ARMv8-M Memory Protection Unit

Version 2.1

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
08 July 2016	0100-00	Non-Confidential	First release
24 August 2016	0101-00	Non-Confidential	Second release
28 February 2017	0200-00	Non-Confidential	Third release
09 April 2019	0201-00	Non-Confidential	Fourth release

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

1 The Memory Protection Unit

The Memory Protection Unit (MPU) in ARMv8-M series processors is a programmable unit inside the processor that allows privileged software, typically an OS kernel, to define memory access permissions to regions within the 4GB memory space. All memory access is monitored by the MPU, including instruction fetches and data accesses from the processor, which can trigger a fault exception when an access violation is detected.

The MPU can:

- Prevent stack overflows in one task from corrupting memory belonging to another task.
- Define regions of memory where access is never allowed by instruction fetches, so preventing any potential malicious code from being executed from those regions.
- Secure regions of RAM and SRAM from accidental corruption by defining those regions as read-only.
- Define regions of memory as "shareable" when multiple masters in the system have access to that region. By being shareable, the system is required to ensure coherency for that region among its masters.

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 MPU can define 0, 4, 8, 12, or 16 regions for any implementation of the Cortex-M23 or Cortex-M33 processors. Previously, the number of regions allowed on a Cortex-M0+, Cortex-M3, or Cortex-M4 was either 0 or 8.

- 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 size 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 eXecute Never (XN) attribute enables separation of code and data regions.

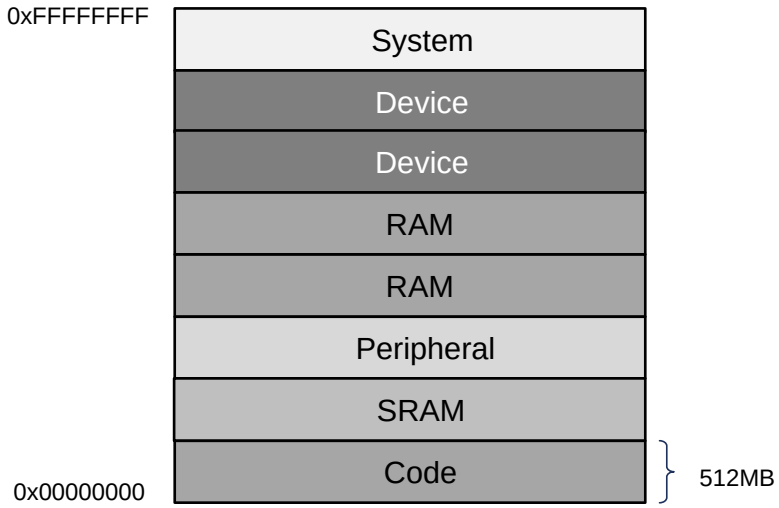
If the ARMv8-M Security Extension is included, the Secure and Normal worlds can each have their own MPU. The number of regions in the Secure and Non-secure MPU can be configured independently and each can be programmed to protect memory for the associated Security state.

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 Normal 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).

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

1.1 The default memory map

ARMv8-M is a memory-mapped architecture with shared address space for physical memory and processor control and status registers. At reset, and before the MPU is configured or initialized, the memory is divided into 8 x 512MB segments.



The definitions of these regions are as follows:

Code	0x00000000-0x1FFFFFFF	Memory to hold instructions Typically ROM or flash memory	Code execution allowed
SRAM	0x20000000-0x3FFFFFFF	Fast SRAM memory, on-chip RAM	
2xRAM	0x60000000-0x9FFFFFFF	Typically RAM memory, usually off-chip RAM	
Peripheral	0x40000000-0x5FFFFFFF	Peripheral memory space, on-chip	Code execution not allowed eXecute Never (XN)
2xDevice	0xA0000000-0xDFFFFFFF	Peripheral memory space, off-chip	
System	0xE0000000-0xFFFFFFFF	Contains memory mapped registers	

You can use the MPU to redefine all of these regions. As the MPU is an optional component, in a system with the ARMv8-M Security Extensions, you can have, for example, a Non-secure MPU defining regions in the Non-secure memory space, while the Secure memory space is left in the default configuration.

2 Protected Memory System Architecture

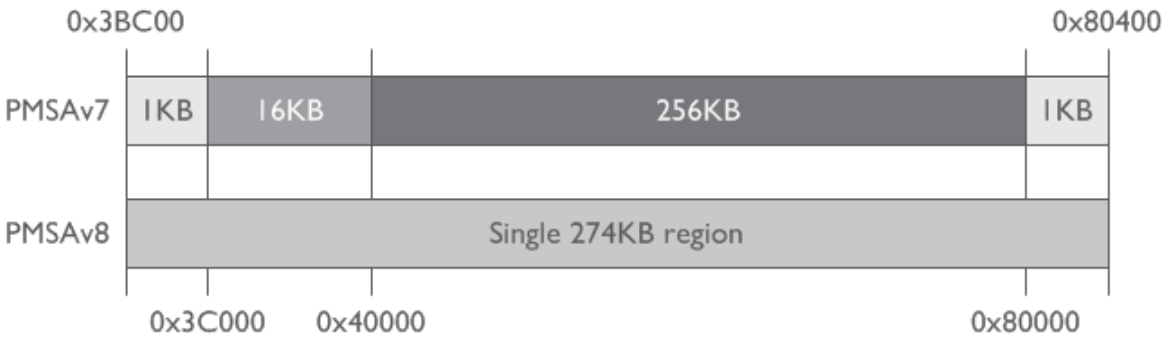
The Protected Memory System Architecture (PMSAv8) is the architecture that defines the operation of the MPU inside the ARM processors.

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 versions of the M-profile ARM architecture.

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.

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, using the PMSAv7 architecture, multiple MPU region registers are required, as the following figure shows:



In the ARMv8-M architecture, the size of an MPU region can be any size (including 274KB) but must have a a granularity of 32 bytes.

- PMSAv8 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.
- An individual MPU region is defined by:

```
Address >= MPU_RBAR.BASE:‘00000’ && Address <= MPU_RLAR.LIMIT:‘11111
```

Note

Many features of the ARMv8-M architecture are not implemented in ARMv8-M implementations, for example, Caches, Write Buffers and *Tightly Coupled Memories* (TCMs) are architectural features that are not currently in any ARMv8-M implementations.

The ARMv8-M architecture with Main Extension has 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.

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. System Space is always Execute never (XN).

3 Memory system and memory partitioning

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

The Secure memory space is further divided into two types:

Secure (S)

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

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

Non-secure Callable (NSC)

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

Non-secure (NS)

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

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

4 MPU registers

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

It is important to realize that all MPU registers are banked. If TrustZone is enabled, there is a set of MPU registers for the Secure state, and a mirror set for the Non-secure state. When accessing the MPU address between 0xE000ED90 and 0xE000EDC4, the type of MPU registers accessed is determined by the current state of the processor.

Non-secure code can access Non-secure MPU registers and Secure code can access Secure MPU registers. In addition, Secure code can access Non-secure MPU registers at their aliased address.

Secure access sees Secure MPU registers, Non-secure access sees Non-secure MPU registers. Secure software can also access Non-secure MPU registers using the alias address.

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

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.

Note

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

All MPU registers:

- Are privileged access only. Unprivileged accesses generate a fault exception.
- 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.

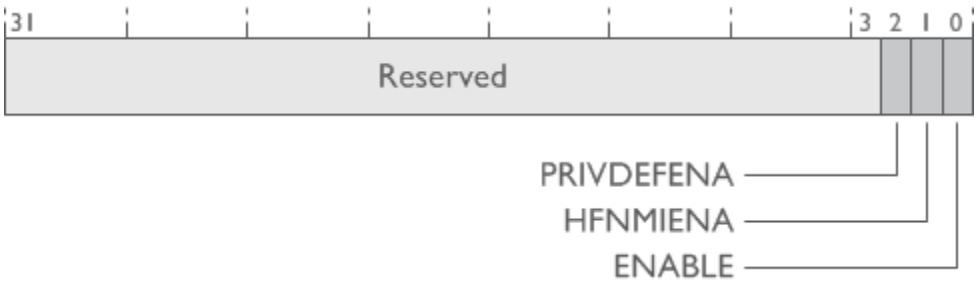
4.1 MPU_TYPE

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

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 selected security state.
7:1	Reserved – read as 0	0	Reserved
0	SEPARATE	0	Indicates support for separate instruction data address regions. ARMv8-M 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.



Bits	Field	Reset	Description

31:3	Reserved – read as 0	0	Reserved
2	PRIVDEFENA	0	Privileged background region enable. When set to 1, this 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. When cleared to 0, all accesses to unmapped addresses result in faults.
1	HFNMIENA	0	MPU Enable for HardFault and NMI (Non-Maskable Interrupt). When set to 1, MPU access rules apply to HardFault and NMI handlers. When cleared to 0, HardFault and NMI handlers bypass MPU configuration as if MPU is disabled.
0	ENABLE	0	Enable control. When set to 1, the MPU is enabled. When cleared to 0, the MPU is disabled.

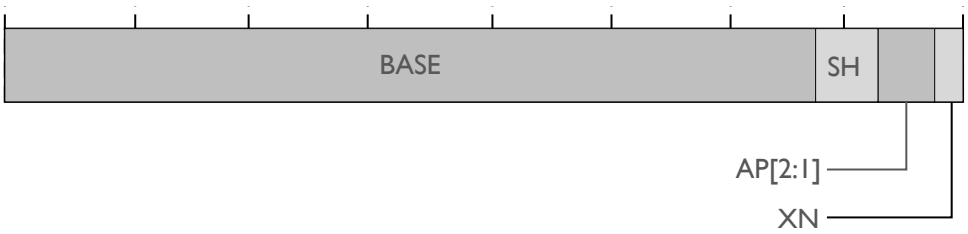
4.3 MPU_RNR

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

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. Bits [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

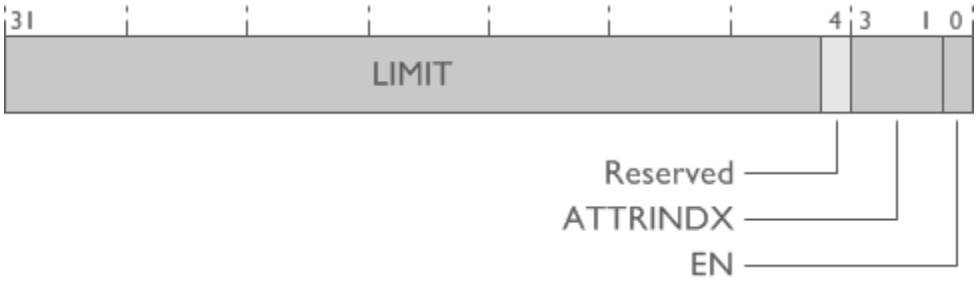
The MPU Region Base Address Register defines the starting address of an MPU region and access permission.



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 00 - Non-shareable 01 - Outer shareable 10 - Inner Shareable This field is ignored if the memory attribute is set to Device memory type.
2:1	AP[2:1]	Unknown	Access permissions. 00 – read/write by privileged code only 01 – read/write by any privilege level 10 – Read only by privileged code only 11 – Read only by any privilege level
0	XN	Unknown	eXecute Never attribute 0 – allow program execution in this region 1 – 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.



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). Bits [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

This is an alias of the MPU_RBAR register to allow faster programming of different MPU regions. The region number that is selected when using MPU_RBAR*n* and MPU_RLAR*n* 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

This 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 memory attributes, which can be referenced using the AttrIndx field in MPU_RLAR to determine the memory attribute for an MPU region.

The format of each memory attribute (ATTR1 through ATTR7) is described by MAIR_ATTR in the Armv8-M Architecture Reference Manual.

5 Memory type definitions in ARMv8-M architecture

Note

If you are not intending to implement or use a cache or write buffer, then some of the information within this module, for example cache attributes, is irrelevant.

In ARMv8-M architecture, memory types are divided into:

- Normal memory.
- Device memory.

Note

The Strongly Ordered (SO) device memory type in ARMv6-M and ARMv7-M is now a subset of the device memory type.

5.1 Normal Memory

The Normal Memory type is intended to 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 re-ordering 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 MMIO registers. The Device memory type is intended for that use.

Note

The Normal Memory definition remains mostly unchanged from the ARMv7-M architecture.

Normal memory can have several attributes 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 attribute can be further divided:

Cache policy	Write-Through / Write-Back.
Allocation	Cache line allocation hints, for read and write accesses.
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.

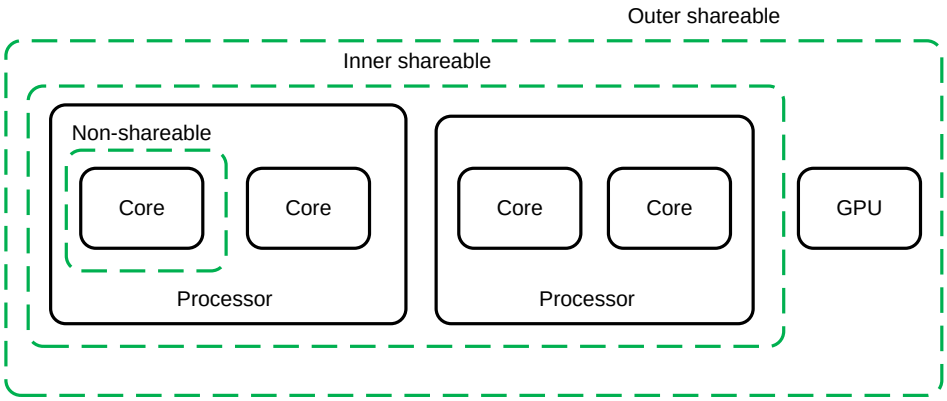
Typically, the inner cache attribute is used by any integrated caches, while the outer cache attributes are exported on using the bus system sideband signals. Depending on the processor implementation, the inner cache attributes can also be exported to the memory system using extra sideband signals.

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 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.

The architecture manages this by grouping all masters into shareability groups. An example of this is shown in the following figure.



The following shareability domain options are available:

Non-shareable

This 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

This 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

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.

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 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 one level of RAM any master can see any memory update.
- In a system with caches, not all masters can access all caches.
    - The system might employ hardware cache coherency to make updates visible.
    - The system might treat any shareable memory as non-cacheable, making updates visible.

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

5.2 Device Memory

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

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

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

Note

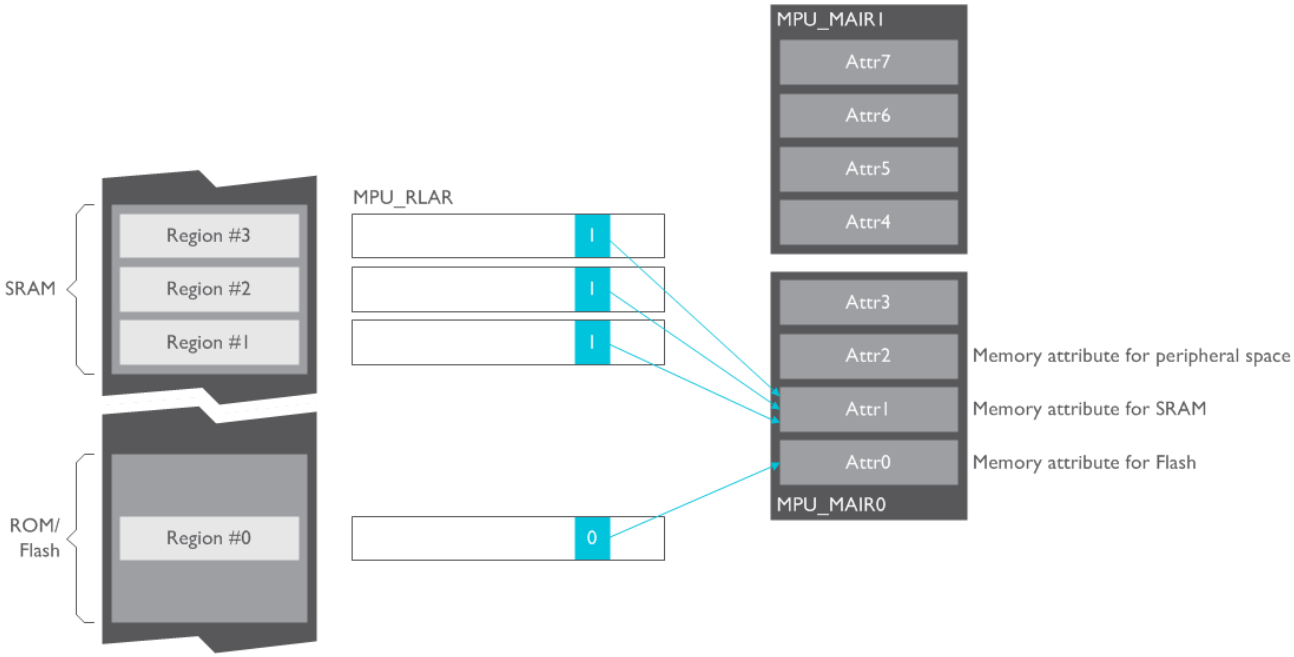
For most simple processor designs, reordering, and gathering (merging of transactions) do not occur even if the memory attribute configuration allows it to do so.

Device memory is shareable, and must be cached.

6 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.

The following figure shows an example of 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.

7 CMSIS

Cortex-M series processors provide software support with an initiative called the Cortex Microcontroller Software Interface Standard (CMSIS). 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.

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 support in CMSIS starts from CMSIS version 5.0. For more information, see www.arm.com/cmsis and http://www.keil.com/pack/doc/CMSIS/Core/html/index.html.

While is possible to use the ARM assembly language to perform the initialization procedure, it is not particularly efficient. To disable the MPU using assembly language you might use:

```
MPU_CTRL EQU 0xE000ED94             // Address of the Non-secure register
LDR R0, =MPU_CTRL                   // Copy MPU_CTRL into R0
LDR R1, [R0]                        // Read MPU_CTRL
ORR R1, R1, #(0x0 << 20)            // Set bit 0 to disable the MPU
STR R1, [R0]                        // Write back the modified value to
// MPU_CTRL
DSB
ISB
```

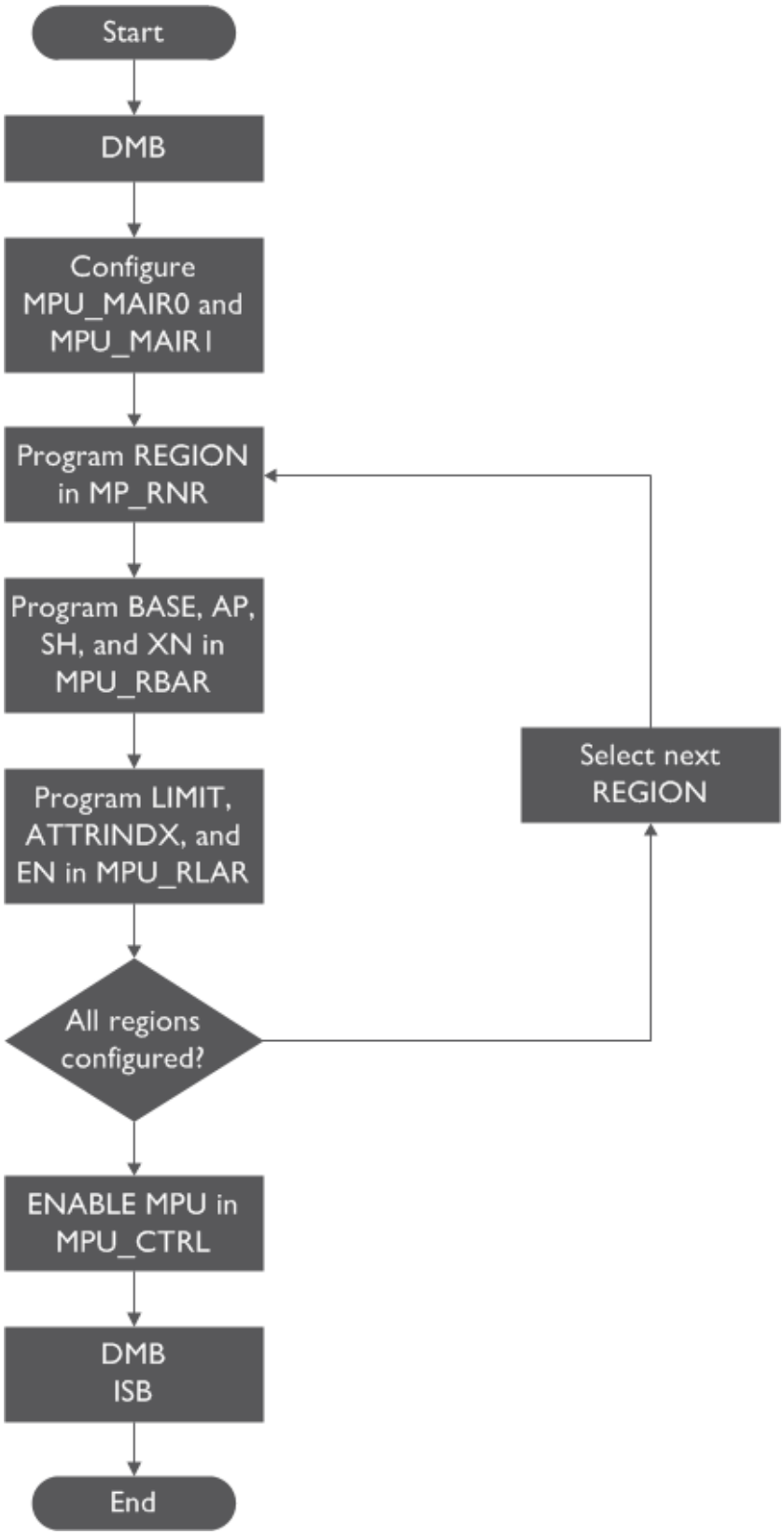
While the CMSIS equivalent is much more concise.

```
MPU->CTRL = 0;
```

ARM therefore recommends that you use CMSIS to control Cortex-M series processors.

8 Initializing and configuring an MPU

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 then 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.

When all the required regions have been configured, the MPU can then 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 example code sets up and configures a number of memory regions and sets the memory attributes for each region:

Include the CMSIS headers and use the CMSIS MPU structure to access the MPU configuration registers.

```
void armv8mml_mpu_config(void);
/* General MPU Masks */
#define REGION_ENABLE   0x00000001
/* Shareability */
#define NON_SHAREABLE   0x00
#define RESERVED        0x08
#define OUTER_SHAREABLE 0x10
#define INNER_SHAREABLE 0x18

/* Access Permissions */
#define EXEC_NEVER      0x01 /* All instruction fetches abort */
#define RW_P_ONLY       0x00 /* Read/Write, Privileged code only */
#define RW_P_U          0x02 /* Read/Write, Any Privilege Level */
#define RO_P_ONLY       0x04 /* Read-Only, Privileged code only */
#define RO_P_U          0x06 /* Read-Only, Any Privilege Level */

/* Read/Write Allocation Configurations for Cacheable Memory
Attr<n>[7:4] and Attr<n>[3:0] are of the format: 0bXXRW */
#define R_NON_W_NON     0x0 /* Do not allocate Read/Write */
#define R_NON_W_ALLOC   0x1 /* Do not allocate Read, Allocate Write */
#define R_ALLOC_W_NON   0x2 /* Allocate Read, Do not allocate Write */
#define R_ALLOC_W_ALLOC 0x3 /* Allocate Read/Write */

/* Memory Attribute Masks */
#define DEVICE          0x0F
#define NORMAL_OUTER    0xF0
#define NORMAL_INNER    0x0F

/* Memory Attributes */
#define DEVICE_NG_NR_NE 0x00 /* Device, Non-Gathering, Non-Reordering, Non-
Early-Write-Acknowledgement */
#define DEVICE_NG_NR_E  0x04 /* Device, Non-Gathering, Non-Reordering, Early-
Write-Acknowledgement */
```

```
#define DEVICE_NG_R_E   0x08 /* Device, Non-Gathering, Reordering, Early-
Write-Acknowledgement */
#define DEVICE_G_R_E    0x0C /* Device, Gathering, Reordering, Early-Write-
Acknowledgement */

#define NORMAL_O_WT_T   0x00 /* Normal, Outer Write-through transient (if RW
not 00) */
#define NORMAL_O_NC     0x40 /* Normal, Outer Non-cacheable (if RW is 00) */
#define NORMAL_O_WB_T   0x40 /* Normal, Outer Write-back transient (if RW not
00) */
#define NORMAL_O_WT_NT  0x80 /* Normal, Outer Write-through non-transient */
#define NORMAL_O_WB_NT  0xC0 /* Normal, Outer Write-back non-transient */

#define NORMAL_I_WT_T   0x00 /* Normal, Inner Write-through transient (if RW
not 00) */
#define NORMAL_I_NC     0x04 /* Normal, Inner Non-cacheable (if RW is 00) */
#define NORMAL_I_WB_T   0x04 /* Normal, Inner Write-back transient (if RW not
00) */
#define NORMAL_I_WT_NT  0x08 /* Normal, Inner Write-through non-transient */
#define NORMAL_I_WB_NT  0x0C /* Normal, Inner Write-back non-transient */
```

Any outstanding memory transactions must be forced to complete by executing a DMB instruction and the MPU disabled before it can be configured.

```
#include <CMSDK_ARMv8MML.h>
#include <core_armv8mml.h>
#include "mpu.h"
void armv8mml_mpu_config(void)
{
   /* Force any outstanding transfers to complete before disabling MPU */
   __asm volatile("dmb\n");
   /* Disable MPU */
   MPU->CTRL = 0;
   /* Enable Memory Management Fault Handler using SHCSR */
   SCB->SHCSR |= (1 << SCB_SHCSR_MEMFAULTENA_Pos);
   /* Check number of supported MPU regions */
   uint32_t MPU_regions = (MPU->TYPE & MPU_TYPE_DREGION_Msk) >>
MPU_TYPE_DREGION_Pos;
```

Any memory types that are to be assigned to a region must be encoded into the MAIR registers. The encoded types can then be indexed by the Region Limit Register for that region. Region 0 is configured as Normal memory, inner or outer non-cacheable. Region 1 is Device-nGnRnE.

```
   /* Configure MPU Memory Attribution Indirection Registers */
   uint32_t non_cacheable_attr = NORMAL_O_NC | NORMAL_I_NC;
   uint32_t device_attr = DEVICE_NG_NR_NE;
   MPU->MAIR0 = MPU->MAIR0 | ((device_attr << MPU_MAIR0_Attr1_Pos) |
(non_cacheable_attr));
   /* MPU_MAIR0 index 0: SRAM1, SRAM2&3, System ROM
      MPU_MAIR0 index 1: Device */
```

Regions are configured by encoding the region start address, shareability, executability, and read/writeability in the Region Base Address Register. The Region Limit Register holds the limit address of the region and an index value that selects the appropriate type from those encoded in the MAIRs.

This example selects region zero and configure it to cover the addresses `0x0` to `0x007FFFFF`. The memory type is indexed to MAIR region 0 (Normal, non-cacheable) and the regions is configured as Non-shareable, executable, and read-only. This memory type might be used to cover the initial boot code and exception handler stored in ROM or Flash.

```
    /* Configure region 0: NS SRAM1 (Normal, Non-Shareable, RO, Any Privilege
Level)*/
    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;
```

Data regions could be configured in a similar way but would usually be configured to be writeable and eXecute Never.

Normal memory is unsuitable for peripheral control registers as it allows speculative read accesses. Code must configure one or more device regions to cover peripheral register space.

Device memory should always be configured as eXecute Never to prevent instruction speculation.

Other regions can be configured in the same way. In a system with caches some regions of Normal memory might be configured as inner or outer cacheable. Some Normal memory regions might be only accessible to privileged code and peripheral region access might also be limited to privileged code.

When all regions have been programmed any unused regions should be disabled. The total number of regions can be read from the MUP_TYPE register.

```
    /* Configure region 1: NS SRAM2&3 (Normal, Non-Shareable, RW, Any
Privilege Level) */
    MPU->RNR = 1;
    MPU->RBAR = (0x20000000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RW_P_U;
    MPU->RLAR = (0x207FFFFF & MPU_RLAR_LIMIT_Msk) | ((0 <<
MPU_RLAR_AttrIndx_Pos) & MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;
    /* Configure region 2: NS CMSDK APB (Device-nGnRnE, Non-Shareable, RW, Any
Privilege Level, XN) */
    MPU->RNR = 2;
    MPU->RBAR = (0x40000000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RW_P_U |
EXEC_NEVER;
    MPU->RLAR = (0x4000FFFF & MPU_RLAR_LIMIT_Msk) | ((1 <<
MPU_RLAR_AttrIndx_Pos) & MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;
    /* Configure region 3: NS GPIO (Device-nGnRnE, Non-Shareable, RW, Any
Privilege Level, XN) */
    MPU->RNR = 3;
```

```
    MPU->RBAR = (0x40010000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RW_P_U |
EXEC_NEVER;
    MPU->RLAR = (0x40013FFF & MPU_RLAR_LIMIT_Msk) | ((1 <<
MPU_RLAR_AttrIndx_Pos) & MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;
    /* Configure region 4: NS Default AHB Slave (Device-nGnRnE, Non-Shareable,
RO, Any Privilege Level, XN) */
    MPU->RNR = 4;
    MPU->RBAR = (0x40014000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RO_P_U |
EXEC_NEVER;
    MPU->RLAR = (0x40017FFF & MPU_RLAR_LIMIT_Msk) | ((1 <<
MPU_RLAR_AttrIndx_Pos) & MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;
    /* Configure region 5: System ROM (Normal, Non-Shareable, RO, Any
Privilege Level) */
    MPU->RNR = 5;
    MPU->RBAR = (0xF0000000 & MPU_RBAR_ADDR_Msk) | NON_SHAREABLE | RO_P_U;
    MPU->RLAR = (0xF0000FFF & MPU_RLAR_LIMIT_Msk) | ((0 <<
MPU_RLAR_AttrIndx_Pos) & MPU_RLAR_AttrIndx_Msk) | REGION_ENABLE;
    /* Disable all other regions */
    for (uint32_t i=6; i<MPU_regions; i++) {
          MPU->RNR = i;
          MPU->RLAR &= 0x0;
    }
```

The final step is to enable the MPU by writing to MPU_CTRL. Code should then execute a memory barrier to ensure that the register updates are seen by any subsequent memory accesses. An Instruction Synchronization Barrier (ISB) ensures the updated configuration used by any subsequent instructions.

```
    /* Enable MPU and the default memory map as a background region (acts as
region number -1) for privileged access only (MPU_CTRL.PRIVDEFENA). */
    MPU->CTRL = 5;
    /* Insert DSB followed by ISB */
    __asm volatile(
          "dsb\n"
      "isb\n"
          );
}

#include <CMSDK_ARMv8MML.h>
#include <core_armv8mml.h>
#include <stdio.h>

extern void armv8mml_mpu_config(void);
int main(void)
{
    armv8mml_mpu_config();    /* configure MPU regions */
    printf("MPU configured successfully!\r\n");
    while(1);
}
```