Secure software guidelines

Version 1.0

for ARM®v8-M based platforms

arm

100720_0100_0100_en

Secure software guidelines

for ARM®v8-M based platforms

Release Information

Document History

Issue	Date	Confidentiality	Change
0100	23 August 2016	Non-Confidential	First release

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

Secure software guidelines for ARM®v8-M based platforms

Preface

This preface introduces the *Secure software guidelines for ARM®v8-M based platforms*.

It contains the following:

About this book

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

Using this book

This book is organized into the following chapters:

Chapter 1 Secure Software Guidelines

You must meet several requirements when creating Secure software for an ARM®v8-M based platform. These include requirements to generate special instructions (`BXNS` and `BLXNS`) to branch to Non-secure code and the requirement to preserve and protect Secure register values before calling Secure functions. CMSE is an extension to the C language that can be implemented by tool vendors to provide a standard way to generate this code.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

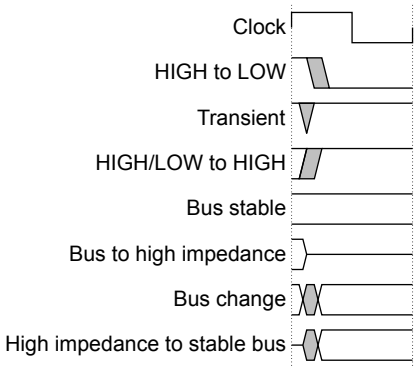


Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name denotes an active-LOW signal.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Secure software guidelines for ARMv8-M based platforms*.
- The number 100720_0100_0100_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Secure Software Guidelines

You must meet several requirements when creating Secure software for an ARM®v8-M based platform. These include requirements to generate special instructions (`BXNS` and `BLXNS`) to branch to Non-secure code and the requirement to preserve and protect Secure register values before calling Secure functions. CMSE is an extension to the C language that can be implemented by tool vendors to provide a standard way to generate this code.

It contains the following sections:

1.1 ARM®v8-M Security Extension

The Security Extension is an optional part of the ARMv8-M architecture. It defines a system-wide division of physical memory into Secure regions and Non-secure regions, and two system-wide security states that are enforced by hardware.

The architecture supports the creation of a Trusted software stack that provides features such as Secure remote firmware updates, while significantly reducing the attack surface of such code. This is an important feature for any network-connected device that can be updated after deployment, including any IoT device.

The ARMv8-M architecture does not support the A32 instruction set.

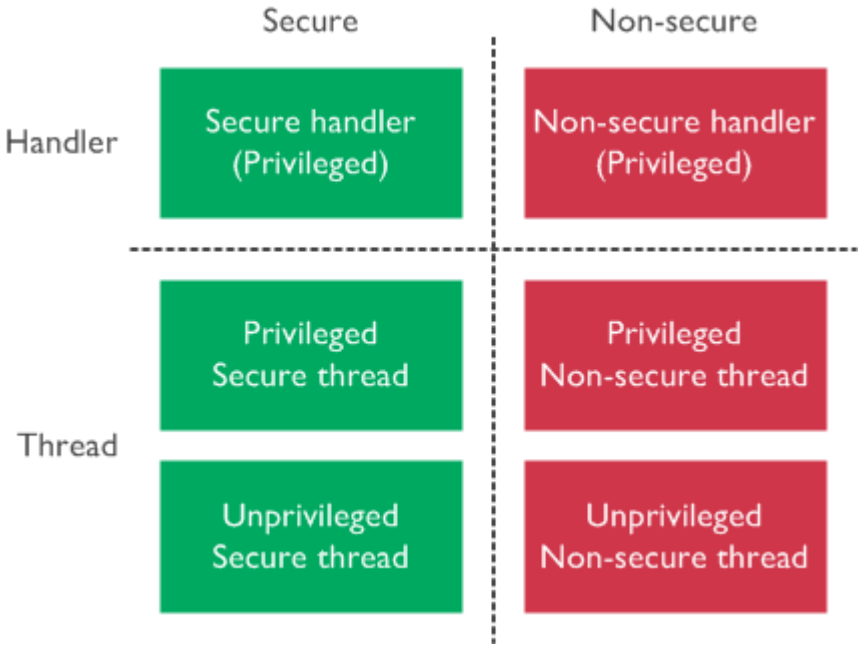
The ARMv8-M architecture is designed to combine Secure and Non-secure software. It gives vendors the ability to protect their software assets, both code and data, by restricting access to the memory where their software assets reside, except for a set of explicitly exported entry points that are defined by the vendor.

There is a direct relation between the memory regions and the security states:

- 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.
- 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.

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 SecureFault in the ARMv8-M architecture with Main Extension, or a HardFault in the ARMv8-M architecture.

The security states are orthogonal to the processor mode, as the following figure shows:



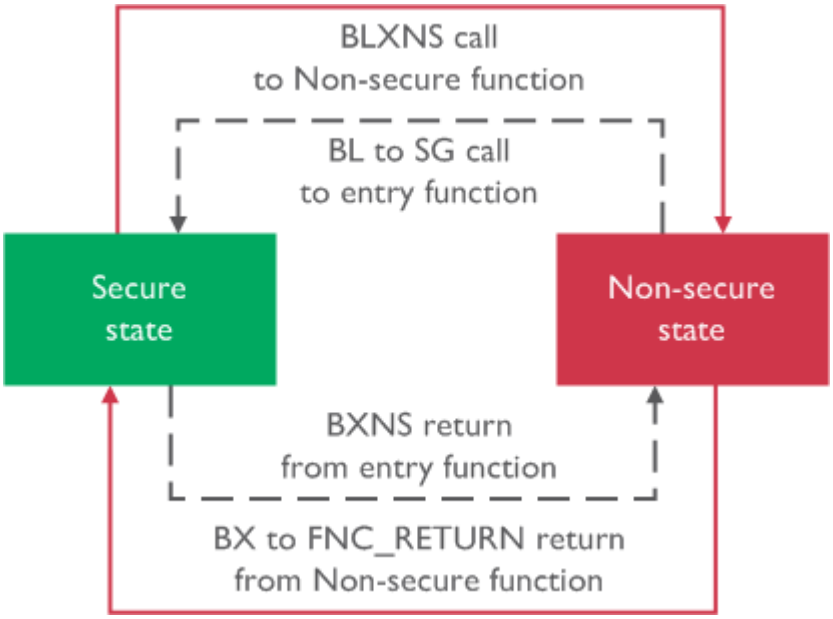
Memory regions can be defined by the system through the *Implementation Defined Attribution Unit* (IDAU) or can be controlled in software through the memory mapped *Secure Attribution Unit* (SAU) registers.

Parts of the system are banked between the security states. The Stack Pointer is banked, resulting in a Stack Pointer for each combination of security state and mode. All parts of the system accessible in Non-secure state can be accessed in Secure state as well, including the banked parts.

1.2 Security state changes using CMSE

Transitions from Secure to Non-secure state can be initiated by software by using either a `BXNS` or `BLXNS` instruction that has the *Least Significant Bit* (LSB) of the target address unset. This enables the LSB of an address to denote the security state.

The system boots in Secure state and can change security states using branches, as the following figure shows:



Note

- Transitions from Non-secure to Secure state can be initiated by software in two ways:
  - A branch to a Secure gateway.
  - A branch to the reserved value `FNC_RETURN`.

A Secure gateway is an occurrence of the Secure Gateway instruction (`SG`) in a special type of Secure region, named a *Non-secure Callable* (NSC) region. When branching to a Secure Gateway from Non-secure state, the `SG` instruction switches to the Secure state and clears the LSB of the return address in the LR. In any other situation, the `SG` instruction does not change the security state or modify the return address. The `SG` instruction must be fetched from NSC memory.

A branch to the reserved value `FNC_RETURN` causes the hardware to switch to Secure state, read an address from the top of the Secure stack, and branch to that address. The reserved value `FNC_RETURN` is written to the LR when executing the `BLXNS` instruction.

Security state transitions can be caused by hardware through the handling of interrupts. Those transitions are transparent to software.

This section contains the following subsections:

- *1.2.1 Secure code requirements* on page 1-11.
- *1.2.2 Development tools* on page 1-13.

1.2.1 Secure code requirements

To prevent Secure code and data from being accessed from Non-secure state, Secure code must meet several requirements. The responsibility for meeting these security requirements is shared between hardware, toolchain, and software developer.

Information leakage

Information leakage from the Secure state to the Non-secure state can occur through parts of the system that are not banked between the security states.

The unbanked registers that are accessible by software are:

- General purpose registers except for the Stack Pointer (R0-R12, R14-R15).
- Floating-point registers (S0-S31, D0-D15).
- The N, Z, C, V, Q, and GE bits of the xPSR register.
- The FPSCR register.

Secure code must clear secret information from unbanked registers before initiating a transition from Secure to Non-secure state.

Non-secure memory access

When Secure code has to access Non-secure memory using an address that is calculated by the Non-secure state, it cannot trust that the address lies in a Non-secure memory region. Furthermore, the *Memory Processing Unit* (MPU) is banked between the security states. Secure and Non-secure code might have different access rights to Non-secure memory.

Secure code that accesses Non-secure memory on behalf of the Non-secure state must only do so if the Non-secure state has permission to perform the same access itself. The Secure code can use the `TT` instruction to check Non-secure memory permissions.

Take care when using Secure code to access Non-secure memory unless it does so on behalf of the Non-secure state. Data belonging to Secure code must reside in Secure memory.

Volatility of Non-secure memory

Non-secure memory can be changed asynchronously to the execution of Secure code.

There are two possible causes:

- Interrupts that are handled in Non-secure state can change Non-secure memory.
- The debug interface can be used to change Non-secure memory.

There can be unexpected consequences when Secure code accesses Non-secure memory. For example:

```
int array[N]


void foo(int *p) {
    if (*p >= 0 && *p < N) {
        // Non-secure memory (*p) is changed at this point
        array[*p] = 0;
    }
}
```

Secure code must treat Non-secure memory as volatile memory.

When the pointer `p` points to Non-secure memory, it is possible for its value to change after the memory accesses used to perform the array bounds check, but before the memory access used to index the array. Such an asynchronous change to Non-secure memory would render this array bounds check useless.

You can handle this as follows:

```
int array[N]


void foo(volatile int *p) {
    int i = *p;
    if (i >= 0 && i < N) {
        array[i] = 0;
    }
}
```

Inadvertent Secure Gateway

An SG instruction can occur inadvertently. If an inadvertent SG instruction occurs in an NSC region, the result is an inadvertent Secure Gateway.

An inadvertent SG instruction can occur in the following cases:

- Uninitialized memory.
- General data in executable memory, for example jump tables.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its first halfword that follows an SG instruction, for example two successive SG instructions.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its last halfword that is followed by an SG instruction, for example an SG instruction that follows an LDR (immediate) instruction.

If an inadvertent SG instruction occurs in an NSC region, the result is an inadvertent Secure Gateway.

Memory in an NSC region must not contain an inadvertent SG instruction.

The Secure Gateway veneers limit the instructions that must be placed in NSC regions. If the NSC regions contain only these veneers, an inadvertent Secure Gateway cannot occur.

1.2.2 Development tools

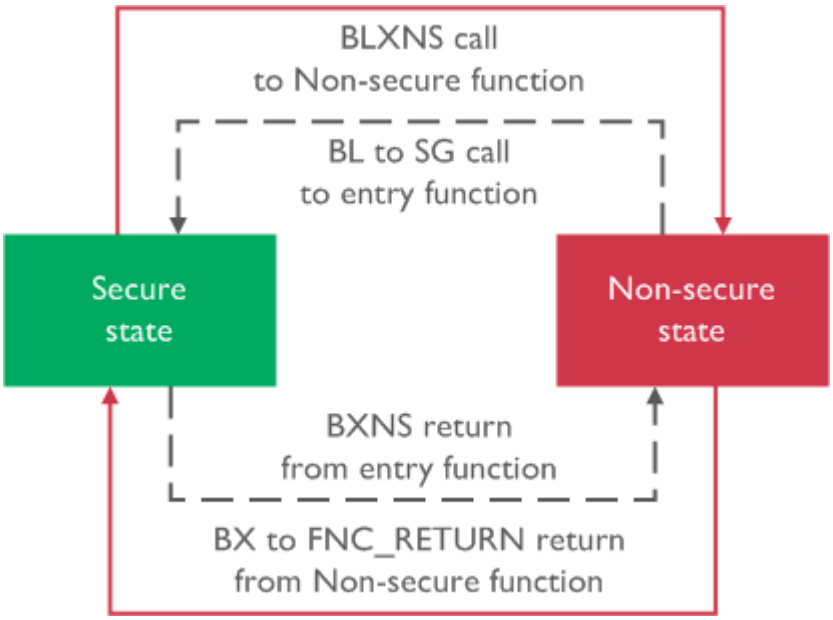
Development tools are expected to provide C and assembly language support for interacting between the security states. Code that is written in C++ must use the `extern C` linkage for any inter-state interaction.

Security state changes must be expressed through function calls and returns.

This use of the `extern C` linkage provides an interface that fits naturally with the C language.

A function in Secure code that can be called from the Non-secure state through its Secure gateway is called an entry function. A function call from Secure state to the Non-secure state is called a Non-secure function call.

The following figure shoes security state transitions:



Executable files

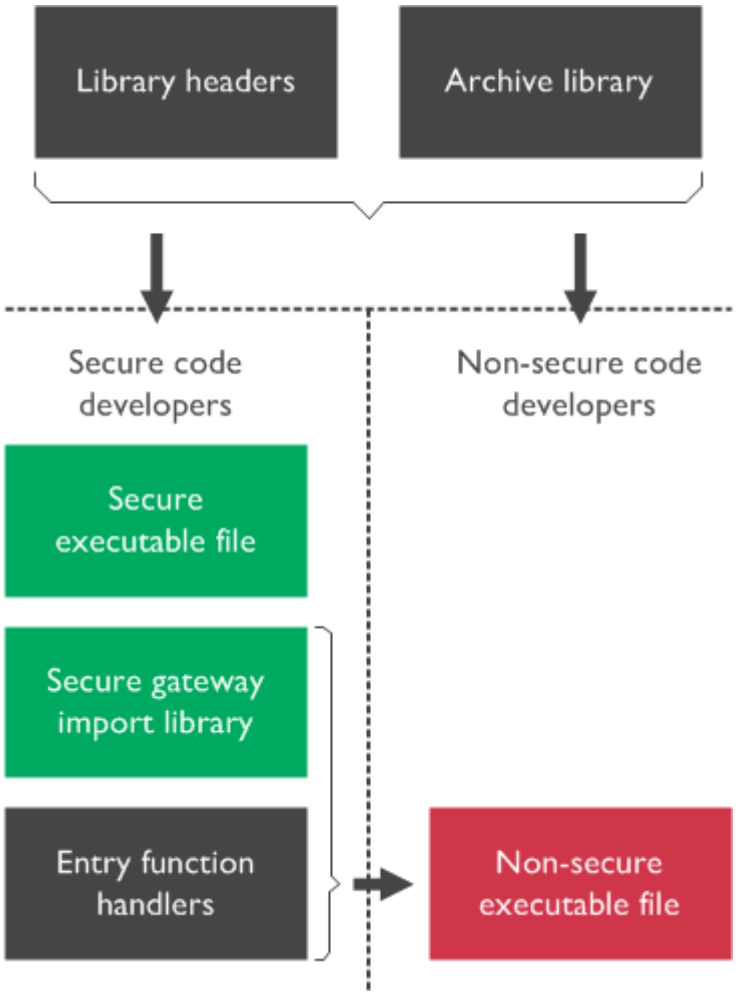
There are two different types of executable files, one for each security state. The Secure state executes Secure code from a Secure executable file. The Non-secure state executes Non-secure code from a Non-secure executable file. The Secure and Non-secure executable files are developed independently of each other.

A Non-secure executable is unaware of security states.

From the point of view of the Non-secure state, a call to a Secure gateway is a regular function call, as is the return from a Non-secure function call. You can develop Non-secure code with a toolchain that is not CMSE aware, that is, you do not require new tool features when you are only building Non-secure code.

Developing a Secure executable file requires toolchain support whenever a function is called from, calls, or returns to Non-secure state and whenever memory is accessed through an address that is provided by the Non-secure state. The Secure code ABI is otherwise identical to the Non-secure code ABI.

The following figure shows the interaction between developers of Secure code, Non-secure code, and (optional) security agnostic library code:



The Secure gateway import library contains the addresses of the Secure gateways of the Secure code. This import library consists of or contains a relocatable file that defines symbols for all the Secure gateways. The Non-secure code links against this import library to use the functionality that is provided by the Secure code.

A relocatable file containing only copies of the (absolute) symbols of the Secure gateways in the Secure executable must be available to link Non-secure code against.

Linking against this import library is the only requirement on the toolchain that is used to develop the Non-secure code. This functionality is similar to calling ROM functions, and is expected to be available in existing toolchains.

Secure gateway veneers

A toolchain must support generating a Secure gateway veneer for each entry function with external linkage. It consists of an `SG` instruction followed by a `B.W` instruction that targets the entry function it veneers.

Secure gateway veneers decouple the addresses of Secure gateways (in NSC regions) from the rest of the Secure code. By maintaining a vector of Secure gateway veneers at a forever-fixed address, the rest of the Secure code can be updated independently of Non-secure code. This also limits the amount of code in NSC regions that potentially can be called by the Non-secure state.

Vectors of Secure gateway veneers are expected to be placed in NSC memory. All other code in the Secure executable is expected to be placed in Secure memory regions. This placement is under the control of the developer.

Preventing inadvertent Secure gateways is a responsibility that is shared between a developer and their toolchain. A toolchain must make it possible for a developer to avoid creating inadvertent Secure gateways.

Excluding the first instruction of a Secure gateway veneer, a veneer must not contain the bit pattern of the SG instruction on a 2-byte boundary.

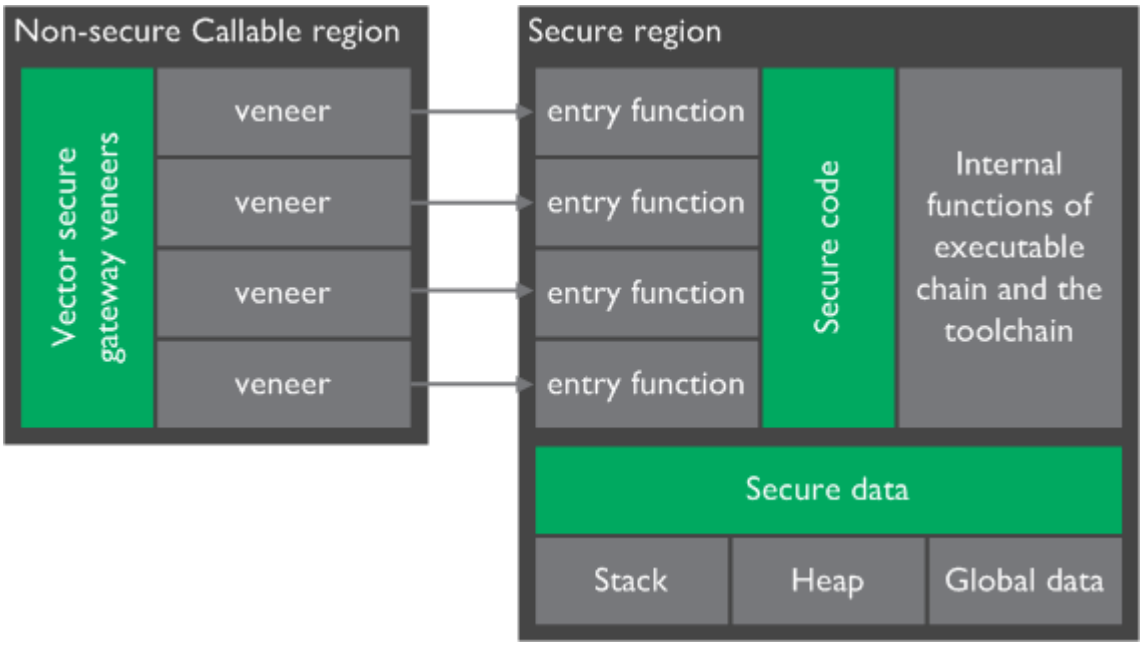
A vector of Secure gateway veneers must be aligned to a 32-byte boundary, and must be zero padded to a 32-byte boundary.

The developer must take care that the code or data before the vector of Secure gateway veneers does not create an inadvertent Secure gateway with the first Secure gateway veneer in the vector. ARM recommends placing the vector of Secure gateway veneers at the start of an NSC region.

The position of Secure gateway veneers in a vector must be controllable by the developer.

This last requirement gives the developer complete control over the address of a Secure gateway veneer. It allows the developer to fix the addresses of the Secure gateway veneers so that Secure code can be updated independently of Non-secure code.

The following figure shows the memory layout of a Secure executable:



1.3 Test Target instruction

To allow software to determine the security attribute of a memory location, the *Test Target* (`TT`) instruction is used.

`TT` queries the security state and access permissions of a memory location.

Test Target Unprivileged (`TTT`) queries the security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (`TTA`) and *Test Target Alternate Domain Unprivileged* (`TTAT`) query the security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

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 whether 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 intrinsics which make this functionality available to the developer.

The `<arm_cmse.h>` header must be included before using the `TT` intrinsics.

This section contains the following subsections:

- *1.3.1 TT intrinsics* on page 1-16.
- *1.3.2 Address range check intrinsic* on page 1-17.

1.3.1 TT intrinsics

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.

Table 1-1 TT intrinsics

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.

Note

ARM recommends that a toolchain behaves as if these intrinsics would write the pointed-to memory. This prevents subsequent accesses to this memory being scheduled before this intrinsic.

The exact type signatures for `cmse_TT_fptr()` and `cmse_TTT_fptr()` are implementation-defined because there is no type that is defined by the C propgramming language that can hold all function pointers.

Note

ARM recommends implementing these intrinsics as macros.

1.3.2 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 defined in this section can be used to perform permission checks on C objects.

Some *Secure Attribution Unit* (SAU), *Implementation Defined Attribution Unit* (IDAU), and *Memory Protection Unit* (MPU) configurations block the efficient implementation of an address range check. This 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.

Regions are aligned at 32-byte boundaries. If the address range fits in one 32-byte address line, a single `TT` instruction suffices.

ARM recommends that software developers use the returned pointer to access the checked memory range. This generates a data dependency between the checked memory and all its subsequent accesses and prevents these accesses from being scheduled before the check.

1.4 CMSE Support

The `<arm_cmse.h>` header defines the language extension that provides support for Secure executable files that are written in the C language. Non-secure executable files do not require any additional toolchain support.

The `<arm_cmse.h>` header must be included before using CMSE support, except for using the `__ARM_FEATURE_CMSE` macro.

Bits 0 and 1 of feature macro `__ARM_FEATURE_CMSE` are set if CMSE support for Secure executable files is available.

The availability of CMSE implies availability of the `TT` instruction.

A compiler might provide a switch to enable support for creating CMSE Secure executable files. ARM recommends such a switch to be named `-mcmse`.

This section contains the following subsections:

- *1.4.1 Non-secure memory usage* on page 1-18.
- *1.4.2 Non-secure function call* on page 1-20.

1.4.1 Non-secure memory usage

Secure code must only use Secure memory except when communicating with the Non-secure state. The security implications of accessing Non-secure memory through a pointer are the responsibility of the developer.

Arguments and return value

A caller from the Non-secure state is not aware it is calling an entry function. If it must use the stack to write arguments or read a result value that uses the Non-secure stack.

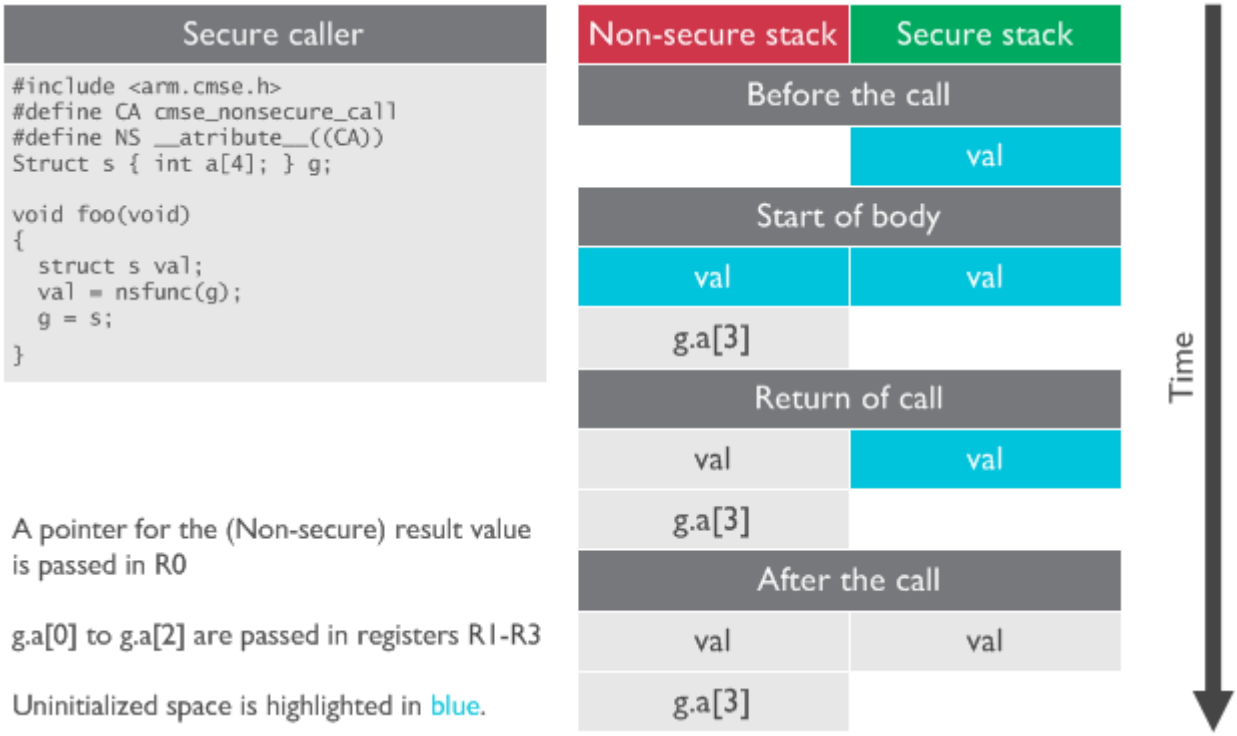
If a toolchain supports stack-based arguments, it must be aware of the volatile behavior of Non-secure memory and the requirements of using Non-secure memory.

In practice, a compiler might generate code that:

- Copies stack-based arguments from the Non-secure stack to the parameter on the Secure stack in the prologue of the entry function.
- Copies the stack-based return value from the Secure stack to the Non-secure stack in the epilogue.

A possible optimization would be to access the Non-secure stack directly for arguments that read at most once, but accessibility checks are still required.

The following figure shows the stack use of an entry function:



Return from an entry function

An entry function must use the `BXNS` instruction to return to its Non-secure caller.

This instruction switches to Non-secure state if the target address has its LSB unset. The LSB of the return address in the LR is automatically cleared by the `SG` instruction when it switches the state from Non-secure to Secure.

Note

To prevent information leakage when an entry function returns, the registers that contain secret information must be cleared.

The code sequence directly preceding the `BXNS` instruction that transitions to Non-secure code must:

- Clear all caller-saved registers except:
  - Registers that hold the result value and the return address of the entry function.
  - Registers that do not contain secret information.
- Clear all registers and flags that have UNDEFINED values at the return of a procedure, according to the *Procedure Call Standard for the ARM Architecture* (AAPCS).
- Restore all callee-saved registers as required by the AAPCS.

Floating-point registers can be cleared conditionally by checking the SFPA bit of the special-purpose CONTROL register.

A toolchain could provide the developer with the means to specify that some types of variables never hold secret information, for example by setting the TS bit of FPCCR. The Security Extension assumes that floating-point registers never hold secret information.

Because of these requirements, performing tail-calls from an entry function is difficult.

Security state of the caller

An entry function can be called from Secure or Non-secure state. Software must distinguish between these cases.

To enable this the Security Extension defines an intrinsic:

Table 1-2 Security state of the caller intrinsic

Intrinsic	Semantics
`int cmse_nonsecure_caller(void)`	Returns non-zero if entry function is called from Non-secure state and zero otherwise.

1.4.2 Non-secure function call

A call to a function that switches state from Secure to Non-secure is called a Non-secure function call. A Non-secure function call must use function pointers. This is a consequence of separating Secure and Non-secure code into separate executable files.

A Non-secure function type must be declared using the function attribute `__attribute__((cmse_nonsecure_call))`.

A Non-secure function type must only be used as a base type of a pointer. This restriction disallows function definitions with this attribute and ensures that a Secure executable file only contains Secure function definitions.

Performing a call

A function call through a pointer with a Non-secure function type as its base type must switch to the Non-secure state. To create a function call that switches to the Non-secure state, an implementation must emit code that clears the LSB of the function address and branches using the `BLXNS` instruction.

Note

A Non-secure function call to an entry function is possible. This call to an entry function behaves like any other Non-secure function call.

All registers that contain secret information must be cleared to prevent information leakage when performing a Non-secure function call. Registers that contain values that are used after the Non-secure function call must be restored after the call returns. Secure code cannot depend on the Non-secure state to restore these registers.

The code sequence directly preceding the `BLXNS` instruction that transitions to Non-secure code must:

- Save all callee- and live caller-saved registers by copying them to Secure memory.
- Clear all callee- and caller-saved registers except:
  - The LR.
  - The registers that hold the arguments of the call.
  - Registers that do not hold secret information.
- Clear all registers and flags that have UNDEFINED values at the entry to a procedure according to the AAPCS.

A toolchain could provide the developer with the means to specify that some types of variables never hold secret information.

When the Non-secure function call returns, caller and callee that are saved registers that are saved before the call must be restored.

An implementation need does not have to save and restore a register if its value is not live across the call. However, callee-saved registers are live across the call in almost all situations. These requirements specify behavior that is similar to a regular function call, except that:

- Callee-saved registers must be saved as if they are caller-saved registers.
- Registers that are not banked and potentially contain secret information must be cleared.

The floating-point registers can efficiently be saved and cleared using the `VLSTM` instruction, and restored using `VLLDM` instruction.

Arguments and return value

The callee of a Non-secure function call is called in Non-secure state. If stack usage is required according to the AAPCS, the Non-secure state expects the arguments on the Non-secure stack and writes the return value to Non-secure memory.

The stack usage during a Non-secure function call is shown in the following figure.

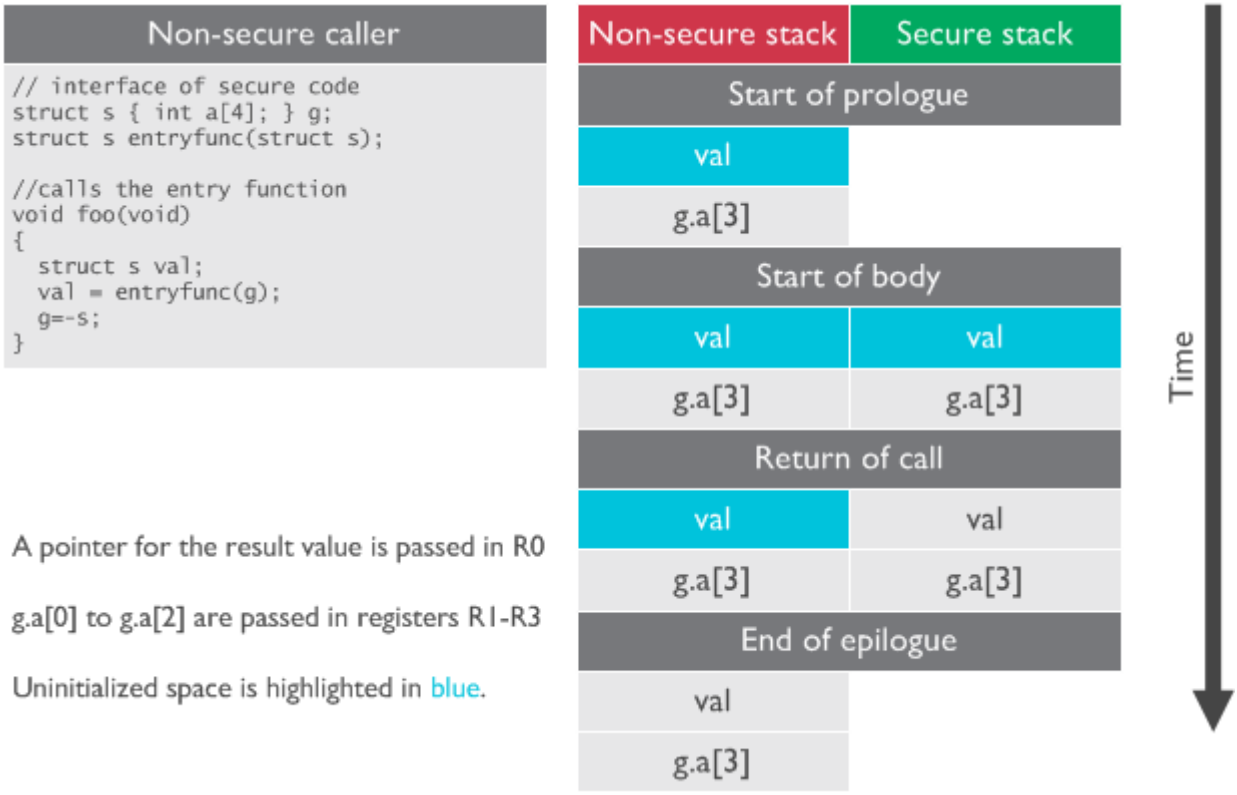


Table 1-3 Non-secure function call intrinsics

Intrinsic	Semantics
`cmse_nsfptr_create(p)`	Returns the value of `p` with its LSB cleared. The argument `p` can be any function pointer type.
`cmse_is_nsfptr(p)`	Returns non-zero if `p` has LSB unset, zero otherwise. The argument `p` can be any function pointer type.

Note

The exact type signatures of these intrinsics are implementation-defined because there is no type defined by the C programming language that can hold all function pointers. ARM recommends implementing these intrinsics as macros and recommends that the return type of `cmse_nsfptr_create()` is identical to the type of its argument.

A Non-secure returning function must be declared by using the attribute `__attribute__((cmse_nonsecure_return))` on a function declaration.

A Non-secure returning function has a special epilogue, identical to that of an entry function.