Memory Protection Unit (MPU)

Version 1.0

ARM

ARM 100699_0100_00_en

Memory Protection Unit (MPU)

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	08 July 2016	Non-Confidential	First release

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents
Memory Protection Unit (MPU)

List of Figures
Memory Protection Unit (MPU)

List of Tables
Memory Protection Unit (MPU)

Preface

This preface introduces the *Memory Protection Unit (MPU)* .

It contains the following:

About this book

Product revision status

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

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

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

Intended audience

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

The *Memory Protection Unit* (MPU) is a programmable unit that allows privileged software to define memory access permissions for up to 16 separate memory regions. This chapter provides an overview of the MPU programmers' model and summarizes its key features.

Chapter 2 Memory type definitions

In the ARMv8-M architecture, memory types are divided into Normal Memory and Device Memory. If the ARMv8-M architecture with Security Extension is implemented, the memory space is partitioned into Secure and Non-secure memory regions.

Chapter 3 Memory configuration

The MPU is configured by a series of memory mapped registers in the *System Control Space* (SCS). This chapter lists the MPU registers, and describes the attribute indirect mechanism that allows multiple MPU regions to share a set of memory attributes.

Chapter 4 Register definitions

This chapter shows the bit assignments for each of the MPU registers.

Chapter 5 CMSIS MPU support

ARMv8-M processors provide software support with an initiative called the *Cortex Microcontroller Software Interface Standard* (CMSIS). This chapter lists the standardized names for MPU registers, and provides configuration settings to initialize CMSIS MPU.

Glossary

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

See the *ARM Glossary* for more information.

Typographic conventions

italic

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

bold

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

`monospace`

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

`mono`space

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

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

monospace bold
: Denotes language keywords when used outside example code.

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

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

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

Timing diagrams

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

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

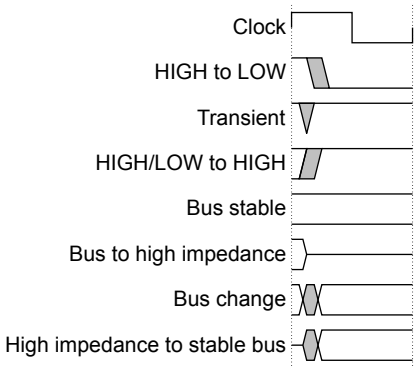

Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

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

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Memory Protection Unit (MPU)* .
- The number ARM 100699_0100_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

——— **Note** ———

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

———————————

Chapter 1
Introduction

The *Memory Protection Unit* (MPU) is a programmable unit that allows privileged software to define memory access permissions for up to 16 separate memory regions. This chapter provides an overview of the MPU programmers' model and summarizes its key features.

It contains the following sections:

1.1 About the MPU

The *Memory Protection Unit* (MPU) is a programmable unit that allows privileged software, typically an OS kernel, to define memory access permission. It monitors transactions, including instruction fetches and data accesses from the processor, which can trigger a fault exception when an access violation is detected.

The *Protected Memory System Architecture* (PMSA) is the architecture that defines the operation of the MPU inside the ARM® processors. With the development of the ARMv8-M architecture, the PMSA has been updated to PMSAv8.

The MPU programmers' model allows the privileged software to define memory regions and assign memory access permission and memory attributes to each of them. Depending on the implementation of the processor, the MPU on ARMv8-M processors supports up to 16 regions. The memory attributes define the ordering and merging behaviors of that region, as well as caching and buffering attributes. Cache attributes can be used by internal caches, if available, and can be exported for use by system caches.

ARMv8-M architecture with Main Extension have a dedicated *Memory Management Fault* (MemManage) that is triggered by accesses that violate the access permissions that are configured for an MPU region. 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 implementations that isolate memory on a per-thread basis, or provide demand stack allocation.

If the MemManage fault is disabled or cannot be triggered because the current execution priority is too high, the fault is escalated to a HardFault. ARMv8-M implementations without the Main Extension can only use the HardFault exception.

If the ARMv8-M Security Extension is included, the Secure and Non-secure worlds have their own MPU. The number of regions in the Secure and Non-secure MPU can be configured independently and each can be programmed to protect memory for the associated Security state.

Certain memory accesses including exception vector fetches, accesses to *System Control Space* (SCS), which include MPU, NVIC, and SysTick, and the *Private Peripheral Bus* (PPB), which includes internal debug components, are not affected by the MPU settings. Also, the MPU configurations do not define the access permissions and attributes for debug accesses.

1.2 Key features of the MPU

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 must be a multiple of 32 bytes.
- All regions must begin on a 32 byte aligned address.
- Regions have independent read/write access permissions for privileged and unprivileged code.
- The eXecure Never (XN) attribute enables separation of code and data regions.

With ARMv8-M architecture with Security Extension it is possible to have one set of MPU configuration registers for the Secure world and another set of MPU configuration registers for the Non-secure world. It is also possible to have the MPU feature available in just one of the security states, or have no MPU at all. Secure software can access a Non-secure MPU using an alias address (address `0xE002ED90`).

1.3 MPU programmers' model changes for the ARM®v8-M architecture

Although the concepts for the MPU operations are similar, the MPU in the ARMv8-M architecture has a different programmers' model to the MPU in previous ARMv8-M architectures.

The following changes have been made to the MPU programmers' model for the ARMv8-M architecture:

- 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. For example, when creating a memory region from an address `0x3BC00-0x80400`, multiple MPU region registers are required, as in the following figure.

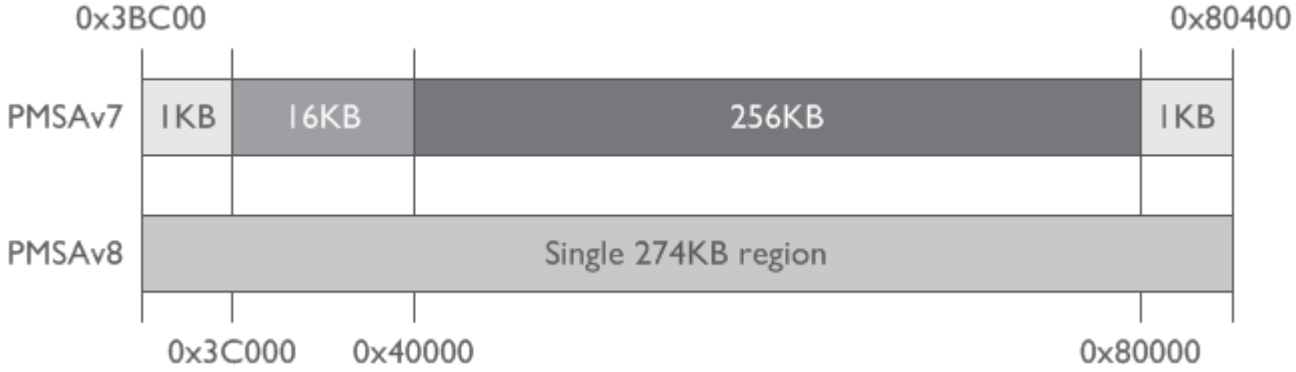

Figure 1-1 MPU memory regions

- In the ARMv8-M architecture the size of an MPU region can be any size (say 274KB) with a granularity of 32 bytes.
- PMSEv8 does not include subregions as the region size is now more flexible.
- Regions are now not allowed to overlap. As the MPU region definition is much more flexible, overlapping MPU regions are not necessary.
- Memory regions define memory attributes using an index value which is then looked up in a set of memory attribute registers.

As the ARMv8-M architecture with Security Extension was not previously available, legacy configuration code must also be updated to reflect the new features.

Chapter 2
Memory type definitions

In the ARMv8-M architecture, memory types are divided into Normal Memory and Device Memory. If the ARMv8-M architecture with Security Extension is implemented, the memory space is partitioned into Secure and Non-secure memory regions.

It contains the following sections:

2.1 Memory type definitions in the ARM®v8-M architecture

In the ARMv8-M architecture, memory types are divided into Normal Memory and Device Memory.

Note

The *Strongly Ordered* (SO) device memory type in the ARMv6-M and ARMv7-M architectures is a subset of the Device memory type in ARMv8-M architecture.

This section contains the following subsections:

2.1.1 Normal memory

The Normal memory type can be used for MPU regions that are used to access general instruction or data memory. Normal memory allows the processor to perform some memory access optimizations, such as access reordering or merging. Normal memory also allows memory to be cached and is suitable for holding executable code.

Normal memory must not be used to access peripheral registers *Memory Mapped I/O* (MMIO); the Device memory type is intended for that use.

Note

The Normal memory definition remains largely unchanged from ARMv7-M architecture.

Normal memory can have several attributes that can be applied to it. The following memory attributes are available:

Cacheability
Memories can be cacheable or non-cacheable.

Shareability
Normal memory can be shareable or Non-shareable.

eXecute Never
Memories can be marked as executable or eXecute Never (XN).

Cacheability

The cacheability can be further divided into cache policy, allocation, and transient hint.

Cache policy
Write-Through or Write-Back

Allocation
Cache line allocation hintas, for read and write access.

Transient hint
A hint to the cache that the data might only be needed in the cache temporarily.

The architecture supports two levels of cache attributes. These are the inner cache and outer cache attributes.

Typically, the inner cache attribute is used by any integrated caches and the outer cache attributes are exported using the bus system sideband signals. Depending on the processor implementation, the inner cache attributes can also be exported to the memory system using extra sideband signals.

For example, in the AMBA® 5 AHB5 specification, the **HPROT** signal is used to propagate cache attributes:

Table 2-1 Cache attributes

HPROT[6] Shareable	HPROT[5] Allocate	HPROT[4] Lookup	HPROT[3] Modifiable	HPROT[2] Bufferable	Memory Type
0	0	0	0	0	Device-nE
0	0	0	0	1	Device-E
0	0	0	1	0	Normal Non-cacheable, Non-shareable
0	0 or 1	1	1	0	Write-Through, Non-shareable
0	0 or 1	1	1	1	Write-Back, Non-shareable
1	0	0	1	0	Normal Non-cacheable, shareable
1	0 or 1	1	1	0	Write-Through, shareable
1	0 or 1	1	1	1	Write-Back, shareable

Note

The transient attribute is included in the architecture to be consistent with ARMv8-A processors. It indicates that the benefit of caching is for a relatively short period. Therefore it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Configuring an MPU region with a cacheable memory type does not mean that the data must be cached but only indicates to the hardware that it might be cached. If a region is defined as cacheable software takes responsibility for performing any necessary cache maintenance operations.

Shareability

Many systems have multiple bus masters, either multiple processors, or a mixture of processors and other masters such as *Direct Memory Access* (DMA) engines. The shareability attribute allows software to advertise to the hardware which of these devices must be able to see any updates to a particular area of memory.

To manage shareability, the ARM architecture groups all masters into one of three domains of memory shareability:

- Non-shareable memory.
- Inner shareable memory.
- Outer Shareable memory.

The following figure shows how masters are divided into shareability groups.

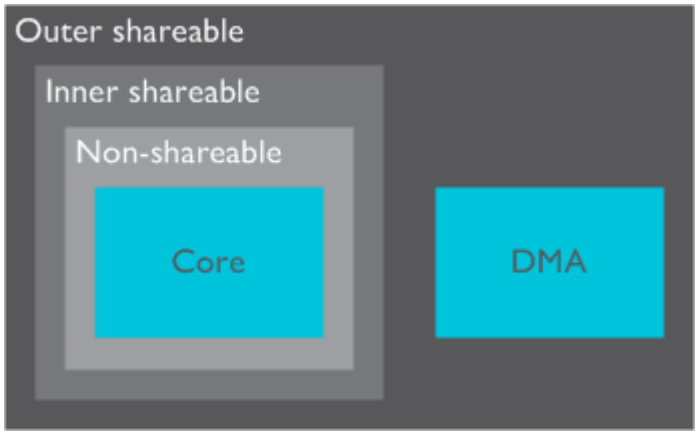

Figure 2-1 Shareability groups

Defining the shareability of a memory region imposes some functional requirements on the hardware but it does not restrict how the hardware implements that functionality.

The Memory Model Feature Register 0 (ID_MMFR0) provides some information about the available shareability domains and their behaviors.

The *Outer Shareable* (OSH) requirement is that all masters in the outer sharable domain can see the effects of any memory updates:

- In a system without caches and just on level of RAM, any master can see any memory update.
- In a system with caches, not all masters can access all caches and the system might either employ hardware cache coherency to make updates visible, or treat any shareable memory as non-cacheable, making updates visible.

Non-shareable memory

Non-shareable represents memory accessible only by a single processor or other agent, so memory accesses never have to be synchronized with other processors. Only the processor itself must see the information, though it can be made visible to other agents.

Inner shareable memory

Inner shareable memory represents a shareability domain that can be shared by multiple masters, but not necessarily all the agents in the system. A system might have multiple Inner Shareable domains. An operation that affects one Inner Shareable domain does not affect other Inner Shareable domains in the system. All agents inside this domain might be able to see the memory.

Outer shareable memory

An *Outer Shareable* (OSH) domain reorder is shared by multiple agents and can consist of one or more Inner shareable domains. An operation that affects an Outer Shareable domain also implicitly affects all Inner shareable domains inside it. However, it does not otherwise behave as an inner shareable operation.

2.1.2 Device Memory

Device memory must be used for memory regions that cover peripheral control registers. Some of the optimizations that are permitted for Normal memory, such as access merging or repeating, would be unsafe for a peripheral register.

The Device memory type has several attributes:

- G or nG – Gathering or non-Gathering. Multiple accesses to a device can be merged into a single transaction except for operations with memory ordering semantics, for example, memory barrier instructions, load acquire/store release.
- R or nR – Reordering or Non-reordering.
- E or nE – Early Write Acknowledge (similar to bufferable).

Only four combinations of these attributes are valid:

- Device-nGnRnE
- Device-nGnRE
- Device-nGRE
- Device-GRE

——— **Note** ———

Device-nGnRnE is equivalent to ARMv7-M Strongly Ordered memory type and Device-nGnRE is equivalent to ARMv7-M Device memory.

Device-nGRE and Device-GRE are new to ARMv8-M architecture.

———————————

Typically peripheral control registers must be either Device-nGnRE or Device-nGnRnE to prevent reordering of the transactions in the programming sequences.

Device-nGRE and Device-GRE memory types can be useful for peripherals that memory access sequence and ordering does not affect results. For example, bitmap or display buffers in display interface. If the bus interface of such peripheral can only accept certain transfer sizes, the peripheral must be set to Device-nGRE.

——— **Note** ———

For most simple processor designs, reordering, and gathering (merging of transactions) do not occur even if the memory attribute configuration allows it to do so.

———————————

2.2 Memory system and memory partitioning

If the ARMv8-M architecture with Security Extension is implemented the 4GB memory space is partitioned into Non-secure and Secure memory regions. The Secure memory space is further divided into two types, Non-secure Callable and Non-secure.

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

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

This section contains the following subsections:

- *2.2.1 Secure (S)* on page 2-21.

2.2.1 Secure (S)

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

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

Chapter 3
Memory configuration

The MPU is configured by a series of memory mapped registers in the *System Control Space* (SCS). This chapter lists the MPU registers, and describes the attribute indirect mechanism that allows multiple MPU regions to share a set of memory attributes.

It contains the following sections:

3.1 MPU registers

The MPU is configured by a series of memory mapped registers in the System Control Space. The MPU registers are banked between Secure and Non-secure states. The programmers' model for Secure MPU and Non-secure MPU are the same, but the number of MPU regions for the two MPUs can be different.

When accessing the MPU address between `0xE000ED90` and `0xE000EDC4`, the type of MPU registers accessed is determined by the current state of the processor. Secure access sees Secure MPU registers, Non-secure access sees Non-secure MPU registers. Secure software can also access Non-secure MPU registers using an alias address.

MPU registers are privileged access only (unprivileged accesses generate a fault exception). MPU registers must be accessed using 32-bit aligned transfers. By default the MPU is disabled after reset.

The memory type is encoded as an 8-bit field that is stored in one of the *Memory Attribute Indirection Registers* (MAIR). Each MAIR register has four 8-bit fields, allowing eight memory types to be defined at any one time.

The following table lists the MPU registers.

Table 3-1 MPU registers

Register	Address	NS Address Alias	Description
MPU_TYPE	`0xE000ED90`	`0xE002ED90`	MPU Type Register
MPU_CTRL	`0xE000ED94`	`0xE002ED94`	MPU Control Register
MPU_RNR	`0xE000ED98`	`0xE002ED98`	MPU Region Number Register
MPU_RBAR	`0xE000ED9C`	`0xE002ED9C`	MPU Region Base Address Register
MPU_RLAR	`0xE000EDA0`	`0xE002EDA0`	MPU Region Base Limit Register
MPU_RBAR_A1	`0xE000EDA4`	`0xE002EDA4`	MPU Region Base Address Register Alias 1
MPU_RBAR_A2	`0xE000EDAC`	`0xE002EDAC`	MPU Region Base Address Register Alias 2
MPU_RBAR_A3	`0xE000EDB4`	`0xE002EDB4`	MPU Region Base Address Register Alias 3
MPU_RLAR_A1	`0xE000EDA8`	`0xE002EDA8`	MPU Region Limit Address Register Alias 1
MPU_RLAR_A2	`0xE000EDB0`	`0xE002EDB0`	MPU Region Limit Address Register Alias 2
MPU_RLAR_A3	`0xE000EDA8`	`0xE002EDB8`	MPU Region Limit Address Register Alias 3
MPU_MAIR0	`0xE000EDC0`	`0xE002EDC0`	MPU Memory Attribute Indirection Register 0
MPU_MAIR0	`0xE000EDC4`	`0xE002EDC4`	MPU Memory Attribute Indirection Register 1

Note

In the ARMv8-M architecture the MPU_TYPE, MPU_CTRL and MPU_RNR registers are identical to those same registers in the ARMv6-M or ARMv7-M architectures.

3.2 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.

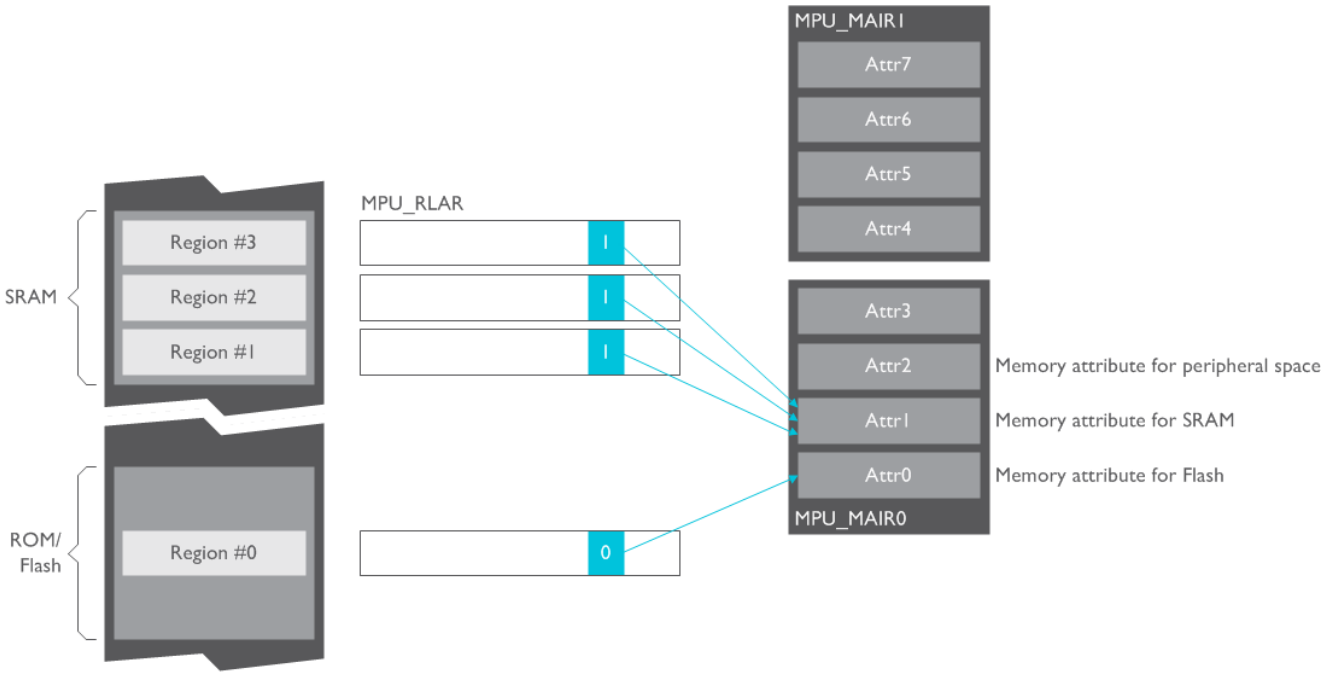

Figure 3-1 Attribute indirection

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.

Chapter 4
Register definitions

This chapter shows the bit assignments for each of the MPU registers.

It contains the following sections:

4.1 MPU_TYPE

The MPU Type register indicates how many regions the MPU supports for the selected security state. This register is read only.

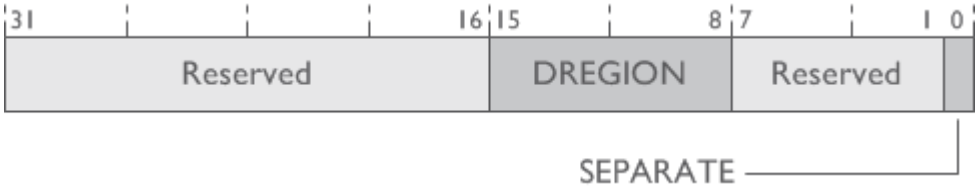

Figure 4-1 MPU_TYPE bit assignments

Table 4-1 MPU_TYPE bit assignments

Bits	Field	Reset	Description
[31:16]	Reserved – read as 0	0	Reserved.
[15:8]	DREGION	Implementation defined	Number of MPU regions that are supported by the MPU in the selected security state.
[7:1]	Reserved – read as 0	0	Reserved.
[0]	SEPARATE	0	Indicates support for separate instruction data address regions. ARMv8-M architecture only supports unified MPU regions and therefore this bit is set to 0.

4.2 MPU_CTRL

The MPU Control register provides various programmable bit fields for MPU enable and features.

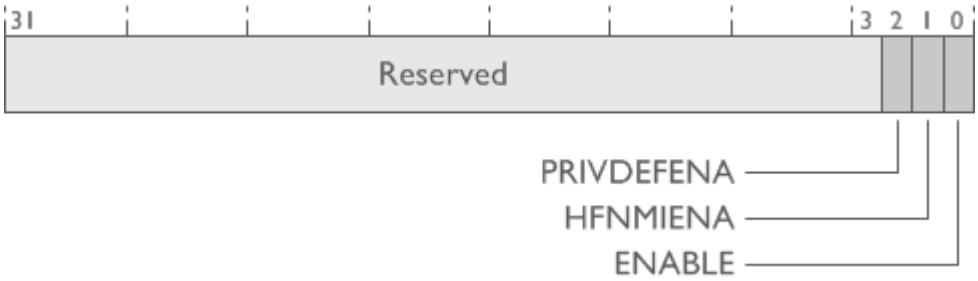

Figure 4-2 MPU_CTRL bit assignments

Table 4-2 MPU_CTRL bit assignments

Bits	Field	Reset	Description
[31:3]	Reserved – read as 0	0	Reserved.
[2]	PRIVDEFENA	0	Privileged background region enable: `0b1` Enables the default memory map for privilege code when the address accessed does not map into any MPU region. Unprivileged accesses to unmapped addresses result in faults. `0b0` All accesses to unmapped addresses result in faults.

Table 4-2 MPU_CTRL bit assignments (continued)

Bits	Field	Reset	Description
[1]	HFNMIENA	0	MPU Enable for HardFault and NMI (Non-Maskable Interrupt): `0b1` MPU access rules apply to HardFault and NMI handlers. `0b0` HardFault and NMI handlers bypass MPU configuration as if MPU is disabled.
[0]	ENABLE	0	Enable control: `0b1` MPU is enabled. `0b0` MPU is disabled.

4.3 MPU_RNR

The MPU Region Number Register selects the region that is accessed by the MPU_RBAR and MPU_RLAR.

Figure 4-3 MPU_RNR bit assignments

Table 4-3 MPU_RNR bit assignments

Bits	Field	Reset	Description
[31:8]	Reserved – read as 0	0	Reserved.
[7:0]	REGION	Unknown	Region number. Selects and indicates the region that is accessed by the MPU_RBAR and MPU_RLAR. Bit [7:2] of the region number is also used to select region number when accessing region setup via alias registers (MPU_RBAR_A{n} and MPU_RLAR_A{n}).

4.4 MPU_RBAR

The MPU Region Base Address Register defines the starting address of an MPU region and access permission.

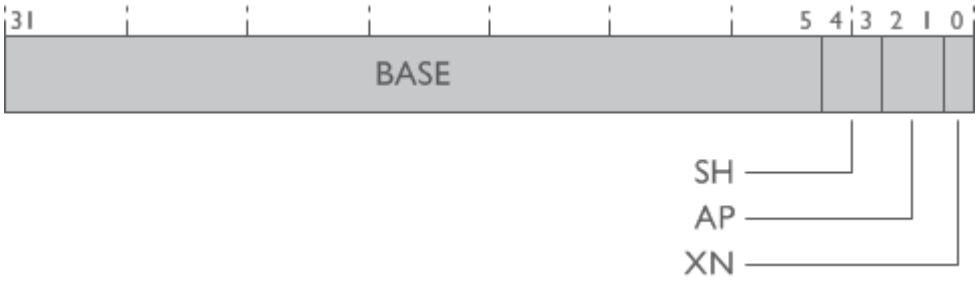

Figure 4-4 MPU_RBAR bit assignments

Table 4-4 MPU_RBAR bit assignments

Bits	Field	Reset	Description
[31:5]	BASE	Unknown	Starting address of MPU region address (bits [31:5] – the address must be aligned to multiple of 32 bytes).
[4:3]	SH	Unknown	Shareability for Normal memory: `0b00` Non-shareable. **01** Outer shareable. **10** Inner shareable. This field is ignored if the memory attribute is set to Device memory type.

Table 4-4 MPU_RBAR bit assignments (continued)

Bits	Field	Reset	Description
[2:1]	AP[2:1]	Unknown	Access permissions: `0b00` Read/write by privileged code only. `0b01` Read/write by any privilege level. `0b10` Read only by privileged code only. `0b11` Read only by any privilege level.
[0]	XN	Unknown	eXecute Never attribute: `0b0` Allow program execution in this region. `0b1` Disallow program execution in this region.

4.5 MPU_RLAR

The MPU Region Limit Address Register defines the ending address of an MPU region, region enable, and an indirection index to memory attribute array.

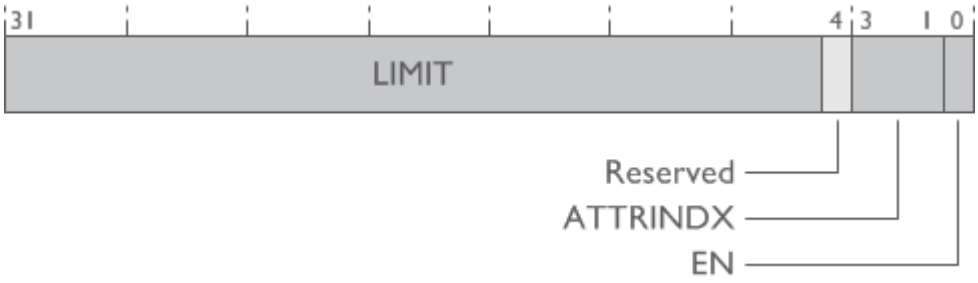

Figure 4-5 MPU_RLAR bit assignments

Table 4-5 MPU_RLAR bit assignments

Bits	Field	Reset	Description
31:5	LIMIT	Unknown	Ending address (upper inclusive limit)of MPU region address (bits [31:5] – the address must be aligned to multiple of 32 bytes). Bit [4:0] of the address value is assigned with 0x1F to provide the limit address to be checked against.
4	Reserved	0	Reserved.
3:1	AttrIndx	Unknown	Attribute Index. Select memory attributes from attribute sets in MPU_MAIR0 and MPU_MAIR1.
0	EN	0	Region enable.

4.6 MPU_RBAR_A1/2/3 and MPU_RLAR_A1/2/3

An alias of MPU_RBAR register to allow faster programming of different MPU regions. The region number that is selected when using MPU_RBARn and MPU_RLARn is equal to (MPU_RNR[7:2]<<2) + n.

For example:

Condition	When Accessing	MPU Region accessed
MPU_RNR=0	MPU_RBAR / MPU_RLAR	0
	MPU_RBAR_A1 / MPU_RLAR_A1	1
	MPU_RBAR_A2 / MPU_RLAR_A2	2
	MPU_RBAR_A3 / MPU_RLAR_A3	3
MPU_RNR=4	MPU_RBAR / MPU_RLAR	4
	MPU_RBAR_A1 / MPU_RLAR_A1	5
	MPU_RBAR_A2 / MPU_RLAR_A2	6
	MPU_RBAR_A3 / MPU_RLAR_A3	7

MPU_RBAR_A1/2/3 and MPU_RLAR_A1/2/3 enables software to program multiple MPU regions quickly without the need to reprogram MPU_RNR every time.

4.7 MPU_MAIR0, MPU_MAIR1

The MPU Attribute Indirection Register 0 and 1 provide eight sets of 8-bit memory attributes, which can be referenced by AttrIndx in MPU_RLAR to determine the memory attribute for an MPU region.

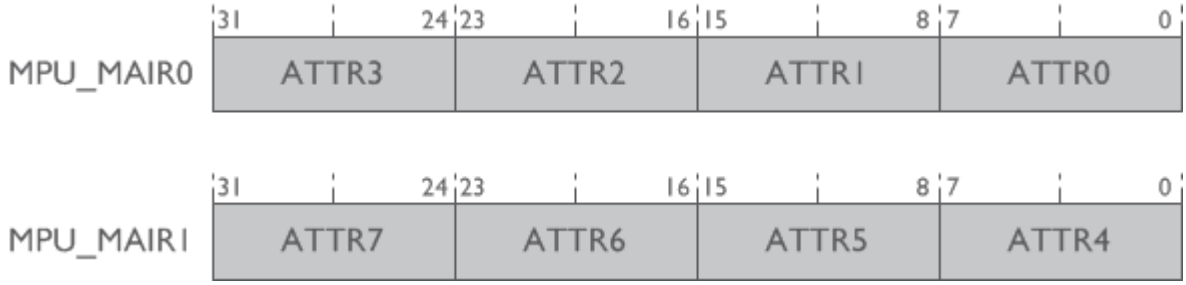

Figure 4-6 MPU_MAIR0, MPU_MAIR1 bit assignments

4.8 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 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 DMB followed by an *Instruction Synchronization Barrier* (ISB). The following figure summaries the various stages that are required to configure an MPU region.

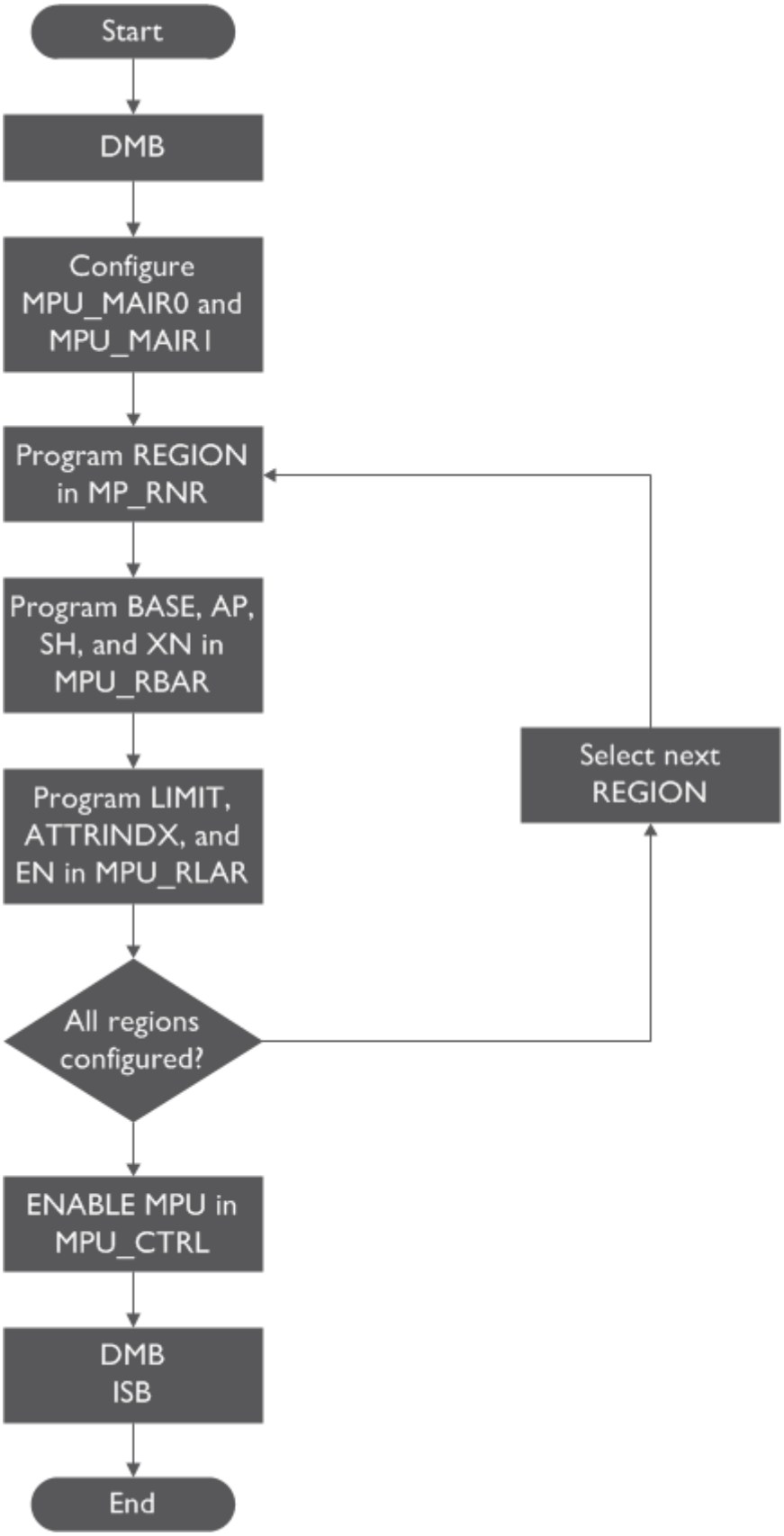

Figure 4-7 Configuring an MPU region

Chapter 5
CMSIS MPU support

ARMv8-M processors provide software support with an initiative called the *Cortex Microcontroller Software Interface Standard* (CMSIS). This chapter lists the standardized names for MPU registers, and provides configuration settings to initialize CMSIS MPU.

It contains the following sections:

5.1 CMSIS-CORE

One of the projects within CMSIS is CMSIS-CORE, a standardized *Hardware Abstraction Layer* (HAL) for accessing processor features. CMSIS-CORE is integrated in device driver code that is provided by microcontroller vendors, and being integrated into various software development suites.

Inside the processor-specific header files in CMSIS-CORE, the MPU registers are defined with a data structure (typedef) which provides standardized names for MPU registers.

Table 5-1 Standardized names for MPU registers

Register	CMSIS symbols	CMSIS symbols for Non-secure alias	Descriptions
MPU_TYPE	MPU->TYPE	MPU_NS->TYPE	MPU Type Register
MPU_CTRL	MPU->CTRL	MPU_NS ->CTRL	MPU Control Register
MPU_RNR	MPU->RNR	MPU_NS ->RNR	MPU Region Number Register
MPU_RBAR	MPU->RBAR	MPU_NS ->RBAR	MPU Region Base Address Register
MPU_RLAR	MPU->RLAR	MPU_NS ->RLAR	MPU Region Base Limit Register
MPU_MAIR0	MPU->MAIR0	MPU_NS ->MAIR0	MPU Memory Attribute Indirection Register 0
MPU_MAIR1	MPU->MAIR1	MPU_NS ->MAIR1	MPU Memory Attribute Indirection Register 1

The ARMv8-M architecture support in CMSIS starts from CMSIS version5.0.

The following settings are used to initialize the MPU in CMSIS:

```
__DMB(); /* Force any outstanding transfers to complete before disabling MPU */

/* Disable MPU */
MPU->CTRL = 0;

/* Configure memory types */

/* MPU_MAIR0 index 0: Normal-Outer-Non-cacheable-Inner-Non-cacheable */
MPU->MAIR0 |= (NORMAL_O_NC | NORMAL_I_NC);

/* MPU_MAIR0 index 1: Device-nGnRnE */
MPU->MAIR0 |= (DEVICE_NG_NR_NE << MPU_MAIR0_Attr1_Pos);

/* Configure region 0: Normal, Non-Shareable, RO, Any Privilege Level)*/
/* Region start = 0x0 Region end = 0x007FFFFF */
MPU->RNR = 0;
MPU->RBAR = (0x00000000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RO_P_U;
MPU->RLAR = (0x007FFFFF & MPU_RLAR_LIMIT_Msk) | ((0 << MPU_RLAR_AttrIndx_Pos) &
MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;

/* Configure region 1: Device-nGnRnE, RW, Any Privilege Level, XN) */
/* Region start = 0x40010000 Region end = 0x40013FFF */
MPU->RNR = 1;
MPU->RBAR = (0x40010000 & MPU_RBAR_ADDR_Msk) | RW_P_U | EXEC_NEVER;
MPU->RLAR = (0x40013FFF & MPU_RLAR_LIMIT_Msk) | ((1 << MPU_RLAR_AttrIndx_Pos) &
MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;

MPU->CTRL |= 1; /* Enable the MPU */

__DSB(); /* Force memory writes before continuing */
__ISB(); /* Flush and refill pipeline with updated permissions */
```