



Armv8-M Memory Model and Memory Protection

Version 1.0

User Guide

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

107565_0100_01_en



Armv8-M Memory Model and Memory Protection User Guide

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	14 December 2022	Non-Confidential	First release

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, 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.

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.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US 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 <https://www.arm.com/company/policies/trademarks>.

Copyright © 2022 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|version 21.0)

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction.....	7
1.1 Memory model.....	7
1.2 Memory Protection Unit (MPU).....	7
1.3 Differences between Armv7-M and Armv8-M MPU.....	8
2. Memory system.....	10
2.1 Memory address space.....	10
2.1.1 Importance of PPB space.....	11
2.2 Memory types and attributes.....	12
2.2.1 Normal memory.....	13
2.2.2 Device memory.....	14
2.3 Memory barriers.....	15
2.3.1 Data Memory Barrier (DMB).....	16
2.3.2 Data Synchronization Barrier (DSB).....	16
2.3.3 Instruction Synchronization Barrier (ISB).....	17
2.3.4 When do you need a DSB followed by an ISB?.....	17
2.3.5 Load-Acquire and Store-Release instructions.....	18
2.4 Alignment behavior.....	20
2.5 Memory endianness.....	20
2.6 Exclusive accesses.....	22
2.7 Caches and memory hierarchy.....	23
2.7.1 Introduction to caches.....	23
2.7.2 Memory hierarchy.....	24
2.7.3 Implications of caches for programmers.....	25
2.8 Tightly Coupled Memory (TCM).....	25
3. Memory protection.....	27
3.1 Memory Protection Unit.....	27
3.2 MPU programmers model.....	30
3.2.1 MPU registers overview.....	30
3.2.2 Configuring an MPU region.....	31
3.3 Attribute indirection.....	33
3.4 PRIVDEFENA bit usage.....	33

3.5 The importance of the HFNMIENA bit.....	33
3.6 Significance of XN and PXN bits.....	34
3.7 Security Extension and MPU.....	34
3.8 Default memory map and MPU.....	35
3.9 MemManage faults in Armv8-M Mainline.....	35
4. Getting started with Armv8-M based systems.....	37
4.1 CMSIS support for MPU.....	37
4.2 Debug tools support.....	38
5. Use case examples.....	42
5.1 Generic Information.....	42
5.1.1 What is inside a program image?.....	42
5.1.2 Memory map.....	45
5.1.3 Tool versions.....	45
5.2 trap_access.....	46
5.2.1 Project structure.....	46
5.2.2 Memory Map and Scatter file definitions.....	47
5.2.3 MPU configurations.....	50
5.2.4 Triggering MemManage faults.....	51
5.2.5 Output in Target Console.....	52
5.3 rtos_context_switch.....	53
5.3.1 Basic Building Blocks.....	54
5.3.2 Putting it all together.....	60
5.3.3 Additional Information.....	63
6. References.....	65
7. Next steps.....	66

1. Introduction

This guide gives an overview of the Armv8-M Memory Model and the Memory Protection Unit (MPU) implemented in Cortex-M processors. This guide uses examples to help explain the concepts it introduces.

This chapter gives an overview of the following topics:

- [Memory model](#)
- [Memory Protection Unit \(MPU\)](#)
- [Differences between Armv7-M and Armv8-M MPU](#)

1.1 Memory model

The Armv8-M architecture supports 32-bit memory addressing and has a 4GB linear address space. The memory space is unified, that is both instructions and data share the same address space. The default memory map or default memory address space is defined by the architecture. The default memory map divides the 4GB address range into a number of regions. For more details, see Chapter B8: The System address map in the [Arm Architecture Reference Manual](#).

Each region within the memory address space has a set of memory attributes and access permissions. These memory attributes include the following:

- Memory type
- Shareability
- Cacheability

Although the pre-defined memory map is fixed, the architecture provides a high degree of flexibility to allow system designers to differentiate their product with different memory types and peripherals.

Memory addresses can be either Little-Endian (LE) or Big-Endian (BE). The Armv8-M architecture supports unaligned data accesses if unaligned access support is enabled using the Configuration Control Register (CCR).

1.2 Memory Protection Unit (MPU)

The Armv8-M architecture supports the optional Protected Memory System Architecture (PMSAv8) as an architecture extension. This extension provides a Memory Protection Unit (MPU) in the Cortex-M processor series. In Arm Cortex-M processors, the number of regions is configurable by silicon designers, and can be up to 16 regions in current Armv8-M Cortex-M processors (Cortex-M33, Cortex-M55, Cortex-M85). The MPU is a programmable device that can define memory access permissions, such as privileged access only, and memory attributes, for example

Cacheability, for different memory regions. The MPU can make an embedded system more robust and in some cases it can make the system more secure by:

- Preventing application tasks from corrupting stack or data memory used by other tasks and the OS kernel
- Preventing unprivileged tasks from accessing certain peripherals that can be critical to reliability or the security of the system
- Defining SRAM or RAM space as non-executable (Execute-never, XN) to prevent code injection attacks
- Using the Privileged Execute-never (PXN) feature to prevent accidental execution of user code and data by the OS

Note:

PXN feature is available only in Armv8.1-M, but not in Armv6-M, Armv7-M and Armv8.0-M architectures.

The MPU must be programmed by privileged software and should be configured and enabled before use. If the MPU is not enabled, then from a software point of view the MPU is not present. The MPU can also be used to define memory attributes such as Cacheability which can be exported to a system-level cache or to other memory controllers. If a memory accesses violates the access permissions defined by the MPU or accesses a memory location that is not defined by an MPU region, then the transfer would be blocked and a MemManage fault exception would be triggered.

More details on Memory Protection Unit will be covered extensively in subsequent sections.

1.3 Differences between Armv7-M and Armv8-M MPU

Although the Memory Protection Unit's operations are conceptually similar between the Armv7-M and Armv8-M architectures, the Armv8-M architecture has a different programmers' model to program the MPU regions compared to the Armv7-M architecture.

The following processors contain an MPU implementation based on the PMSAv7 architecture:

- Cortex-M0+
- Cortex-M3
- Cortex-M4
- Cortex-M7

The following processors contain an MPU implementation based on the PMSAv8 architecture:

- Cortex-M23
- Cortex-M33
- Cortex-M55

- Cortex-M85
- Star

When migrating from Armv7-M-based processors to Armv8-M-based processors, the MPU-related code should be changed to support PMSAv8.

The following table shows the main differences between Armv8-M-based and older MPUs:

Armv6-M and Armv7-M MPU	Armv8-M MPU
The MPU in the ARMv6-M and ARMv7-M architectures requires that an MPU memory region must be aligned to an address which is a multiple of the region size, and that the region size must be a power of two.	In the Armv8-M architecture the start and end address of a region only need to be aligned to a 32 byte boundary. That is, MPU regions can be any size.
MPU regions can overlap. Higher regions number have higher priority when MPU regions overlapped	Regions are not allowed to overlap. As the MPU region definition is much more flexible, overlapping MPU regions are not necessary.
Memory attributes for each region are programmed in the corresponding MPU_RASR register	Memory regions define memory attributes using an index into a set of memory attribute registers.
The concept of sub-regions is widely used within a single MPU region	There is no concept of sub-regions in PMSAv8. Because PMSAv8 gives more flexibility in region address configuration, there is no need to retain the sub-region concept used in Armv6-M- and Armv7-M-based processors

2. Memory system

This chapter describes the following topics:

- [Memory address space](#)
- [Memory types and attributes](#)
- [Memory barriers](#)
- [Alignment behavior](#)
- [Memory endianness](#)
- [Exclusive accesses](#)
- [Caches and memory hierarchy](#)
- [Tightly Coupled Memory \(TCM\)](#)

2.1 Memory address space

The Armv8-M architecture is a memory-mapped architecture. Cortex-M processors support 32-bit memory addressing, which allows 4GB of memory space. The address is 2^{32} bytes, but typically the memory system is at least 32-bit wide (word size). Each of those addresses is word-aligned, meaning that the address is divisible by 4.

Imagine a word with a word-aligned address that we will call A . This word consists of the four bytes with addresses A , $A+1$, $A+2$, and $A+3$. While instruction fetches are always halfword-aligned, some load and store instructions support unaligned addresses. Address calculations are normally performed using ordinary integer instructions. This means that they normally wrap around if they overflow or underflow the address space. This unified memory space can be used to store both instructions and data for a program. All memory addresses used in Armv8-M are physical addresses (PAs). There is no concept of virtual addresses (VAs).

The architecturally predefined 32-bit memory address space is subdivided for code, data, and peripherals, with regions for on-chip and off-chip resources. On-chip refers to resources that are closely coupled to the processor. The address space supports eight primary partitions of 0.5GB each, as follows:

- Code
- SRAM
- Peripheral
- Two RAM regions
- Two Device regions
- System

The default memory map is divided into eight memory segments, which are summarized in the following table:

Region	Address range	Example usage	Default eXecute-Never (XN) attribute
Code	0x00000000 – 0x1FFFFFFF	Memory to hold program image, typically ROM or flash memory	No
SRAM	0x20000000 – 0x3FFFFFFF	Fast SRAM memory, usually on-chip RAM	No
Peripheral	0x40000000 – 0x5FFFFFFF	Peripheral memory space, on-chip	Yes
2x RAM	0x60000000 – 0x9FFFFFFF	Typical RAM memory, usually off-chip RAM	No
2x Device	0xA0000000 – 0xDFFFFFFF	Peripheral memory space, off-chip	Yes
System	0xE0000000 – 0xFFFFFFFF	Contains memory mapped registers and the Vendor system region, Vendor_SYS	Yes

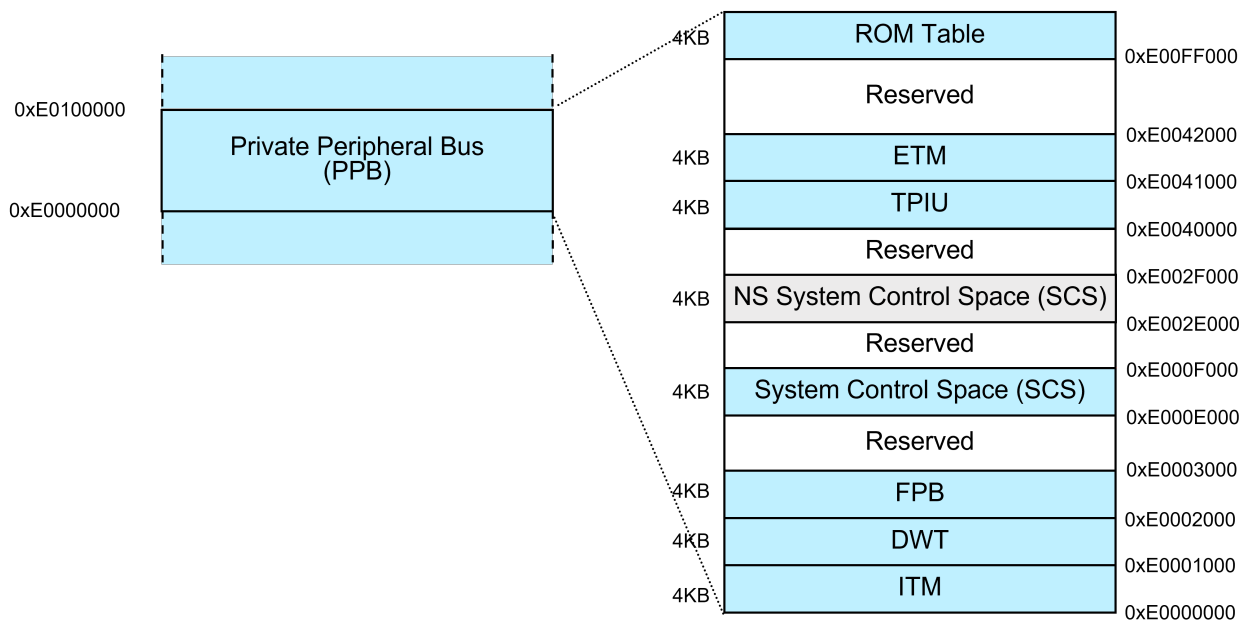
The System region of the memory map, starting at 0xE0000000, subdivides as follows:

- The 1MB region at offset +0x00000000 is reserved as a Private Peripheral Bus (PPB)
- The region from offset +0x00100000 is the Vendor system region, Vendor_SYS

2.1.1 Importance of PPB space

The 1MB region reserved as PPB plays a vital role in the system memory map. The PPB memory region contains register space for supporting key processor resources including NVIC, MPU, and debug features. Accesses to the PPB region are always marked as little-endian only regardless of the processor's endian state. By default, only privileged accesses are allowed to write to or read from the PPB address space.

Figure 2-1: PPB within the System region



Within the PPB memory region, the System Control Space (SCS) is a memory-mapped 4KB address space that provides 32-bit registers for control, configuration, and status reporting. If the Security Extension is implemented, both Secure and Non-secure SCS regions are present. The SCS includes the System Control Block (SCB), which provides configuration registers for the processor. Examples of some of the registers that are part of the SCB include the following:

- System management registers (E.g. System Control Register (SCR), Configuration Control Register (CCR))
- ID registers (E.g. CPUID)
- Fault Status and Address registers
- Vector Table Offset register
- Interrupt Control State register

For more information about the different default memory access attributes corresponding to each of the eight memory regions, see section B8.1 System address map in the [Armv8-M Architecture Reference Manual](#)

2.2 Memory types and attributes

When describing the memory model, a memory access is an instruction fetch from memory, or a load or store data access. A memory access is governed by:

- Whether the access is a read, write, or instruction fetch
- The address alignment
- Data endianness
- Memory attributes

Every memory region has a memory type that affects how the processor can access the addresses in that region. The two memory types are Normal memory and Device memory, and they are mutually exclusive. Additionally, memory attributes are specified to control:

Memory attribute	Description
Access Permission	Restricts whether an address is read/write or read-only and also whether it is only accessible in privileged mode
Execute Permission	Determines whether code can be executed from a memory region. This is referred as eExecute Never attribute, see default memory map table in Section:Memory address space
Sharability	The sharability attribute is used in multiprocessor systems. However, this attribute is not generally used with Cortex-M processors.
Cacheability	The memory attribute settings can support two cache levels: inner cache and outer cache. Both inner and outer cache memories can have their own unique cache policies (for example, write-through cache or write-back cache). When a region is marked as cacheable, it means that the data may be cached but not that it must be cached. Device memory is not cacheable.

2.2.1 Normal memory

Address regions defined as Normal memory type have the following properties:

- The Processing Element (PE) assumes that read and write accesses can be repeated with no side effects. They return the last value that was written to the accessed resource.
- Accesses can be merged before accessing the target memory system.
- A weakly ordered memory model is implemented. This means that there is no requirement for Normal accesses to complete in order with respect to other Normal and Device accesses. However, the PE must handle dependencies, which sets a series of constraints on the reordering of memory accesses targeting the same address. These constraints are listed and described in the section B7.2.3 Ordering and observability in the [Armv8-M Architecture Reference Manual](#). Additionally, ordering can be enforced if required by using barriers, which will be covered in a later section in this guide.
- Speculative accesses are permitted. Speculative access means that the data or instruction can be fetched before explicitly being referenced. This can occur, for example, when performing branch prediction or speculative cache line fills.



There is no technical limitation of the memory types (e.g. SRAM, DRAM, MRAM) being used as Normal memory.

2.2.1.1 Dependency handling example

Even if Normal memory is weakly ordered, constraints are imposed so that accesses to the same address are correctly handled. For example, in the following code:

```
MOVW r0, #0xAAAA
MOVT r0, #0xBBBB
MOV r1, #0x1000
MOV r2, #0xCC
STR r0, [r1]
STRB r2, [r1, #3]
LDRH r0, [r1, #2]
```

The `STR` instruction stores the word `0xBBBBAAAA` in address `0x1000`. Then, the `STRB` instruction stores a byte `0xCC` in address `0x1003`. Therefore, `STR` and `STRB` must complete in the order in which they appear in the code, so that `LDRH` returns the most up-to-date halfword value `0xCCBB` from address `0x1002`.

Normal memory is less restrictive than Device memory and therefore offers better performance.

2.2.2 Device memory

Device memory is a memory type that is assigned to regions of memory where accesses can have side effects. Some of the main features of Device memory are:

- It is not cacheable.
- It is always treated as shareable.
- It does not support speculative data accesses.
- Any unaligned access to Device memory generates an UNALIGNED UsageFault exception.

It is recommended that Device regions are marked as execute-never, because speculative instruction fetches would otherwise be permitted, which could corrupt the values of read-sensitive registers. In addition, instruction fetches from Device memory may trigger MemManage faults on some systems.

Device memory is assigned a combination of Device memory attributes. In Armv8-M architecture, there are four variants of Device memory as the result of a limited combination of three different attributes.

The Device memory attributes are:

- Gathering (G) and non-Gathering (nG): Device memory regions, marked with the G attribute, allow multiple accesses of the same type (read or write) to the same or different locations to be merged into a single transaction.
- Reordering (R) and non-Reordering (nR): For regions marked with the nR attribute, accesses to a peripheral must occur in program order.
- Early Write Acknowledgement (E) and no Early Write Acknowledgement (nE): Assigning the nE attribute recommends that only the endpoint of the write access returns a write acknowledgment of the access, and that no earlier point in the memory system (for example, a buffer) returns a write acknowledgment.

Based on the Device memory attributes, Device memory can implement four different variants, which are:

1. Device-nGnRnE
2. Device-nGnRE
3. Device-nGRE
4. Device-GRE

Device accesses ordering example

Consider a memory map with the following three Device regions:

- Region A: Device-nGnRnE
- Region B: Device-GRE
- Region C: Device-nGnRnE

For the following sequence of load instructions:

```
LDR r0, A
LDR r1, B
LDR r2, A+8
LDR r3, C
LDR r4, B+8
```

- Region A is marked as nR, therefore the two accesses to region A are guaranteed to be in order with respect to each other.
- Region B is marked as R, therefore the two loads from region B are not guaranteed to be in program order with respect to each other. Note that none of the accesses to region B are guaranteed to occur in program order with respect to any of the accesses to regions A and C.
- Region C is marked as nR. It is not guaranteed whether the accesses to region A will occur in program order with respect to accesses to region C.

The terms strong and weak memory type are commonly used when describing Device memory. Device-nGnRnE is the strongest memory type, as it defines rules that memory accesses must obey. Therefore, it can also be said that it is the most restrictive Device memory type. On the contrary, Device-GRE is the weakest Device memory type.

The rules of stronger Device memory types are always valid for weaker Device memory types. This means that memory marked with G/R/E attributes may, but not must, be gathered, reordered, or early acknowledged.



In Arm Cortex-M processors based of Armv8-M architecture, Device memory is always treated as shareable. However this behaviour could be different in Cortex-M processors based on Armv6-M/Armv7-M architectures.

2.3 Memory barriers

The Armv8-M architecture supports out-of-order completion of instructions and data accesses to optimize execution. However, the Armv8-M architecture supports memory barriers to ensure that an operation has completed before continuing execution. By using barrier instructions, a PE can guarantee completion of any preceding load and store instructions and flush any prefetched instructions.

The most common barrier instructions are:

- Data Memory Barrier (DMB)
- Data Synchronization Barrier (DSB)
- Instruction Synchronization Barrier (ISB)

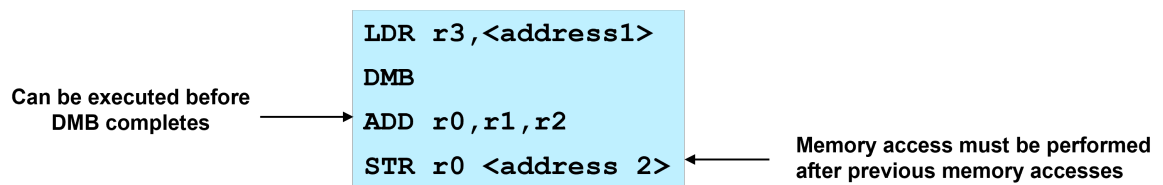
In Armv8.1-M architecture with Reliability, Availability and Serviceability (RAS) extension, Error Synchronization Barriers (ESB) are introduced. For additional information on the different memory

barriers, see section B7.2.9 Memory barriers and section B16.4 in the [Armv8-M Architecture Reference Manual](#).

2.3.1 Data Memory Barrier (DMB)

Data Memory Barriers are used to ensure that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. Note that `DMB` applies only to memory access instructions, not to any other instructions, as shown in the following figure:

Figure 2-2: Data Memory Barrier

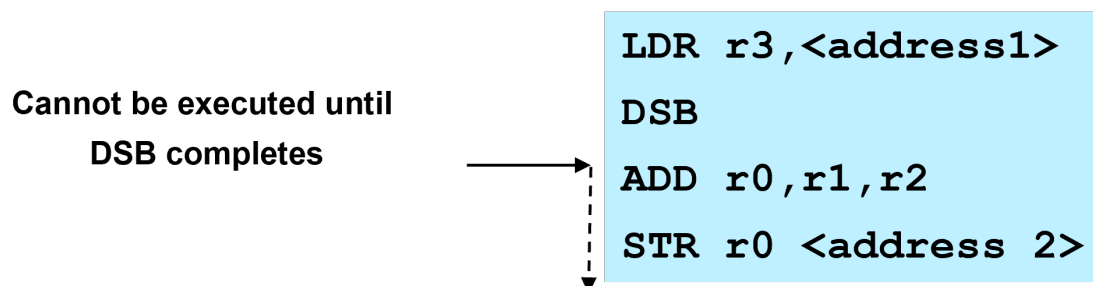


2.3.2 Data Synchronization Barrier (DSB)

A Data Synchronization Barrier ensures that memory accesses before the `DSB` instruction have completed before the completion of the `DSB` instruction.

Any instruction that appears in program order after the `DSB` cannot execute until the `DSB` completes. The `DSB` is a "stronger" barrier than the `DMB`.

Figure 2-3: Data Synchronization Barrier



2.3.3 Instruction Synchronization Barrier (ISB)

An Instruction Synchronization Barrier ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed.

Using an `ISB` instruction guarantees that the effects of context-changing operations (for example, changing CONTROL register bits) executed before the `ISB` are visible to the instructions fetched after the `ISB`.

Performing an `ISB` operation flushes the pipeline in the PE and is a context synchronization event.



The implicit context synchronization events include Exception Entry, Exception Exit, Debug Entry, and Debug Exit

2.3.4 When do you need a DSB followed by an ISB?

Here are few example scenarios where you need a Data Synchronization Barrier (`DSB`) followed by an Instruction Synchronization Barrier (`ISB`).

1. MPU configuration:

A `DSB` is used after enabling the MPU to ensure that the subsequent `ISB` instruction is executed only after the write to the MPU Control register is completed. The `ISB` instruction is used after the `DSB` to ensure the processor pipeline is flushed and subsequent instructions are re-fetched with new MPU configuration settings.

2. Enable or disable the floating-point unit (FPU):

Before using an FPU, you need to program CPACR (Coprocesor Access Control Register) to enable the FPU. The CPACR register lets you enable or disable the FPU. Because writing to the CPACR register affects subsequent floating-point instructions, a `DSB` instruction is executed to ensure that the write to the CPACR register is completed. The `ISB` instruction is used after the `DSB` to ensure that new FPU settings are applied to subsequent floating-point instructions.

3. Enabling interrupts using NVIC:

If a pended interrupt request needs to be recognized immediately after being enabled in the NVIC, a `DSB` instruction followed by an `ISB` instruction is recommended. The `DSB` instruction ensures that the write to the NVIC enable register is complete, while the `ISB` instruction ensures that IRQ is executed.

4. Vector table configuration:

In Cortex-M processors, typically the location of the vector table is determined by the Vector Table Offset Register (VTOR). If you need to change the vector table base address, then a `DSB` instruction should be used after writing a new value to the VTOR register. This ensures that the

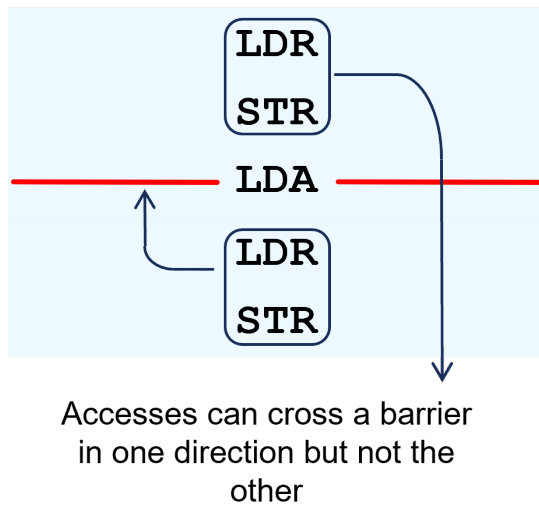
write to the VTOR register is complete. An `ISB` followed by a `DSB` is required to ensure that any subsequent exceptions and interrupts use the new vector table base address.

2.3.5 Load-Acquire and Store-Release instructions

Load-Acquire and Store-Release instructions are load and store instructions with implicit barrier semantics.

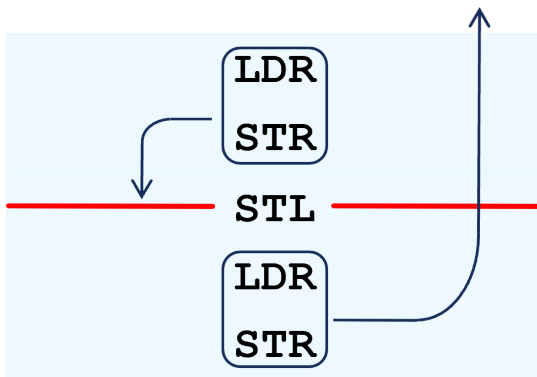
A Load-Acquire (`LDA`) instruction ensures that all reads and writes caused by loads and stores that appear in program order after the `LDA` are observed after the `LDA`. However, accesses before the `LDA` are not affected.

Figure 2-4: Load-Acquire



A Store-Release (`STL`) instruction ensures that all reads and writes caused by loads and stores that appear in program order before the `STL` are observed before the `STL`. However, accesses after the `STL` are not affected.

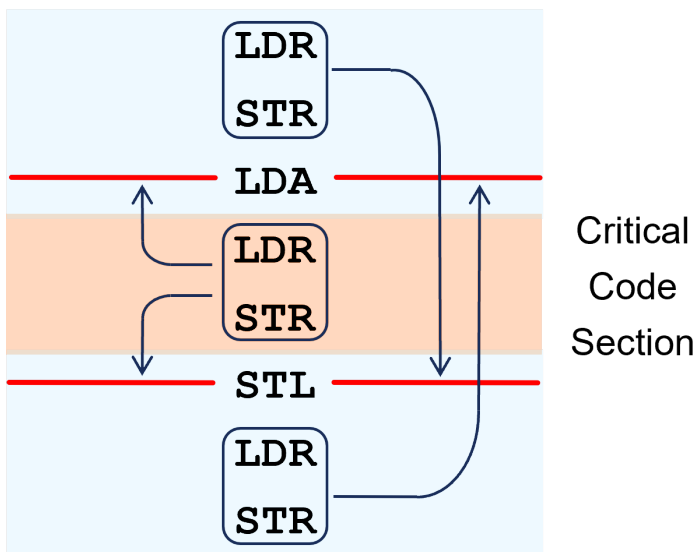
Figure 2-5: Store-Release



Accesses can cross a barrier in one direction but not the other

LDA and STL instructions can be combined to protect critical sections of code, as shown in the following figure:

Figure 2-6: Combining LDA and STL to protect critical code



Ordering is not enforced within the critical section of code.

2.4 Alignment behavior

In Armv8-M, all instruction fetches are halfword-aligned.

Unaligned data accesses in Normal memory are only supported if the Main Extension is implemented, otherwise they generate an alignment HardFault. Accesses to Device memory must always be aligned.

Specific instructions always generate alignment faults for some unaligned data accesses. For example, `LDAH` (Load-Acquire Halfword) and `STLH` (Store-Release Halfword) instructions always generate an alignment fault for non halfword-aligned data accesses.

Additionally, some instructions generate alignment faults for some unaligned data accesses when CCR(Configuration Control Register).UNALIGN_TRP bit is set to 1. For example, non-word-aligned data accesses using the `LDR{T}` and `STR{T}` instructions generate an alignment fault if CCR.UNALIGN_TRP is set to 1.

For more detailed information on alignment behavior, see the section B7.6 Alignment behavior in the [Armv8-M Architecture Reference Manual](#).

2.5 Memory endianness

Cortex-M processors are used in either Little-Endian or Big-Endian memory system. The following figures show, for both little-endian and big-endian, the relationship in memory between:

- The word at address A .
- The halfwords at addresses A and $A+2$.
- The bytes at addresses A , $A+1$, $A+2$, and $A+3$.

Figure 2-7: Little-endian format

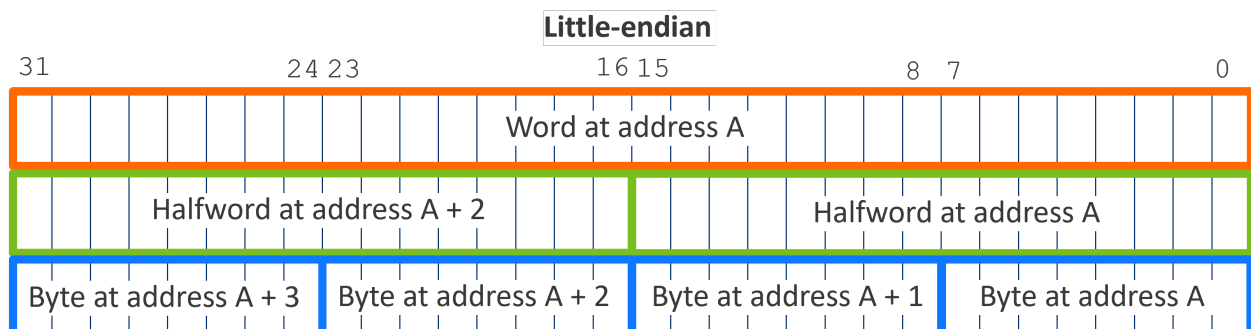
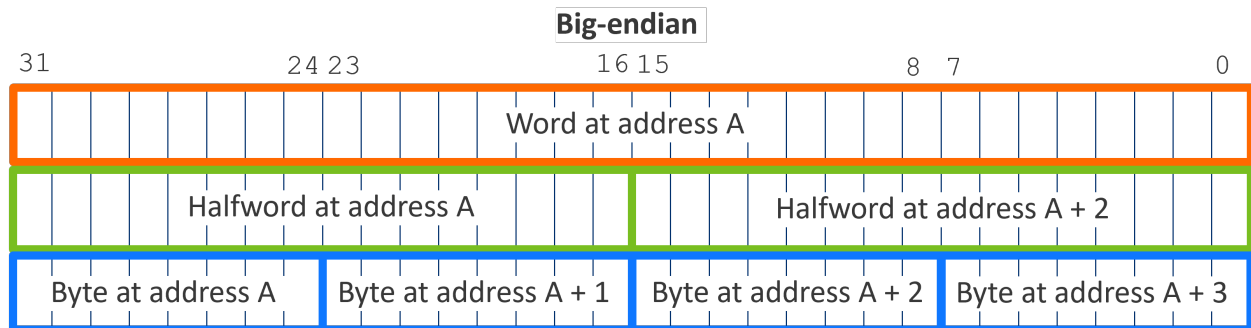


Figure 2-8: Big-endian format



In a little-endian memory system the least significant byte is at the lowest address. In a big-endian memory system the most significant byte is at the lowest address.

The endianness of data accesses is indicated by Application Interrupt and Reset Control Register(AIRCR). AIRCR.ENDIANNESS bit can either implemented with a static value or configured by a hardware input on reset. In Arm Cortex-M processors, Endianness cannot be changed at runtime.

The endian mapping has following restrictions:

- The endianness setting only applies to data accesses. Instruction fetches are always little-endian.
- All accesses to the Private Peripheral Bus (PPB) are always little-endian.



Note

In most of the Cortex-M based systems, the Endianness of the memory system is fixed, and majority of the Cortex-M systems on the market use Little Endian memory system.

Endianness example

Consider the following code. We want to determine the final value in the `r2` register, depending on the endianness of the system.

```
MOV r0, #0x4223056C
MOV r1, #0x1000
STR r0, [r1]
LDRB r2, [r1]
```

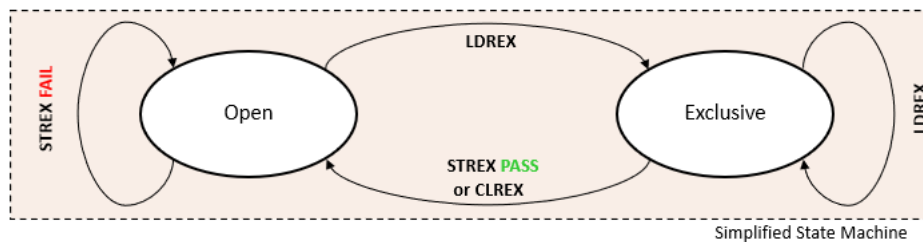
- In the case of little-endian, the value `0x6C` is stored in the least significant byte in the memory address `0x1000`. `LDRB` loads the least significant byte in address `0x1000`. Therefore the final value in `r2` is `0x6C`.
- In the case of big-endian, the value `0x42` is stored in the least significant byte in the memory address `0x1000`. `LDRB` loads the least significant byte in address `0x1000`. Therefore the final value in `r2` is `0x42`.

2.6 Exclusive accesses

Semaphores are commonly used for allocating shared resources to applications. When a shared resource can only be safely accessed by one thread at a time, it is commonly called Mutual Exclusion (MUTEX). In such cases, when a resource is being used by one process, it is locked to that process and cannot serve another process until the lock is released. To create a MUTEX semaphore, a memory location is defined as the lock flag to indicate whether a shared resource is locked by a process. When a process or application wants to use the resource, it needs to check whether the resource has been locked first. If it is not being used, then it can set the lock flag to indicate that the resource is now locked. In traditional Arm processors, the access to the lock flag is carried out by a single `swp` instruction. It allows the lock flag read and write to be atomic, preventing the resource from being locked by two processes at the same time. To overcome this situation, exclusive access instruction pairs (`LDREX/STREX`, `LDREXB/STREXB`, and `LDREXH/STREXH`) were introduced.

The exclusive state machine operates in a pair of instructions. Out of reset, the exclusive state machine is always in `open` state. On executing the `LDREX` instruction, the state machine enters `Exclusive` state. An `STREX` instruction can successfully write into a memory location (marked as lock flag) only when the state machine is in `Exclusive` state. This exclusive state machine is depicted in the following figure:

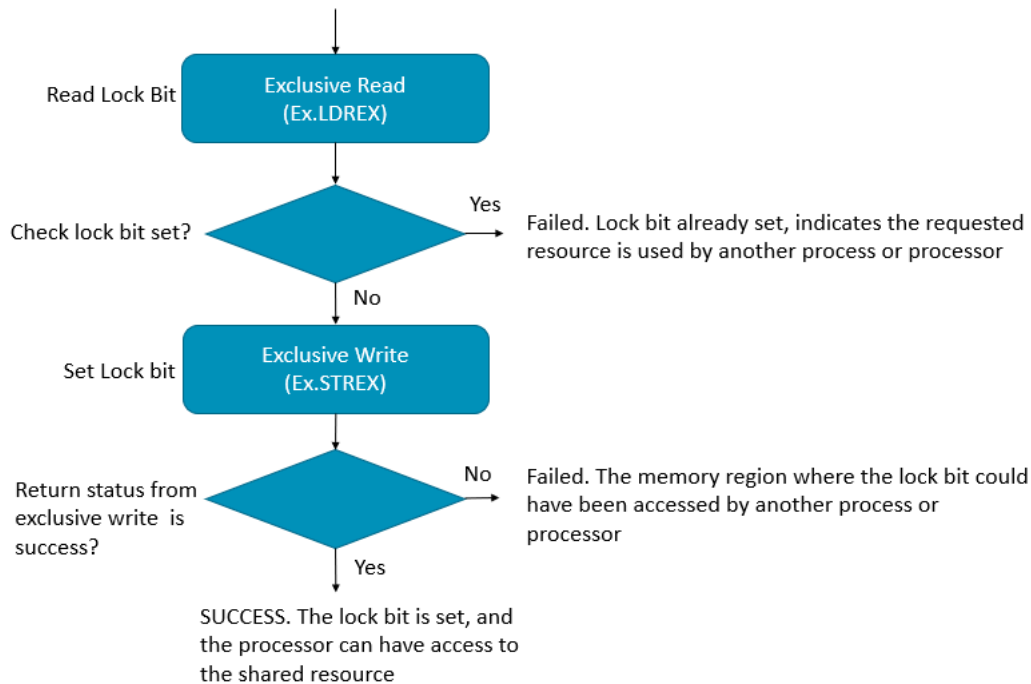
Figure 2-9: Exclusive State Machine



The exclusive write (for example `STREX`) can fail in the following conditions:

- An `LDREX` instruction was not executed before an `STREX` instruction
- A `CLREX` instruction was executed
- A context switch operation occurred, resulting in clearing the exclusive state

The sequence to acquire a lock using exclusive instructions is shown in the following diagram:

Figure 2-10: Using exclusive access in a semaphore

Exclusive access instructions allow software to create atomic behaviors in software. However, in hardware level the bus transfers are not forced into a locked sequence. Exclusive accesses are preferable than using locked bus transfers because they have much lower impact to the interrupt latency.

2.7 Caches and memory hierarchy

The Armv8-M architecture defines support for caches within the architecture and with memory attributes. Memory attributes can be exported on a supporting bus protocol such as AMBA (AHB or AXI protocols) to support system caches. In situations where a breakdown in coherency can occur, software must manage the caches using cache maintenance operations that are memory mapped.

2.7.1 Introduction to caches

A cache is a block of high-speed memory locations containing both address information (commonly known as a TAG) and the associated data. The purpose is to increase the average speed of a memory access. Caches operate on two principles of locality:

Spatial locality

An access to one location is likely to be followed by accesses from adjacent locations, for example, sequential instruction execution or usage of a data structure.

Temporal locality

An access to an area of memory is likely to be repeated within a short time period, for example, execution of a code loop.

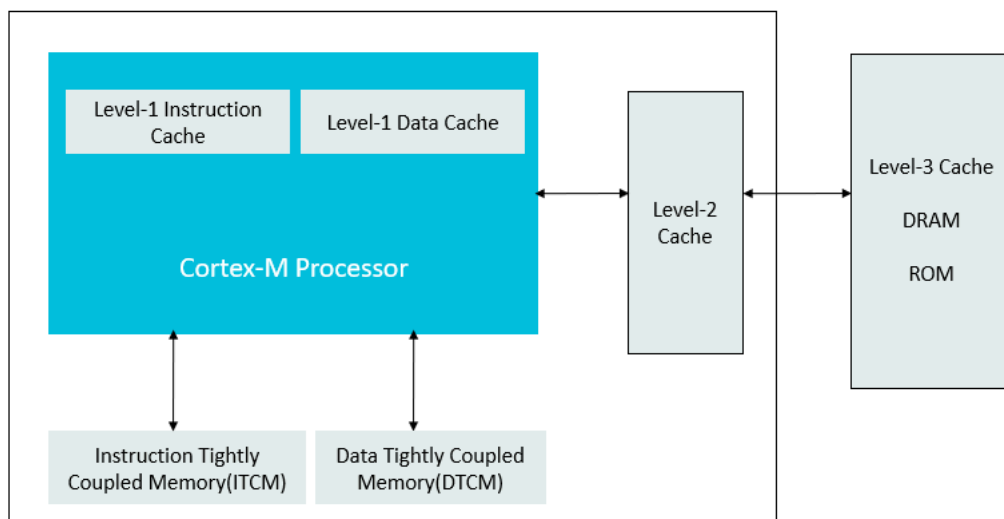
To minimize the quantity of control information stored, the spatial locality property is used to group several locations together under the same tag. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses. Normally, caches are self-managing, with the updates occurring automatically. Whenever the processor wants to access a cacheable location, the cache is checked. If the access is a cache hit, the access occurs immediately, otherwise a location is allocated and the cache line loaded from memory.

2.7.2 Memory hierarchy

Memory close to a processor has very low latency, but is limited in size and expensive to implement. Further from the processor it is easier to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8-M memory system can include multiple levels of cache in a hierarchical memory system.

The following figure shows an example system with multiple levels of caches.

Figure 2-11: Memory hierarchy example



2.7.3 Implications of caches for programmers

Caches are largely invisible to the application programmer, but can become visible due to a breakdown in coherency. Such a breakdown can occur when:

- Memory locations are updated by other agents in the systems.
- Other bus managers in the system are not aware of memories updated by the applications and ended up using outdated data.

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
 - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
 - Not enabling caches in the system.
- By using cache maintenance operations to manage the coherency issues in system software.
- By using sophisticated hardware coherency mechanisms implemented at a system level to ensure the coherency of data accesses to memory for cacheable locations by observers within the different Shareability domains.

If software requires coherency between instruction execution and memory, it must manage this coherency using the `ISB` and `DSB` memory barriers and cache maintenance operations.

2.8 Tightly Coupled Memory (TCM)

TCM is designed to provide low-latency memory that can be used by the processor. This memory can be used to hold critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache would be highly undesirable. In addition, you can use TCM to hold ordinary variables, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks. A TCM is physically located very close to the processor core. Accesses to the TCM will typically be configured to capture or return data in a single cycle. By storing time-critical routines such as exception handlers in the TCM, the processor can have immediate access to the subroutine rather than having to wait for an initial code fetch from external memory.

TCM is expected to be used as part of the physical memory map of the system, and is not expected to be backed by a level of external memory with the same physical addresses. Particular memory locations must be contained either in the TCM or in the cache. In particular, no coherency mechanisms are supported between the TCM and the cache. This means that it is important when allocating the base address of the TCM to ensure that the same address ranges are not contained in the cache.

For example in the Cortex-M7 processor, the memory system includes support for TCM. If TCM is implemented, then TCM ports connect a low-latency memory to the processor, and the TCM ports provide Instruction TCM (ITCM) and Data TCM (DTCM) interfaces. ITCM is a 64-bit memory interface and DTCM is two 32-bit memory interfaces (D0TCM and D1TCM). Each

TCM has a fixed base address and a size for each TCM should be configured during the reset initialization routine (TCM settings can also be defined by silicon designers and not necessary to program TCM configuration registers). Typically, RAM or RAM-like memory (for example SRAM or FRAM) are connected to the TCM port, that is Normal-type memory in Arm architecture. For best performance, DTCM is typically used to store critical variables and frequently updated variables and ITCM is typically used to access critical functions, exception vector table(s), and interrupt service routines. However, there is no functional restriction regarding which TCM should be used to place code and data. Note that the memory systems in the Cortex-M7/M55/M85 processors are designed in such a way that the addresses that target TCM will reach only the ITCM/DTCM interface while the remaining address range (which does not fall under physical TCM address) will by default go through the AXI interface. Compiler-generated code will typically perform data loads from the ITCM to access literal pool data and other constant values in the program image.



It is important to note that Tightly Coupled Memory and its interface are not defined by the Armv8-M architecture. However, Cortex-M processor implementations which include TCM have the necessary enable and control registers for TCM. For example, for Cortex-M55, ITCMCR and DTCMCR are used to enable the TCM.

If TCMs are enabled out of reset, to identify whether TCMs are available in your memory system, you can read `ID_MMFR0` (Memory Model Feature Register 0) register and check that bits [19:16] read as 4'b0001.

Please see the use case example [TCM for latency critical code] in [Use case examples](#) to see how TCM is used to place a critical section of code.

3. Memory protection

Memory protection is used to restrict access to code and data depending on the specific execution context.

For example, an Operating System (OS) commonly runs in privileged mode. In privileged mode, OS code can be executed and OS data can be accessed. Additionally, application memory space can be accessed. However, when executing an application, the application should not be able to access OS data or code. Therefore, applications should be considered unprivileged and have restricted access to specific memory regions. This is one basic example to illustrate the need of memory protection mechanisms. Please see the use case example [rtos_context_switch](#) in [Use case examples](#) that shows how to create a very simple RTOS (Real Time Operating System) capable of dealing with context switching and thread isolation. If memory protection is supported in the processor, illegal memory accesses can be trapped before leaving the processor.

This chapter describes the following:

- Memory Protection Unit (MPU)
- MPU programmers model
- MPU registers overview
 - The significance of XN and PXN bits
 - Where can the PRIVDEFENA bit be used?
 - The importance of the HFNMIENA bit
- Programming memory regions with the MPU
- MPU faults and categories

3.1 Memory Protection Unit

The MPU is an optional component in Cortex-M processor systems. In systems that require high reliability, the MPU can protect memory regions by defining access permissions for different privilege states. The MPU can also define other memory attributes such as Cacheability, which can be exported to system-level cache unit or memory controllers.

In systems without an embedded OS, the MPU can be programmed to have a static configuration. The configuration can be used for functions including the following:

- Setting a RAM/SRAM region to be read-only to protect important data from accidental corruption
- Making a portion of RAM/SRAM space at the bottom of the stack inaccessible to detect stack overflow
- Setting a RAM/SRAM region to be XN to mitigate code injection attacks
- Defining memory attribute settings that can be used by system level cache or memory controllers

In systems with an embedded OS, the MPU can be programmed at each context switch so that each application task can have a different MPU configuration. In this way, you can:

- Define memory access permissions so that stack operations of an application task can only access their own allocated space, to prevent stack corruption of other stacks in case of a stack leak
- Define memory access permissions so that an application task can only have access to its own data and a limited set of peripherals

The Protected Memory System Architecture (PMSA) is the architecture that defines the operation of the MPU for Cortex-M processors. With the development of the Armv8-M architecture, the PMSA has been updated to the PMSAv8. The MPU programmers' model allows privileged software to define memory regions and assign memory access permissions and memory attributes to each of them. The number of supported MPU regions can vary across devices. Typically eight MPU regions are implemented in most systems.

The MPU is software programmable and can be configured in a number of different ways using the MPU regions available on a device. The MPU monitors instruction fetches and data accesses from the processor and triggers a fault exception when an access violation is detected. If the MPU is enabled, and if a memory access violates the access permissions defined by the MPU or if an access to a region which is not programmed by MPU is attempted, then the transfer would be blocked and a MemManage fault would be triggered. Some examples of MPU violations that can result in MemManage faults include the following:

- The address accessed matches more than one MPU region.
- The transaction does not match all of the access conditions for the MPU region being accessed.
- The address accessed only matches the system address map and the system address map is not enabled.

Armv8-M implementations with the Main Extension have a dedicated Memory Management Fault (MemManage) that is triggered by accesses that violate the MPU configuration settings. The Main Extension also provides the MemManage Fault Status Register (MMFSR) and the MemManage Fault Address Register (MMFAR) which provide information about the cause of the fault and the address being accessed in the case of data faults. These provide useful information to RTOS kernels that isolate memory on a per-thread basis, or provide on-demand stack allocation. If the MemManage fault is disabled or cannot be triggered because the current execution priority is too high, then the fault is escalated to a HardFault. Armv8-M implementations without the Main Extension can only use the HardFault exception.

For each MPU region, the following information is needed for a complete description of memory space allocated for that MPU region:

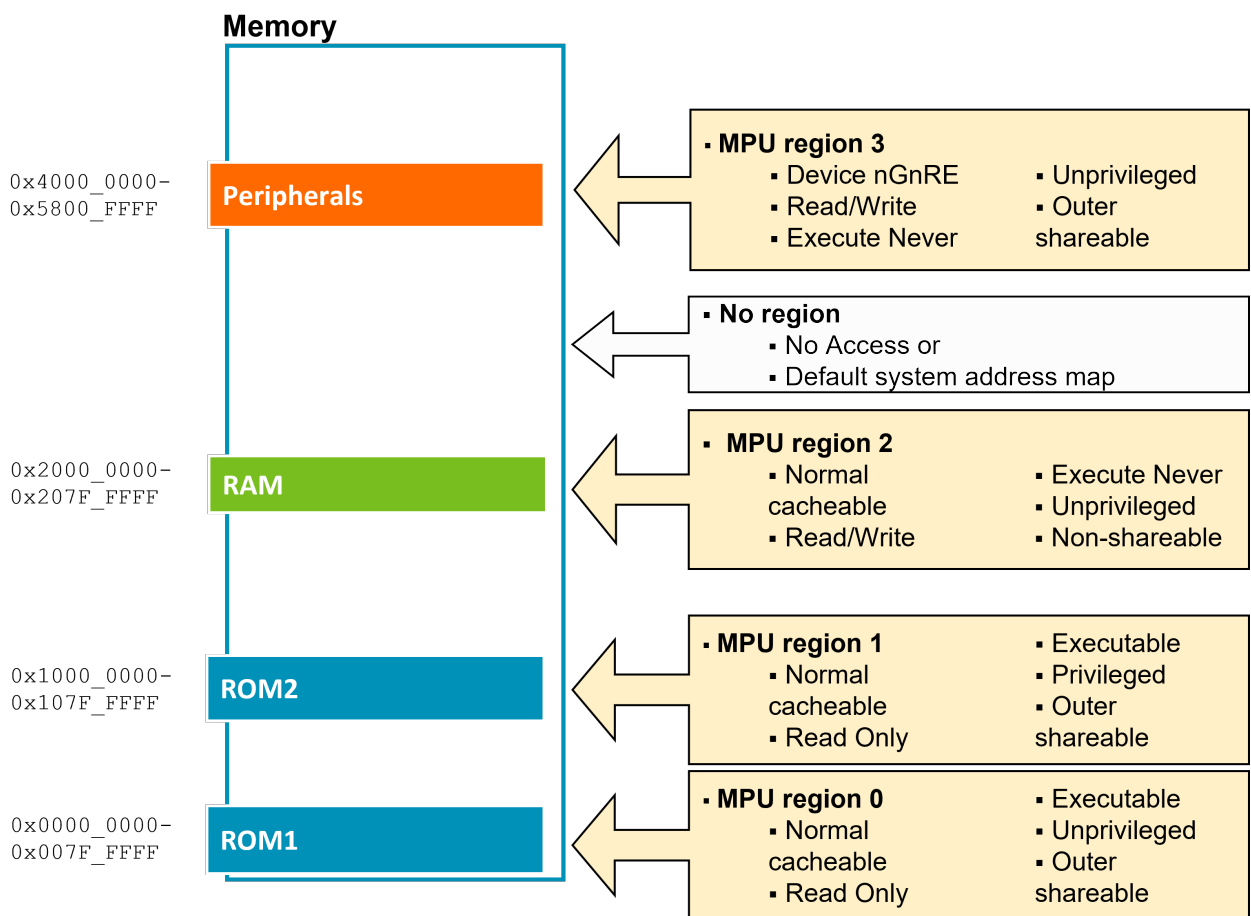
- Base and limit addresses
- Memory type: Normal or Device
- Read-only or Read-Write
- Privileged or unprivileged
- Execution permission

- Shareability
- Cacheability, for Normal memory
- Device attributes, for Device memory

The MPU must be enabled after being programmed. If the MPU is not enabled, then the processor does not have visibility of MPU configuration. For each region, the memory attributes define the ordering and merging behaviors of that region, as well as caching and buffering attributes.

The following figure shows an example of a memory map with four MPU regions and the attributes for each:

Figure 3-1: MPU regions example



Note

With the MPU enabled, an access to an address that does not match any MPU regions (no region in the above figure) will fault unless the access is privileged and MPU_CTRL.PRIVDEFENA is set (see [MPU_CTRL, MPU Control Register]). In this case the default memory map would be applied.

3.2 MPU programmers model

The Armv8-M MPU supports a configurable number of programmable regions with a typical implementation supporting between zero and eight regions per security state.

- The smallest size that can be programmed for an MPU region is 32 bytes.
- The maximum size of any MPU region is 4GB, but the MPU region size must be a multiple of 32 bytes.
- All regions must start at a 32 byte-aligned address.
- Regions can have separate read/write access permissions for privileged and unprivileged code.
- The execute-never (XN) attribute enables separation of code and data regions.

The MPU is configured by a series of memory mapped-registers in the System Control Space (SCS).

3.2.1 MPU registers overview

To program MPU regions and enable the MPU, it is necessary to know which registers should be configured. Note that the MPU registers are memory-mapped. The following table shows a summary of all the MPU registers available in the MPU programmers' model.

Address	Register name	Type	Description	Notes
0xE000ED90	MPU_TYPE	RO	MPU Type Register	The MPU Type register indicates how many regions the MPU supports for the selected security state. This register is read-only
0xE000ED94	MPU_CTRL	RW	MPU Control Register	The MPU Control register provides various programmable bit fields for MPU enable and features. Refer to the sections PRIVDEFENA bit usage and The importance of the HFNMIENA bit for more information.
0xE000ED98	MPU_RNR	RW	MPU Region Number Register	The MPU Region Number Register selects the region that is accessed by the MPU_RBAR and MPU_RLAR registers
0xE000ED9C	MPU_RBAR	RW	MPU Region Base Address Register	The MPU Region Base Address Register defines the starting address and access permissions of an MPU region
0xE000EDA0	MPU_RLAR	RW	MPU Region Limit Address Register	The MPU Region Limit Address Register defines the end address of an MPU region, region enable, and an indirection index to memory attribute array
0xE000EDA4	MPU_RBAR_A1	RW	MPU Region Base Address Register Alias 1	Provides indirect read and write access to the base address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions
0xE000EDA8	MPU_RLAR_A1	RW	MPU Region Limit Address Register Alias 1	Provides indirect read and write access to the limit address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions
0xE000EDAC	MPU_RBAR_A2	RW	MPU Region Base Address Register Alias 2	Provides indirect read and write access to the base address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions

Address	Register name	Type	Description	Notes
0xE000EDB0	MPU_RLAR_A2	RW	MPU Region Limit Address Register Alias 2	Provides indirect read and write access to the limit address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions
0xE000EDB4	MPU_RBAR_A3	RW	MPU Region Base Address Register Alias 3	Provides indirect read and write access to the base address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions
0xE000EDB8	MPU_RLAR_A3	RW	MPU Region Limit Address Register Alias 3	Provides indirect read and write access to the limit address of the currently selected MPU region selected by MPU_RNR. This helps in faster programming of different MPU regions
0xE000EDC0	MPU_MAIRO	RW	MPU Memory Attribute Indirection Register 0	The MPU Attribute Indirection Register 0 provides four sets of 8-bit memory attributes which can be referenced by AttrIndx in MPU_RLAR to determine the memory attribute for an MPU region. Refer to the section Attribute indirection for more details.
0xE000EDC4	MPU_MAIR1	RW	MPU Memory Attribute Indirection Register 1	The MPU Attribute Indirection Register 1 provides four sets of 8-bit memory attributes, which can be referenced by AttrIndx in MPU_RLAR to determine the memory attribute for an MPU region. Refer to the section Attribute indirection for more details.



Note

MPU_RBAR_A1/2/3 and MPU_RLAR_A1/2/3 are aliases of the MPU_RBAR and MPU_RLAR registers to allow faster programming of different MPU regions. The region number that is selected when using MPU_RBARn and MPU_RLARn is equal to $(\text{MPU_RNR}[7:2] \ll 2) + n$. MPU_RBAR_A1/2/3 and MPU_RLAR_A1/2/3 enable software to program multiple MPU regions quickly without the need to reprogram MPU_RNR every time.

For detailed register descriptions and bit positions, see Section B10.1 Memory Protection Unit in the [Armv8-M Architecture Reference Manual](#).

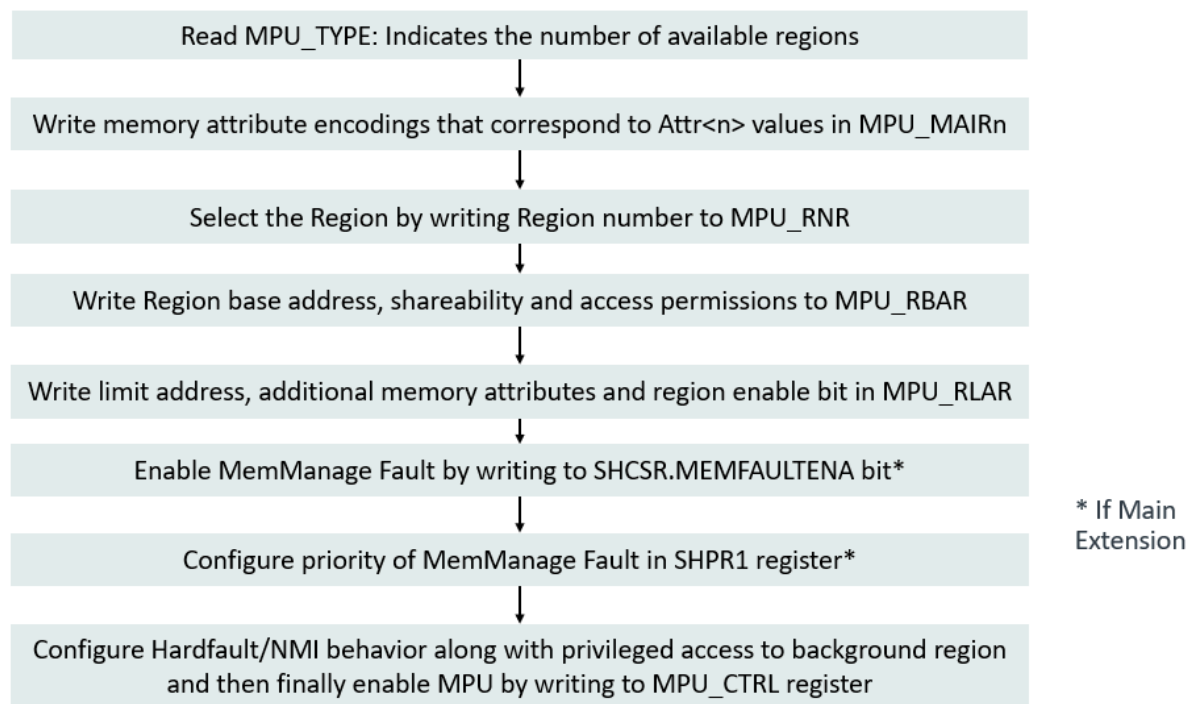
3.2.2 Configuring an MPU region

The MPU must be configured before it is enabled. A Data Memory Barrier (DMB) operation is recommended to force any outstanding writes to memory before enabling the MPU.

The necessary memory types must be encoded into the MAIR registers so that they can be referenced from the MPU_RLAR register for each region. MPU_RNR selects which region MPU_RBAR and MPU_RLAR are currently configuring. The start and end address of each region can be programmed into the MPU_RBAR and MPU_RLAR registers, along with the required access permissions, shareability, and executability.

When all the required regions have been configured, the MPU can be enabled by setting the ENABLE bit in MPU_CTRL. To ensure that any subsequent memory accesses use the new MPU configuration, software must execute a DSB followed by an Instruction Synchronization Barrier (ISB). The following figure summarizes the various stages that are required to configure an MPU region:

Figure 3-2: Configuring memory regions in the MPU

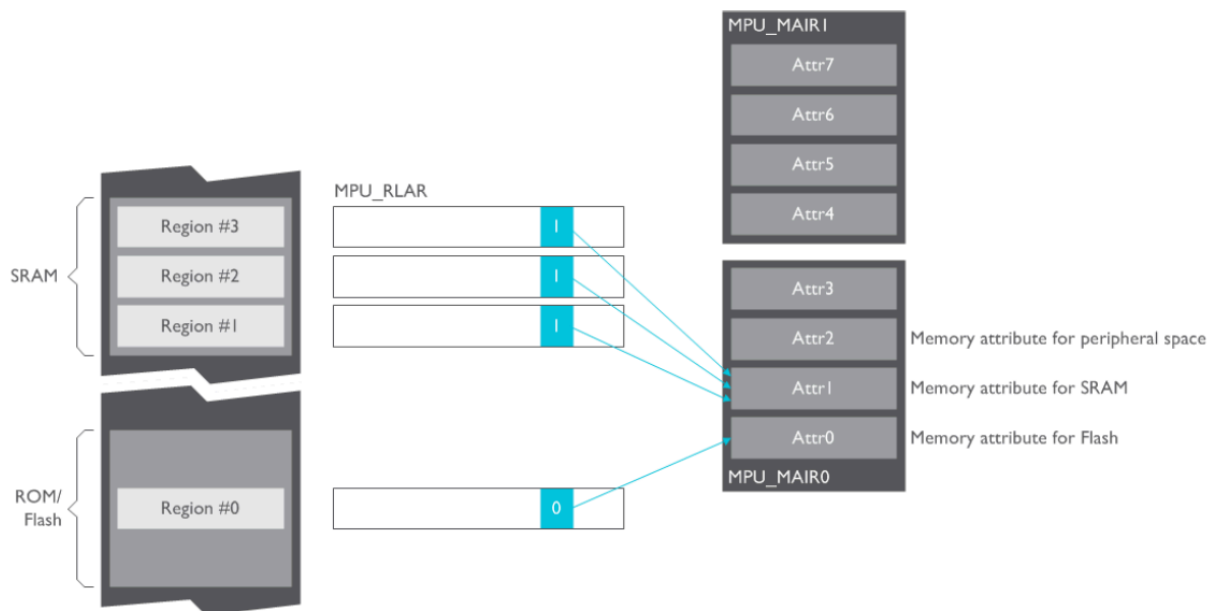


See [Use case examples](#) for actual code sequences used for MPU region programming and configuration settings.

3.3 Attribute indirection

The attribute indirection mechanism allows multiple MPU regions to share a set of memory attributes. For example, in the following figure MPU regions 1, 2 and 3 are all assigned to SRAM, so they can share cache-related memory attributes.

Figure 3-3: MPU attribute indirection



At the same time, regions 1, 2, and 3 can still have their own access permission, XN, and shareability attributes. This is required as each region can have different uses in the application.

3.4 PRIVDEFENA bit usage

The PRIVDEFENA bit in the MPU Control (MPU_CTRL) register is used to enable the background region (for example, region number -1). By using PRIVDEFENA and if no other regions are set up, privileged programs will be able to access all memory locations, and only unprivileged programs will be blocked. However, if other MPU regions are programmed and enabled, they can override the background region. For example, for two systems with similar region setups but only one with PRIVDEFENA set to 1, the one with PRIVDEFENA set to 1 will allow privileged access to background regions. This corresponds to the right-hand side of the diagram shown in [Attribute indirection](#).

3.5 The importance of the HFNMIENA bit

The HFNMIENA bit in the MPU Control (MPU_CTRL) register defines the behavior of the MPU during execution of NMI, HardFault handlers, or when FAULTMASK is set. By default, the MPU is

bypassed (disabled) in these cases. This allows the HardFault handler and NMI handler to execute even if the MPU was set up incorrectly.

3.6 Significance of XN and PXN bits

Execute-never (XN) indicates an execute-never region. The execute-never (XN) configuration is used to prevent certain types of malicious attack from taking control over program execution by inserting their code into memory, for example the Stack.

Any attempt to execute code from an XN region faults, generating a MemManage exception. System space in the default memory map (See B8.1 System address map in the [Armv8-M Architecture Reference Manual](#)) is always marked as Execute-never (XN). A memory region can also be configured as execute-never by the execute-never (XN) bit in the MPU_RBAR register. This bit configuration indicates whether the processor can execute instructions from the MPU region or not. Note that an enabled MPU cannot change the XN property of the System memory region.

As new MPU region attribute called Privileged execute-never (PXN) was introduced in the Armv8.1-M architecture.

If an MPU region is configured with the PXN attribute set, and the processor attempts to execute code in this region while at privileged level, a Memory Management Fault exception is triggered, with IACCVIOL bit in MemManage Fault Status Register set to 1. The PXN attribute bit is in bit 4 of MPU_RLAR (Region Limit Address Register) and its alias registers. It is available in both Secure and Non-secure MPU, and this bit was previously fixed to 0 in Armv8.0-M.

The PXN feature allows privileged software to ensure that specific application tasks (threads) execute only in unprivileged level. For example, a hacker cannot use stack corruption in a privileged peripheral handler to branch into unprivileged code and execute them with privileged level.

This feature is particularly useful for TrustZone enabled systems with Secure firmware components from various software vendors. In those cases, some of the security firmware components might not be fully trusted and need to be restricted to unprivileged execution only. With Armv8.0-M, the unprivileged software components must not have their own Secure entry points which are callable from Non-secure state because the software components would execute in privileged state if being called directly from Non-secure Handler mode. As a result, the entry points need to be implemented separately with security checking which increases software overhead. With the PXN attribute available in Armv8.1-M, these unprivileged software components can safely have their own Secure API entry points.

3.7 Security Extension and MPU

If the Armv8-M Security Extension is implemented, there are two optional MPUs:

1. Secure MPU (MPU_S)
2. Non-secure MPU (MPU_NS)

Within each security state, the MPU can be configured independently. For example, a Cortex-M processor with Security Extension could support four Secure MPU regions in Secure state and eight Non-secure MPU regions in Non-secure state. For more information, refer Armv8-M Security Extension User Guide.

3.8 Default memory map and MPU

The default memory map shown in [Memory system](#) is always applied in the following cases when:

- The MPU is not implemented
- The MPU is implemented but is disabled
- Exception vector reads from the Vector Address Table
- Exception handler is NMI or HardFault or FAULTMASK=1 and MPU_CTRL.HFNMIENA is 0 (see [MPU_CTRL, MPU Control Register]).
- Accesses to the PPB, within the range 0xE0000000–0xE00FFFFF
- Access is privileged, MPU_CTRL.PRIVDEFENA is set and no enabled regions match (see [MPU_CTRL, MPU Control Register]).
- The access is a non-UDE debug access. (UDE - Unprivileged Debug Extension)

Additionally, the MPU's capabilities when programming addresses in System space (addresses 0xE0000000 and higher) are restricted. System space is always execute-never. System space has a default memory type of Device-nGnRE which can only be remapped to Device-nGnRnE using an MPU.

3.9 MemManage faults in Armv8-M Mainline

MemManage faults occur when an MPU mismatch occurs if the Armv8-M Main Extension is implemented. MemManage fault is enabled via SHCSR.MEMFAULTENA configuration settings. When a MemManage fault is taken, we can obtain additional information about the cause of the fault by looking at the contents of the following registers:

- MMFAR, MemManage Fault Address Register: shows the address of the memory location that caused an MPU fault.
- MMFSR, MemManage Fault Status Register: shows the status of MPU faults.
- DEMCR, Debug Exception and Monitor Control Register: manages vector catch behavior and DebugMonitor handling when debugging. It includes MemManage vector catch fields.

The following table shows a summary of the different MemManage faults and their conditions and status bits set in the MMFSR and DEMCR registers when they occur:

Type	MMFSR status bit	DEMCR vector catch bit	Conditions
Data access	DACCVIOL	VC_MMERR	Violation or fault on MPU as result of data access. The address of the access that caused the fault is also recorded in the MMFAR register
Instruction access	IACCVIOL	VC_MMERR	Violation or fault on MPU as result of instruction access
Exception entry stack memory operations	MSTKERR	VC_INTERR	Failure on a hardware save of context, because of an MPU access violation.
Exception return stack memory operations	MUNSTKERR	VC_INTERR	Failure on a hardware restore of context, because of an MPU access violation.
Lazy state preservation error flag	MLSPERR	VC_INTERR	Records whether a MemManage fault occurred during FP lazy state preservation

4. Getting started with Armv8-M based systems

This chapter gives a brief introduction about the platforms, compilers, and tools support available for Armv8-M MPU.



For more details on hardware platforms, simulation model support, and compiler support see the [Introduction to the Armv8-M architecture and its programmers model user guide](#).

4.1 CMSIS support for MPU

CMSIS-Core is part of the Common Microcontroller Software Interface Standard (CMSIS) and provides a standardized API for different aspects of software development for Cortex-M devices, including:

- Startup and initialization code templates.
- Processor core instruction intrinsics.
- Processor core peripheral functions and macros.
- Device-specific system clock and peripheral macros and functions.

CMSIS-Core source code and documentation is available from the following CMSIS GitHub repository:

- [CMSIS Version 5](#)

Few of the popular compilers that CMSIS-Core supports are listed below:

- Arm Compiler 6.
- GNU Arm Embedded Toolchain.
- IAR C/C++ Compiler.

Arm Compiler 6 is available as part of the following products:

- Arm Development Studio (Arm DS).
- Keil Microcontroller Development Kit (Keil MDK).

The fixed system address map and memory-mapped MPU registers in the Armv8-M architecture make it easy to program the MPU in software. To make things even easier for developers to work with the MPU, CMSIS-Core provides ready-to-use MPU functions and macros written in C. For more information, see [MPU Functions for Armv8-M](#) in the [CMSIS-Core \(Cortex-M\)](#) documentation.

The following CMSIS-Core C header files support the MPU:

- [mpu_armv8.h](#)
- [core_armv8mml.h](#)

You can refer to CMSIS-Core C header files corresponding to your Cortex-M processor core as well. For example, if Cortex-M55 processor is used in your system, then refer [core_cm55.h](#)

The following macro must be set appropriately before using these headers to program the MPU:

- `__MPU_PRESENT`

This macro is defined by the CMSIS-Core device header file, which is normally provided by Arm microcontroller device vendors. The device header file is typically available as part of a CMSIS Device Family Pack (DFP) that includes other files, such as the startup and initialization code mentioned in the list of header files above, enabling the user to develop a CMSIS-compliant embedded application. The DFP, which is essentially an archive file, is created by the device vendor.

Arm also acts as a device vendor by providing some device headers and DFPs targeted at its models and platforms described in section Hardware Platform and Simulation Model Support for MPU.

Generic Armv8-M CMSIS-Core device headers can be found at [ARMv8MML.h](#). Depending on your Cortex-M processor selection, the corresponding processor's device header files can be referred. For example, if Cortex-M55 processor is used in your system, then refer [ARMCM55.h](#)

DFPs are supported by different embedded development tools such as Keil MDK and Arm DS. These packs and archives can be downloaded from the following repository on the Arm website.

- [CMSIS Packs on Arm Developer](#)

Also, some IDEs provide an integrated pack installer to make it even easier to download, install, and use the DFPs and the CMSIS-Core files.

4.2 Debug tools support

Designed for the Arm architecture, Arm Development Studio is the most comprehensive embedded C/C++ dedicated software development solution which supports debug for Cortex-M CPUs. Its components include the following:

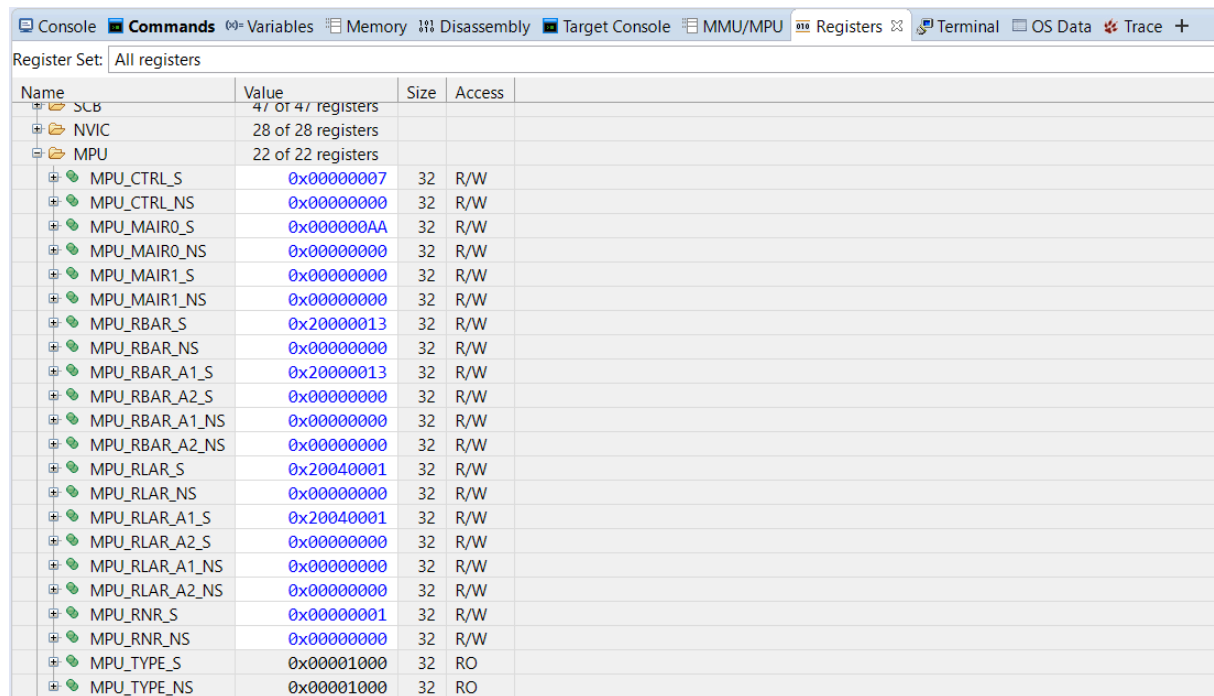
- Arm Compiler for Embedded 6 for compiling bare-metal embedded applications. Includes support for the latest Arm architectures.
- Arm Compiler for Embedded FuSa to accelerate the building of safety-critical code and simplify the TÜV SÜD certification process.
- Complete library of reference Fixed Virtual Platforms (FVPs) along with pre-built examples.
- Entitlement to Keil MDK Professional Edition is included in Silver, Gold, and Platinum editions.

Arm Development Studio 2022.1 has been used to develop the use case examples in this guide. It is not guaranteed that future Arm Development Studio versions correspond completely to the content shown in this guide.

Arm Development Studio features that are particularly relevant for this guide include the following:

- MPU Registers view

Figure 4-1: MPU Registers View in Arm DS



Name	Value	Size	Access
SCB	47 of 47 registers		
NVIC	28 of 28 registers		
MPU	22 of 22 registers		
MPU_CTRL_S	0x00000007	32	R/W
MPU_CTRL_NS	0x00000000	32	R/W
MPU_MAIR0_S	0x000000AA	32	R/W
MPU_MAIR0_NS	0x00000000	32	R/W
MPU_MAIR1_S	0x00000000	32	R/W
MPU_MAIR1_NS	0x00000000	32	R/W
MPU_RBAR_S	0x20000013	32	R/W
MPU_RBAR_NS	0x00000000	32	R/W
MPU_RBAR_A1_S	0x20000013	32	R/W
MPU_RBAR_A2_S	0x00000000	32	R/W
MPU_RBAR_A1_NS	0x00000000	32	R/W
MPU_RBAR_A2_NS	0x00000000	32	R/W
MPU_RLAR_S	0x20040001	32	R/W
MPU_RLAR_NS	0x00000000	32	R/W
MPU_RLAR_A1_S	0x20040001	32	R/W
MPU_RLAR_A2_S	0x00000000	32	R/W
MPU_RLAR_A1_NS	0x00000000	32	R/W
MPU_RLAR_A2_NS	0x00000000	32	R/W
MPU_RNR_S	0x00000001	32	R/W
MPU_RNR_NS	0x00000000	32	R/W
MPU_TYPE_S	0x00001000	32	RO
MPU_TYPE_NS	0x00001000	32	RO

- MemManage Fault Address and Status Registers view

Figure 4-2: MemManage Fault Address and Status Registers View in Arm DS

Register Set: All registers				
Name	Value	Size	Access	
SCB	47 of 47 registers			
AFSR	0x00000000	32	R/W	
AIRCR_S	0xFA050000	32	R/W	
AIRCR_NS	0xFA050000	32	R/W	
BFAR	0x00000000	32	R/W	
CCR_S	0x00040201	32	R/W	
CCR_NS	0x00040201	32	R/W	
CCSIDR	0xF01FE019	32	RO	
CFSR_S	0x00000082	32	R/W	
UFSR	0x0000	16	R/W	
BFSR	0x00	8	R/W	
MMFSR	0x82	8	R/W	
CFSR_NS	0x00000000	32	R/W	
CLIDR	0x09200003	32	RO	
CPACR_S	0x00000000	32	R/W	
CPACR_NS	0x00000000	32	R/W	
CPUID	0x00000000	32	RO	
CSSELR	0x00000000	32	R/W	
CTR	0x8303C003	32	RO	
DFSR	0x00000000	32	R/W	
HFSR	0x00000000	32	R/W	
ICSR_S	0x00000804	32	R/W	
ICSR_NS	0x00000804	32	R/W	
ID_AFR0	0x00000000	32	RO	
ID_DFR0	0x00200000	32	RO	
ID_ISAR0	0x01141110	32	RO	
ID_ISAR1	0x03112000	32	RO	
ID_ISAR2	0x20232232	32	RO	
ID_ISAR3	0x01112131	32	RO	
ID_ISAR4	0x01010142	32	RO	
ID_MMFR0	0x00111040	32	RO	
ID_MMFR1	0x00000000	32	RO	
ID_MMFR2	0x01000000	32	RO	
ID_MMFR3	0x00000000	32	RO	
ID_PFR0	0x00000030	32	RO	
ID_PFR1	0x00000210	32	RO	
MMFAR_S	0x00001000	32	R/W	
MMFAR_NS	0x00000000	32	R/W	

- MPU Regions View

Figure 4-3: MPU Regions View in Arm DS

Console

Commands

Variables

Memory

Disassembly

Target Console

MMU/MPU

Registers

Terminal

OS Data

Trace

+

Tables

Memory Map

Base	Limit	Type	Attributes
MPU (Secure)			ENABLE=1, HFNMIENA=1, PRIVDEFENA=1, MAIR 0=0xA...
0x00000000	0x000031BF	Region	SH=2, AP=3, XN=0, PXN=0, AttrIndex=0, EN=1
0x20000000	0x2004001F	Region	SH=2, AP=1, XN=1, PXN=0, AttrIndex=0, EN=1
0x00000000	0x00000000	Region (x14)	SH=0, AP=0, XN=0, PXN=0, AttrIndex=0, EN=0
MPU (Non-Secure)			ENABLE=0, HFNMIENA=0, PRIVDEFENA=0, MAIR 0=0x0,...
SAU			ALLNS=0, ENABLE=0
IDAU			[Not Configured]

Region

0

Address Range

0x00000000 - 0x000031BF (12 KB)

Memory type

Normal

Shareable

Outer Shareable

Cache (inner)

Write-through, read-allocate, no-write-allocate

Cache (outer)

Write-through, read-allocate, no-write-allocate

Access Permissions (Priv)

RO

Access Permissions (Unpriv)

RO

Execute Never (Priv)

False

Execute Never (Unpriv)

False

Current: MPU

5. Use case examples

This chapter aims to show some user case examples related to MPU configuration.

The use case examples included are:

- `trap_access` : A simple example to show how memory can be protected by the MPU regions
- `rtos_context_switch` : A basic example to show simple real-time kernel context switching operation

The source code for these examples can be found in [GitHub repository](#)

Before getting into the use case examples, let us try to understand generic information required for these example projects.

5.1 Generic Information

In most basic applications, the programs can be completely written in C language. The C compiler compiles the C program code into object files and then generates the executable program image file using the linker.

5.1.1 What is inside a program image?

When a project is built using toolchain, it generates a program image. Inside this program image, in addition to actual scenario that we would want to run, there is also a range of other software components. They are:

- A Vector Table
- A reset handler/startup code
- C Startup code
- C runtime library functions
- Actual application code

Let us try to get a brief understanding of these software components in below sections.

5.1.1.1 Vector table

In Arm Cortex-M processors, the vector table contains the starting addresses of each exception and interrupt. One of the exceptions is reset, which means that after reset, the processor will fetch the reset vector (starting address of the reset handler) from the vector table and start the execution from reset handler. The first word in the vector table defines the starting value of Main

Stack Pointer (MSP). If the vector table is not set up correctly in the program image, the device cannot start.

In Arm-DS project examples shown with this guide, the vector table is defined in device-specific startup code at <example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c. Here is a snippet of vector table that can be found in the startup code.

```
extern const VECTOR_TABLE_Type __VECTOR_TABLE[496];
const VECTOR_TABLE_Type __VECTOR_TABLE[496] __VECTOR_TABLE_ATTRIBUTE = {
    (VECTOR_TABLE_Type)(&__INITIAL_SP), /* Initial Stack Pointer */
    Reset_Handler, /* Reset Handler */
    NMI_Handler, /* NMI Handler */
    HardFault_Handler, /* Hard Fault Handler */
    MemManage_Handler, /* MPU Fault Handler */
    BusFault_Handler, /* Bus Fault Handler */
    UsageFault_Handler, /* Usage Fault Handler */
    SecureFault_Handler, /* Secure Fault Handler */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler, /* SVC Handler */
    DebugMon_Handler, /* Debug Monitor Handler */
    0, /* Reserved */
    PendSV_Handler, /* PendSV Handler */
    SysTick_Handler, /* SysTick Handler */
    /* Interrupts */
    Interrupt0_Handler, /* Interrupt 0 */
    Interrupt1_Handler, /* Interrupt 1 */
    Interrupt2_Handler, /* Interrupt 2 */
    Interrupt3_Handler, /* Interrupt 3 */
    Interrupt4_Handler, /* Interrupt 4 */
    Interrupt5_Handler, /* Interrupt 5 */
    Interrupt6_Handler, /* Interrupt 6 */
    Interrupt7_Handler, /* Interrupt 7 */
    Interrupt8_Handler, /* Interrupt 8 */
    Interrupt9_Handler, /* Interrupt 9 */
    /* Interrupts 10 .. 480 are left out */
};
```

__VECTOR_TABLE - This symbol name is used for defining the static interrupt vector table. The name must comply with any compiler/linker conventions, e.g. if used for vector table relocation. CMSIS-Core specifies common default for supported compilers.



__VECTOR_TABLE_ATTRIBUTE - This symbol name defines the additional declaration specifications to be used when defining the static interrupt vector table.

Both **__VECTOR_TABLE** and **__VECTOR_TABLE_ATTRIBUTE** are expected to be used only by startup file (i.e.) RTE/Device/ARMv8MML/startup_ARMv8MML.c

__INITIAL_SP - The initial stack pointer value is defined in CMSIS Pack files (e.g. cmsis_armclang.h, cmsis_gcc.h).

5.1.1.2 Reset_Handler()

The reset handler or startup code is the first piece of software to execute after a system reset. Typically, the reset handler is used for setting up configuration data for the C startup code (such as address range for stack and heap memories), which then branches into C startup code. Also, it is considered a best practise to initialize stack pointers with its limit before entering C startup code. Since this project uses CMSIS-CORE framework, the reset handler executes `SystemInit()` function which sets up the configuration for clocks and PLLs, before branching to C startup code.

```
__NO_RETURN void Reset_Handler(void)
{
    __set_PSP((uint32_t) (& __INITIAL_SP));
    __set_MSP_LIM((uint32_t) (& __STACK_LIMIT));
    __set_PSP_LIM((uint32_t) (& __STACK_LIMIT));
    #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
        TZ_set_STACKSEAL_S((uint32_t *) (& __STACK_SEAL));
    #endif
    SystemInit(); /* CMSIS System Initialization */
    __PROGRAM_START(); /* Enter PreMain (C library entry point)
    */
}
```



Note

Depending on the development tools being used, the reset handler can be optional. If the reset handler is omitted, the C startup code is executed directly instead. The startup code is typically provided by microcontroller vendors and is also often bundled inside toolchains. They can be either in the form of assembly or C code

5.1.1.3 Default_Handler()

The device-specific startup code at `<example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c` provides weak aliases for each exception handler to `Default_Handler`. (Note that the weak symbol is a special linker symbol that denotes a function that can be overridden during link time.) As the exception handlers are marked as weak aliases, any function with the same name will override this definition. This helps the programmer to define their own handlers without the need to change the device-specific startup code.

For example, consider below code in startup code:

```
void Interrupt0_Handler      (void) __attribute__((weak, alias("Default_Handler")));
```

This code indicates the linker to assign the `Default_Handler` to `Interrupt0_Handler` if a programmer does not provide a `Interrupt0_Handler` function themselves.

Also, if you look through the whole start-up code, you will find similar code for every possible interrupt handler. This allows the code to create a default handler without requiring the programmer to assign specific handler for each interrupt explicitly.

5.1.1.4 C startup code

When using high level languages like C/C++, the processor will need to execute a piece of program code to setup the program execution environment. This includes:

- Setting up the initial data values in SRAM (for e.g. global variables)
- Zero initialization of data memory for variables that are uninitialized at load time
- Initializing the data variables controlling heap memory (for e.g. `malloc()` usage)

After initialization, the C startup code branches to the start of the `main()` program. The C startup code is automatically inserted by the toolchain and is toolchain specific, and is not inserted by toolchain if the program is written in assembly.



For Arm compilers, the C startup code is labeled as `__main`, while the startup code generated by GNU C compilers is normally labeled as `_start`

5.1.1.5 C runtime library functions

C library code is inserted into the program image by the linker when certain C/C++ functions are used. C library code can also be included by way of data processing tasks such as floating point calculations.

5.1.2 Memory map

Example projects use MPS2 FVP (Fixed Virtual Platform) to run the program. The memory map for the MPS2 FVP can be found in [MPS2 - memory map for models with the Arm®v8-M additions](#). It is important to consider the memory map of the platform used to define specific regions for ROM, RAM etc., defined in scatter file (or linker script).

For additional details on scatter file definitions, please read the following:

- [Arm Compiler for Embedded Reference Guide](#)
- [Arm Compiler for Embedded User Guide](#)

5.1.3 Tool versions

This example project is created, built and run using following tool versions.

- Arm Development Studio 2022.1
- Arm Compiler for Embedded 6.18
- Fast Models Fixed Virtual Platforms (FVP) 11.18

- CMSIS 5.8.0 (available in [GitHub repository](#))

5.2 trap_access

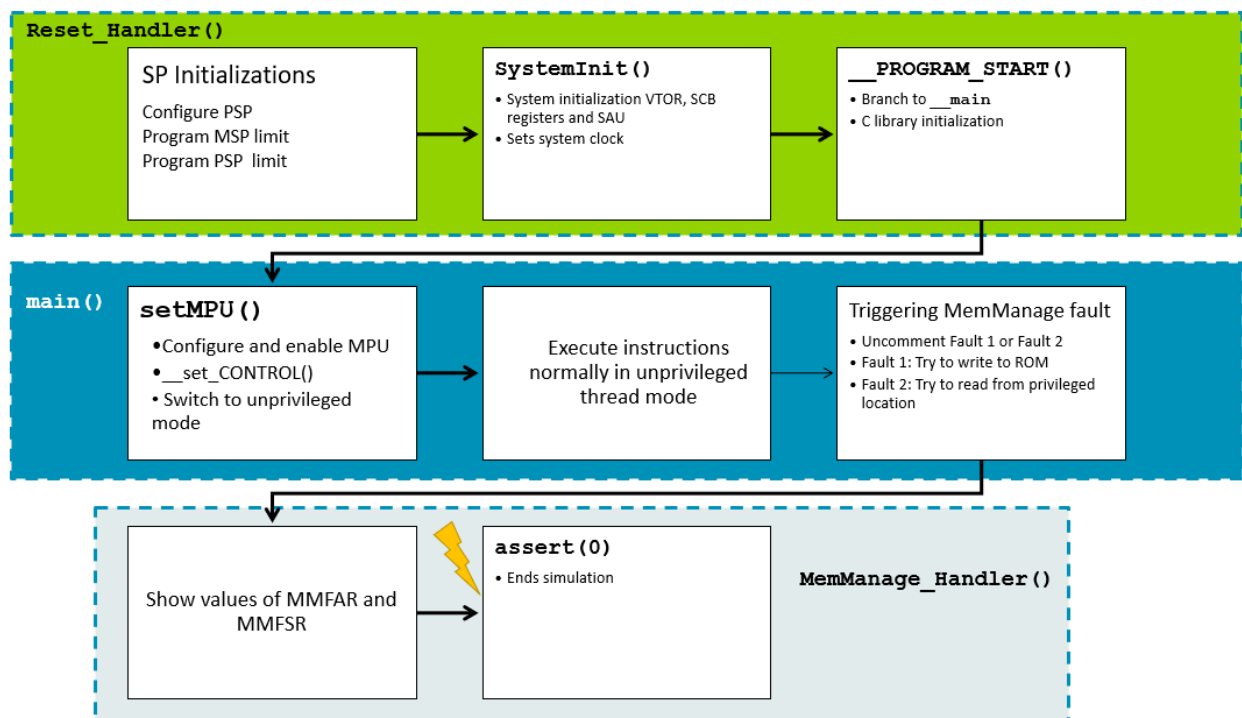
This example aims to show:

- Basic programming of MPU regions.
- Enabling the MPU and MemManage faults.
- Deliberately triggering different MemManage faults to show how memory is protected by the MPU.

The source code for this example is available at [Memory_model/trap_access](#).

An execution flow chart for this example project is shown below:

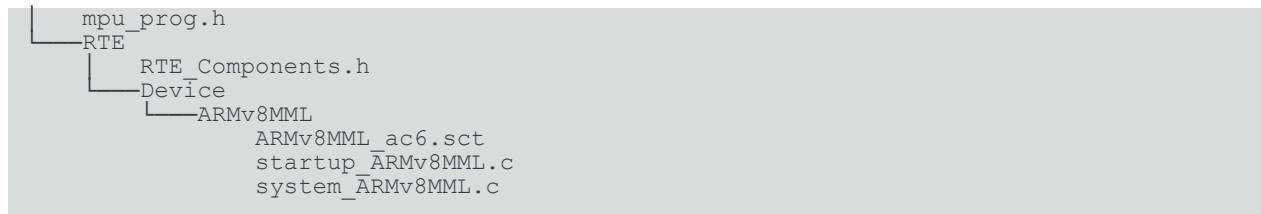
Figure 5-1: Execution flow chart for trap_access



5.2.1 Project structure

The file structure of this [example project](#) is here shown:

```
main.c
mpu_defs.h
mpu_prog.c
```

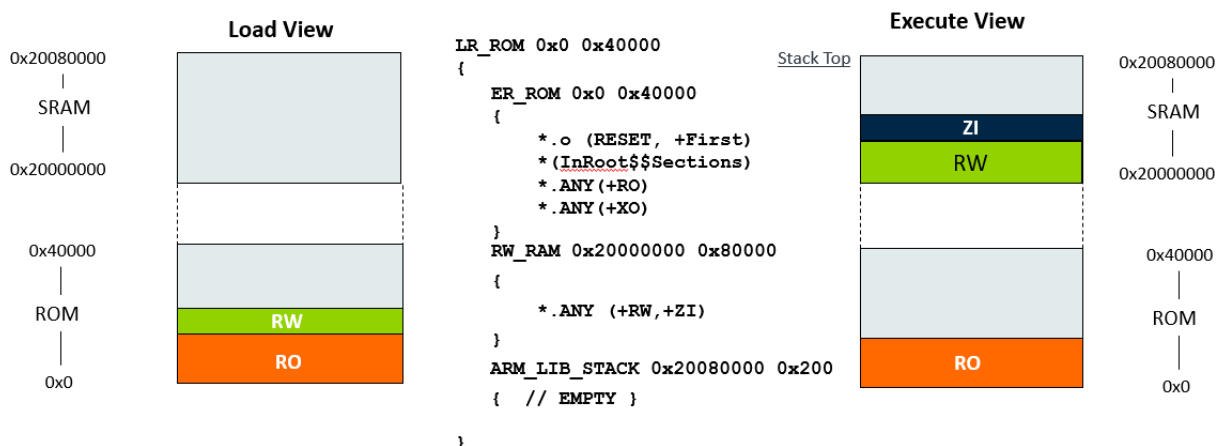


- main.c: Sets the MPU, switches to unprivileged mode and triggers MemManage faults.
- mpu_prog.c: Programs MPU regions and enables the MPU.
- mpu_defs.h: Definitions for MPU region attributes.
- RTE/Device/ARMv8MML/startup_ARMv8MML.c: Vector table, interrupt/exception handlers.
- RTE/Device/ARMv8MML/ARMv8MML_ac6.sct: Scatter file.
- RTE/Device/ARMv8MML/system_ARMv8MML.c: Target definitions.

5.2.2 Memory Map and Scatter file definitions

A scatter file gives you the ability to control where the linker places different parts of your image for your particular target including the location and size of various memory regions that are mapped to ROM, RAM, and FLASH. Considering the target memory map of MPS2 FVP model (Refer [Memory map](#)), the following execution regions are defined in the scatter file for this project.

Figure 5-2: Scatter file layout



Here is a snippet of scatter file definitions for this project. Please see the [scatter file](#) for the full description of the different regions.

```

<...>
#define __ROM_BASE      0x00000000
#define __RAM_BASE      0x20000000
#define __ROM_SIZE      0x00080000
#define __RAM_SIZE      0x00040000
/*-----

```

```
Scatter Region definition
*-----*/
LR_ROM __RO_BASE __RO_SIZE {                ; load region size_region
ER_ROM __RO_BASE __RO_SIZE {                ; load address = execution address
  *.o (RESET, +First)
  *(InRoot$$Sections)
  .ANY (+RO)
  .ANY (+XO)
}
RW_RAM __RW_BASE __RW_SIZE {
  ; RW data
  .ANY (+RW +ZI)
}
ARM_LIB_STACK __STACK_TOP EMPTY - __STACK_SIZE {
  ; Reserve empty region for stack
}
<...>
```

In subsequent sections, you will be able to understand scatter file definitions used for this example project and how it maps to actual memory address.

5.2.2.1 ROM and RAM regions

For this example project, two regions are defined in scatter file:

- **LR_ROM:** A read-only load region (LR_ROM) starts at __RO_BASE (0x0). In this example, the load address is same as execution address (ER_ROM)
- **RW_RAM:** Read-write data region (RW_RAM) starts at __RW_BASE used for general memory read and write operations.

The linker script generates the region-related symbols for each region specified in the scatter file. Image\$\$ and Load\$\$ symbols are generated for each execution region by linker script.

Considering that ER_ROM and RW_RAM execution regions defined in scatter file, following linker generated symbols are referenced as extern symbols in source code [mpu_prog.c](#):

- Image\$\$ER_ROM\$\$Base - Denotes execution base address of the region ER_ROM located at 0x0000_0000
- Image\$\$ER_ROM\$\$Limit - Denotes the address of the byte beyond the end of the execution region ER_ROM which equates to 0x0008_0000
- Image\$\$RW_RAM\$\$Base - Denotes execution base address of region RW_RAM located at 0x2000_0000
- Image\$\$RW_RAM\$\$Limit - Denotes the address of the byte beyond the end of execution region RW_RAM which equates to 0x2004_0000

For more details on linker generated symbols corresponding to its scatter file refer Section 4.5 of [Arm Compiler for Embedded Reference Guide](#)



Typically, the Static Random Access Memory (SRAM) in the processor system is used in a number of ways:

- Data : Data usually contains global and static variables
- Stack : The role of stack memory includes storing temporary data when handling function calls.
- Heap : The use of heap memory is optional and is only needed when the application uses functions like malloc() that perform dynamic memory allocation.

5.2.2.2 RESET

The vector table is defined with a named region called “RESET”. This RESET region will be defined by CMSIS Pack installation. Using this name, the linker script can specify where the vector table is placed. A linker script (scatter-loading file) example for the Arm toolchain using the RESET named region shown in the scatter file at [RTE/Device/ARMv8MML/ARMv8MML.sct](#). The line *.o (RESET, +First) of the scatter file example shown above specify that RESET named region is placed as first item in the internal ROM starting __RO_BASE. This address value should match the initial Vector Table Offset Register (VTOR) of the hardware platform being used. If it does not, the startup sequence will fail as the processor will not be able to read the vector table (i.e.) the initial value of Main Stack Pointer (MSP) and the starting address of the reset handler.

5.2.2.3 Stack and Heap

For each software project, it is essential to ensure sufficient memory space is allocated for stack and heap operations. In Arm Compiler toolchain, memory space for stack and heap sections is defined using ARM_LIB_STACK and ARM_LIB_HEAP regions respectively. This is defined in the scatter file as below:

```
<...>
#define __STACK_SIZE      0x00000200
#define __HEAP_SIZE       0x00000C00
<...>
; =====
; ARM_LIB_STACK 0x2004_0000 EMPTY -0x200 ; Stack growing down
; =====
ARM_LIB_STACK __STACK_TOP EMPTY -__STACK_SIZE { ; Reserve empty region for stack
}
<...>
```

As ARM_LIB_STACK and ARM_LIB_HEAP are also part of execution regions, the linker script generates Image\$\$ symbols similar to ER_ROM/RW_RAM regions explained in previous sections. In this example project, the symbol Image\$\$ARM_LIB_STACK\$\$ZI\$\$Limit is referenced in MPU region configuration. (Refer section [MPU configurations](#))

When defining the allocation of memory space, there is a need to consider the amount of additional memory space required for exception stack frame(s) and for the stack space required by exception handlers. In this example, since Floating point extension and Security extension are

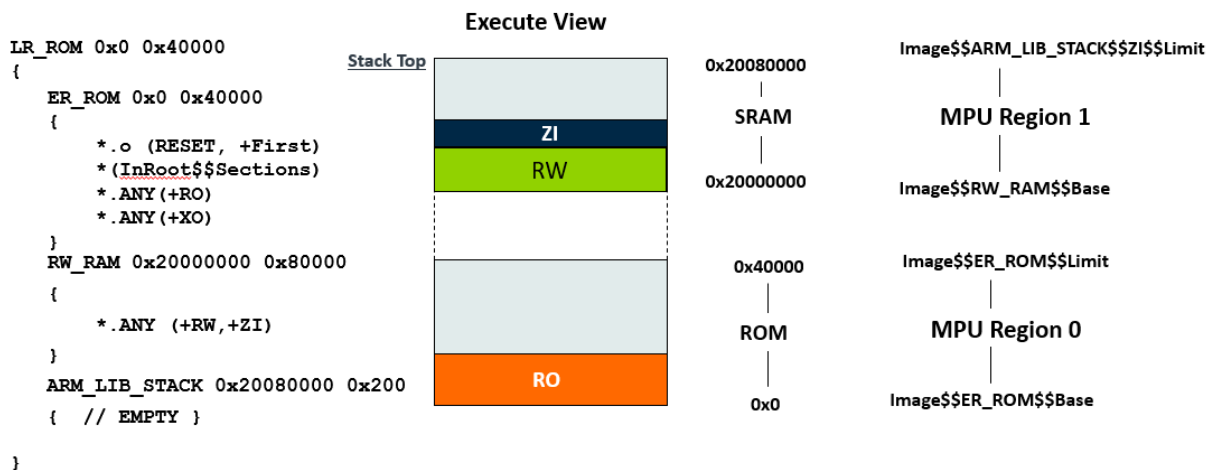
not enabled, just 8 words (i.e. integer stack frame) will be allocated for an exception stack frame. However, if Floating point extension or Security extension is implemented, then you will have to consider space for additional extended stack frames. For more details on the types of exception stack frame, refer Section B3.19 of [Armv8-M Architecture Reference Manual](#)

5.2.3 MPU configurations

When entering `main()`, the MPU regions are programmed using [MPU Functions for Armv8-M](#) by calling the function `setMPU()`, which is defined in `mpu_prog.c`. Two MPU regions are used in this example project are:

MPU region	Information	Address range	eXecute Never	Shareability	Access Permissions	Attr Index
0	ROM	ER_ROM Base- ER_ROM Limit	XN = 0	ARM_MPU_SH_OUTER	Read only, any privilege	AttrIdx 0 (Normal memory, cacheable, outer and inner Write-Through non-transient with read allocate)
1	RAM	RW_RAM Base- ARM_LIB_STACK ZI Limit	XN = 1	ARM_MPU_SH_OUTER	Read/ write, any privilege	AttrIdx 0 (Normal memory, cacheable, outer and inner Write-Through non-transient with read allocate)

Figure 5-3: MPU region and Scatter file layout



```

int setMPU(void) {
<...>
  // =====
  // Set MPU regions
  // =====
  // The base and limit addresses of both MPU regions are set based on symbols
  // created by the linker based on the scatter file. Using this approach
  // helps in keeping the MPU configuration in sync with the addresses being
  // used, and removes the need to duplicate the information.
  //
  //
  unsigned int ROMAddr = (unsigned int) &Image$$ER_ROM$$Base;
  unsigned int ROMLimit = (unsigned int) &Image$$ER_ROM$$Limit;
  unsigned int RAMAddr = (unsigned int) &Image$$RW_RAM$$Base;
  unsigned int RAMLimit = (unsigned int) &Image$$ARM_LIB_STACK$$ZI$$Limit;
}

```

```
// =====
// Set region 0
// =====
ARM_MPU_SetRegion(0UL,
                  ARM_MPU_RBAR(ROMAddr,
                                ARM_MPU_SH_OUTER,
                                ARM_MPU_RO,
                                ARM_MPU_NON_PRIV,
                                ARM_MPU_EXEC),
                  ARM_MPU_RLAR(ROMLimit, 0UL)
);
// =====
// Set region 1
// =====
ARM_MPU_SetRegion(1UL,
                  ARM_MPU_RBAR(RAMAddr,
                                ARM_MPU_SH_OUTER,
                                ARM_MPU_RW,
                                ARM_MPU_NON_PRIV,
                                ARM_MPU_XN),
                  ARM_MPU_RLAR(RAMLimit, 0UL)
);
// =====
// Enable MemManage Faults
// =====
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk;
// Enable MPU with all region definitions and
// background regions for privileged access.
ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk | MPU_CTRL_HFNMIENA_Msk);
return 0;
}
```

Note that, inside the CMSIS function `ARM_MPU_Enable`, a `DMB` barrier is inserted at the beginning and `DSB` and `ISB` barriers are inserted at the end of the function. The `DSB` barrier is used to guarantee that the register writes complete, while the `ISB` barrier is used to make sure the updates take effect before the execution of next instruction.

5.2.4 Triggering MemManage faults

In `main()`, after setting the MPU regions and enabling the MPU, execution is switched to unprivileged mode by writing on the `CONTROL` register. Now we are ready to execute the example fault scenario in `main()`. There are two example fault scenarios included in `main()` function. It is required that only one fault scenario is activated at a time. Hence, depending on your requirement, choose one fault scenario for this example project.

```
#define PRIVILEGED_LOCATION 0x5800E000;
int main (void) {
    // Configure and enable the MPU, defined in mpu_prog.c
    setMPU();
    // Switch to unprivileged mode
    set_CONTROL( __get_CONTROL() | CONTROL_nPRIV_Msk );
    // =====
    // NOTE:
    // Uncomment the scenario you want to trigger
    // =====
    int x = 1;
    // Scenario:1
    // =====
    // Try writing to ROM
    int* test1 = (int*)&Image$$ER_ROM$$Base;
    *test1 = x;
    // Scenario:2
}
```

```
// =====
// Try reading from background region marked as Privileged memory
// int* test2 = (int*)PRIVILEGED_LOCATION;
// x= *test2;
// return 0;
}
```



Inside the CMSIS function `__set_CONTROL`, an ISB barrier is inserted at the end of the function to guarantee the updates take effect before the next instruction.

Once you trigger the fault scenario, it is expected that you will enter into MemManage Fault. It is good to note that the MemManage handler is also defined in `main.c`:

```
/*-----
   MemManage Handler
   -----*/
void MemManage_Handler(void){
    printf("    We are in the MemManage handler. \n");
    printf("    MemManage Fault Address Register: \n
        SCB->MMFAR = 0x%08x\n", SCB->MMFAR );
    printf("    MemManage Fault Status Register: \n
        SCB->CFSR->MMFSR = 0x%08x\n", (0x000000FF) & (SCB->CFSR));
    assert(0);
}
```

5.2.5 Output in Target Console

Consider the following scenarios:

- Scenario:1 - Trying to write on a location in ROM. A MemManage fault is triggered and the MemManage handler is executed, showing the following output in target console:

```
We are in the MemManage handler.
MemManage Fault Address Register:
    SCB->MMFAR = 0x00000000
MemManage Fault Status Register:
    SCB->CFSR->MMFSR = 0x00000082
```

- Scenario:2 - Trying to write on a background region, which is privileged. As we are in unprivileged mode, A MemManage fault is triggered and the MemManage handler is executed, showing the following output in target console

```
We are in the MemManage handler.
MemManage Fault Address Register:
    SCB->MMFAR = 0x5800e000
MemManage Fault Status Register:
    SCB->CFSR->MMFSR = 0x00000082
```

In both of the faults above, the `MMFAR` register shows the address causing the MemManage fault. The `MMFSR` register provides the following information of the MemManage fault, from the value `0x82`:

- The processor attempted a load or store to a memory location which is not permitted.
- `MMFAR` holds a valid fault address.

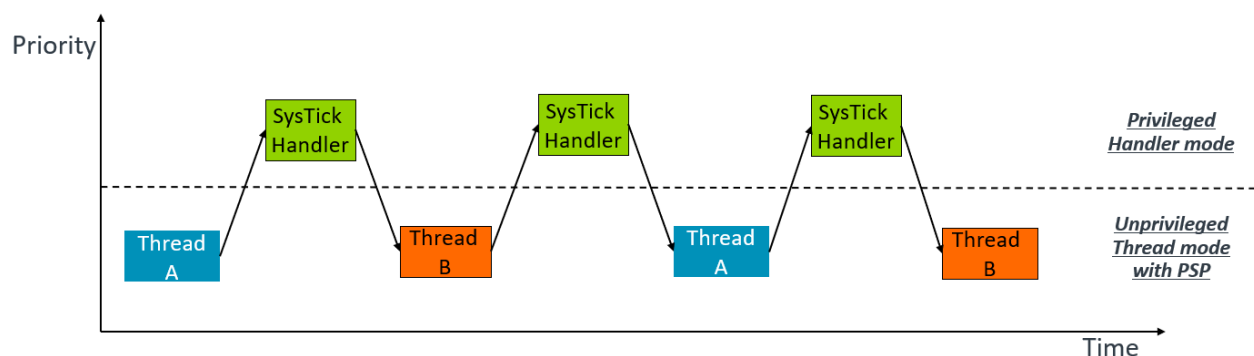
5.3 rtos_context_switch

The goal of this example is to show a simple and easy to understand the real-time kernel context switching operations; using MPU regions concept available in Cortex-M processors. This goal is accomplished by having two threads A and B, that switches alternatively between them and a SysTick Interrupt Service Routine(ISR) that acts as a kernel code.

Basic context switching and thread isolation requirements considered for this example are listed below:

- Two isolated threads called as thread A and thread B will be created
- Both thread A and thread B will be executed in unprivileged mode.
- Each thread has its own dedicated process stack.
- The MPU regions are setup in such a way that the data and code corresponding to Thread A is not accessible to Thread B and vice-versa.
- A System Tick Timer (SysTick) generates interrupts to switch between threads.
- The SysTick handler acts as a real-time kernel code and is responsible for context switching. The SysTick handler is also responsible for MPU reprogramming that is needed for thread isolation.
- The MPU regions for SysTick handler is setup in such a way that it neither of the threads can access the memory used by kernel code (i.e.) SysTick handler.

Figure 5-4: context_switch layout



The source code for this example is available at [Memory_model/rtos_context_switch](#).

To understand this example, the implementation details are divided into two sections:

- Basic Building Blocks
- Putting it all together

5.3.1 Basic Building Blocks

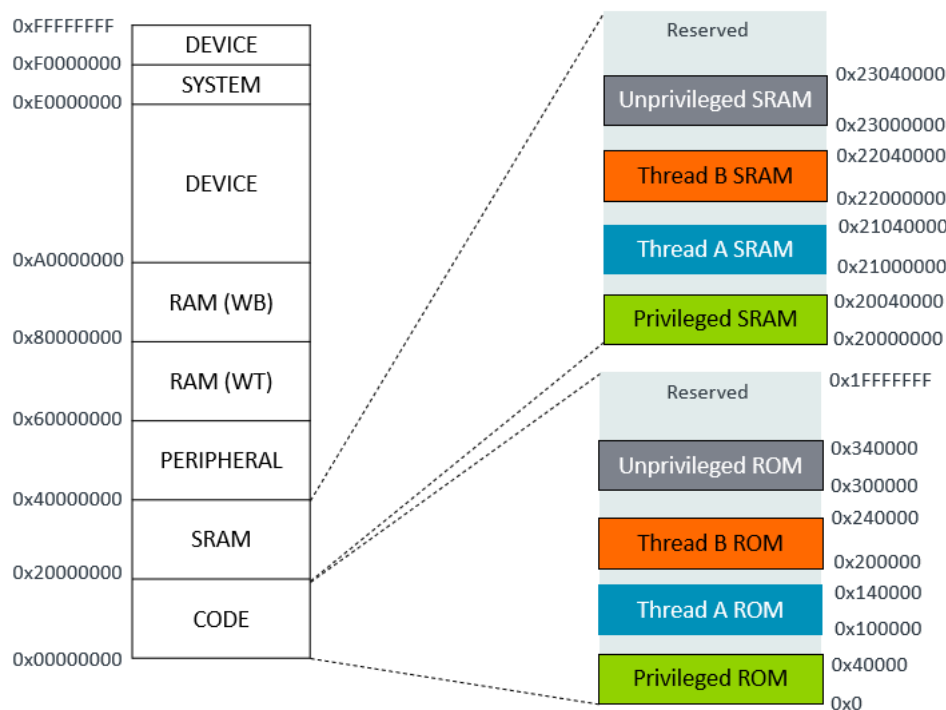
The building blocks for this example are:

- Memory map and MPU Configurations
- The SysTick Timer
- Thread Initialization
- Thread Context Switch in SysTick handler

5.3.1.1 Memory map and MPU Configurations

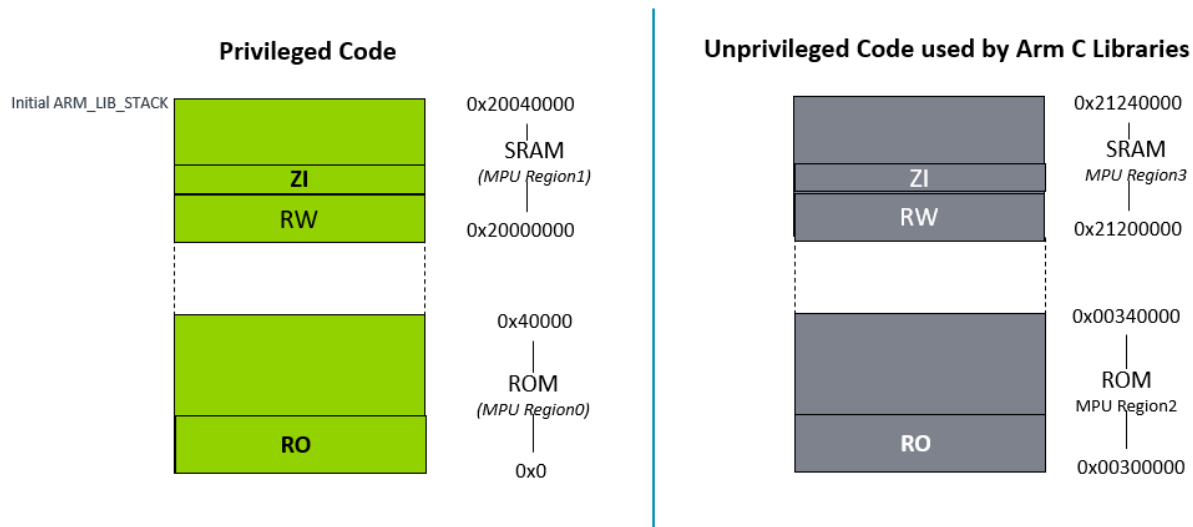
Below figure shows the overall memory map that will be used in this example.

Figure 5-5: Memory map



Considering the target memory map, the following execution regions are defined in the scatter file. Please see the scatter file [RTE/Device/ARMv8MML/ARMv8MML.sct](#) for the full description of the different regions.

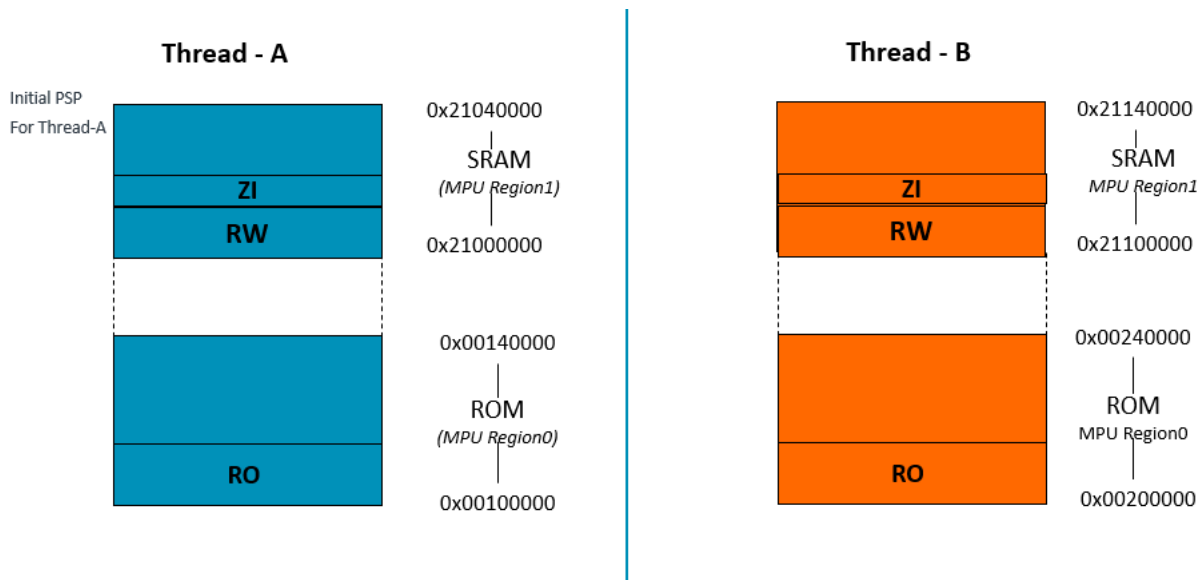
Figure 5-6: Scatter file layout 1



Before triggering the context switch operations, initial set of MPU regions are programmed in `setMPU()` function available in [mpu_prog.c](#). The initial set of MPU regions include from MPU region 0 to MPU region 3 which are used define memory for Privileged and Unprivileged access with the program as shown in above figure. `setMPU` function is called only once when executing `startos()` function. The `startos()` function is called from [main.c](#). Later in SysTick handler, the MPU regions (MPU region 0 and MPU region 1) are reconfigured to new thread that is going to be executed out of a context switch operation.

The MPU is configured in such a way that the background region is accessible only by privileged access by setting `MPU_CTRL.PRIVDEFENA` bit. Since Thread-A, Thread-B are required to be executed in unprivileged mode, individual MPU regions are setup for unprivileged thread execution and for its data accesses. This is shown in below figure:

Figure 5-7: Scatter file layout 2



5.3.1.2 The SysTick Timer

As a part of Armv8-M architecture, the architecture provides an in-built system timer called as SysTick. A SysTick provides a simple, 24-bit decrementing, wrap-on-zero counter. Though SysTick can be used for various purposes, in this example, SysTick acts as an RTOS tick timer that generates interrupt requests at a programmable rate on a regular basis.

The variable `systemCoreClock` holds the clock speed, and is updated by `SystemCoreClockUpdate()` function based on hardware configuration. The SysTick is then setup to generate SysTick exception at 2KHz. Timing interval (in number of clock cycle) will be `SystemCoreClock/2000`.

```
SystemCoreClockUpdate();
SysTick_Config(SystemCoreClock/2000);
```



It is good to note that `SystemCoreClockUpdate()` function operates based on device's vendor implementation. Check your vendor's implementation to get more details on clock frequency and its interval.

Using SysTick as an interrupt source, Thread A and Thread B are switched alternatively from SysTick handler routine.

The SysTick handler `sysTick_Handler()` is defined in [myRTOS.c](#) and written using [embedded assembly](#) as given in below code sequence:

```
extern __attribute__((naked)) void SysTick_Handler() {
    __asm(
        "PUSH    {r4-r11}\n"
```



```

"MOV    r1,sp\n"
"MOV    r0,lr\n"
"BL     SysTick_Handler_body\n"
"POP    {r4-r11}\n"
"BX     r0"
);

```



Note

Why **SysTick_Handler()** marked as **naked**? As SysTick exception handler (SysTick_Handler()) is marked as weak alias, any function with the same name will override this definition. When a function is declared with `__attribute__((naked))`, the compiler does not generate prologue and epilogue sequences for that functions. In this example, SysTick_Handler() is declared with `__attribute__((naked))` to ensure that context saving operations required for threads A and B are handled completely in the body of SysTick_Handler().

5.3.1.3 Thread Initialization

For the RTOS to switch between threads, the context of each thread should be defined. To define the context of a thread, the following data are required:

- Stack pointer
- Stack pointer limit
- Program counter
- Link register
- MPU configuration for the thread
- Thread identifier
- Callee registers (Saved and restored in SysTick handler)

`newThread()` is a function defined in [myRTOS.c](#) is used to create new threads by initialising the different fields of the thread context:

```

int newThread(void * function, uint32_t * threadStackLim, ThreadContext
*threadContext, MpuRegionCfg *mpuConf, unsigned int threadID){
    for(int i = 0; i < NUM_CALLEE_REGS; i++){
        threadContext->calleeRegs[i] = 0x0;
    }
    threadContext -> spLimit = threadStackLim;
    threadContext -> pc = function;
    threadContext -> lr = MY_EXC_RETURN;
    threadContext -> mpuCfg = mpuConf;
    threadContext -> id = threadID;
    // Initialise thread stack
    threadContext -> sp = (threadStackLim +
        THREAD_STACK_SIZE - NORM_STACK_FRAME_SIZE);
    threadContext -> sp[NORM_STACK_FRAME_XPSR_OFFSET] = DEF_MY_XPSR;
    threadContext -> sp[NORM_STACK_FRAME_PC_OFFSET] = (uint32_t)function;
    return 0;
}

```

Each thread have defined memory regions allocated for its code execution and data accesses separately. The file [threadDefs.h](#) includes the definitions required for each thread.



If Security extension is implemented in a system, then while context switching to next thread, the new thread's stack pointer top should be sealed with a value of `0xFE5EDA5`. For more details on context switching operations across Secure and Non-secure environments, refer Armv8-M Security Extension User Guide.

- Thread A calculates and prints numbers of the Fibonacci sequence. The thread A function (`thrA()`), thread A data and thread A stack (`uint32_t threadA_stk[THREAD_STACK_SIZE]`) are in [threadA.c](#).
- Thread B calculates and prints numbers of the Pentagonal sequence. The thread B function (`thrB()`), thread B data and thread B stack (`uint32_t threadB_stk[THREAD_STACK_SIZE]`) are in [threadB.c](#).

5.3.1.4 Thread Context Switch in SysTick handler

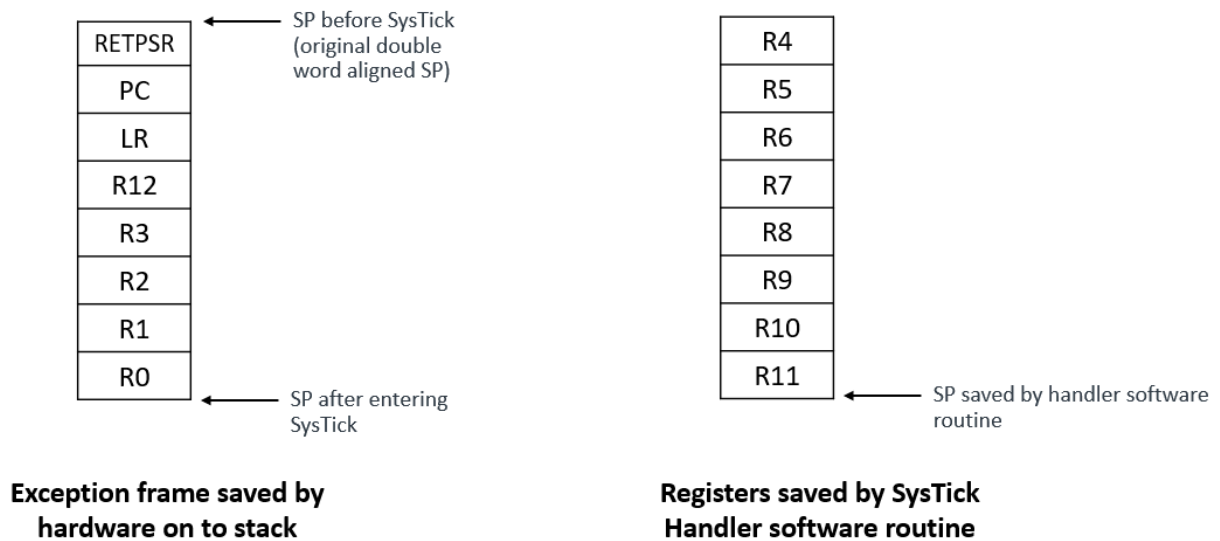
The context switch happens in SysTick exception handler. Once a SysTick interrupt occurs, the processor hardware automatically stacks an exception frame (RETPSR, PC, LR, r12 and r3-r0) onto the Process Stack (PSP) and branches to the SysTick handler routine in Handler Mode (which uses the Main Stack).

As a part of context switching operation, the SysTick handler performs following operations:

- Manually save remaining registers r4-r11 on the Main Stack
- Save current thread's PSP location
- Load next thread's stack pointer memory location and assign it to PSP
- Manually unstack registers r4-r11
- Perform a SysTick exception return operation by executing "BX LR". Note that LR is written with a value of `0xFFFFFFF` which makes the processor switch to unprivileged thread mode, unstack next thread's exception frame from PSP and continue on its PC.

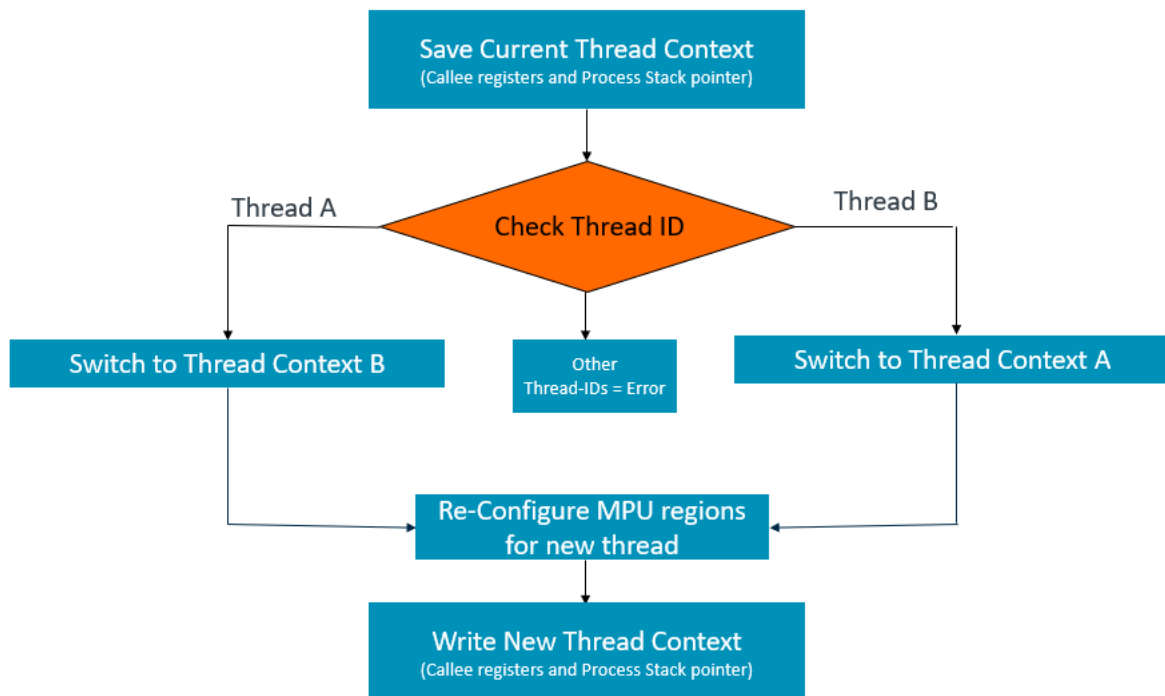
This figure shows the layout on how each registers are saved on the stack (i.e.) 1. Registers saved by Exception stack frame 2. Registers saved by SysTick handler software routine

Figure 5-8: Register save layout



Below flow-chart shows the steps performed in `sysTick_Handler()` as a part of Thread context switching operations:

Figure 5-9: SysTick_handler_switch layout



Once the current context is changed to the other thread, the MPU regions are reconfigured to its corresponding thread by calling the function `reprogMPU()`.

For example, if the current context is Thread-A and the context switch happens to Thread-B, then MPU regions are reconfigured like below:

MPU Register	Current context - Thread A	New context - Thread B
MPU_RBAR0 (Base)	0x00100000	0x00200000
MPU_RLAR0 (Limit)	0x00140000	0x00240000
MPU_RBAR1 (Base)	0x21000000	0x21100000
MPU_RLAR1 (Limit)	0x21040000	0x21140000

5.3.2 Putting it all together

This section demonstrates the steps required for context switching using the building blocks described in [Basic Building Blocks](#).

- Step:1 - Reset Handler and System initialization
- Step:2 - Define and create new threads in `main()`
- Step:3 - Start SysTick Timer
- Step:4 - Enter SysTick Handler
- Step:5 - SysTick Handler routine
- Step:6 - Enter new thread in unprivileged mode.

5.3.2.1 Step:1

As a part of `Reset_Handler()` routine, following registers are initialized before branching to `__PROGRAM_START` which is Arm C library entry point.

- Process Stack Pointer (PSP)
- Process Stack Pointer Limit (PSPLIM)
- Main Stack Pointer Limit (MSPLIM)
- Vector Table Offset Register (VTOR)
- CCR.UNALIGNED_TRP bit is set to 1 to trap any unaligned accesses

5.3.2.2 Step:2

Post performing Arm C library initialization routines successfully in `__PROGRAM_START`, the program now enters `main()`. As a part of this `main()` routine, the MPU regions are initialized as per figure

Scatter file layout-1 in [Memory map and MPU Configurations](#). Thread-A and Thread-B are created by calling `newThread()` function.

In this example, it is expected that the first thread to get executed is Thread-A. Hence, before triggering the SysTick interrupt which performs the context switch, the current context is initialized as Thread-B.

```
currentContext = &threadBContext;
```

Hence on SysTick interrupt handler return, Thread-A will be executed as a first thread.

5.3.2.3 Step:3

The SysTick timer interrupt is enabled by `startos()` function (i.e.) the SysTick timer interrupt is enabled by writing 1 to SYST_CSR.ENABLE register and processor will start SysTick timer by decrementing the counter. When the counter reaches to a value of zero, SysTick interrupt is triggered before reloading a non-zero value in the counter. The context switch operation handled by SysTick handler.



The idle thread is run until the SysTick timer generates an exception that will start running and alternating between threads.

5.3.2.4 Step:4

On a SysTick interrupt, the processor hardware save the registers (R0-R3,R12,LR,PC and RETPSR) as a part of exception stacking process and enter SysTick Handler routine.



If you assuming that a SysTick interrupt occurs on Thread-A execution, then PSP initialized for Thread-A will be used for exception stacking process

5.3.2.5 Step:5

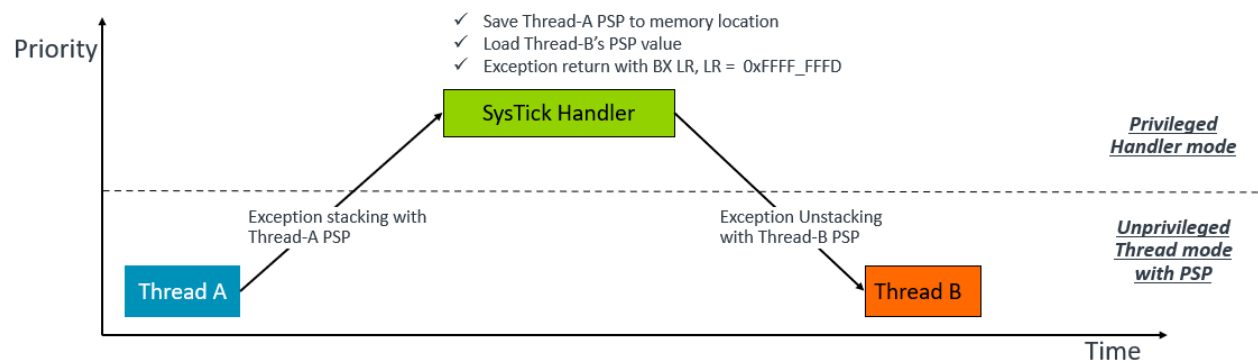
In SysTick handler, following steps are performed:

- Save R4-R11 registers on to Main stack pointer.
- Save the PSP value into a memory location such that it can be used later.
- Load PSP value for the next thread from memory
- Restore R4-R11 values from Main stack pointer.

- Perform exception return with EXC_RETURN as 0xFFFF_FFFD (i.e.) return to thread mode with new PSP stack pointer

Following figure depicts the context switch operation from Thread-A to Thread-B as an example:

Figure 5-10: Thread A to Thread B example



5.3.2.6 Step:6

Enter next thread in unprivileged thread mode using new PSP.

Assuming that the SysTick exception interrupted Thread-A, you will notice that the processor is returning to Thread-B; hence, the context switch is complete!

5.3.2.6.1 Output in Target Console

Here is a snippet of expected output in your target console:

```
Fibonacci number: 0
Fibonacci number: 1
Fibonacci number: 2
Fibonacci number: 3
...
Pentagonal number: 2752
Pentagonal number: 2882
Pentagonal number: 3015
Pentagonal number: 3151
...
Fibonacci number: 8
Fibonacci number: 13
Fibonacci number: 21
Fibonacci number: 34
```

5.3.3 Additional Information

Here are few additional notes considered for this example:

- For simplicity, SysTick exception is used to switch between the threads in this example. However, it is recommended to use PendSV exception for more complex context switching operations (Refer Armv8-M Exception Model User Guide for more information).
- Though in a more realistic system, there will be number of threads to handle, and there might be more sophisticated mechanisms to assign priority to these threads into different categories, this example does not assign any priority between the threads.
- Some systems might choose to have the context switching written in assembly language for performance reasons, but C language is used in this example for easy read.
- It does not use any other higher priority interrupts (like NMI) to disrupt the switching between the threads.

5.3.3.1 Triggering a MemManage fault

The MemManage Faults can be enabled before enabling the MPU in `setMPU()` in `mpu_prog.c`:

If we uncomment the following code in `threadB.c`:

```
// Uncomment to trigger MemManage fault
threadA_stk[1] = 0x0;
```

then, thread B tries to write on thread A stack which will trigger a MemManage fault. The MemManage fault will enter `MemManage_Handler()` available in `myRTOS.c`.

5.3.3.2 Project structure

The file structure of this [example project](#) is shown below:

```
main.c
mpu_configs.c
mpu_configs.h
mpu_defs.h
mpu_prog.c
mpu_prog.h
mpu_reprog.c
mpu_reprog.h
myRTOS.c
myRTOS.h
threadA.c
threadB.c
threadDefs.h
└─ RTE
   └─ RTE_Components.h
      └─ Device
         └─ ARMv8MML
            ├── ARMv8MML_ac6.sct
            ├── startup_ARMv8MML.c
            └── system_ARMv8MML.c
```

- `main.c`: Threads creation and initialization, starting the RTOS.
- `mpu_configs.c`: Setting thread A and thread B MPU configurations.
- `mpu_defs.h`: Definitions for MPU region attributes.
- `mpu_prog.c`: Initial MPU programming, enabling MemManage faults, enabling the MPU.
- `mpu_reprog.c`: Reprogramming the MPU when switching between threads.
- `myRTOS.c`:
 - Starting the OS.
 - Creating new threads.
 - SysTick handler for thread management and context switching.
 - MemManage handler.
- `threadA.c`: Calculates and prints numbers of the Fibonacci sequence.
- `threadB.c`: Calculates and prints numbers of the Pentagonal sequence.
- `threadDefs.h`: Definitions and type definitions for thread contexts.

6. References

Here are some resources related to material in this guide:

- [Armv8-M Architecture Reference Manual](#)
- Books:
 - The Definitive Guide to Arm Cortex-M3 and Cortex-M4 Processors - Joseph Yiu
 - The Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors - Joseph Yiu
- [Cortex-M resources](#)
- [Procedure Call Standard for the Arm Architecture](#)

7. Next steps

Refer to the following guides for more details about specific architectural extensions:

- [Armv8-M Exception Model User Guide](#)
- [Armv8-M Security Extension User Guide](#)