prevent leaking of Secure information.

In some cases, MCU vendors can provide additional protection features such as eXecute Only Memory (XOM). XOM is a firmware protection technique in which a portion of the program memory space only supports program execution and the contents of the memory cannot be read back by a debugger or by any software. Please refer to the MCU vendor's documentation to see the range of protection features available.

IV. Making the most of the ARMv8-M Processors

A. Overview

Since there are a number of features in Cortex-M23/M33 that are not available in previous Cortex-M0/M0+/M3/M4 processors, software developers will need to modify their code to fully utilize the features.

B. Instruction set

The instruction set supported by Cortex-M23 is much richer when compared with Cortex-M0/M0+. As a result:

- Application code should be recompiled to make use of the additional instructions, potentially enabling better performance and smaller code size.
- The RTOS can be updated to use exclusive accesses for semaphore operations. Unlike SVC-based semaphore operations, using exclusive access avoids any impact to interrupt latency. In addition the RTOS can use the same semaphore code for both ARMv8-M Mainline and Baseline processors.

Applications migrated from Cortex-M3/M4 to Cortex-M33 should experience a performance uplift due to the more efficient pipeline design in Cortex-M33. However, some compilers may be further optimized for the pipeline characteristics of Cortex-M33 and recompiling the software with these newer compilers may bring additional performance gains.

C. Stack Limit feature

As an additional security measure it is highly recommended that Secure software utilizes the stack limit feature to detect stack overflows. Without this, a stack overflow in Secure code could lead to a leak of Secure information and could potentially introduce a vulnerability allowing Non-secure users to compromise Secure software (e.g. by modifying the return stack).

For Non-secure software, the stack limit feature is available on Cortex-M33 (ARMv8-M Mainline) but is not available on Cortex-M23 (ARMv8-M Baseline). If you are using Cortex-M23, stack overflow detection could be implemented by:

- using the MPU (e.g. some RTOS utilize MPU for stack overflow detection);
- using an RTOS which implements stack range checking during a context switch;

- using the Non-secure Process Stack Pointer (PSP_NS) for main applications and placing the Process Stack at the beginning of SRAM space - if a stack overflow occurs in the thread code, invalid SRAM space would be accessed and this would trigger a HardFault exception.

D. *New Memory Protection Unit (MPU) features*

The MPU is a programmable hardware unit that can be used to define memory access permissions and memory attributes. Typically, in an RTOS environment, application tasks are executed at an unprivileged access level, and some RTOS are able to configure the MPU so that application tasks can only access memory spaces allocated to them. By having this protection, the tasks cannot corrupt data used by the OS or other tasks, hence enhancing the reliability of the system.

In an RTOS environment, the MPU is reconfigured at each context switch so that different application tasks can access different parts of the memory system.

The MPU also defines memory attributes, for example, defining which memory spaces are cacheable.

The ARMv8-M architecture defines a new programmer's model for the MPU that enables much more flexible region definitions. Instead of restricting the MPU region size to a power of two and requiring the MPU region start address to be a multiple of the region size, the new design specifies the MPU region using a starting address and an ending address with a granularity of 32 bytes.

The MPU configuration options have also been enhanced. The MPU in Cortex-M0+/M3/M4 can support up to 8 regions, and in Cortex-M7, 8 or 16 regions. In Cortex-M23 and Cortex-M33, there can be two MPUs (one for Secure code and one for Non-secure code) and each of them can have either 4, 8, 12 or 16 regions independently.

The MPU in ARMv8-M has some differences in memory attribute definitions compared to ARMv6-M or ARMv7-M. In the earlier architectures, memory is divided into: normal memory (e.g. ROM, RAM); devices memory; and strongly ordered memory. In the ARMv8-M architecture, only normal and device memory types are available, with the device memory type further sub-divided into 4 classes based on three properties:

- E/nE: Early response allowed or disallowed
- R/nR: Reordering of accesses allowed or disallowed
- G/nG: Gathering of accesses (e.g. merging) allowed or disallowed

The four classes of Devices are therefore:

- Device-nGnRnE: equivalent to strongly ordered memory in earlier architectures
- Device-nGnRE: equivalent to bufferable devices in earlier architectures
- Device-nGRE
- Device-GRE

More details on the programming model of the MPU can be found in the document "Memory Protection Unit for ARMv8-M based platforms": (https://developer.arm.com/products/architecture/m-profile/docs/100699/latest/introduction)

E. *RTOS*

As outlined, there are a number of features in ARMv-8M which enhance the operation of an RTOS. RTOS software developers should therefore read "RTOS design considerations for ARMv8-M based platforms" which is available for reference.

F. *Coprocessor*

The Cortex-M33 processor has a coprocessor interface that supports up to 8 coprocessors. Chip designers can utilize the coprocessor interface to add hardware accelerators which can speed up a range of specialized operations which might otherwise need a longer sequence of generic instructions to perform.

Each of the coprocessors can have a number of hardware registers. Conceptually these registers are just like peripheral registers, but in a separate address space (not part of the 4GB memory address space). The coprocessor interface provides a high speed mechanism to transfer data between the processor's register bank and the coprocessor registers – it can read/write 32-bit or 64-bit in a single cycle. Since the transfer is not affected by other bus traffic on the main bus interface and does not need additional instructions to set up a register with an address, it can deliver higher performance than using memory mapped hardware accelerators.

To use the coprocessor feature:

1. Enable the coprocessor – this can be controlled by the CPACR (Coprocessor Access Control Register);
2. Access the coprocessor using coprocessor instructions (e.g. inline assembly) or via intrinsics defined in ACLE.

The range of coprocessor instructions includes those in table below (table I). Each data transfer instruction can also include an operand to define the coprocessor operations required if any. The exact definition of the coprocessor number, operands and coprocessor registers are device-specific.

TABLE I. COPROCESSOR INSTRUCTIONS SUPPORTED IN THE CORTEX-M33 PROCESSOR

Instructions	Function
MCR <cp>, #<opc1>, <Rt>, <CRn>,<CRm> {, #<opc2>} MCR2 <cp>, #<opc1>, <Rt>, <CRn>,<CRm> {, #<opc2>}	Move 32-bit data from register to coprocessor register
MCRR <cp>, #<opc1>, <Rt>, <Rt2>, <CRm> MCRR2 <cp>, #<opc1>, <Rt>, <Rt2>,<CRm>	Move 64-bit data from register to coprocessor register
MRC <cp>, #<opc1>, <Rt>, <CRn>,<CRm> {, #<opc2>} MRC2 <cp>, #<opc1>, <Rt>, <CRn>,<CRm> {, #<opc2>}	Move 32-bit data from coprocessor register to register

MRRC <cp>, #<opc1>, <Rt>, <Rt2>, <CRm> MRRC2 <cp>, #<opc1>, <Rt>, <Rt2>,<CRm>	Move 64-bit data from coprocessor register to register
CDP <cp>, #<opc1>, <CRd>, <CRn>,<CRm> {, #<opc2>} CDP2 <cp>, #<opc1>, <CRd>, <CRn>,<CRm> {, #<opc2>}	Coprocessor data processing – tell a coprocessor to performance an operation

A range of intrinsic functions are defined in ACLE to make accessing of these instructions easier in C/C++ programming environment (table II):

TABLE II. INTRINSIC FUNCTIONS DEFINED IN ACLE FOR COPROCESSOR ACCESSES.

Instructions	Intrinsic Function
MCR MCR2	void __arm_mcr(coproc, opc1, uint32_t value, CRn, CRm, opc2) void __arm_mcr2(coproc, opc1, uint32_t value, CRn, CRm, opc2)
MCRR MCRR2	void __arm_mcrr(coproc, opc1, uint64_t value, CRm) void __arm_mcrr2(coproc, opc1, uint64_t value, CRm)
MRC MRC2	uint32_t __arm_mrc(coproc, opc1, CRn, CRm, opc2) uint32_t __arm_mrc2(coproc, opc1, CRn, CRm, opc2)
MRRC MRRC2	uint64_t __arm_mrrc(coproc, opc1, CRm) uint64_t __arm_mrrc2(coproc, opc1, CRm)
CDP CDP2	void __arm_cdp(coproc, opc1, CRd, CRn, CRm, opc2) void __arm_cdp2(coproc, opc1, CRd, CRn, CRm, opc2)

The coprocessor interface is TrustZone-aware. Secure software can define each coprocessor as Secure or Non-secure by programming the Non-secure Access Control Register (NSACR) register. If a coprocessor is configured as Secure, an access to the coprocessor from Non-secure software triggers a fault exception. In addition, the interface has a side band signal to indicate the security attribute, which can enable finer grain control of which coprocessor registers are Secure or Non-secure.

V. FAULT HANDLING

Fault exceptions have existed for a long time in Cortex-M processors based on ARMv6-M and ARMv7-M. The ARMv8-M architecture further enhances the fault exceptions available. In ARMv8-M, if the security extension is implemented, BusFault and HardFault exceptions target the Secure domain by default. It is possible to change the behavior by setting a bit called BFHFNMINS (BusFault HardFault NMI Non-secure) in the AIRCR (Application Interrupt Reset Control Register) This is normally only used if the Secure side software is not being used by the application.

In the Cortex-M23 processor, if a security violation occurs, the HardFault exception is triggered. For the Cortex-M33 processor, a new configurable fault exception called SecureFault (exception type 7) is introduced. The SecureFault exception is disabled by default, and can be enabled by setting bit 19 (SECUREFAULTENA) of the System Handler Control and State Register (SHCSR).

If the security extension is implemented, the fault exception handlers might need to be updated if the stacked PC needs to be extracted from the stack frame. For software running in the Non-secure state, it cannot locate the stack frame in the Secure stack, and hence needs to check the new S bit in EXC_RETURN:

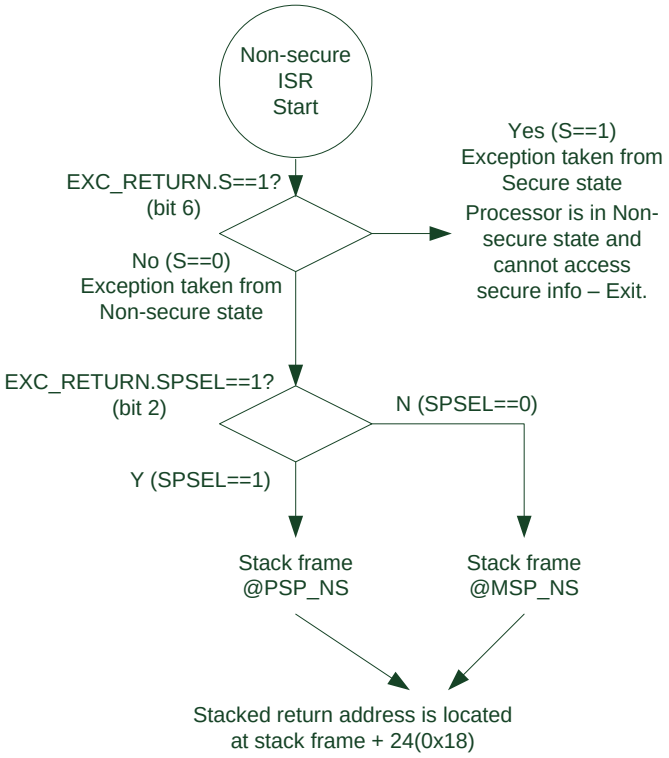

Fig. 4. Example software flow for fault handlers in Non-secure software

If the exception handler is running as Secure software, then it needs to do more work to identify the correct stack pointer and to detect if the stack frame contains the extended section:

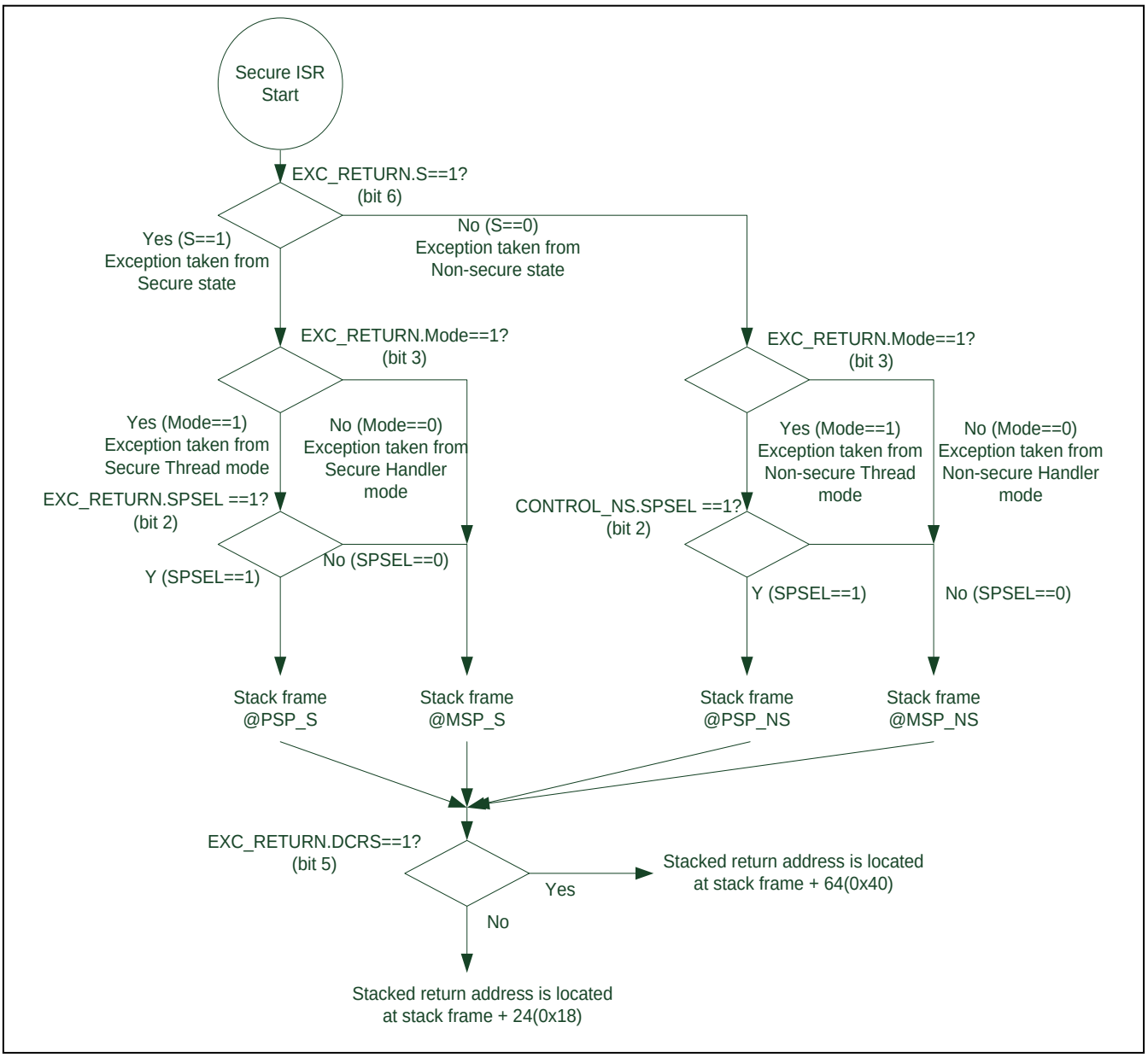

Fig. 5. Example software flow for fault handlers in Secure software

The Cortex-M33 processor also has a new Secure Fault Status Register (SFSR) to allow software or a debugger to detect the possible cause of a security violation.

VI. SECURE SOFTWARE DEVELOPMENT

For the majority of software developers using off-the-shelf microcontrollers, software application development will be focused on the Non-secure side. However, a range of software developers, including firmware engineers in silicon companies, OEMs or middleware vendors, may need to develop software that runs on the Secure side. In this section of the paper we will introduce some of the topics that these developers will find useful.

A. C compiler support for Secure software development

To support development of Secure software on Cortex-M23 and Cortex-M33, the compiler needs to support a range of ACLE features for the Cortex-M Security Extension (CMSE). The ACLE feature defines:

- A number of function attributes to allow functions to be declared as Secure entry points and to allow function pointers to be declared as Non-secure;
- A number of intrinsic functions to provide pointer checking and processor state accesses;
- A number of C macros to support various CMSE features.

The ACLE specification is open and is supported by multiple compiler vendors, helping to ensure that application code is portable across vendors. However it should be noted that a small number of CMSE support features are optional (e.g. parameter passing using the stack in cross security domain function calls) and hence support should be checked when porting between vendor solutions.

When compiling application code for Secure state:

- The command line option specifying that the compilation is for Secure state should be used (e.g. "-mcmse" in the ARM Compiler);
- The C/C++ code needs to include a specific CMSE header file (#include "arm_cmse.h");
- The linker command line option or linker script need to specify where to place the branch veneers for Secure entry points;
- At the linking stage, a command line option may need to specify the name of an export library (the library file will be used by Non-secure software developers to link their program images).

If you are using a CMSIS-compliant device driver for Secure software development, you might notice there is an extra "partition_<device>.h" file. This file contains an initialization function "TZ_SAU_Setup()" for programming the SAU and system level controllers (if any) for memory partitioning. This function also defines which interrupts are Secure/Non-secure and defines if certain other resources should be accessible from Non-secure state. This function is called by the SystemInit() function as a part of the reset handler.

B. Creating cross domain function calls

CMSE defines two new important function attributes:

TABLE III. FUNCTION ATTRIBUTES DEFINED IN CMSE

Function attribute	Usage
__attribute__((cmse_nonsecure_entry))	To indicate that the function can be called from Non-secure software.
__attribute__((cmse_nonsecure_call))	To indicate that a function/function pointer is in Non-secure

To create a Secure API that is callable from Non-secure side, the "cmse_nonsecure_entry" attribute should be added and this will cause the tool chain to automatically generate the required branch veneers with SG (Secure Gateway) instructions that indicate valid entry points. Also, the compiler automatically generates code to ensure that registers which may contain secure data are cleared or restored back to the original Non-secure values from before the API call.

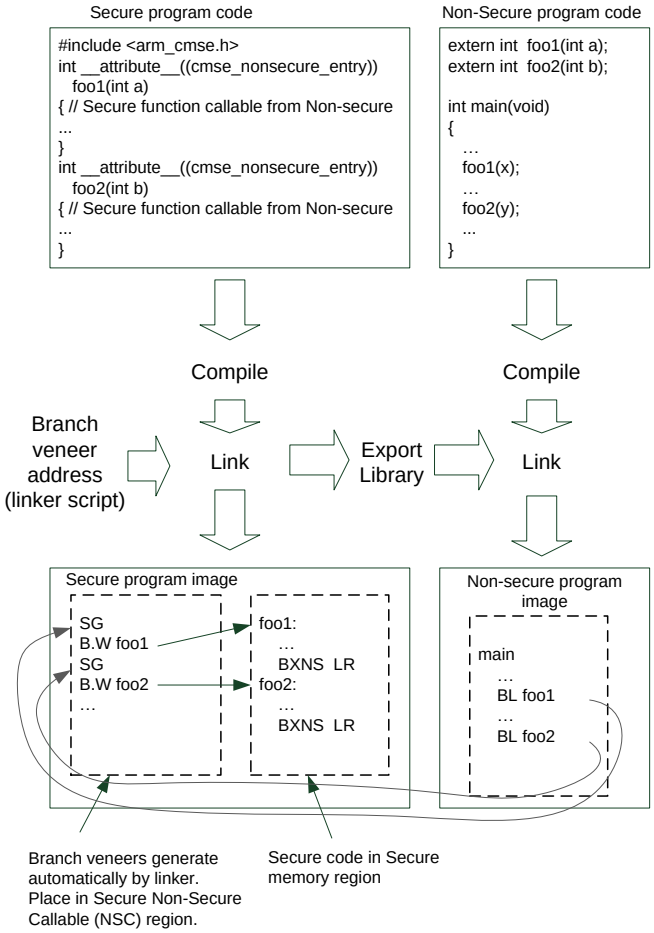

Fig. 6. Example software development flow for testing Secure software

For Secure software to call a Non-secure API, for example, when Secure middleware needs to call a hardware driver function in the Non-secure side, the situation is a little bit more involved. This is because quite often the Secure software was developed before the Non-secure software and does not know the address of the Non-secure APIs during compilation. As a result, in many cases Non-secure code needs to call a Secure API to pass over function pointers before the call back function can be made.

For example, the process is demonstrated in the following code:

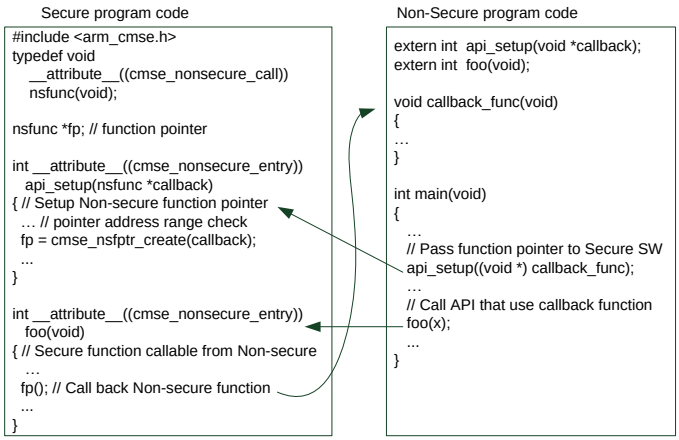

Fig. 7. Example software interaction when Secure software calls Non-secure functions

When calling Non-secure functions, C compilers will automatically generate code to make sure all Secure data (except function parameters) are saved and cleared from the register bank before the call is made. As a result, secure data is not exposed to the Non-secure world.

VII. SECURITY CONSIDERATIONS

Although the ARMv8-M architecture and CMSE features provide a strong foundation for security, Secure software still needs to be written carefully to avoid vulnerabilities. For example:

- Although Secure software can access Non-secure RAM space, Secure software should use only Secure memory for general data (e.g. stacks, heaps), except when passing information between Secure and Non-secure software.
- As in most applications, the validation of input parameters is crucial for many Secure APIs.
- Input parameters for secure APIs might need to be copied to Secure space before validation and processing. This is because data in Non-secure memory space could be modified by Non-secure exceptions asynchronously. More information on this covered in an ARM document called "Secure software guidelines for ARMv8-M based platforms".
- Secure software can utilize intrinsic functions provided by CMSE to validate pointers. There are two groups of intrinsic functions as described below.

A. *TT intrinsics that execute the TT instruction with an input pointer:*

TABLE IV. INTRINSIC FUNCTIONS FOR TT INSTRUCTIONS

Intrinsic	Semantics
cmse_address_info_t cmse_TT(void *p)	Generates a TT instruction.
cmse_address_info_t cmse_TT_fptr(p)	Generates a TT instruction. The argument p can be any function pointer type.
cmse_address_info_t cmse_TTT(void *p)	Generates a TT instruction with the T flag.
cmse_address_info_t cmse_TTT_fptr(p)	Generates a TT instruction with the T flag. The argument p can be any function pointer type.
cmse_address_info_t cmse_TTA(void *p)	Generates a TT instruction with the A flag.
cmse_address_info_t cmse_TTA_fptr(p)	Generates a TT instruction with the A flag. The argument p can be any function pointer type.
cmse_address_info_t cmse_TTAT(void *p)	Generates a TT instruction with the A and T flag.
cmse_address_info_t cmse_TTAT_fptr(p)	Generates a TT instruction with the A and T flag. The argument p can be any function pointer type.

B. *Pointer checking intrinsics*

There are two CMSE-defined address range check intrinsics.

TABLE V. INTRINSIC FUNCTIONS FOR ADDRESS (POINTER) CHECKINGS

Intrinsic	Semantics
*cmse_check_address_range(void *p, size_t size, int flags)	Checks the address range from p to p + size – 1 is accessible. The flags defined the required access permission.
*cmse_check_pointed_object(void *p, int flags)	Returns the same value as cmse_check_address_range(p, sizeof(*p), f).

General security considerations:

- Secure APIs should check if the function call is from Non-secure unprivileged code. If so, the pointer check function calls should specify unprivileged checking.
- Secure software should utilize the stack limit check feature to detect stack overflow errors.
- Only the memory space that contains Secure entry points (branch veneers) should be declared as Non-secure Callable (NSC). This prevents other binary data in the secure code image inadvertently becoming secure gateways.
- Any unused space in NSC regions should be initialized or zero-filled (or filled with other values that do not match the SG instruction) to avoid inadvertent secure gateways.
- SRAM initial contents are unpredictable, so if an SRAM is to be setup as a NSC region, it must be initialized first.
- Since the passing of parameters across security boundaries on the stack might not be supported in some compilation tools, it is best to restrict the number of parameters in a function to a small number to ensure better portability.

- Some systems allow memory resources or peripherals to be switched between Secure and Non-secure domains at runtime. Such switching must be handled carefully to avoid leakage of Secure information and reduce the risk of code injection attacks. When a peripheral's security domain is switched, don't forget to make sure that its interrupt security domain is updated at the same time.
- Please note that dynamic switching of security domains in memory/peripheral resources could lead to unexpected behavior for debug tools if the debug tool tries to access those address locations.
- If Secure software utilizes the floating point unit (FPU), the TS bit in Floating-Point Context Control Register (FPCCR) should be set to 1.
- Typically, the BFHFNMINS (BusFault HardFault NMI Non-secure) bit in the AIRCR (Application Interrupt Reset Control Register) should be 0 (default) unless Secure software is not being used.
- Privileged level configurations and Secure stack limits (if used) should be set up before Secure software enables the execution of any Non-secure software (including setting BFHFNMINS). This is to prevent Non-secure software (including Non-secure exception handlers) attempting to call Secure APIs before the Secure world is ready.
- If the Secure world is not used and BFHFNMINS is set to 1, Secure software developers should consider disabling all NSC regions to prevent Non-secure world software from calling Secure APIs.

VIII. Unavailable features

A few features in Cortex-M3 and Cortex-M4 processors are not available in Cortex-M33. They include:

- Bit band – this is removed because the bit band operation requires remapping of the address space. Since the TrustZone security operation is based on address space partitioning, the address remapping nature of bit band can cause complications. Some MCU vendors have added bit level handling features in their peripherals (e.g. separate bit set & bit clear registers) to handle this instead.
- Flash patch – the break point unit in Cortex-M3 and Cortex-M4 processors provide a ROM patching mechanism. When using this feature patch addresses are remapped from the CODE region to the SRAM region and can cause complications to TrustZone. This feature is therefore removed. The new breakpoint unit design in ARMv8-M provides superior breakpoint capabilities – covering all address ranges and allowing all eight comparators to be used for instruction address checks.
- VECTRESET bit in AIRCR – The VECTRESET bit allows debuggers to reset just the processor and not the rest of the system. In ARMv8-M systems, the debugger and software should use SYSRESETREQ (System Reset Request) instead.

IX. Summary

Using TrustZone technology, the Cortex-M23 and Cortex-M33 processors bring efficient, optimized security to microcontroller class devices, providing for the protection of critical security data and resources, and hence enabling the next generation of IoT devices to be more secure. To support these security features, software may need to be updated to make the most of the new processor features, and secure software needs to be written carefully to avoid vulnerabilities.

TrustZone for ARMv8-M makes writing secure software easier, increasing productivity when creating secure solutions. To make it easier for secure software developers, the ARM C Language Extensions (ACLE) has been updated to include Cortex-M Security Extension (CMSE) features to enable software developers to access security features in portable C/C++ code. These features enable efficient interaction between Secure and Non-secure software components while ensuring secure handling of information.

To support ARMv8-M, many compilers have been updated to include optimizations for Cortex-M23 and Cortex-M33 processors, and to include CMSE functionality. To fully utilize the new capabilities of the Cortex-M23 and Cortex-M33 processors, existing code should be recompiled. Debug software has also been updated to support debug authentication requirements. A number of RTOSes have also been updated to take advantage of the new features in the architecture.

To enable greater processing acceleration, the Cortex-M33 processor supports a coprocessor interface enabling Cortex-M33 microcontroller vendors to include their hardware in their SoCs to accelerate certain specialized processing tasks.

References

[1] Joseph Yiu, ARM, "The Next Steps in the Evolution of Embedded Processors for the Smart Connected Era", Embedded World 2016

[2] Joseph Yiu "ARM, "Whitepaper – ARMv8-M Architecture Technical Overview", ARM TechCon 2015.

[3] ARM, "ARMv8-M Architecture Reference Manual"

[4] ARM, "Memory Protection Unit for ARMv8-M based platforms"

[5] ARM, "RTOS design considerations for ARMv8-M based platforms"

[6] ARM, "Secure software guidelines for ARMv8-M based platforms"

[7] ARM, "ARMv8-M Security Extension: Requirements on Development Tools"