

# TrustZone® technology for ARM®v8-M

## Architecture

Version 1.1



# TrustZone® technology for ARM®v8-M Architecture

Copyright © 2016 ARM Limited or its affiliates. All rights reserved.

## Release Information

## Document History

Issue	Date	Confidentiality	Change
0000-00	08 July 2016	Non-Confidential	First release.
0101-00	23 August 2016	Non-Confidential	Second release.

## Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow ARM’s trademark usage guidelines at <http://www.arm.com/about/trademark-usage-guidelines.php>

Copyright © 2016, ARM Limited or its affiliates. All rights reserved.

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

## TrustZone® technology for ARM®v8-M Architecture

### **Preface**

<i>About this book</i> .....	6
<i>Feedback</i> .....	8

### **Chapter 1**

#### **Introduction**

1.1	<i>Secure and Normal worlds</i> .....	1-10
1.2	<i>Security states of the processor</i> .....	1-12
1.3	<i>Memory system and memory partitioning</i> .....	1-14
1.4	<i>Switching between Secure and Non-secure states</i> .....	1-18
1.5	<i>Test Target instructions</i> .....	1-21

### **Chapter 2**

#### **Security**

2.1	<i>Security requirements addressed by TrustZone® technology for ARM®v8-M</i> .....	2-24
2.2	<i>Attack types</i> .....	2-27

### **Chapter 3**

#### **Attribution units**

3.1	<i>SAU and IDAU</i> .....	3-30
3.2	<i>SAU register summary</i> .....	3-31
3.3	<i>IDAU interface, IDAU, and memory map</i> .....	3-35

# Preface

This preface introduces the *TrustZone® technology for ARM®v8-M Architecture* .

It contains the following:

- [About this book on page 6.](#)
- [Feedback on page 8.](#)

## About this book

Write a short description in the book map to render in the "About this book" section of the preface.

### Product revision status

The *rmpr* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

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

*pr* 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 Introduction

ARM® 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. 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.

#### Chapter 2 Security

This topic describes the security features of the TrustZone technology for ARMv8-M. It also provides examples on different attack scenarios and the ways the TrustZone technology for ARMv8-M can prevent them.

#### Chapter 3 Attribution units

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.

### 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.

### Typographic conventions

*italic*

Introduces special terminology, denotes cross-references, and citations.

**bold**

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

monospace

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

monospace

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

*monospace italic*

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

### **monospace bold**

Denotes language keywords when used outside example code.

### **<and>**

Encloses replaceable terms for assembler syntax where they appear in code or code fragments.  
For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

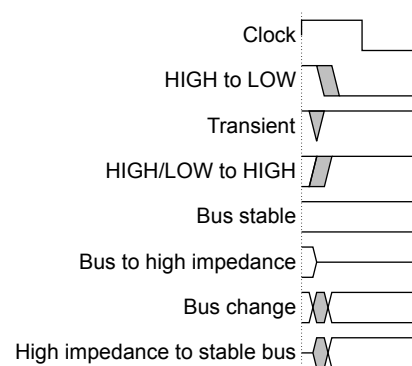
### **SMALL CAPITALS**

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

## **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](mailto:errata@arm.com). Give:

- The title *TrustZone® technology for ARM®v8-M Architecture* .
- The number ARM 100690\_0101\_00\_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

## Introduction

ARM® 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. 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.

It contains the following sections:

- [1.1 Secure and Normal worlds on page 1-10.](#)
- [1.2 Security states of the processor on page 1-12.](#)
- [1.3 Memory system and memory partitioning on page 1-14.](#)
- [1.4 Switching between Secure and Non-secure states on page 1-18.](#)
- [1.5 Test Target instructions on page 1-21.](#)

## 1.1 Secure and Normal worlds

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.

---

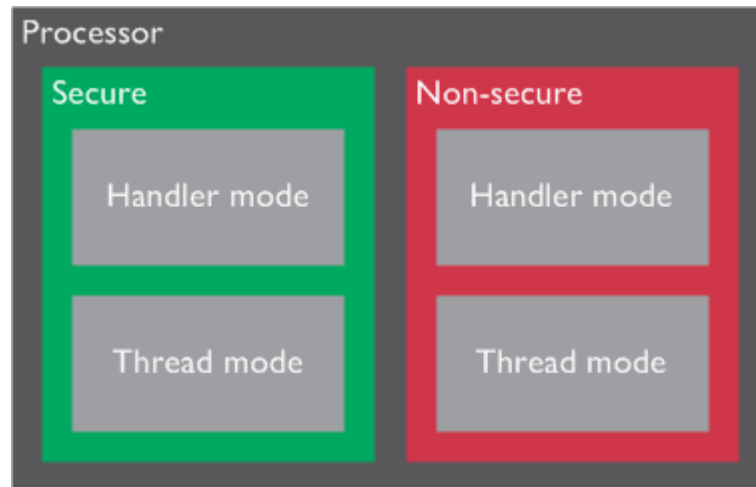
**Note**

Thread mode can also be either Privileged or Unprivileged.

---

The ARMv8-M architecture with Security Extension is an optional architecture extension. 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. ARM TrustZone technology does not cover all aspects of security. For example, it does not include cryptography.

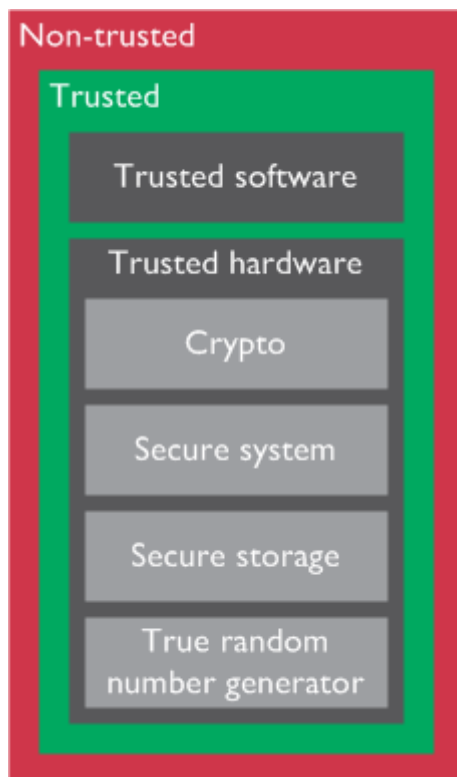
The following figure shows how TrustZone technology for ARMv8-M adds Secure and Non-secure states to processor operation:



In designs with ARMv8-M architecture with 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.

### 1.1.1 Security requirements of the Secure platform

To ensure a Secure platform, the security arrangement is not limited to processor architecture. The whole system level design is enhanced to address security requirements.

The security requirements are as follows:

- The bus protocol has been extended to support the security attributes.
- Extra system components are also required to manage system security aspects, for example, to manage the access permissions for Secure and Non-secure peripherals.
- The debug architecture must also be security aware to prevent access to secret information using debug connections or trace features.
- Software running in the Secure world must be written carefully to prevent vulnerabilities.

### 1.1.2 Relationship between ARM® TrustZone® technology for ARM®v8-M and ARM® Cortex®-A processors

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.

ARM TrustZone technology for ARMv8-M is different to the Cortex-A version, in that the design is optimized for microcontrollers and low-power SoC applications. In these applications, the requirements of low power, low memory footprint, and low latency are important factors. To get the best results, TrustZone technology for ARMv8-M was designed from the ground up instead of reusing the existing TrustZone technology for ARM Cortex-A processors. As a result, the underlying operations of TrustZone technology in the ARMv8-M architecture are different from TrustZone technology in Cortex-A processors.

## 1.2 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.

---

This section contains the following subsections:

- [1.2.1 Design characteristics on page 1-12.](#)
- [1.2.2 Difference in the implementation of TrustZone® technology for ARM®v8-M and TrustZone® technology in ARM® Cortex®-A processors on page 1-13.](#)
- [1.2.3 Difference between TrustZone® technology for ARM®v8-M and virtualization approach in the ARM®v8-R architecture on page 1-13.](#)

### 1.2.1 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 needs to 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.

### 1.2.2 Difference in the implementation of TrustZone® technology for ARM®v8-M and TrustZone® technology in ARM® Cortex®-A 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. Therefore 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.

### 1.2.3 Difference between TrustZone® technology for ARM®v8-M and virtualization approach in the ARM®v8-R architecture

TrustZone technology for ARMv8-M is also different from the virtualization approach as supported in the ARMv8-R architecture.

In systems with virtualization, each of the virtualized software environments (Virtual Machines, or VMs) is isolated from each other, and there is no direct interaction between VMs other than going through the hypervisor, interrupts, or shared memories.

As a result, interaction between software in different VMs requires additional execution cycles and software overhead, and while this is perfectly acceptable in the high-performance ARM Cortex-R processors, it is not ideal for resource constrained ARM Cortex-M applications. In addition, the memory footprints for virtualized systems are often significantly larger, as they require hypervisor software and often contain multiple operating systems.

## 1.3 Memory system and memory partitioning

The 4GB memory space is partitioned into Non-secure and Secure memory regions.

### Non-secure (NS)

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

Non-secure transactions 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 Non-secure addresses, and the system must ensure that Non-secure transactions are denied access to Secure addresses.

The Secure memory space is further divided into two types:

### Secure (S)

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

Secure transactions 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 memory location. This memory type is the only type which an ARMv8-M processor permits to hold an *SG* instruction that allows software to transition from Non-secure to Secure state. The inclusion of NSC locations removes the need for Secure software creators to policy for the accidental inclusion of SG instructions (or data sharing an encoding value) in normal Secure memory by restricting the functionality of the SG instruction to NSC memory.

### 1.3.1 NSC memory regions

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.

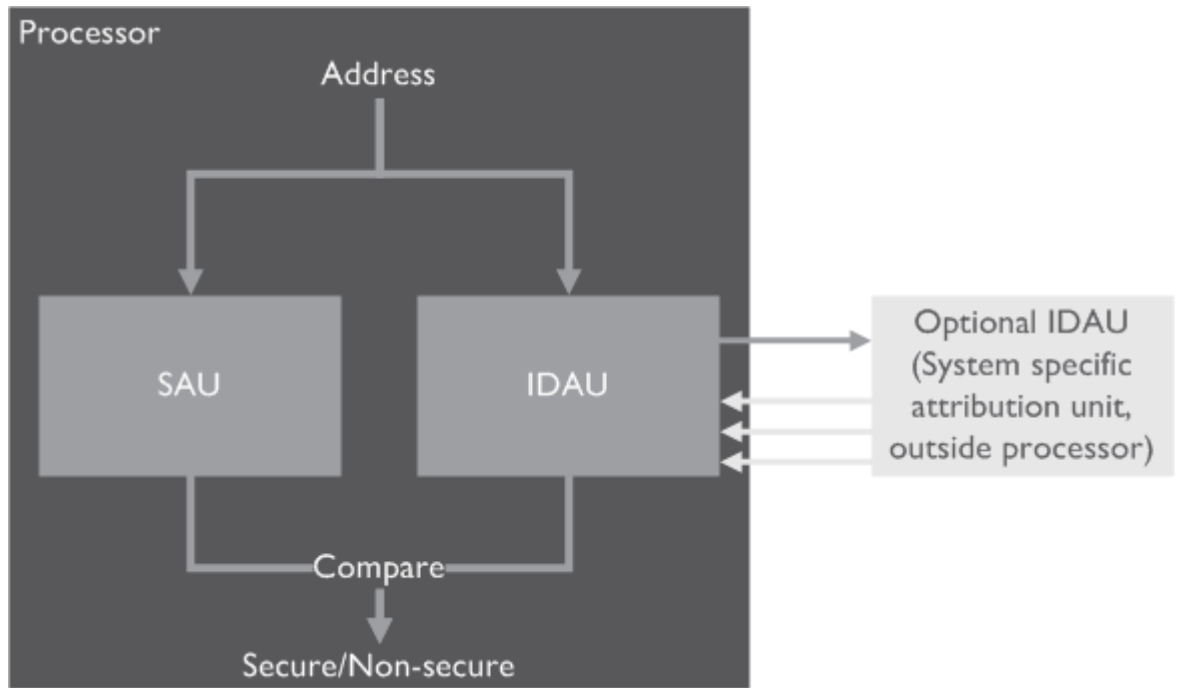
When a Non-secure program calls a function in the Secure side:

- The first instruction in the API must be an *SG* instruction.
- The SG instruction must be in an NSC region, 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.

### 1.3.2 Attribution units (SAU and IDAU)

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 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.

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.

### 1.3.3 Banked internal resources

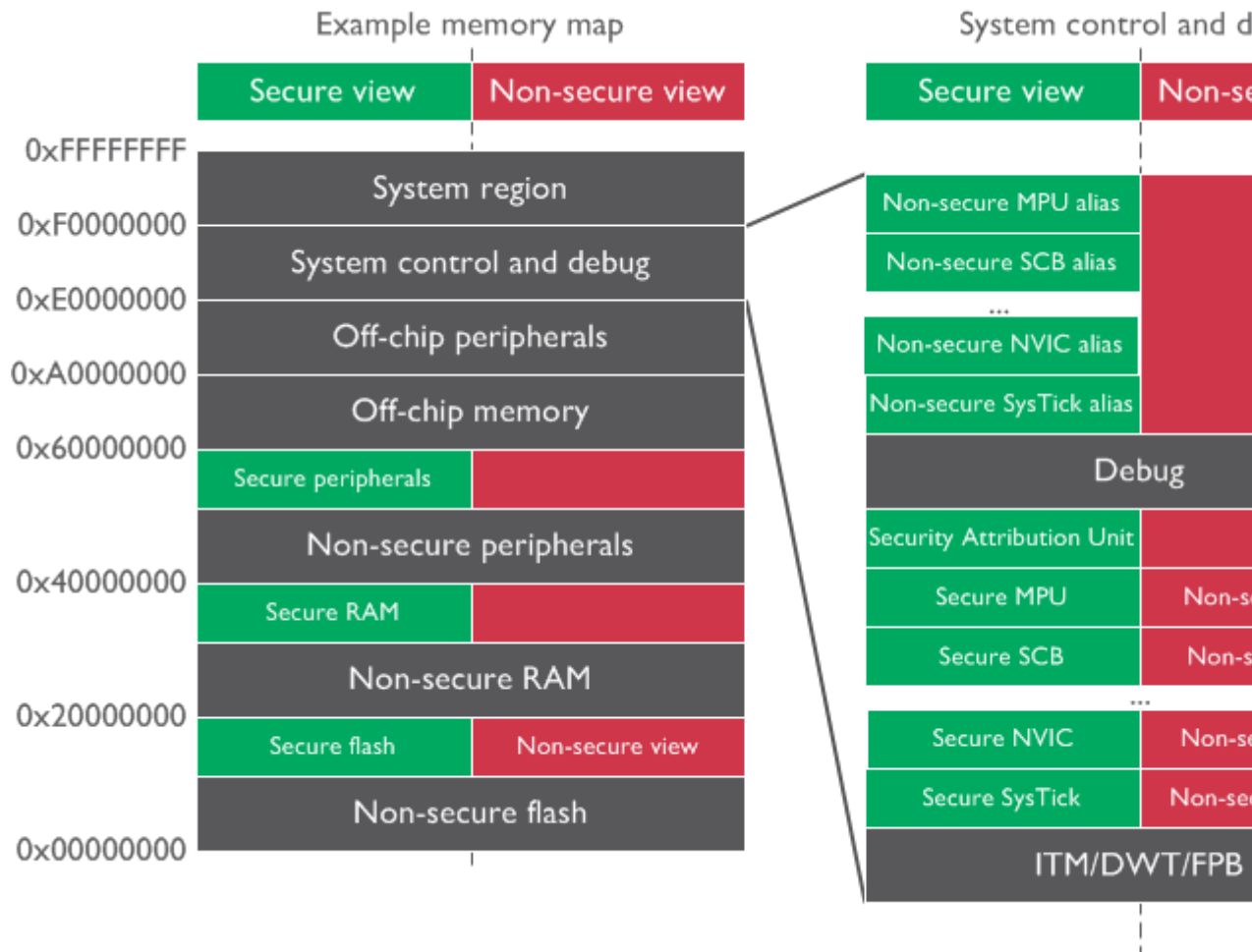
Several internal resources are banked between Secure and Non-secure states.

For registers inside the processor core:

- The Stack Pointers are banked between Secure and Non-secure states. This enables separations of Secure and Non-secure stacks.
- Interrupt masking registers like PRIMASK, FAULTMASK, and BASEPRI are banked.
- FAULTMASK and BASEPRI are available on the ARMv8-M architecture with Main Extension only. This allows existing software to be reused, but Non-secure software cannot influence the operation of Secure software.
- Bit 0 and 1 of the special CONTROL register are banked. Secure and Non-secure software can have different Stack Pointer control and privileged settings.

In addition, the MPU, SysTick timer, and some of the registers in the *System Control Block* (SCB) are also banked. For example, the *Vector Table Offset Register* (VTOR) is banked to allow the vector tables for Secure software and Non-secure software to be separated. Software that accesses these registers using an existing address accesses the corresponding view of the peripheral based on the current processor state. Secure accesses see Secure peripherals. Non-secure accesses see Non-secure peripherals. Secure software can also access Non-secure versions of these components using alias addresses.

The following figure shows the system control and debug area of a typical memory map:

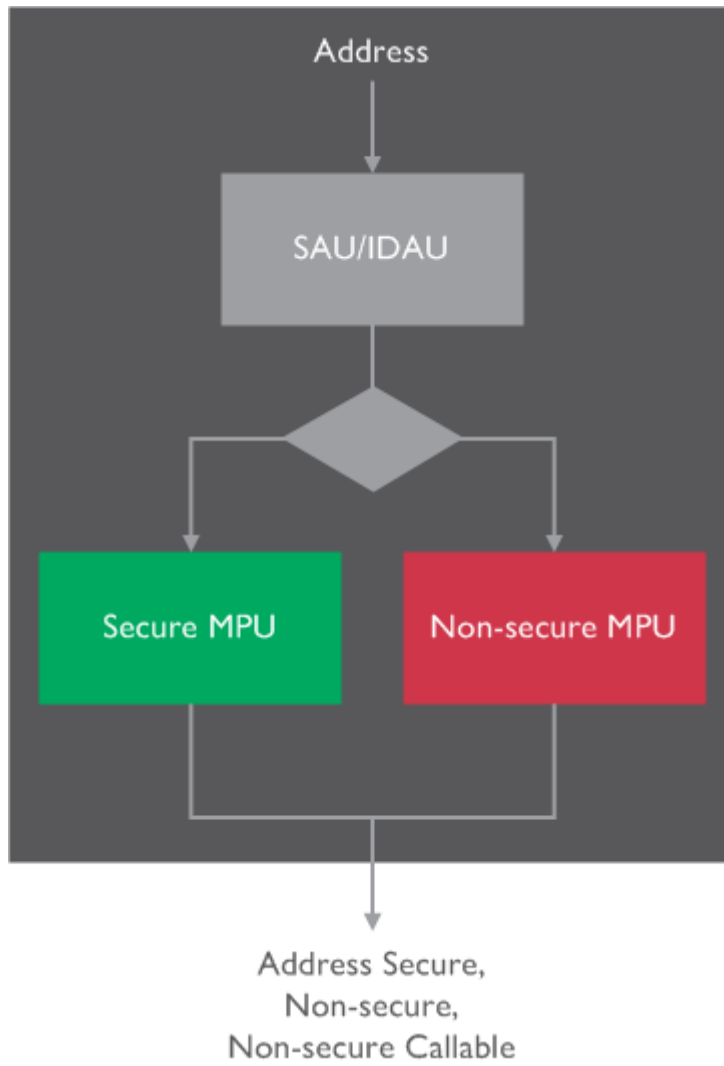


#### 1.3.4 Secure and Non-secure MPUs

As in earlier M-series processors, the *Memory Protection Unit* (MPU) is optional. Based on application requirements, designers can exclude the MPU to reduce area and power, or include either Secure or Non-secure MPU, or both if necessary.

The Secure and Non-secure MPU can be implemented with a different number of MPU regions. Bus transfers generated are looked up using the Secure or Non-secure MPU based on the current state of the processor.





## 1.4 Switching between Secure and Non-secure states

The ARMv8-M architecture with Security Extension allows direct calling between Secure and Non-secure software.

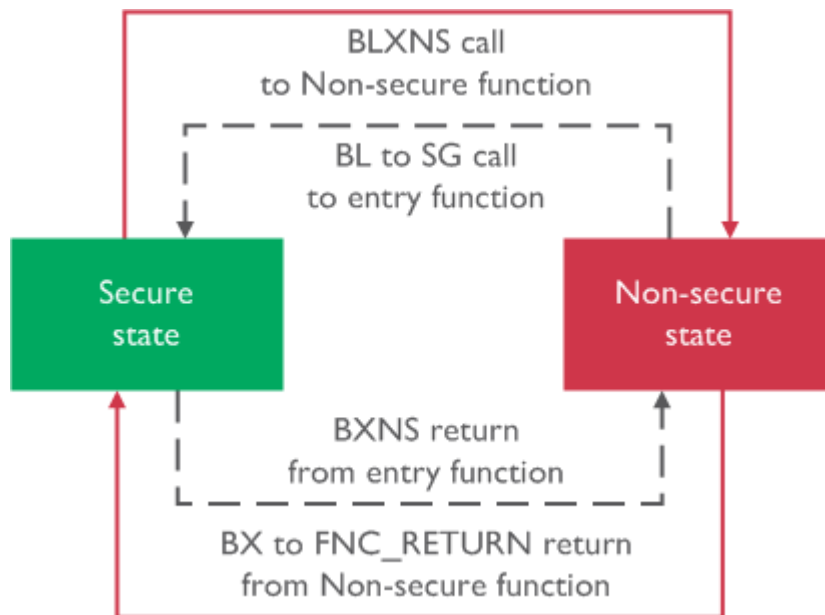
This section contains the following subsections:

- [1.4.1 Security state transitions on page 1-18.](#)
- [1.4.2 Calling Non-secure software on page 1-19.](#)
- [1.4.3 State transition by exceptions and interrupts on page 1-19.](#)

### 1.4.1 Security state transitions

Secure gateway (SG), Branch with exchange to Non-secure state (BXNS), and Branch with link and exchange to Non-secure state (BLXNS) instructions are available for state transition handling in ARMv8-M processors.

The following figure shows the security state transitions:



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

#### Secure gateway (SG)

This instruction is used for switching from Non-secure to Secure state at the first instruction of Secure entry point.

#### Branch with exchange to Non-secure state (BXNS)

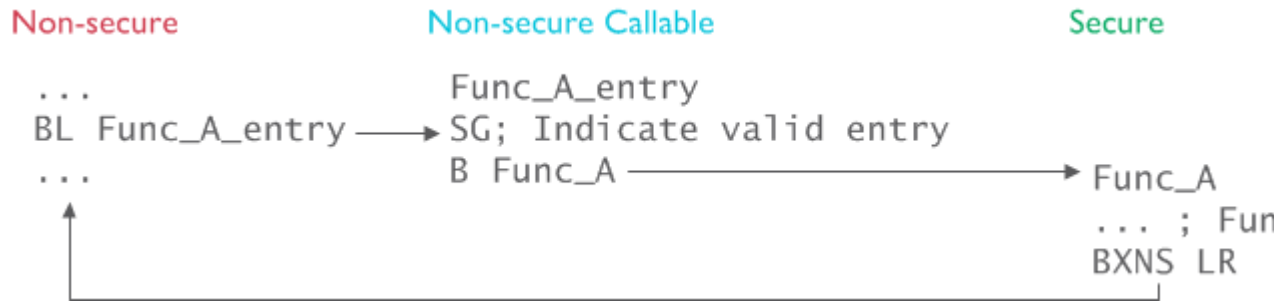
This instruction is used by Secure software to branch or return to Non-secure program.

#### Branch with link and exchange to Non-secure state (BLXNS)

This instruction is used by Secure software to call Non-secure functions.

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.

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



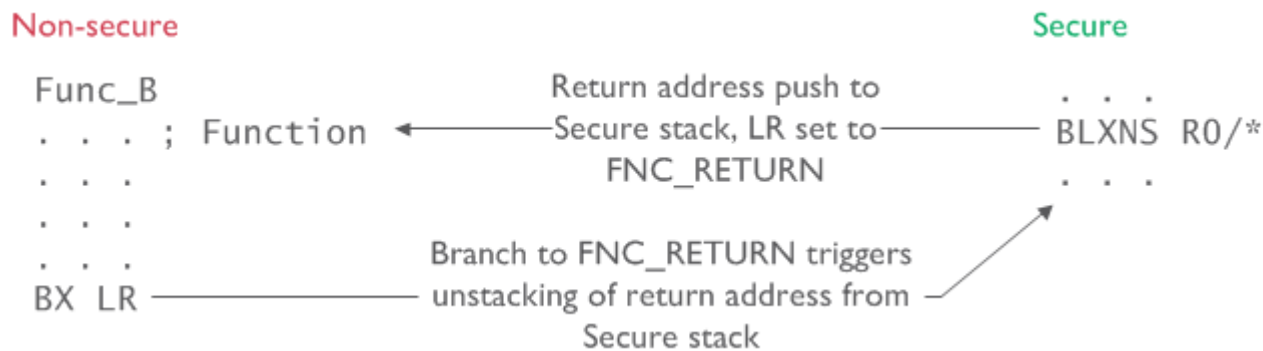
When a Non-secure program calls a Secure API, the API completes by returning to a Non-secure state using a *BXNS* instruction. If a Non-secure program attempts to branch, or call a Secure program address without using a valid entry point, a fault event is generated. In ARMv8-M architecture the *HardFault*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

### 1.4.2 Calling Non-secure software

The ARMv8-M architecture with Security Extension 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 figures shows the software flow when a Secure program calls a Non-secure function:



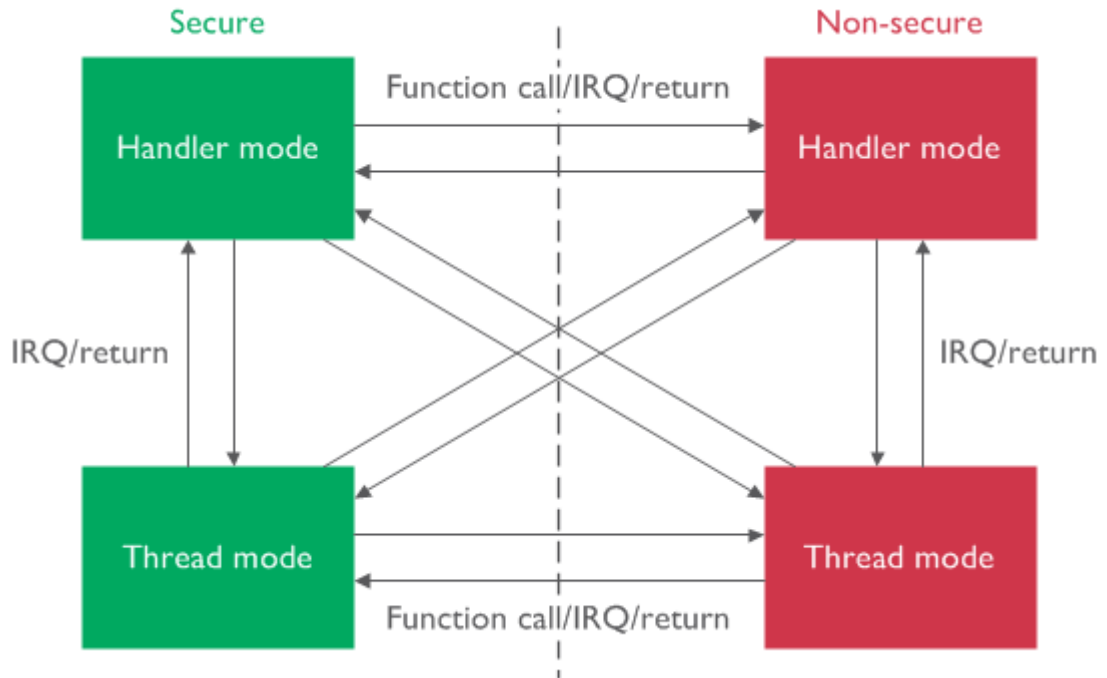
The Non-secure function completes by performing a branch to the *FNC\_RETURN* address. This automatically triggers the unstacking of the true return address from the Secure stack and returns to the calling function. The state transition mechanism automatically hides the return address of the Secure software. 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.

### 1.4.3 State transition by exceptions and interrupts

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.

The following figure shows the forms of transition between Secure and Normal 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.

## 1.5 Test Target instructions

Description of how different Test Target instructions query the security state of a memory location.

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.

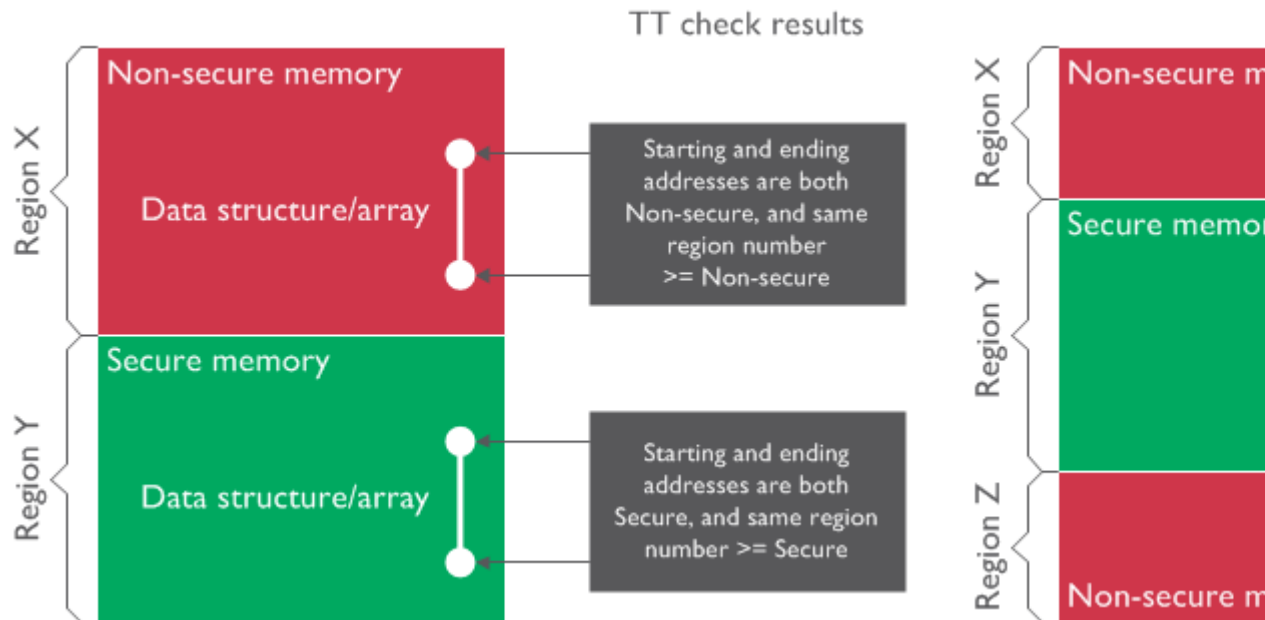
*Test Target Unprivileged* (TTT) queries the security state and access permissions of a memory location for an unprivileged access to that location.

*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 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 the following figure shows:



### 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 the ARMv8-M architecture with Main Extension, all Stack Pointers have corresponding stack limit registers. There are no ARMv8-M registers for Non-secure. Non-secure programs can use the *Memory Protection Unit* (MPU) for stack overflow prevention.

## Chapter 2

# Security

This topic describes the security features of the TrustZone technology for ARMv8-M. It also provides examples on different attack scenarios and the ways the TrustZone technology for ARMv8-M can prevent them.

It contains the following sections:

- [2.1 Security requirements addressed by TrustZone® technology for ARM®v8-M on page 2-24.](#)
- [2.2 Attack types on page 2-27.](#)

## 2.1 Security requirements addressed by TrustZone® technology for ARM®v8-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**

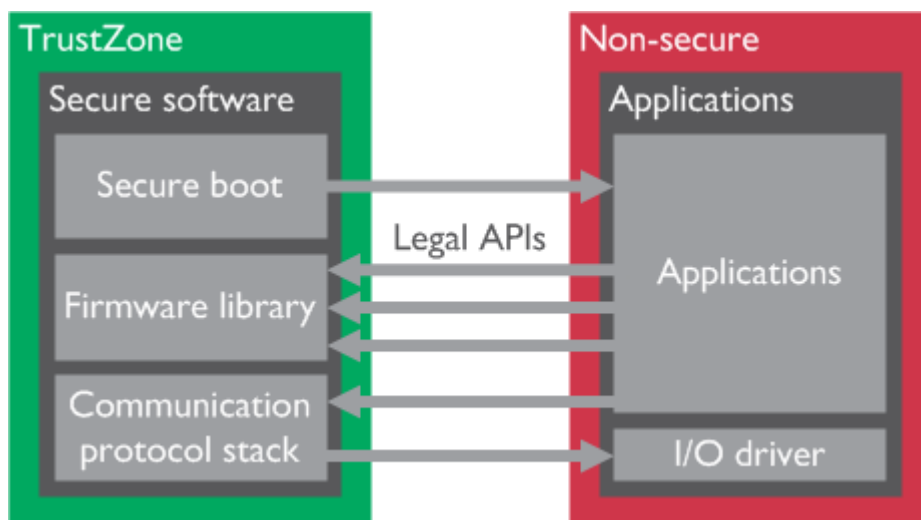
The Secure boot mechanism enables you to have confidence in the platform, as it will always boot from Secure memory.

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:



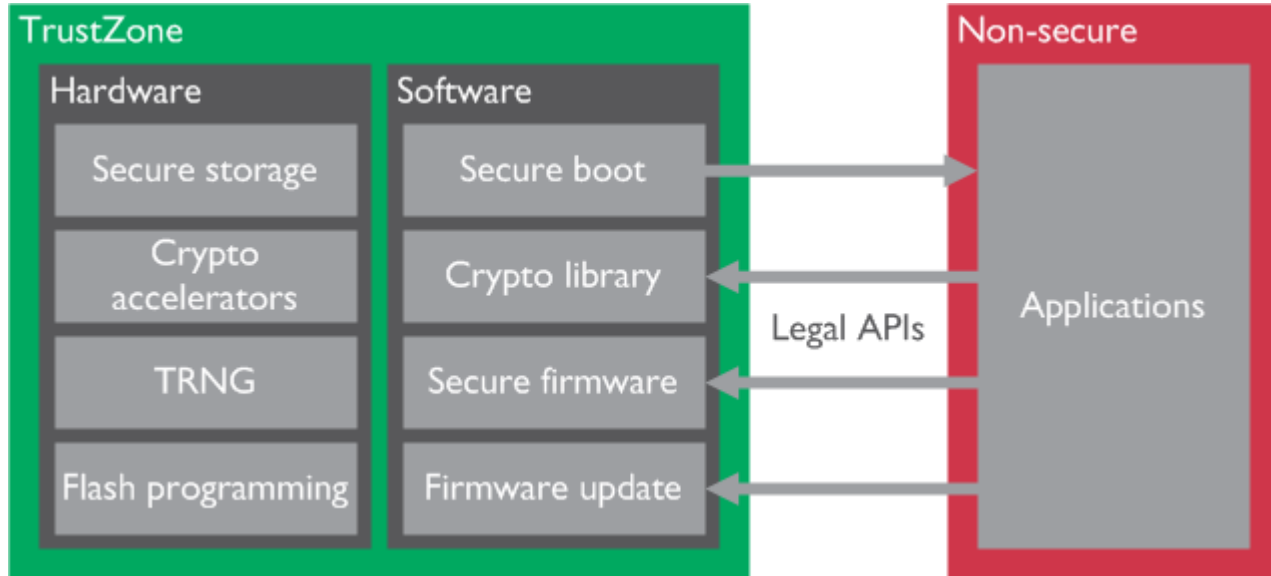


Many microcontrollers already have built-in firmware such as USB or Bluetooth stacks, and TrustZone technology makes the firmware protection implementation easier and more Secure, by ensuring that untrusted software cannot branch to the middle of Secure APIs to bypass any initial checking.

### 2.1.1 Security for IoT products

TrustZone technology 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:



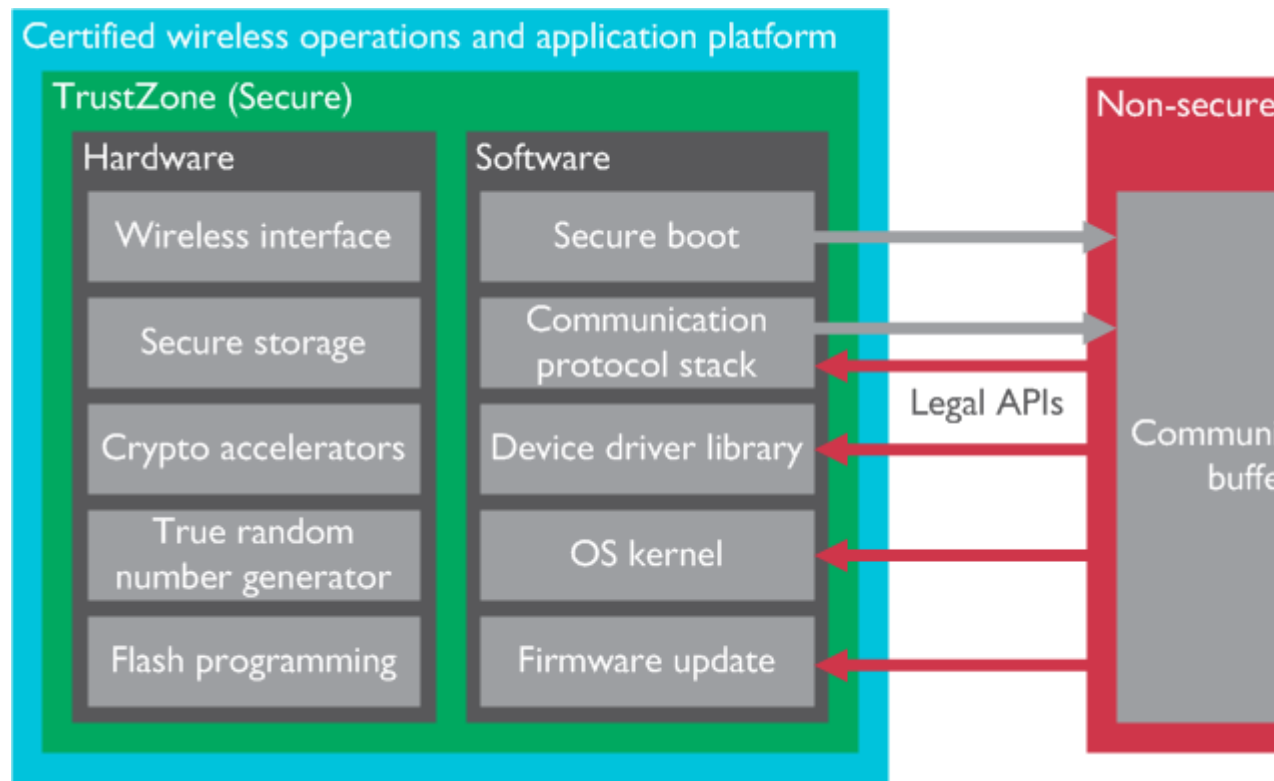
By using TrustZone technology to safeguard these security features, designers can:

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

### 2.1.2 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:



## 2.2 Attack types

Security is one of the common questions when talking about Secure system designs. Many aspects of attack scenarios have been considered in the development of TrustZone technology for ARMv8-M.

These attack scenarios include:

### 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 *Non-secure Callable* (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.

### 2.2.1 Security requirements and architecture

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. Software developers writing Secure software should utilize these features to enable their development tools to generate code images for ARMv8-M devices.

## Chapter 3

# Attribution units

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.

It contains the following sections:

- [3.1 SAU and IDAU on page 3-30.](#)
- [3.2 SAU register summary on page 3-31.](#)
- [3.3 IDAU interface, IDAU, and memory map on page 3-35.](#)

## 3.1 SAU and IDAU

If the ARMv8-M Security Extension is included in the processor, then the internal *Secure Attribution Unit* (SAU) or an external *Implementation Defined Attribution Unit* (IDAU) determines the Security state attributed to each address.

The number of SAU regions is defined during the implementation of the processor. The SAU is disabled at reset.

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.

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 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.

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 check by applying the NSC attribute.

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

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

## 3.2 SAU register summary

List of SAU registers containing the reset value, processor security state and detailed description.

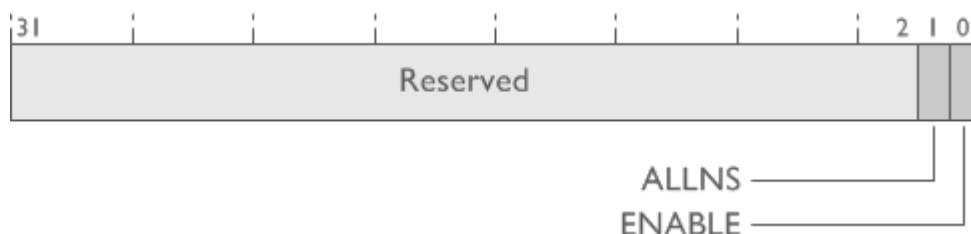
Each of the SAU registers is 32 bits wide. The following table shows the SAU register summary.

**Table 3-1 SAU register summary**

Address	Name	Type	Reset value	Processor security state	Description
0xE000EDD0	SAU_CTRL	RW	0x000000	Secure	SAU Control register
				Non-secure	Read as 0, writes ignored
0xE000EDD4	SAU_TYPE	RO	0x0000000x	Secure	SAU Type register. Indicates the number of available regions
				Non-secure	Read as 0, writes ignored
0xE000EDD8	SAU_RNR	RW	UNKNOWN	Secure	SAU Region Number Register. Selects a region.
				Non-secure	Read as 0, writes ignored
0xE000EDDC	SAU_RBAR	RW	UNKNOWN	Secure	SAU Region Base Address Register
				Non-secure	Read as 0, writes ignored
0xE000EDE0	SAU_RLAR	RW	UNKNOWN	Secure	SAU Region Limit Address Register
				Non-secure	Read as 0, writes ignored

### SAU\_CTRL register

The following figure and table show the SAU\_CTRL register characteristics:



**Table 3-2 SAU\_CTRL register**

Bits	Field	Description
[31:2]	Reserved	Reserved, read as 0.
[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.

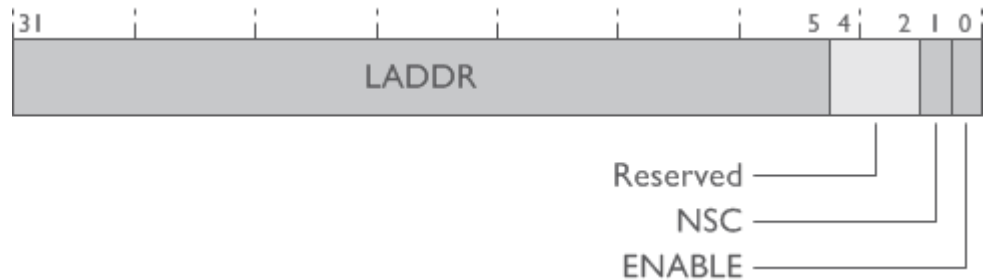
### SAU\_RBAR register

The following figure and table show the SAU\_RBAR register characteristics:



**Table 3-3 SAU\_RBAR register**

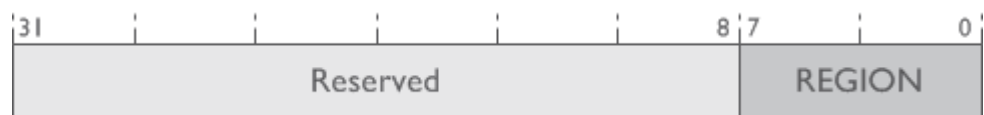
Bits	Field	Description
[31:5]	BADDR	Base address. Holds bits [31:5] of the base address for the selected SAU region.
[4:0]	Reserved	Reserved, read as 0.



**Figure 3-1 SAU\_RLAR register**

**Table 3-4 SAU\_RLAR register**

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



**Figure 3-2 SAU\_RNR register**

**Table 3-5 SAU\_RNR register**

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



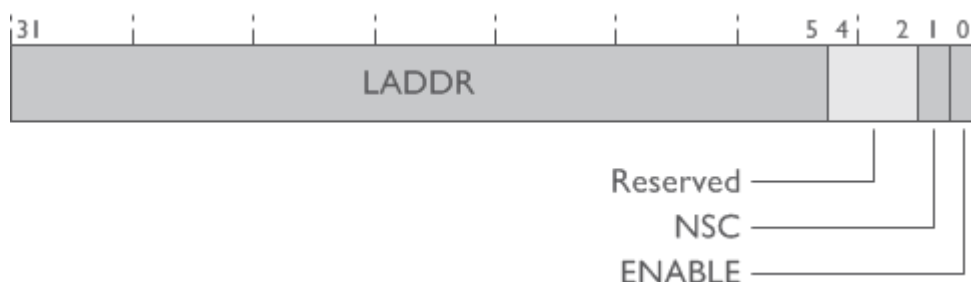
**Figure 3-3 SAU\_TYPE register**



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

### SAU\_RLAR register

The following figure and table show the SAU\_RLAR register characteristics:



**Table 3-6 SAU\_RLAR register**

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

### SAU\_RNR register

The following figure and table show the SAU\_RNR register characteristics:



**Table 3-7 SAU\_RNR register**

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

### SAU\_TYPE register

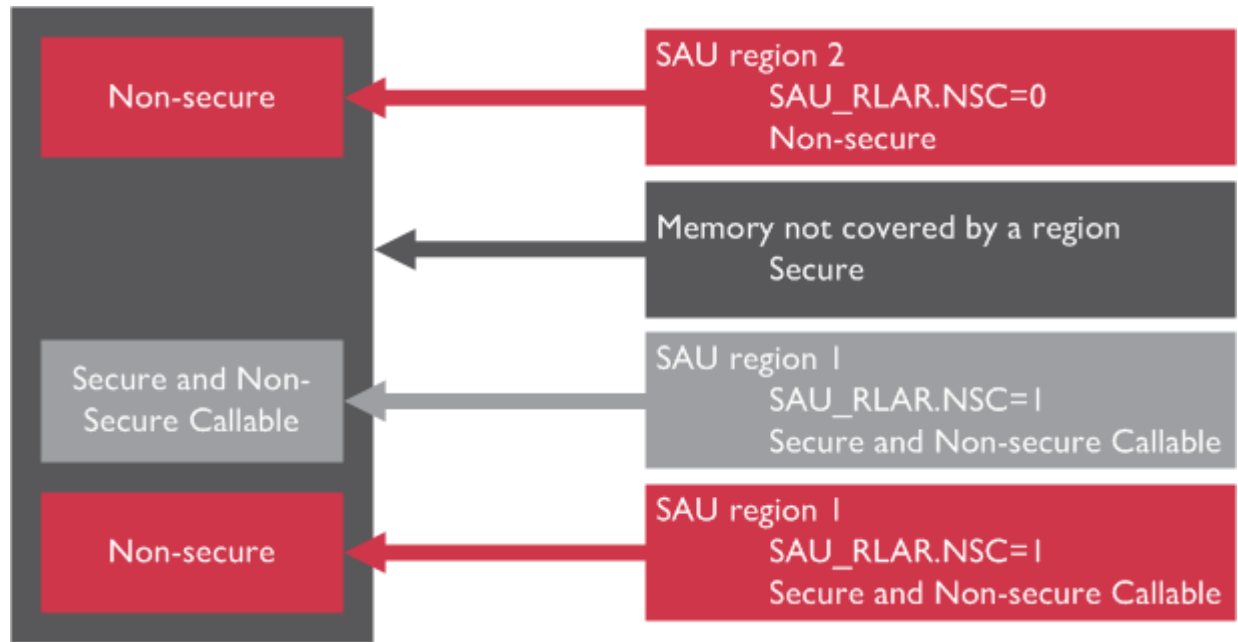
The following figure and table show the SAU\_TYPE register characteristics:



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

### 3.2.1 SAU Region configuration

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

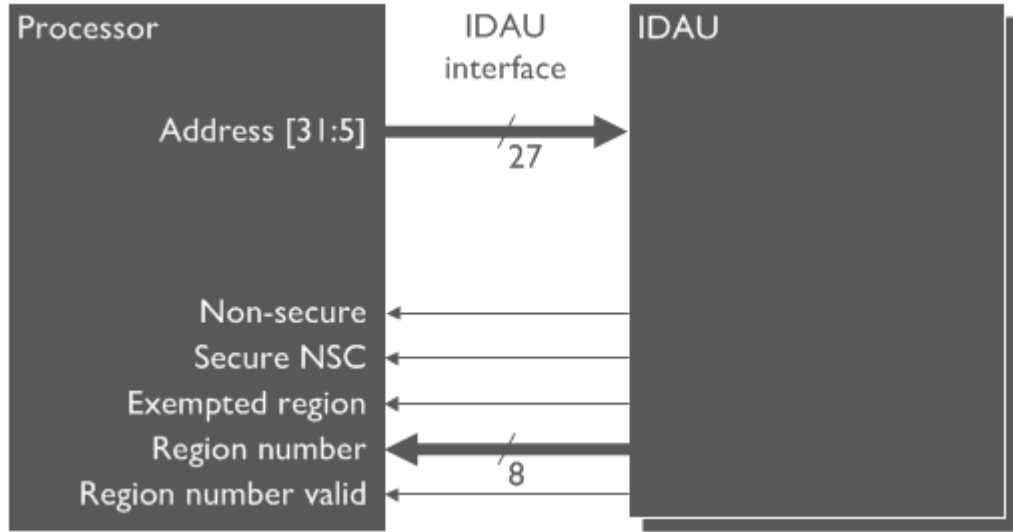


- Regions are enabled individually using SAU\_RLAR.
- The region is Non-secure when SAU\_RLAR.ENABLE = 1 and SAU\_RLAR.NSC=0.
- The region is Secure and Non-secure callable when SAU\_RLAR.ENABLE = 1 and SAU\_RLAR.NSC=1.

### 3.3 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.



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 can result in a higher gate count in the design. As a result, the IDAUs provide simple memory mapping with limited configurability.

#### 3.3.1 Memory map example

ARMv8-M defines a memory map divided on 512MB boundaries. The example memory map adds support for Security by aliasing each of these boundaries at their halfway point. The lower half provides access to a 256MB Non-secure window and 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.

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:

Address	Type	Security
0xFFFFFFFF	Device system	Various (CPU controlled)
0xF0000000		
0xE0000000		
0xD0000000	Device system	Secure
0xC0000000		Non-secure
0xB0000000		Secure
0xA0000000		Non-secure
0x90000000	RAM (WB)	Secure
0x80000000		Non-secure
0x70000000	RAM (WT)	Secure
0x60000000		Non-secure
0x50000000	Device	Secure
0x40000000		Non-secure
0x30000000	SRAM	Secure
0x20000000		Non-secure
0x10000000	Code	Secure
0x00000000		Non-secure

**Note**

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

This 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 *Non-secure Callable* (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 use the internal *Secure Attribution Unit* (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 *Implementation Defined Attribution Unit* (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, the IDAU response signal should be generated based on the system-wide security arrangement. The simple memory map arrangement that is described in this example would be insufficient.