TrustZone technology for the ARMv8-M architecture

Version 2.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
08 July 2016	0000-00	Non-Confidential	First release
23 August 2016	0101-00	Non-Confidential	Second release
03 March 2017	0200-00	Non-Confidential	Third release

Proprietary Notice

Confidentiality Status

This document is 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

1 ARM TrustZone technology

TrustZone technology for ARMv8-M is an optional Security Extension that is designed to provide a foundation for improved system security in a wide range of embedded applications.

The concept of TrustZone technology is not new. The technology has been available on ARM Cortex-A series processors for several years and has now been extended to cover ARMv8-M processors.

At a high level, the concepts of TrustZone technology for ARMv8-M are similar to the TrustZone technology in ARM Cortex-A processors. In both designs, the processor has Secure and Non-secure states, with Non-secure software able to access to Non-secure memories only. TrustZone technology for ARMv8-M is designed with small energy-efficient systems in mind. Unlike TrustZone technology in Cortex-A processors, the division of Secure and Normal worlds is memory map based and the transitions takes place automatically in exception handling code.

However, there are several differences in the implementation:

- TrustZone technology for ARMv8-M supports multiple Secure function entry points, whereas in TrustZone technology for Cortex-A processors, the Secure Monitor handler is the sole entry point.
- Non-secure interrupts can still be serviced when executing a Secure function.

As such TrustZone technology for ARMv8-M is optimized for low-power microcontroller type applications:

- In many microcontroller applications with real-time processing, deterministic behavior and low interrupt latency are important requirements. The ability to service interrupt requests while running Secure code is critical.
- By allowing the register banks to be shared between Secure and Non-secure states, the power consumption of ARMv8-M implementations can be similar to ARMv6-M or ARMv7-M implementations.
- The low overhead of state switching allows Secure and Non-secure software to interact frequently, which is expected to be common place when Secure firmware contains software libraries such as GUI firmware or communication protocol stacks.

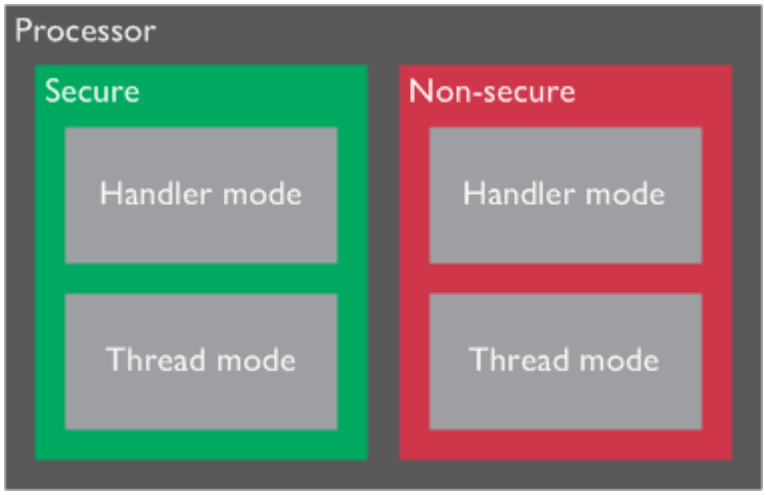
ARM TrustZone technology enables the system and the software to be partitioned into Secure and Normal worlds. Secure software can access both Secure and Non-secure memories and resources, while Normal software can only access Non-secure memories and resources. These security states are orthogonal to the existing Thread and Handler modes, enabling both a Thread and Handler mode in both Secure and Non-secure states.

Note

Thread mode can also be either Privileged or Unprivileged.

If the Security Extension is implemented, the system starts up in Secure state by default. If the Security Extension is not implemented, the system is always in Non-secure state. TrustZone technology enables the processor to be aware of the security states available. ARM TrustZone technology does not cover all aspects of security. For example, it does not include cryptography.

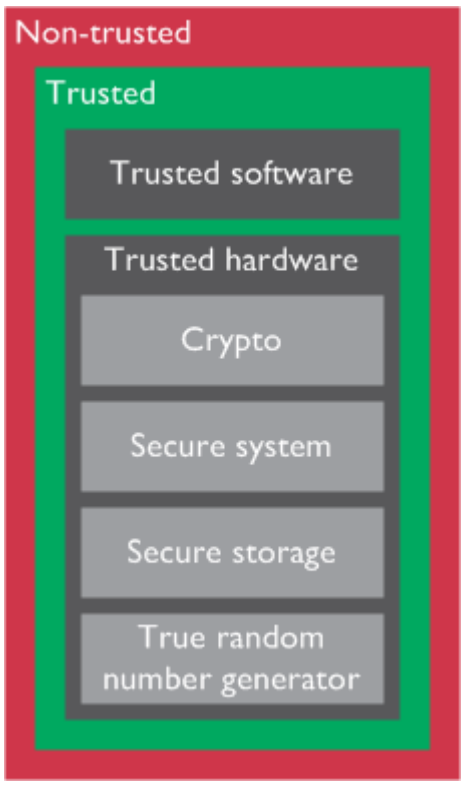
The following figure shows how TrustZone technology for ARMv8-M adds Secure and Non-secure states to processor operation:



In designs with the ARMv8-M architecture Security Extension, components that are critical to the security of the system such can be placed in the Secure world. These critical components include:

- A Secure boot loader.
- Secret keys.
- Flash programming support.
- High value assets.

The remaining applications are placed in the Normal world.



Secure (Trusted) and Non-secure (Non-trusted) software can work together, but Non-secure applications cannot access Secure resources directly. Instead, any access to Secure resources can go through APIs provided by Secure software, and these APIs can implement authentications to decide if the access to the Secure service is permitted. By having this arrangement, even if there are vulnerabilities in the Non-secure applications, hackers cannot compromise the whole chip.

2 Security requirements addressed by TrustZone technology for ARMv8-M

The word security can mean many different things in embedded system designs. In most embedded systems, security can include, but is not limited to:

Communication protection

This protection prevents data transfers from being visible to, or intercepted by unauthorized parties and might include other techniques such as cryptography.

Data protection

This protection prevents unauthorized parties accessing secret data that is stored inside devices.

Firmware protection

This protection prevents on-chip firmware from being reverse engineered.

Operation protection

This protection prevents critical operations from malicious intentional failure.

Tamper protection

In many security sensitive products, anti-tampering features are required to prevent the operation or protection mechanisms of the device from being overridden.

TrustZone technology can address some of the following security requirements of embedded systems directly:

Data protection

Sensitive data can be stored in Secure memory spaces and can only be accessed by Secure software. Non-secure software can only gain access to Secure APIs providing services to the Non-secure domain, and only after security checks or authentication.

Firmware protection

Firmware that is preloaded can be stored in Secure memories to prevent it from being reverse engineered and compromised by malicious attacks. TrustZone technology for ARMv8-M can also work with extra protection techniques. For example, device level read-out protection, a technique that is commonly used in the industry today, can be used with TrustZone technology for ARMv8-M to protect the completed firmware of the final product.

Operation protection

Software for critical operations can be preloaded as Secure firmware and the appropriate peripherals can be configured to permit access from the Secure state only. In this way, the operations can be protected from intrusion from the Non-secure side.

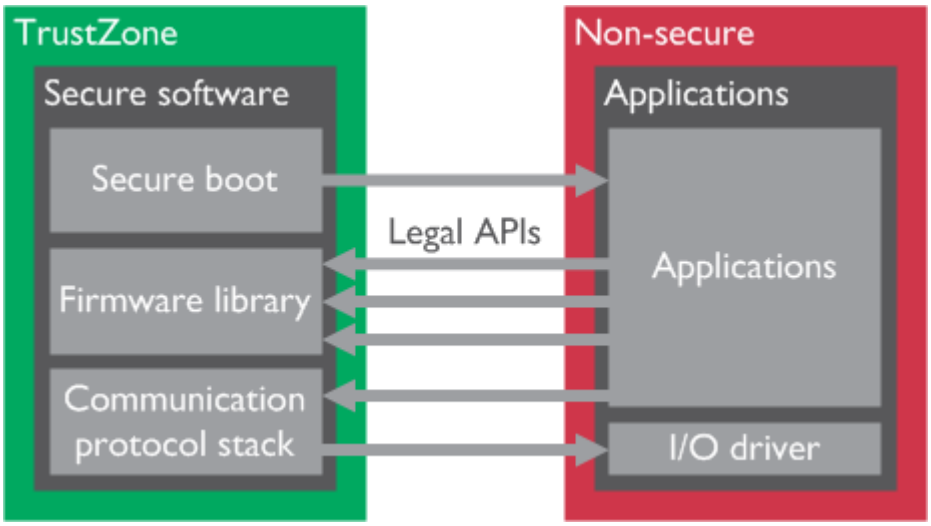
Secure boot

The Secure boot mechanism enables you to have confidence in the platform, as it will always boot from Secure memory.

Since TrustZone technology for ARMv8-M is only a barrier between security domains, some security requirements cannot be addressed by TrustZone technology alone. For example:

- Communication protection still requires cryptography techniques which might be handled by software or assisted by hardware crypto-accelerators, for example, ARM TrustZone Cryptocell products. TrustZone technology can help support such techniques, as certain crypto-software and hardware can be configured to only be accessible within the Secure state.
- Anti-tampering, if necessary in a product, still requires specialized design techniques and product level design arrangements, for example, circuit boards and product enclosures. Whether anti- tampering is applied depends on system requirements and the value of the assets being protected.

Nonetheless, TrustZone technology for ARMv8-M enables a better foundation for system level security. In the simplest example, TrustZone technology for ARMv8-M can be used to protect firmware from being reverse engineered, as the following figure shows:

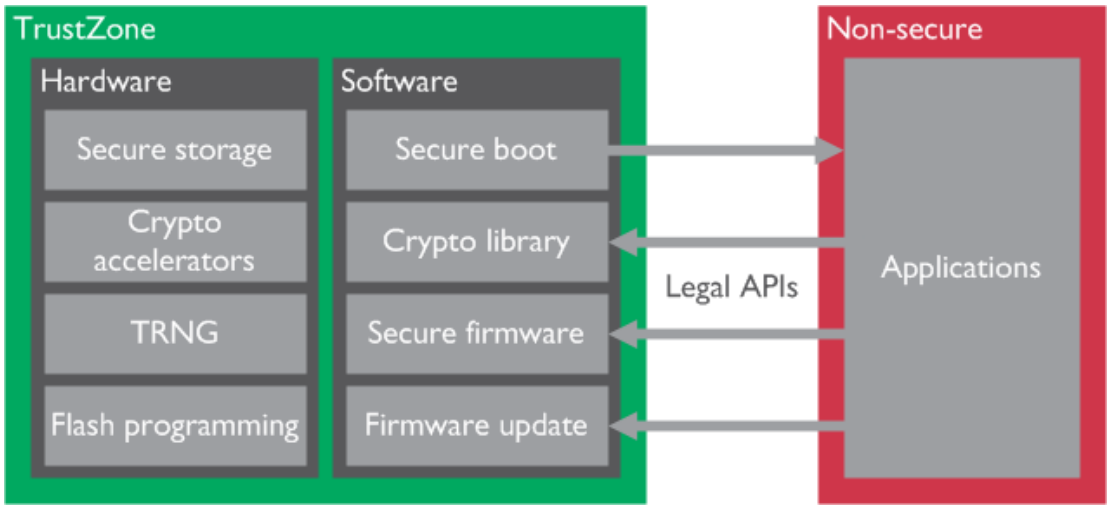


Many microcontrollers already have built-in firmware such as USB or Bluetooth stacks, and TrustZone technology makes the firmware protection implementation easier and more Secure, by ensuring that untrusted software cannot branch to the middle of Secure APIs to bypass any initial checking.

Security for IoT products

TrustZone technology for can also be used with the additional protection features used in advanced microcontrollers targeting the next generation Internet of Things (IoT) products. For example, a microcontroller that is developed for IoT applications can include a range of security features.

The use of TrustZone technology can help ensure that all those features can only be accessed using APIs with valid entry points, as the following figure shows:



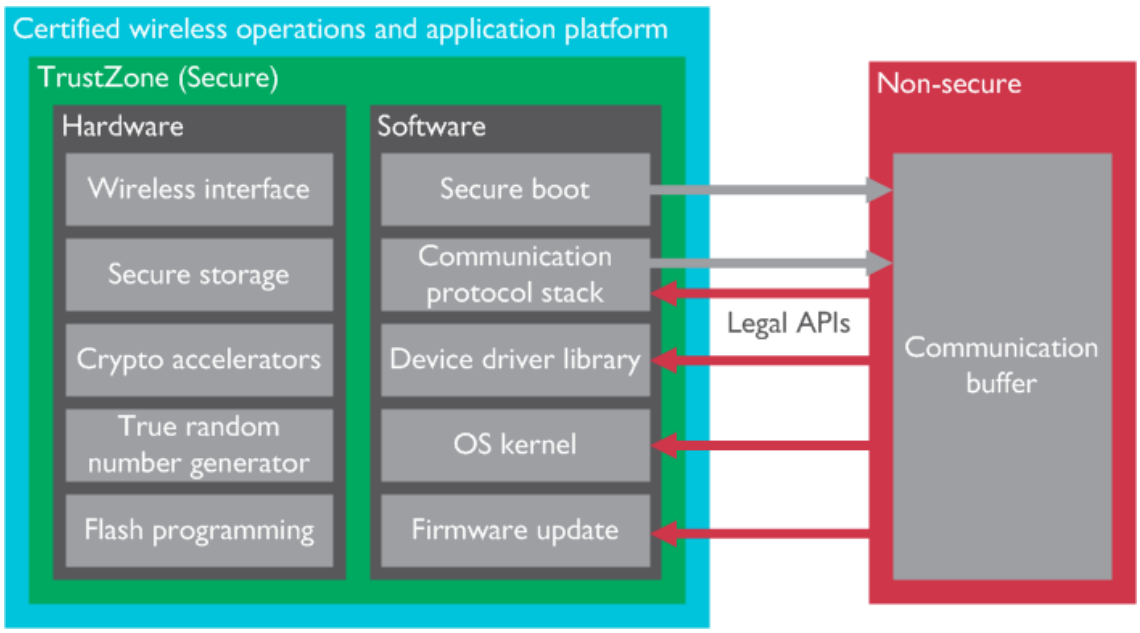
By using TrustZone technology to safe guard these security features, designers can:

- Prevent untrusted applications from directly accessing security critical resources.
- Ensure that a Flash image is reprogramed only after authentication and checking.
- Prevent firmware from being reverse engineered.
- Store secret information with protection at the software level.

Security for wireless communication interface

In some other application scenarios, such as a wireless SoC with a certified built-in radio stack, TrustZone technology can protect the standardized operations, such as wireless communication behavior.

TrustZone technology can ensure that customer defined applications cannot void the certification, as the following figure shows.



3 Memory system and memory partitioning

If the Security Extension is implemented the 4GB memory space is partitioned into Secure and Non-secure memory regions.

The Secure memory space is further divided into two types:

Secure (S)

Secure addresses are used for memory and peripherals that are only accessible by Secure software or Secure masters.

Secure transactions are those that originate from masters operating as, or deemed to be, Secure when targeting a Secure address.

Non-secure Callable (NSC)

NSC is a special type of Secure location. This type of memory is the only type which an ARMv8-M processor permits to hold an SG instruction that enables software to transition from Non-secure to Secure state. The inclusion of NSC memory locations removes the need for Secure software creators to allow for the accidental inclusion of SG instructions, or data sharing encoding values, in normal Secure memory by restricting the functionality of the SG instruction to NSC memory only.

Typically NSC memory regions contain tables of small branch veneers (entry points). To prevent Non-secure applications from branching into invalid entry points, there is the Secure Gateway(SG) instruction.

When a Non-secure program calls a function in the Secure side:

- The first instruction in the API must be an SG instruction.
- The SG instruction must be in an NSC region, which is defined by the Security Attribution Unit(SAU) or Implementation Defined Attribution Unit (IDAU).

The reason for introducing NSC memory is to prevent other binary data, for example, a lookup table, which has a value the same as the opcode as the SG instruction, being used as an entry function in to the Secure state. By separating NSC and Secure memory types, Secure program code containing binary data can be securely placed in a Secure region without direct exposure to the Normal world, and can only be accessed using valid entry points in NSC memory.

Non-secure (NS)

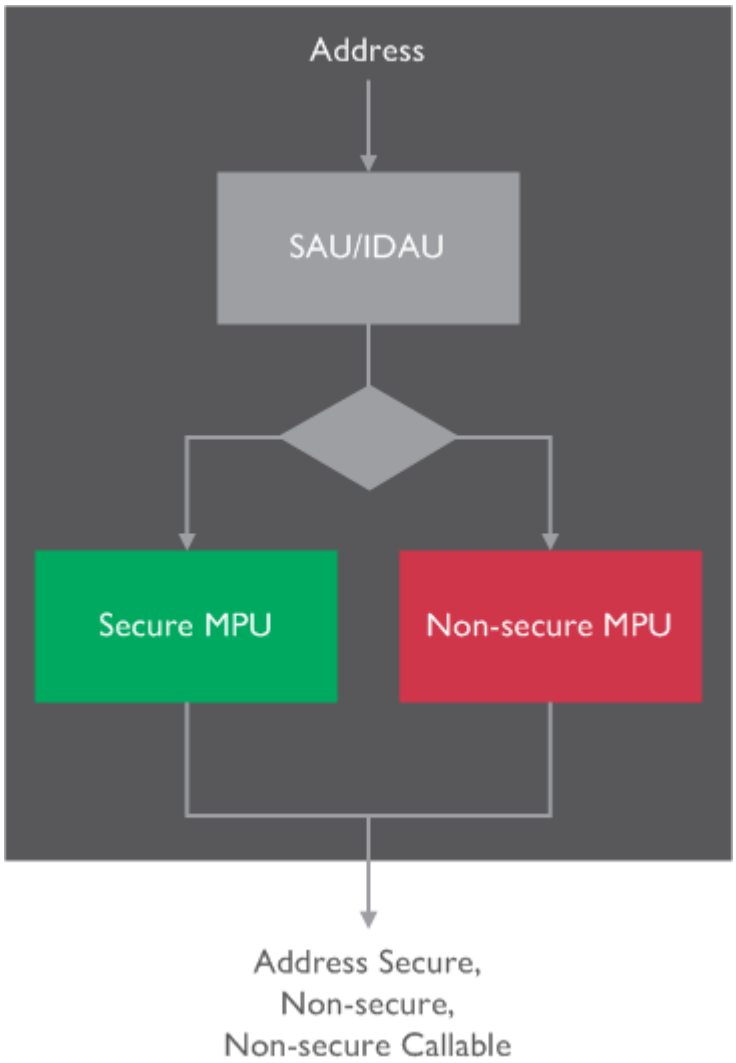
Non-secure addresses are used for memory and peripherals accessible by all software running on the device.

Non-secure transactions are those that originate from masters operating as, or deemed to be, Non-secure or from Secure masters accessing a Non-secure address. Non-secure transactions are only permitted to access NS addresses, and the system must ensure that NS transactions are denied access to Secure addresses.

4 Attribution units (SAU and IDAU)

If the ARMv8-M Security Extension is included in the processor, then security state of a memory region is controlled by a combination of the internal *Secure Attribution Unit* (SAU) or an external *Implementation Defined Attribution Unit* (IDAU). The number of SAU regions is defined during the implementation of the processor. The SAU is disabled at reset.

If no SAU regions are defined, or the SAU is disabled, and no IDAU is included in the system then the entire memory address space is defined as Secure and the processor is not able to switch to Non-secure state. Any attempt to switch to Non-secure state results in a fault. This is the default state of the processor.



The SAU is programmable in Secure state and has a programmers' model similar to the Memory Protection Unit (MPU). The SAU implementation is configurable by designers. The SAU is always present but the designer defines the number of regions. Alternatively, designers can use an IDAU to define a fixed memory map, and use a SAU to override the security attributes for some parts of the memory. A simple use could be to use the IDAU to split memory into 500Mb chunks of alternating Secure and Non-secure memory.

The designer of a microcontroller or SoC device divides the memory spaces into Secure and Non-secure areas. Software defines some of the regions using the Secure Attribution Unit (SAU), or by device-specific controller logic that is connected to a special Implementation Defined Attribution Unit (IDAU) interface on the processor. The memory partitioning is also used to define peripherals as Secure or Non-secure.

The SAU and IDAU also define region numbers for each of the memory regions. The region numbers are 8-bit, and are used by the Test Target(TT) instruction to allow software to determine access permissions and security attribute of objects in memory.

The SAU is only implemented if the ARMv8-M Security Extension is included in the processor. The number of regions that are included in the SAU can be configured to be either 0, 4 or 8.

The SAU can only be programmed in Secure state. Regions are programmed using the SAU Region number Register (SAU_RNR), SAU Region base Address Register (SAU_RBAR), and SAU Region Limit Address Register (SAU_RLAR). The SAU can be enabled using the SAU Control Register (SAU_CTRL).

Note

When programming the SAU Non-secure regions, you must ensure that Secure data and code is not exposed to Non-secure applications.

Security attribution and memory protection in the processor are provided by the optional SAU and the optional Memory Protection Units (MPUs).

For instructions and data, the SAU returns the security attribute that is associated with the address.

For instructions, the attribute determines the allowable Security state of the processor when the instruction is executed. It can also identify if code at a Secure address can be called from Non-secure state. It does this by applying the NSC attribute.

For data, the attribute determines whether a memory address can be accessed from Non-secure state, and also whether the external memory request is marked as Secure or Non-secure.

If a data access is made from Non-secure state to an address marked as Secure, then the processor takes a Secure Fault exception. If a data access is made from Secure state to an address marked as Non-secure, then the associated external memory access is marked as Non-secure.

5 SAU register summary

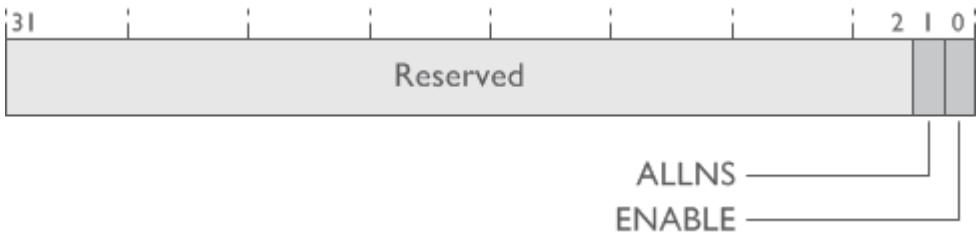
Each of the SAU registers is 32 bits wide. The following table shows the SAU register summary.

Address	Name	Type	Reset value	Processor security state	Description
0xE000EDD0	SAU_CTRL	RW	0x00000000	Secure	SAU Control register
				Non-secure	RAZ/WI
0xE000EDD4	SAU_TYPE	RO	0x0000000x	Secure	SAU Type register. Indicates the number of available regions

				Non-secure	RAZ/WI
0xE000EDD8	SAU_RNR	RW	UNKNOWN	Secure	SAU Region Number Register. Selects a region.
				Non-secure	RAZ/WI
0xE000EDDC	SAU_RBAR	RW	UNKNOWN	Secure	SAU Region Base Address Register
				Non-secure	RAZ/WI
0xE000EDE0	SAU_RLAR	RW	UNKNOWN	Secure	SAU Region Limit Address Register
				Non-secure	RAZ/WI

5.1 SAU_CTRL register

The following figure and table show the SAU_CTRL register characteristics:



Bits	Field	Description
[31:2]	Reserved – read as 0	Reserved
1	ALLNS	All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure.
0	ENABLE	Enable. Enables the SAU.

5.2 SAU_TYPE register

Bits	Field	Description
[31:8]	Reserved-read as 0	-
[7:0]	SREGION	SAU regions. The number of implemented SAU regions.

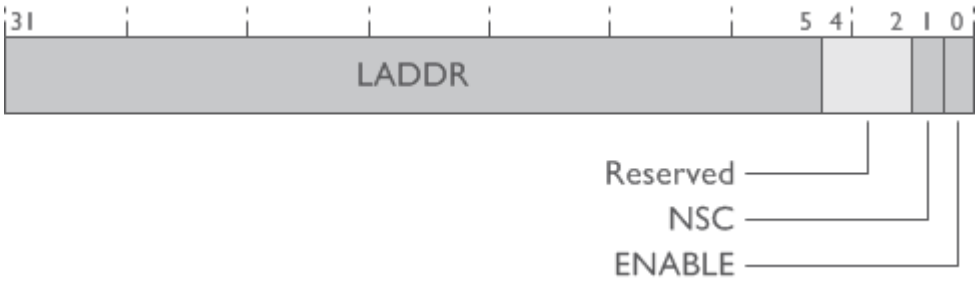
5.3 SAU_RNR register

Bits	Field	Description
[31:8]	Reserved – read as 0	-
[7:0]	REGION	Region number. Indicates the SAU region that SAU_RBAR and SAU_RLAR accesses.

5.4 SAU_RBAR register

Bits	Field	Description
[31:5]	BADDR	Base address. Holds bits [31:5] of the base address for the selected SAU region.
[4:0]	Reserved – read as 0	-

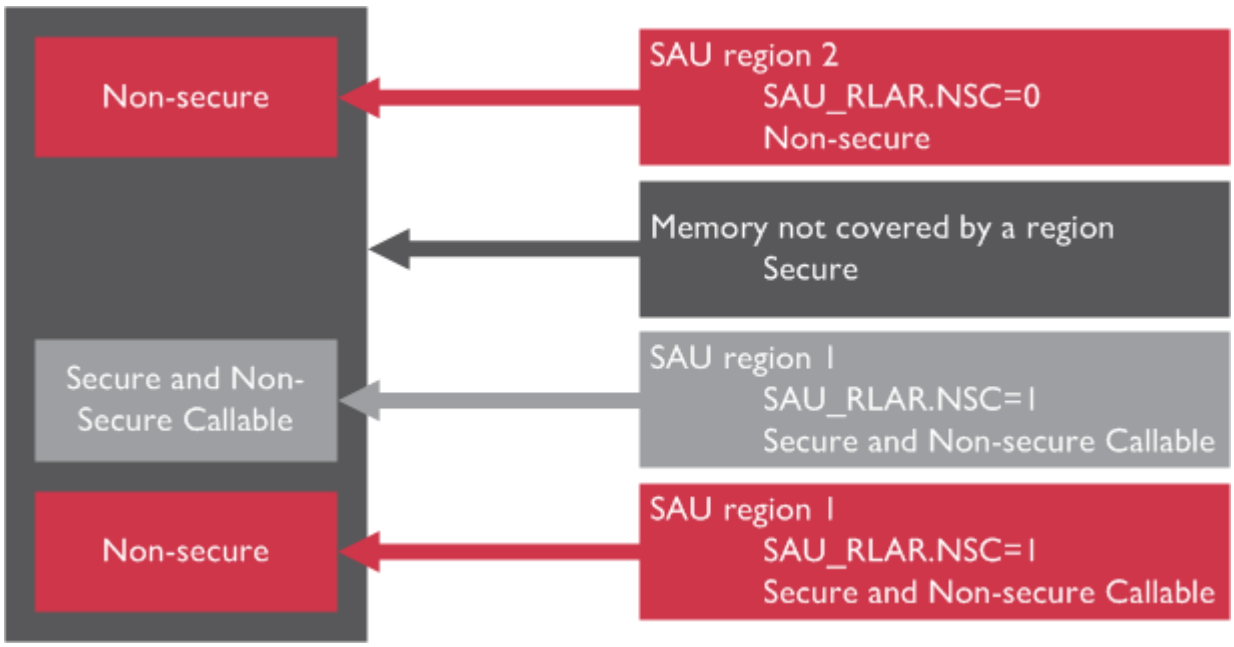
5.5 SAU_RLAR register



Bits	Field	Notes
31:5	LADDR	Limit address [31:5]. Bits [4:0] of the limit address are defined as 0x1F
4:2	RES0	
1	NSC	0 Region is not Non-secure callable1 Region is Non-secure callable
0	ENABLE	0 SAU region is enabled1 SAU region is enabled

5.6 SAU Region configuration

When the SAU is enabled, memory that is not covered by an enabled SAU region is Secure.



- Regions are enabled individually using SAU_RLAR.
- The region is Non-secure when SAU_RLAR.ENABLE = 1 and SUA_RLAR.NSC=0.
- The region is Secure and Non-secure callable when SAU_RLAR.ENABLE = 1 and SUA_RLAR.NSC=1.

5.7 Configuration example

The following example CMSIS code shows how the you can configure the SAU for two regions

```
// Configure SAU using CMSIS

// Configure SAU Region 0
// Start Address 0x00200000
// Limit Address 0x003FFFE0
// Secure non-scure callable

// Use CMSIS to access SAU Region Number Register (SAU_RNR)
// Select region 0
SAU->RNR  =  (0);
// Set SAU Region Base Address Register (SAU_RBAR)
SAU->RBAR =  (0x00200000U & SAU_RBAR_BADDR_Msk);
// Set SAU Region Limit Address Register (SAU_RLAR)
SAU->RLAR =  (0x003FFFE0U & SAU_RLAR_LADDR_Msk) | 1U << SAU_RLAR_NSC_Pos)  &
SAU_RLAR_NSC_Msk)   | 1U
```

```
// Configure SAU Region 1
// Start Address 0x20200000
// Limit Address 0x203FFFE0
// Non-secure

// Select region 1
SAU->RNR  =  (1);
// Set SAU Region Base Address Register (SAU_RBAR)
SAU->RBAR =  (0x20200000U & SAU_RBAR_BADDR_Msk);
// Set SAU Region Limit Address Register (SAU_RLAR)
SAU->RLAR =  (0x203FFFE0U & SAU_RLAR_LADDR_Msk) | 0U << SAU_RLAR_NSC_Pos)  &
SAU_RLAR_NSC_Msk)   | 1U


// Enable SAU
// Use CMSIS to access SAU Control Register (SAU_CTRL)
// Set ENABLE bit[0] to 1
// Set ALLNS bit[1] to 1 All memory is secure when SAU is disabled

SAU->CTRL = ((SAU_INIT_CTRL_ENABLE << SAU_CTRL_ENABLE_Pos) &
SAU->SAU_CTRL_ENABLE_Msk) |
           ((SAU_INIT_CTRL_ALLNS  << SAU_CTRL_ALLNS_Pos)  &
SAU_CTRL_ALLNS_Msk)   ;
```

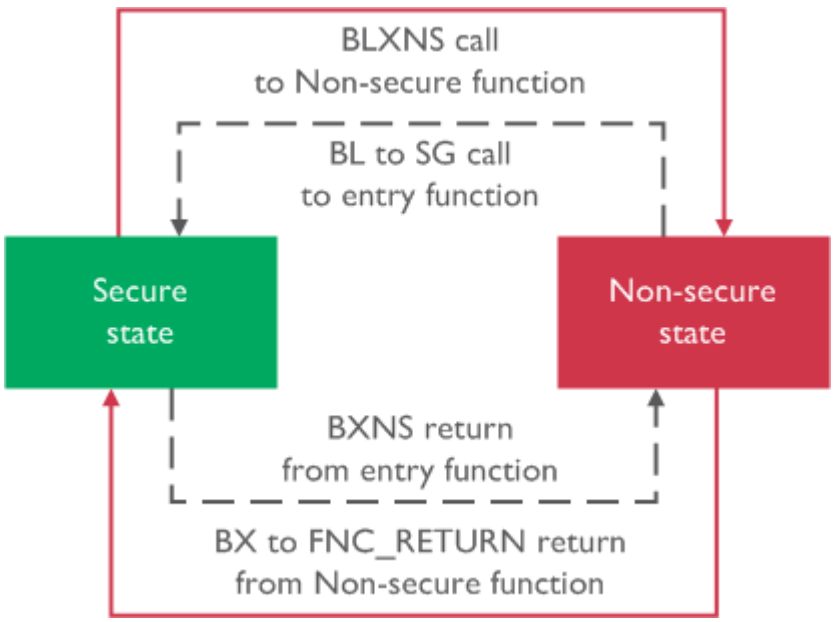
6 Switching between Secure and Non-secure states

The ARMv8-M Security Extensions allow direct calling between Secure and Non-secure software.

Several instructions are available for state transition handling in ARMv8-M processors:

SG	Secure gateway. Used for switching from Non-secure to Secure state at the first instruction of Secure entry point.
BXNS	Branch with exchange to Non-secure state. Used by Secure software to branch or return to Non-secure program.
BLXNS	Branch with link and exchange to Non-secure state. Used by Secure software to call Non-secure functions.

The following figure shows the security state transitions.



A direct API function call from Non-secure to Secure software entry points is allowed if the first instruction of the entry point is SG, and it is in a Non-secure callable memory location.

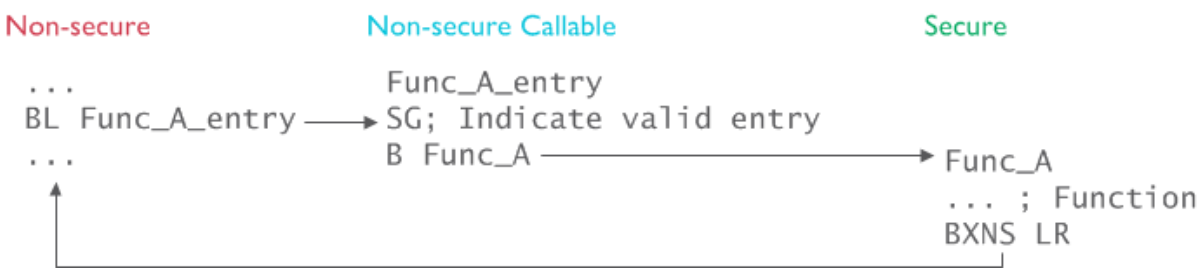


Figure 7: Software flow when a Non-secure program calls a function in the Secure world

When a Non-secure program calls a Secure API, the API completes by returning to a Non-secure state using a BXNS instruction. If a Non-secure program attempts to branch, or call a Secure program address without using a valid entry point, a fault event is generated. In ARMv8-M architecture the HardFault in Secure state handles the fault event. In ARMv8-M architecture with Main Extension, the SecureFault exception type is used.

When a Non-secure program calls a Secure API, the API completes by returning to a Non-secure state using a BXNS instruction. If a Non-secure program attempts to branch, or call a Secure program address without using a valid entry point, a fault event is generated. In ARMv8-M Mainline, the SecureFault exception type is used. For ARMv8-M Baseline, the HardFault in Secure state handles the fault event.

The ARMv8-M Security Extensions also allow a Secure program to call Non-secure software. In such a case, the Secure program uses a BLXNS instruction to call a Non-secure program. During the state transition, the return address and some processor state information are pushed onto the Secure stack, while the return address on the Link Register (LR) is set to a special value called FNC_RETURN. The Least Significant Bit (LSB) of the function address must be 0.

The following figure shows the software flow when a secure program calls a Non-secure function:

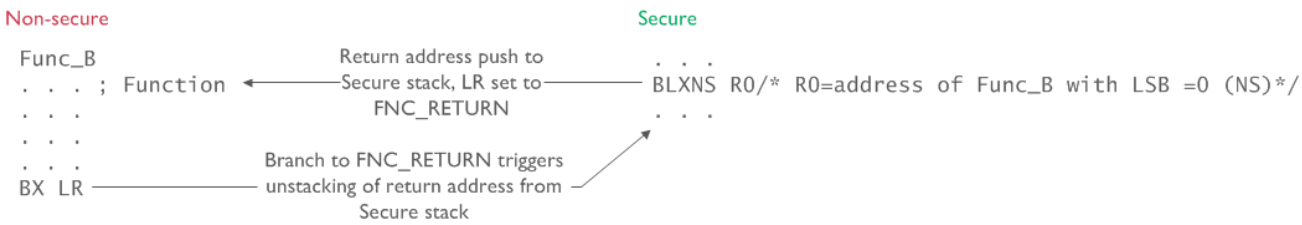


Figure 8: Software flow when a Secure program calls a Non-secure function

The Non-secure function completes by performing a branch to the FNC_RETURN address. This automatically triggers the unstacking of the true return address from the Secure stack and returns to the calling function. The state transition mechanism automatically hides the return address of the Secure software. Secure software can choose to transfer some of the register values to the Non-secure side as parameters, and clears other Secure data from the register banks before the function call.

State transitions can also happen due to exceptions and interrupts. Each interrupt can be configured as Secure or Non-secure, and is determined by the Interrupt Target Non-secure (NVIC_ITNS) register, which is only programmable in the Secure world. There are no restrictions regarding whether a Non-secure or Secure interrupt can take place when the processing is running Non-secure or Secure code.

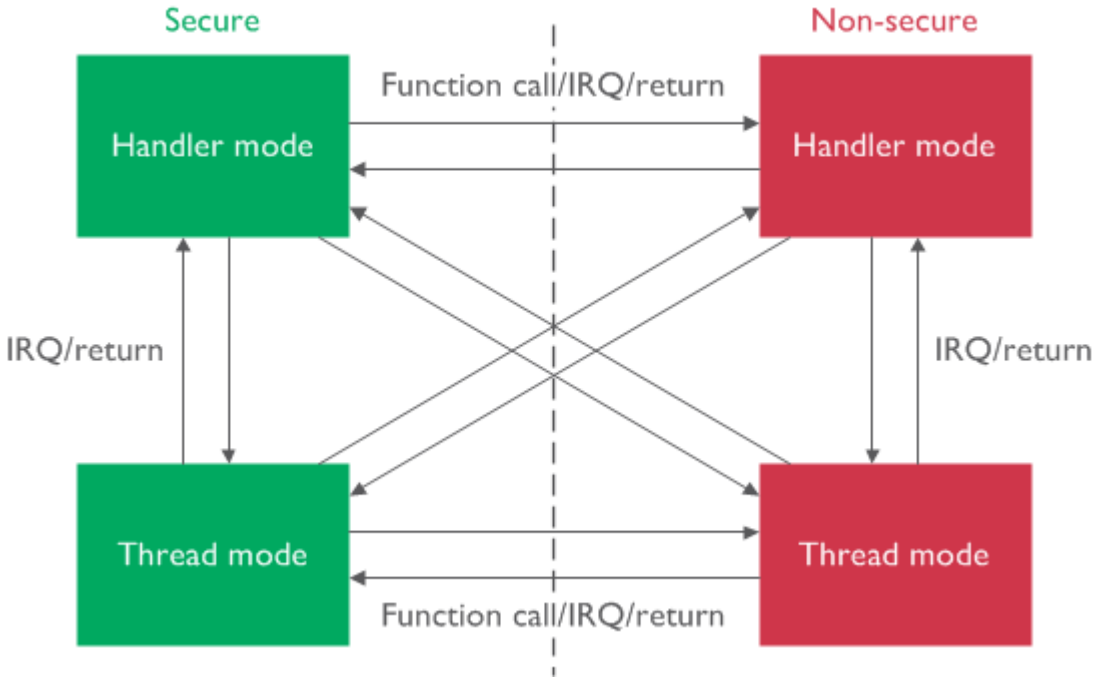


Figure 9: Forms of transition between Secure and Non-secure worlds

If the arriving exception or interrupt has the same state as the current processor state, the exception sequence is almost identical to the current M-series processors, enabling low interrupt latency. The main difference occurs when a Non-secure interrupt takes place, and is handled by the processor during execution of Secure code.

In this case, the processor automatically pushes all Secure information onto the Secure stack and erases the contents from the register banks, therefore avoiding an information leak.

All existing interrupt handling features such as nesting of interrupts, vectored interrupt handling, and vector table relocation are supported. TrustZone technology for ARMv8-M maintains the low interrupt latency characteristics of the existing M-series processors, with Secure to Non-secure interrupts incurring a slightly longer interrupt latency due to the need to push all Secure contents to the Secure stack.

The enhancement of the exception model also works with the lazy stacking of registers in the Floating-Point Unit (FPU). Lazy stacking is used to reduce the interrupt latency in exception sequences so that stacking of floating-point registers is avoided unless the interrupt handler also uses the FPU. In the ARMv8-M architecture, the same concept is applied to avoid the stacking of the Secure floating-point context. In cases where the Secure software does use the FPU and the Non-secure interrupt handler does not use the FPU, the stacking and unstacking of FPU registers is skipped to provide faster interrupt handling sequences. The following table shows some information.

6.1 Security states of the processor

In a simplified view, the program address determines the security state of the processor, which can be either Secure or Non-secure.

- If the processor is running program code in Non-secure memory, the processor is in Non-secure state.
- If the processor is running program code in Secure memory, the processor is in Secure state.
- If the processor is in Secure state, it must fetch instructions from Secure memory.

The ARMv8-M architecture permits function calls or branching between Secure and Non-secure software. However, restrictions are imposed for Non-secure to Secure transitions to ensure that only valid Secure API entry points can be used for calling from the Normal world to Secure code, or when the transition is caused by returning from a Non-secure API back to Secure code.

> **Note**
>
> There is special case when switching from Non-secure to Secure state. The first instruction of the transition must be the SG instruction, and the processor must be in the Non-secure state when the SG instruction is executed

Design characteristics

The design of TrustZone technology for ARMv8-M has several key characteristics.

These characteristics are:

- Non-secure code can call Secure functions using valid entry points only. There is no limitation on the number of entry points.
- Low switching overhead in cross security domain calls. There is only one extra instruction (SG) when calling from the Non-secure to the Secure domain, and only a few extra clock cycles when calling from the Secure state to Non-secure functions.
- Non-secure interrupt requests can still be served during the execution of Secure code, with minimal impact on the interrupt latency, stacking of the full register banks is required instead of just the caller saving registers.
- The processor starts up in Secure state by default. This start up mode enables root-of-trust implementations such as Secure boot.
- Low power:
    - There is no need for separate register banks for Secure and Non-secure states, while Non-secure interrupt handlers are still prevented from snooping into data used by Secure operations.
- Ease of use:
    - Interrupt handlers remain programmable in C, and Non-secure software can access Secure APIs with standard C/C++ function calls.
- High flexibility:

- The design allows Secure software to call Non-secure functions. This function is often required when protected middleware on the Secure side must access device driver code in the Non-secure side. The Secure state can also have privileged and unprivileged execution states, so this state can support multiple Secure software components with a protection mechanism between them.

7 Test target instruction

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

Test Target (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 if a contiguous range of memory shares common security attributes.

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

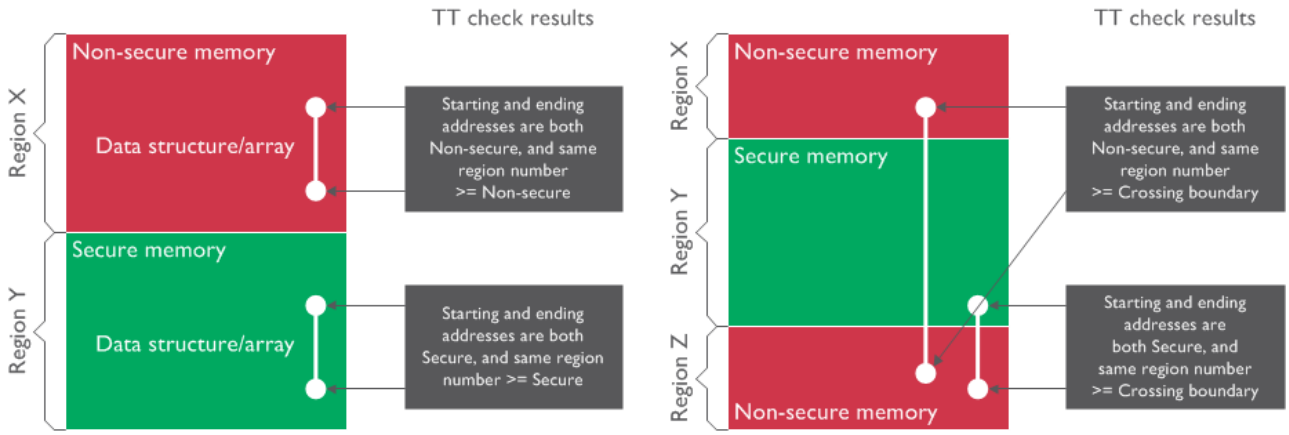


Figure 10: TT instruction allows software to determine if a data object is placed entirely in a Non-secure region

Note

The MPU, SAU and IDAU in ARMv8-M do not allow regions to overlap.

Using this mechanism, Secure code servicing APIs in the Secure world can determine if the memory referenced by a pointer from Non-secure software has the appropriate security attribute

for the API. This prevents Non-secure software from using APIs in Secure software to read out or corrupt Secure information.

As part of ARM TrustZone technology for ARMv8-M, there is also a stack limit checking feature. This detects the erroneous case where an application uses more stack than expected, which can potentially cause a security lapse and the possibility of a system failure. For ARMv8-M Mainline, all stack pointers have corresponding stack limit registers. There are no Baseline Limit registers for Non-secure. Non-secure programs can use the MPU for stack overflow prevention.

8 How Secure is TrustZone technology for ARMv8-M? Attck types

One of the common questions when talking about Secure system designs is: 'How Secure is it?' Many aspects of attack scenarios have been considered in the development of TrustZone technology for ARMv8-M. For example:

Software accesses

With extra system level components, memories can be partitioned between Secure and Non-secure spaces, and can disable Non-secure software from accessing Secure memories and resources.

Branch to arbitrary Secure address locations

The SG instruction and NSC memory attribute ensures that a Non-secure to Secure branch can happen only at valid entry points.

Inadvertent SG instruction in binary data

NSC memory attribute ensures that only Secure address spaces that are intended to be used as entry points can be used to switch the processor into the Secure state. Branching to an inadvertent SG instruction in an address location that is not marked as NSC results in a fault exception.

Faking of a return address when calling a Secure API

When the SG instruction is executed, the return state of the function is stored in the LSB of the return address in the Link Register (LR). At the return of the function, this bit is checked against the return state to prevent the Secure API function (which was called from Non-secure side) from returning to a fake return address pointing to a Secure address.

Attempting to switch to the Secure side using FNC_RETURN (function return code)

When switching from non-returnable Secure code (for example a Secure bootloader) to Non-secure, the BLXNS instruction must be used to ensure that there is a valid return stack. The return stack can then be used to enter an error handler.

This prevents Non-secure malicious code from trying to switch the processor to Secure code using the FNC_RETURN mechanism and crashing the Secure software if there is no valid return address in the Secure stack.

This recommendation does not apply when returning from a Secure API to Non-secure software, as this can use the BXNS instruction.

Faking of EXC_RETURN (exception return code) to return to Secure state illegally

If a Non-secure interrupt takes place during Secure code execution, the processor automatically adds a signature value to the Secure stack frame.

If the Non-secure software attempts to use the interrupt return to switch to the Secure side illegally, the signature check at the exception return fails and hence the error is detected.

Attempt to create stack overflow in Secure software

A stack limit feature is implemented for Secure stack pointers in both ARMv8-M Mainline and Baseline sub-profiles. Therefore the fault exception handler detects and handles such stack overflows.

On the debug side, the architecture also handles the security requirements.

Debug access management

Debug authentication signals are implemented on the processors so that designers can control if debug and trace operations are allowed for Secure and Non-secure states respectively.

AMBA bus interface protocols also support sideband signals in bus transactions, so the system can filter transfers to prevent debuggers from directly accessing Secure memories.

Debug and trace management

The debug authentication signals can be set up to disable debug and trace operations when the processor is running in the Secure state.

Although the architecture is designed to handle many types of attack scenarios, Secure software must always be written with care and must utilize security features, for example, stack limit checks, to prevent vulnerabilities. The ARM C Language Extensions (ACLE) have been extended to include extra features to support the ARMv8-M architecture, and software developers writing Secure software should utilize these features to enable their development tools to generate code images for ARMv8-M devices.

9 IDAU interface, IDAU, and memory map

The IDAU is used to indicate to the processor if a particular memory address is Secure, Non-secure Callable (NSC), or Non-secure, and provides the region number within which the memory address resides. It can also mark a memory region to be exempted from security checking, for example, a ROM table. The IDAU interface in general is processor-specific. However, there is a high similarity between the IDAU interfaces on different Cortex-M processors.

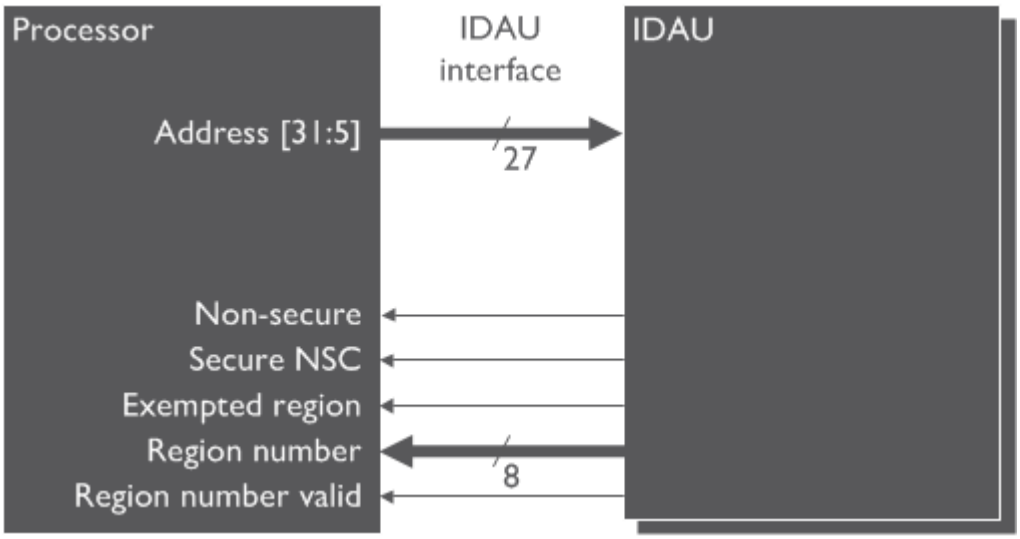


Figure 1: Example IDAU interface.

In theory it is possible to design the IDAU to be programmable. However, the signals on the IDAU interface are likely to be on timing critical paths which can make a complex IDAU impractical and could result in a higher gate count in the design. As a result, the IDAUs provide simple memory mapping with limited configurability.

ARMv8-M defines a memory map divided on 512MB boundaries. The example memory map in the following figure adds support for Security by aliasing each of these at their halfway point such that the lower half provides access to a 256MB Non-secure window. The upper half provides a Secure view of the same 256MB region. Control points elsewhere in the system determine what is accessible in each of the Secure and Non-secure windows.

For example, a designer could use bit [28] of the address to define if a memory is Secure or Non-secure, resulting in the following example memory map.

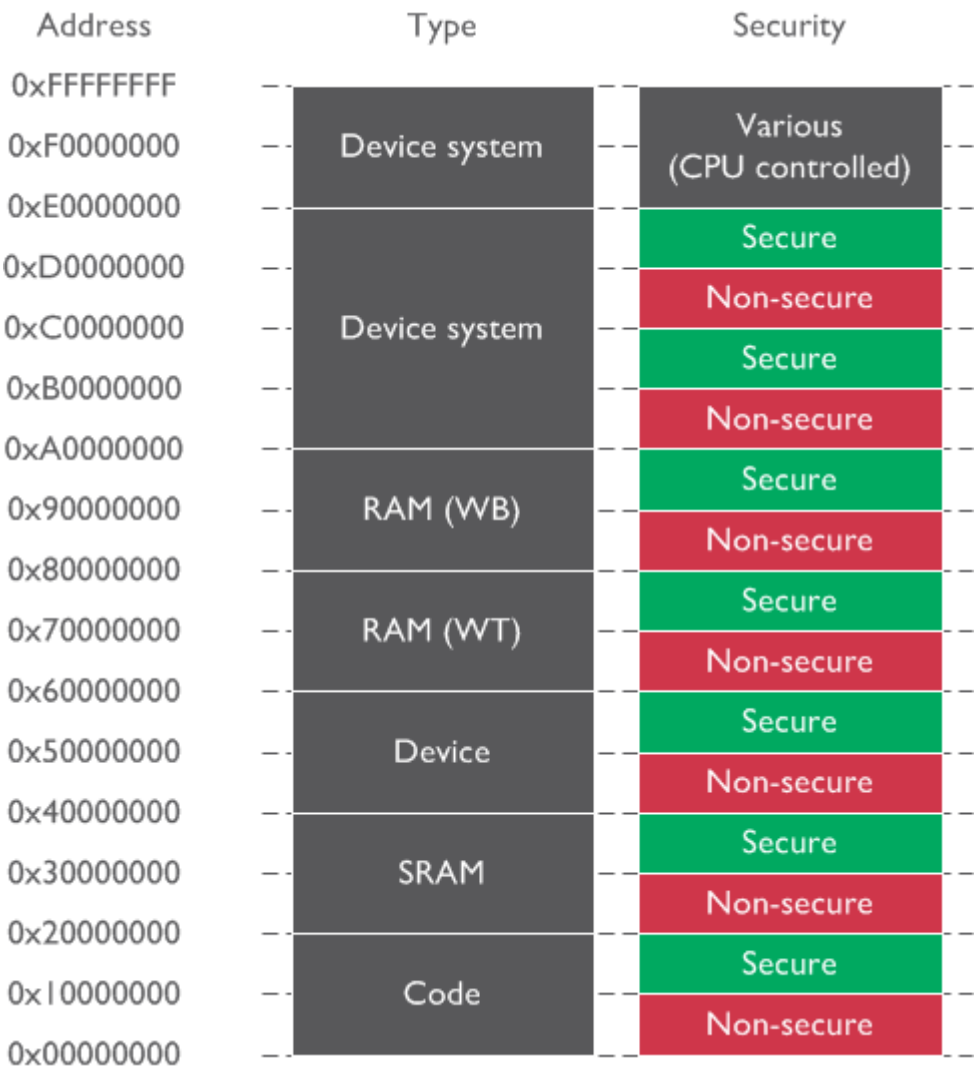


Figure 2: Simple memory map created by a minimal IDAU design

Note

The use of bit[28] for both aliasing and defining Secure vs Non-secure is an example only and must not be relied upon by software.

Such an IDAU can generate the required signals in the following ways:

- The Secure or Non-secure indication can be generated using address bit [28].
- The Secure NSC indication can reuse the Secure indication. This results in all Secure regions being indistinguishable from Secure NSC regions, and are therefore callable from Non-secure software by default. The Secure software must therefore use the internal SAU to force most of the Secure NSC regions to be Secure regions (not NSC) before allowing any Non-secure software to run.

It is important to ensure that only those memory areas that contain valid Secure entry functions (using the SG instruction) are configured to be NSC. Other Secure memories, for example, the stack, could contain data pattern that matches the SG instruction and therefore must not be configured as an NSC region.

- The region number can be generated from bits [31:28] of the address value, with the Region Valid signal tied high. It is important that region numbers are unique for each memory region, unless the memory region number is indicated as invalid by the IDAU interface.

- The Exempted Region control is set to 1 if the address is in the CoreSight ROM table address ranges, allowing the debugger to access to the ROM tables for device identification, even if the debugger is restricted to Non-secure debug only.

There is no restriction on whether Secure memory must be in the upper or lower half of each memory region. If the processor being used has an initial boot address that is restricted to address 0x00000000, then it is better to have the lower half of the address marked as Secure so that the processor can boot in the Secure state.

For application scenarios where an ARMv8-M processor is used together with an ARMv8-A system with a shared memory security attribute configuration, then the IDAU response signal should be generated based on the system-wide security arrangement; the simple memory map arrangement that is described here would be insufficient.