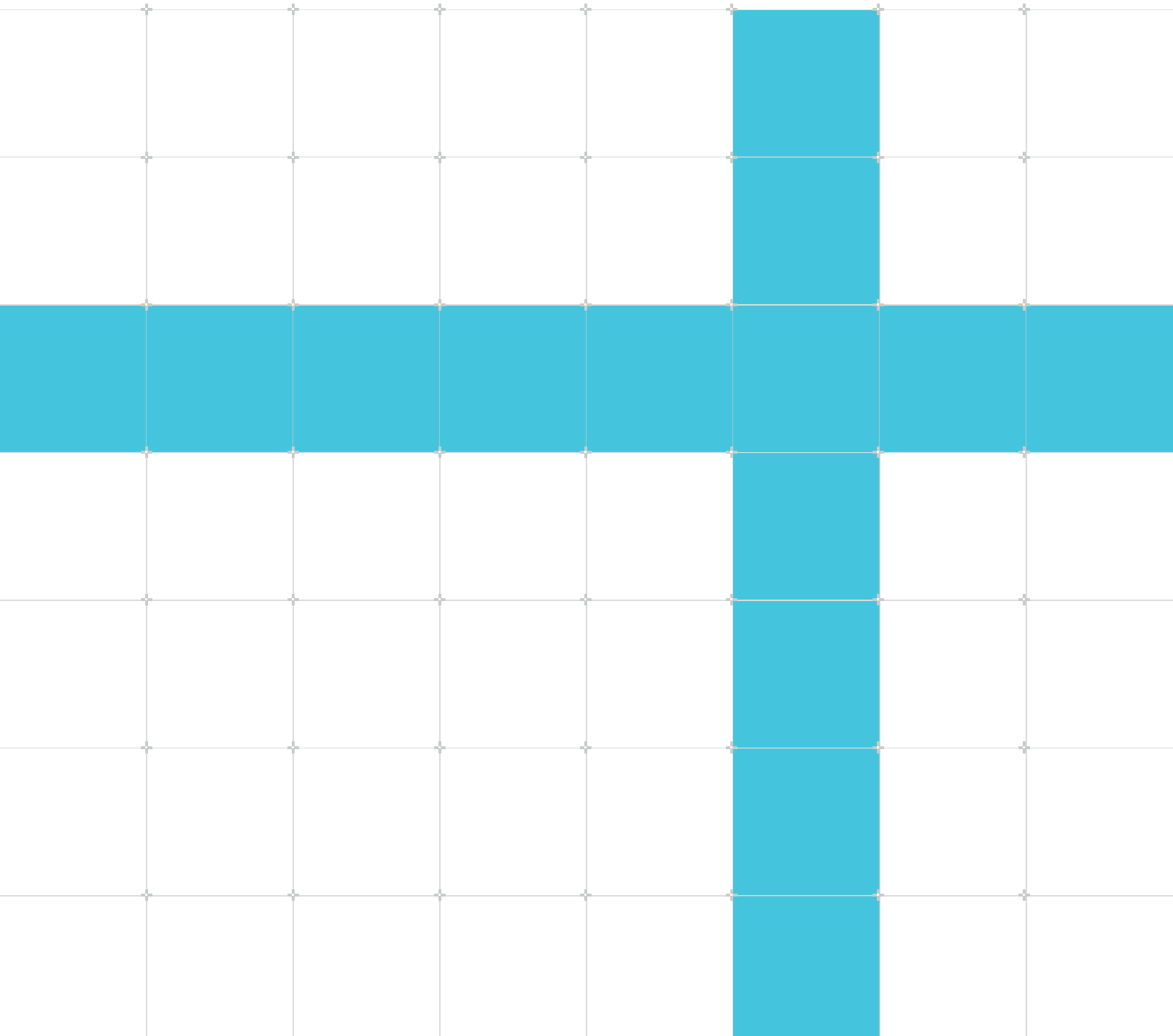
arm

Learn the architecture - AArch64 memory attributes and properties

Version 2.0

Issue 01
102376_0200_01_en



Learn the architecture - AArch64 memory attributes and properties

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	1 April 2019	Non-Confidential	Initial release
0200-01	14 November 2022	Non-Confidential	Update to memory attributes and properties. Added permission indirection and overlays.

Proprietary Notice

(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. Overview

This guide introduces the memory attributes and properties in Armv8-A and Armv9-A. It begins by explaining where attributes that describe memory come from and how they are assigned to regions of memory. Then it introduces the different attributes that are available and explains the basics of memory ordering.

This information is useful for anyone developing low-level code, like boot code or drivers. It is particularly relevant to anyone writing code to setup or manage the Memory Management Unit (MMU).

At the end of this guide, you can Check your knowledge. You will have learned about the different memory types and their key differences and you will also be able to list the memory attributes that can be applied to a given address.

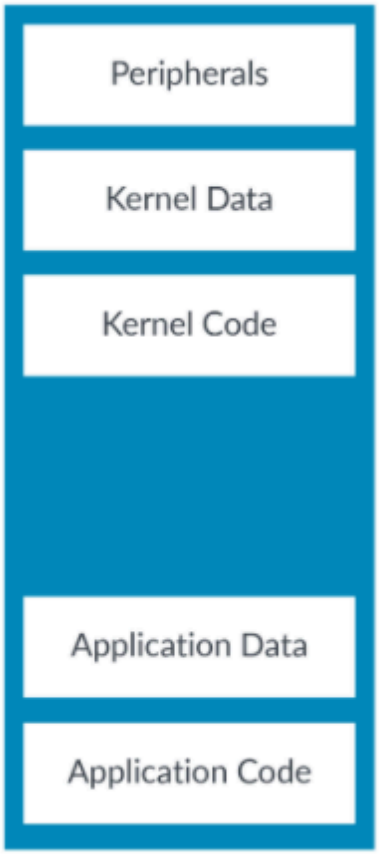
2. What are memory attributes and properties, and why are they needed

Memory attributes and properties are a way of defining how memory behaves. They provide a structure and a set of rules for you to follow when you configure how memory addresses, or regions of memory addresses, are accessed and used in your system.

The memory attributes and properties can be applied to an address, and define the rules associated with memory access.

Consider a simple system with the address space you can see in this diagram:

Figure 2-1: A diagram showing a simple address space.



The arrangement of memory regions in the address space is called an address map. In this example, the address map contains:

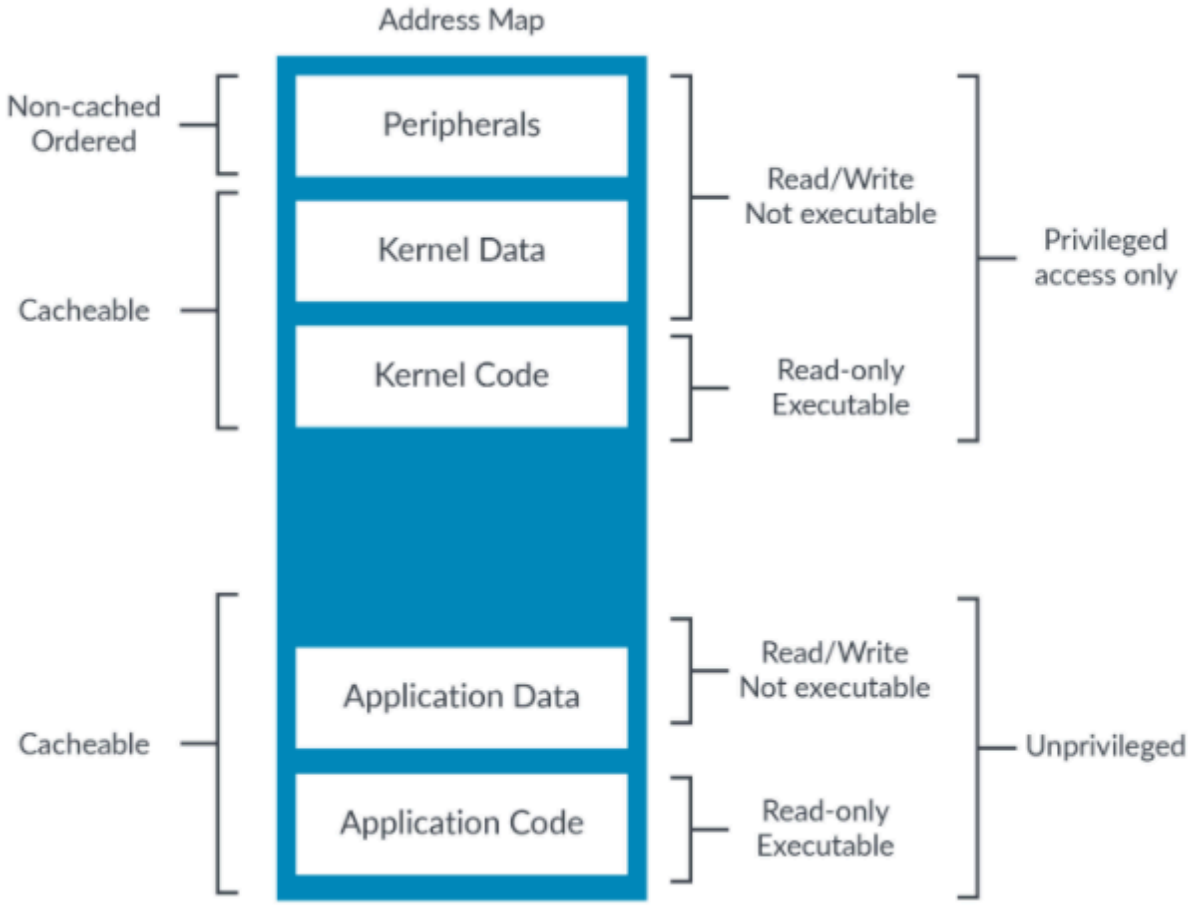
- Memory and peripherals
- In the memories, code and data

- Resources belonging to the OS and resources belonging to user applications

The way that the processor interacts with a peripheral is different to how it should interact with memory. For example, you usually want to cache memories but you do not want to cache peripherals. Caching is the act of storing a copy of information from memory into a hardware structure, which is called a cache. The cache is closer to the core and faster for the core to access. Similarly, you will usually want the processor to block user access to kernel resources such as peripherals.

The following diagram shows the address map with some different memory attributes that you might want to apply to the memory regions:

Figure 2-2: A diagram showing address map for memory regions



You need to be able to describe these different attributes to the processor, so that the processor accesses each location appropriately.

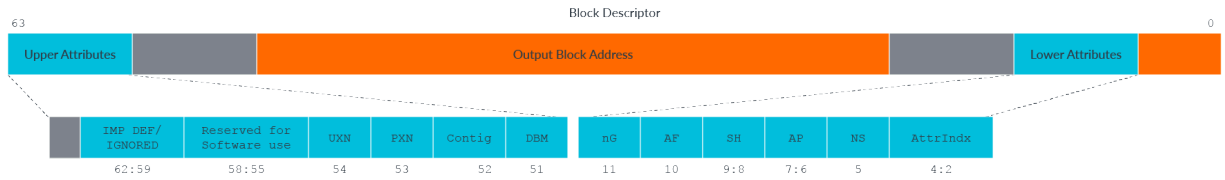
3. Describing memory in AArch64

The mapping between virtual and physical address spaces is defined in a set of translation tables, also sometimes called page tables. For each block or page of virtual addresses, the translation tables provide the corresponding physical address and the attributes for accessing that page.

Each translation table entry is called a block or page descriptor. In most cases, the attributes come from this descriptor.

This diagram shows an example block descriptor, and the attribute fields within it:

Figure 3-1: A diagram showing a block descriptor.



Important attributes discussed in this guide include the following:

- UXN and PXN - Execution permissions
- DBM - The dirty bit modifier
- AF - Access flag
- AP - Access permission
- AttrIndx - Selector for memory type and attributes

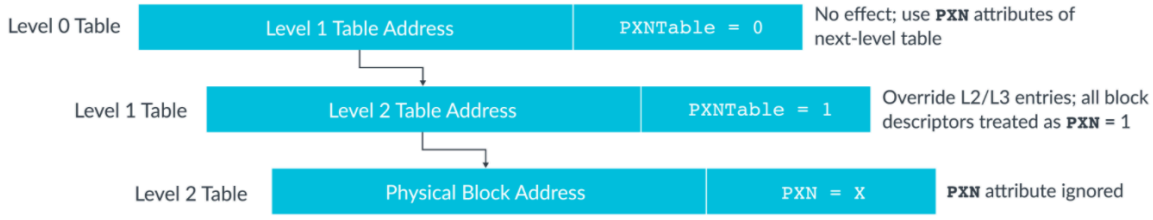
We will look at these attributes in more detail later in this guide.

3.1 Hierarchical attributes

Some memory attributes can be specified in the Table descriptors in higher-level tables. These are hierarchical attributes. This applies to Access Permission, Execution Permissions, and the Physical Address space.

If these bits are set then they override the lower-level entries, and if the bits are clear the lower-level entries are used unmodified. An example, using PXNTable (execution permission) is shown here:

Figure 3-2: A diagram showing a PXNTable with execution permission.



From Armv8.1-A, you can disable support for setting the access and execution permissions using the hierarchical attributes in a table descriptor. This is controlled by the Hierarchical Permission Disable (HPD) bits in the relevant TCR_ELx register. When disabled, the bits used for the hierarchical controls are available to software to use for other purposes.

3.2 MMU disabled

To summarize, the attributes for an address come from the translation tables. Translation tables are situated in memory and are used to store the mappings between virtual and physical addresses. The tables also contain the attributes for physical memory locations.

The translation tables are accessed by the Memory Management Unit (MMU).

What happens if the MMU is disabled? This is an important question to address when writing code that will run immediately after reset.

When the stage 1 MMU is disabled:

- All data accesses are Device-nGnRnE. This is explained in the section Device memory later in this guide.
- All instruction fetches are treated as either non-cacheable or cacheable, according to the value of the SCTLR_ELx.I (instruction cacheability control) field.
- All addresses have read/write access and are executable.

For Exception levels covered by virtualization, when stage 2 is disabled the attributes from stage 1 are used unmodified.

4. Memory types

All addresses in a system that are not marked as faulting are assigned a memory type. The memory type is a high-level description of how the processor should interact with the address region. There are two memory types in Armv8-A, and Armv9-A: Normal memory and Device memory.

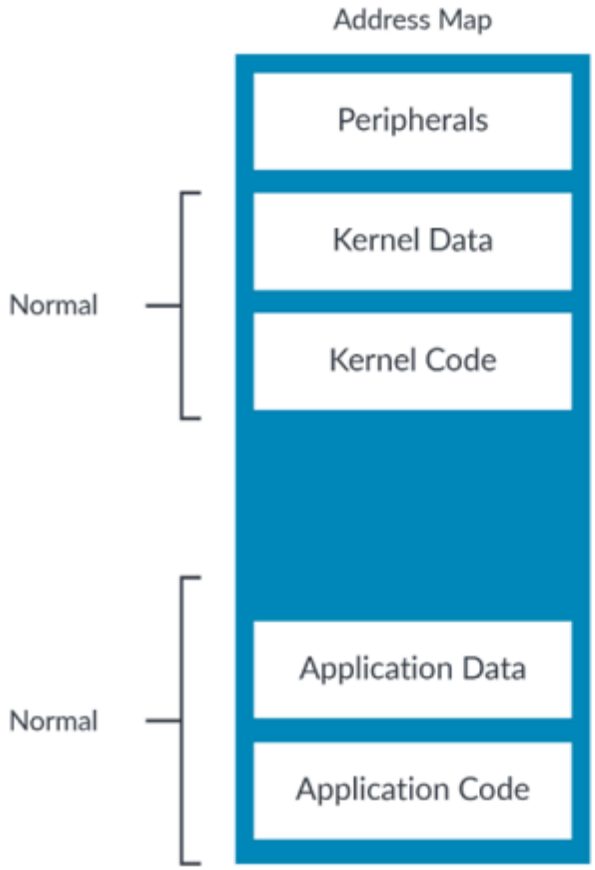
Armv6 and Armv7 include a third memory type: Strongly Ordered. In Armv8, this maps to Device-nGnRnE.

5. Normal memory

The Normal memory type is used for anything that behaves like a memory, including RAM, Flash, or ROM. Code should only be placed in locations marked as Normal.

Normal is usually the most common memory type in a system, as shown in this diagram:

Figure 5-1: A diagram showing the normal type.



5.1 Memory access ordering

Traditionally, computer processors execute instructions in the order that they were specified in the program. Things happen the number of times specified in the program and they happen one at a time. This is called the Simple Sequential Execution (SSE) model. Most modern processors may appear to follow this model, but in reality a number of optimizations are both applied and made available to you, to help speed up performance.

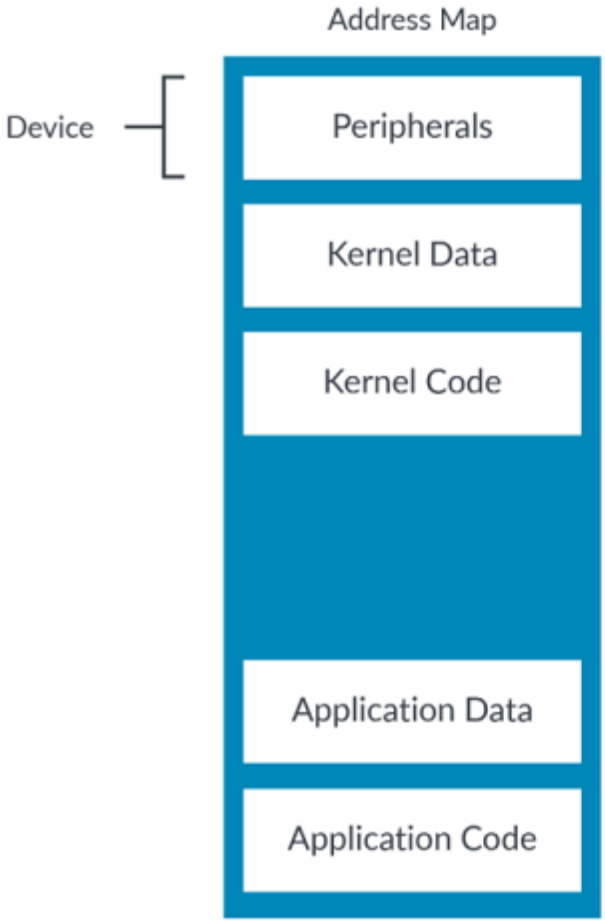
A memory location that is marked as Normal is assumed to have no direct side-effects when it is accessed. This means that reading the location just returns the data, but does not cause the data to change, or directly trigger another process. Because of this, for memory locations marked as Normal, a processor may merge accesses, perform accesses speculatively, or re-order accesses.

For more detail about memory access reordering, see the Memory system, ordering, and barriers guide.

6. Device memory

The Device memory type is used for describing peripherals. Peripheral registers are often referred to as Memory-Mapped I/O (MMIO). Here we can see what would be typically be marked as Device in our example address map:

Figure 6-1: A diagram showing memory mapped device type.



To review, the Normal memory type means that there are no side-effects to the access. For the Device type memory, the opposite is true. The Device memory type is used for locations that can have side-effects.

For example, a read to a FIFO would normally cause it to advance to the next piece of data. This means that the number of accesses to the FIFO is important, and therefore the processor must adhere to what is specified by the program.

Device regions are never cacheable. This is because it is very unlikely that you would want to cache accesses to a peripheral.

Speculative data accesses are not permitted to regions marked as Device. The processor can only access the location if it is architecturally accessed. That means that an instruction that has been architecturally executed has accessed the location.

Instructions should not be placed in regions marked as Device. We recommend that Device regions are always marked as not executable. Otherwise, it is possible that the processor might speculatively fetch instructions from it, which could cause problems for read-sensitive devices like FIFOs.

There is a subtle distinction here that is easy to miss. Marking a region as Device prevents speculative data accesses only. Marking a region as non-executable prevents speculative instruction accesses. This means that, to prevent any speculative accesses, a region must be marked as both Device and non-executable.

6.1 Sub-types of Device

There are four sub-types of Device, with varying levels of restrictions. These sub-types are the most permissive:

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

This sub-type is the most restrictive:

- Device-nGnRnE

The letters after Device represent a combination of attributes:

- Gathering (G, nG). This specifies that accesses can be merged (G) or not (nG). This could be merging multiple accesses to the same location into one access or merging multiple smaller accesses into one larger access.
- Re-ordering (R, nR). This specifies that accesses to the same peripheral can be re-ordered (R) or not (nR). When re-ordering is permitted, the restrictions apply in the same way as for the Normal type. You can find more detail on normal memory access reordering in the Memory system, ordering, and barriers guide.
- Early Write Acknowledgement (E, nE). This determines when a write is considered complete. If Early Acknowledgement is allowed (E), an access can be shown as complete once it is visible to other observers, but before it reaches its destination. For example, a write might become visible to other Processing Elements (PEs) once it reaches a write buffer in the interconnect. When Early Acknowledgement is not allowed (nE), the write must have reached the destination.

Here are two examples:

- Device-GRE. This allows gathering, re-ordering, and early write acknowledgement.
- Device-nGnRnE. This does not allow gathering, re-ordering, and early write acknowledgement.

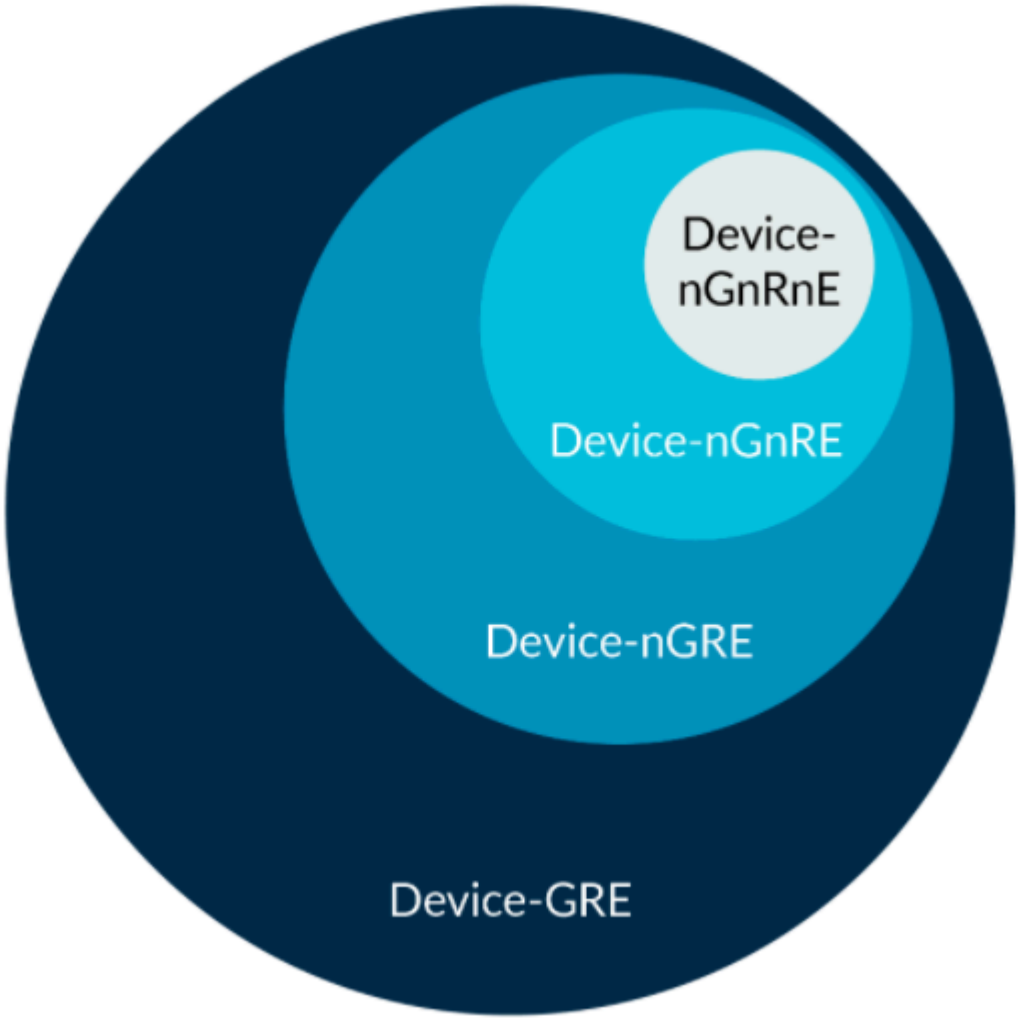
We have already seen how re-ordering works, but we have not introduced gathering or early write acknowledgement. Gathering allows memory accesses to similar locations to be merged into a single bus transaction, optimizing the access. Early write acknowledgement indicates to the memory system whether a buffer can send write acknowledgements at a point on the bus between the core and the peripheral at the address, such that all PEs can observe the write, even if the peripheral has not yet received the write.

Normal Non-cacheable and Device-GRE might appear to be the same, but they are not. Normal Non-cacheable still allows speculative data accesses, but Device-GRE does not.

6.2 Does the processor really do something different for each type?

The memory type describes the set of allowable behaviors for a location. Looking at just the Device type, this image represents the allowable behaviors:

Figure 6-2: A diagram showing device type.



You can see that Device-nGnRnE is the most restrictive sub-type, and has the fewest allowed behaviors. Device-GRE is the least restrictive, and therefore has the most allowed behaviors.

Importantly, all the behaviors allowed for Device-nGnRnE are also permitted for Device-GRE. For example, it is not a requirement for a Device-GRE memory to use Gathering - it is just allowed. Therefore, it would be permissible for the processor to treat Device-GRE as Device-nGnRnE.

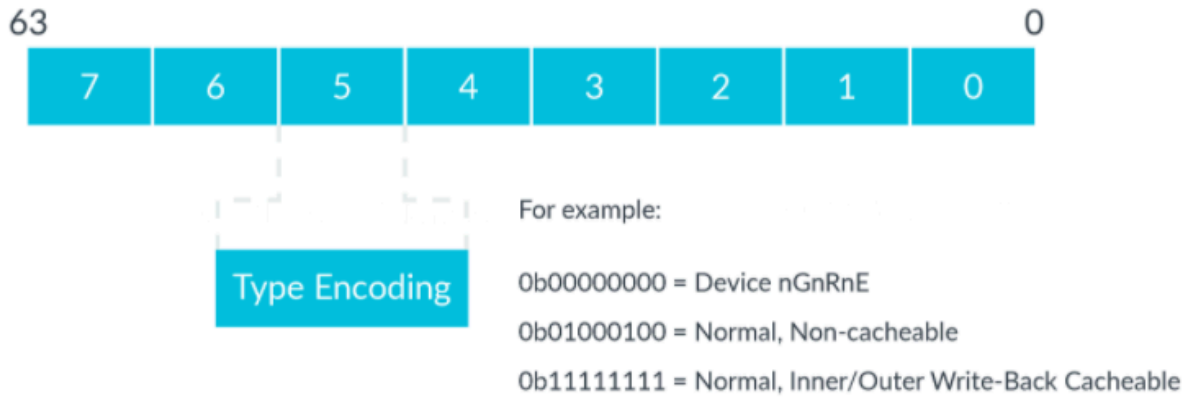
This example is extreme and unlikely to occur with Arm Cortex-A processors. However, it is common for processors to not differentiate between all the types and sub-types, for example treating Device-GRE and Device-nGRE in the same way. This is only allowed if the type or sub-type is always made more restrictive.

Some interconnects cannot fully support the requirements of Device-nGnRnE. For example, a Device-nGnRnE write to PCIe Base Address Register (BAR) space will become a posted write (a write that does not expect a write completion response) once it reaches the PCIe topology. In this case, the write access will only have the properties of Device-nGnRE because the write response cannot be provided by the target endpoint, and will instead be provided by some intermediate component like the PCIe Root Port. However, a Device-nGnRnE write to PCIe configuration space is a non-posted write (a write that expects a completion response), therefore the requirements of Device-nGnRnE are upheld for those types of accesses.

7. Describing the memory type

The memory type is not directly encoded into the translation table entry. Instead, the AttrIndx field in the translation table entry is used to select an entry from the MAIR_ELx (Memory Attribute Indirection Register).

Figure 7-1: A diagram showing memory attribute indirection register.



The selected field determines the memory type and cacheability information.

Why is an index to a register used, instead of encoding the memory type directly into the translation table entries? Because the number of bits in the translation table entries is limited. It requires eight bits to encode the memory type, but only three bits to encode the index into MAIR_ELx. This allows the architecture to efficiently use fewer bits in the table entries.

8. Cacheability and shareability attributes

Locations marked as Normal also have cacheability and shareability attributes. The cacheability attributes control whether a location can be cached. If a location can be cached, the shareability attributes control which other agents need to see a coherent copy of the memory. This allows for some complex configuration, which is beyond the scope of this guide. However, Arm expects operating systems to mark the majority of DRAM memory as Normal Write-back cacheable, Inner shareable.

9. Permissions

Access permissions can be encoded using either the direct permission scheme, or the indirect permission scheme. This chapter discusses the direct permission scheme. See the Permission indirection and permission overlay extensions section of this guide for details of the indirect permission scheme.

In the direct permissions scheme, the Access Permissions (AP) attribute controls whether a location can be read and written, and what privilege is necessary. The following table shows the AP bit settings:

AP	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No access	Read/write
01	Read/write	Read/write
10	No access	Read-only
11	Read-only	Read-only

If an access breaks the specified permissions, for example a write to a read-only region, an exception (labelled as a permission fault) is generated.

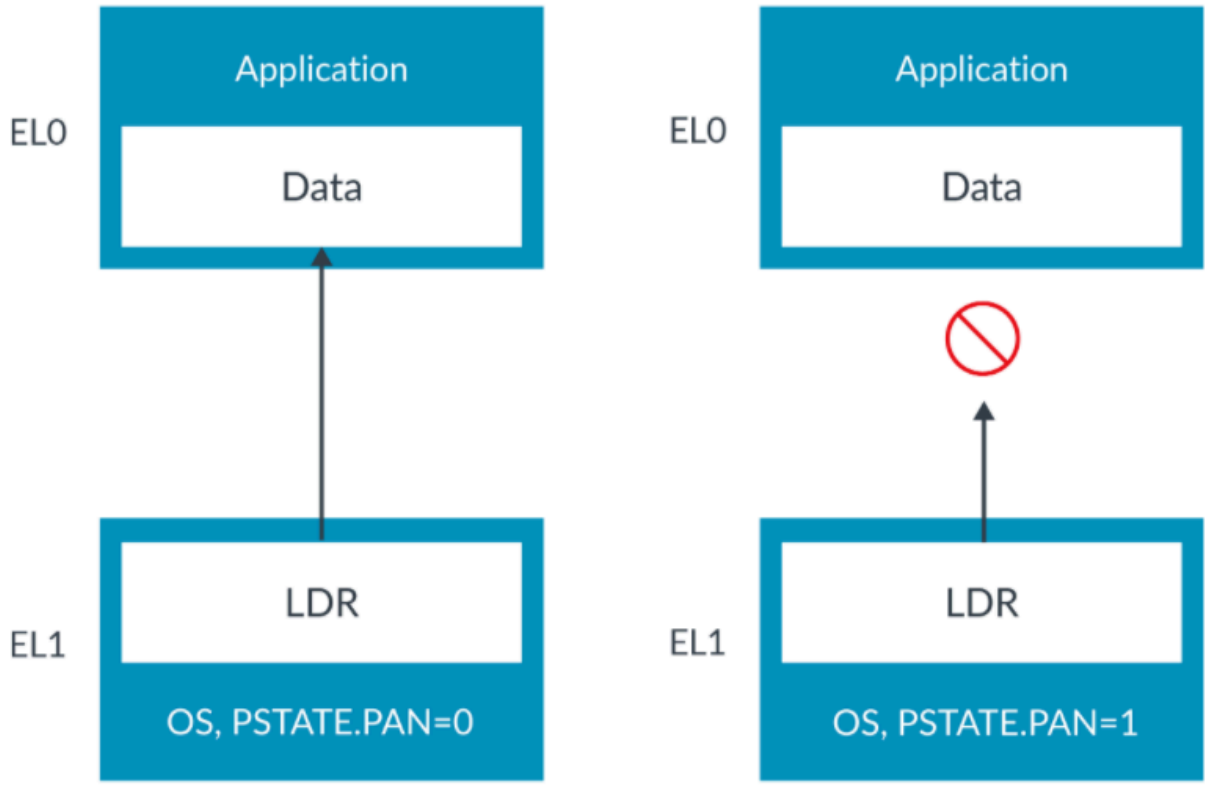
9.1 Privileged accesses to unprivileged data

The standard permission model is that a more privileged entity can access anything belonging to a less privileged entity. For example, an Operating System (OS) can see all the resources that are allocated to an application, or a hypervisor can see all the resources that are allocated to a virtual machine (VM). This is because executing at a higher exception level means that the level of privilege is also higher.

However, this is not always desirable. Malicious applications might try to trick an OS into accessing data on behalf of the application, which the application should not be able to see. This requires the OS to check pointers in systems calls.

The Arm architecture provides several controls to make this simpler. First, there is the PSTATE.PAN (Privileged Access Never) bit. When this bit is set, loads and stores from EL1 (or EL2 when `E2H==1`) to unprivileged regions will generate an exception (Permission Fault),like this diagram illustrates:

Figure 9-1: A diagram showing privileged access never.



Note

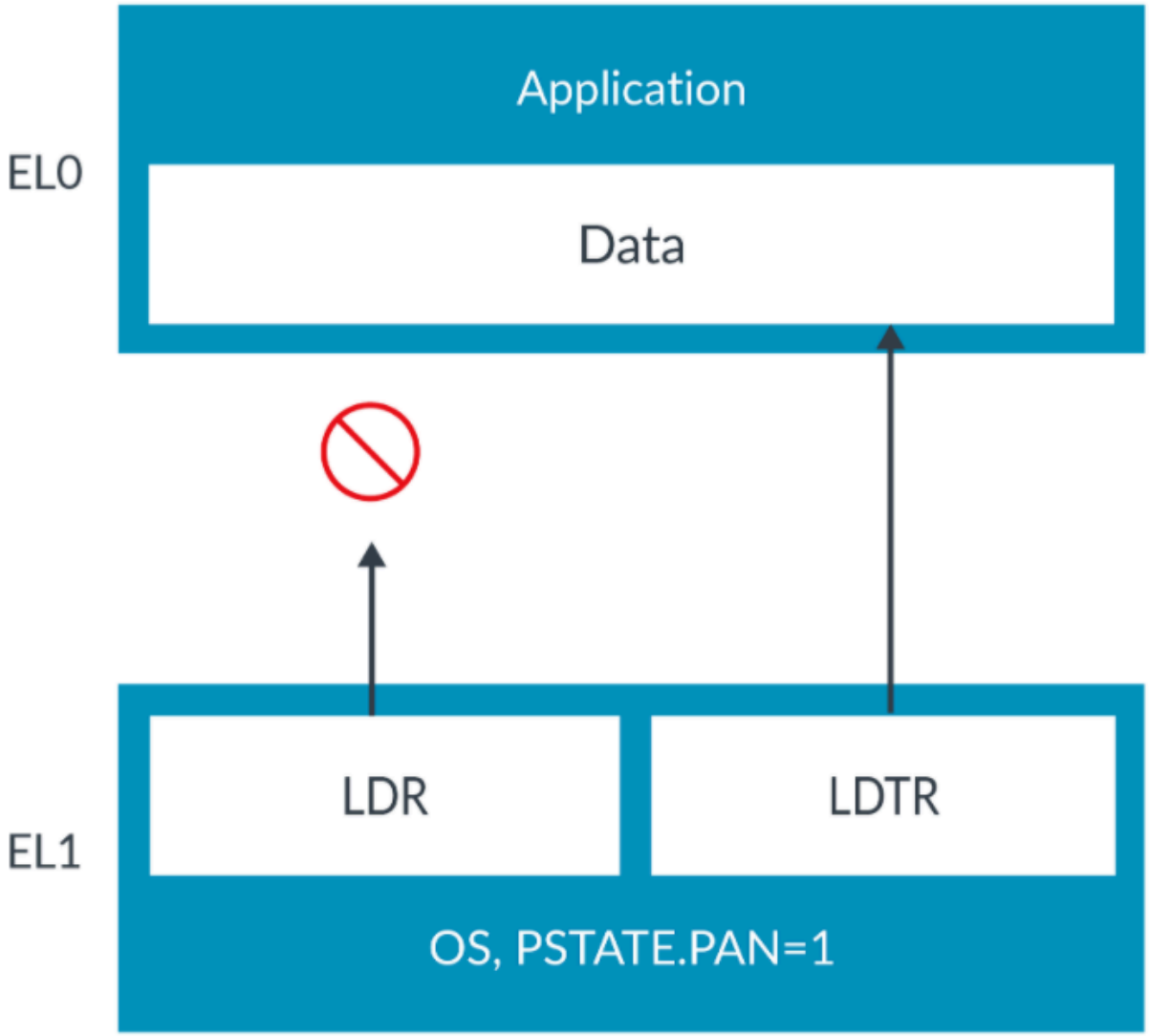
PAN was added in Armv8.1-A.

PAN allows unintended accesses to unprivileged data to be trapped. For example, the OS performs an access thinking that the destination is privileged. In fact, the destination is unprivileged. This means that there is a mismatch between the OS expectation (that the destination is privileged) and reality (the destination is unprivileged). This could occur due to a programming error, or could be the result of an attack on the system. PAN allows us to trap the access before it occurs, catching the error or preventing the attack.

Sometimes the OS does need to access unprivileged regions, for example, to write to a buffer owned by an application. To support this, the instruction set provides the LDTR and STTR instructions.

LDTR and STTR are unprivileged loads and stores. They are checked against EL0 permission checking even when executed by the OS at EL1 or EL2. Because these are explicitly unprivileged accesses, they are not blocked by PAN, like this diagram shows:

Figure 9-2: A diagram showing unprivileged access.



This allows the OS to distinguish between accesses that are intended to access privileged data and those which are expected to access unprivileged data. This also allows the hardware to use that information to check the accesses.

Note

The T in LDTR stands for translation. This is because the first Arm processors to support virtual to physical translation only did so for User mode applications, not for the OS. For the OS to access application data it needed a special load, a load with translation. Today of course, all software sees virtual addresses, but the name has remained.

9.2 Execution permissions

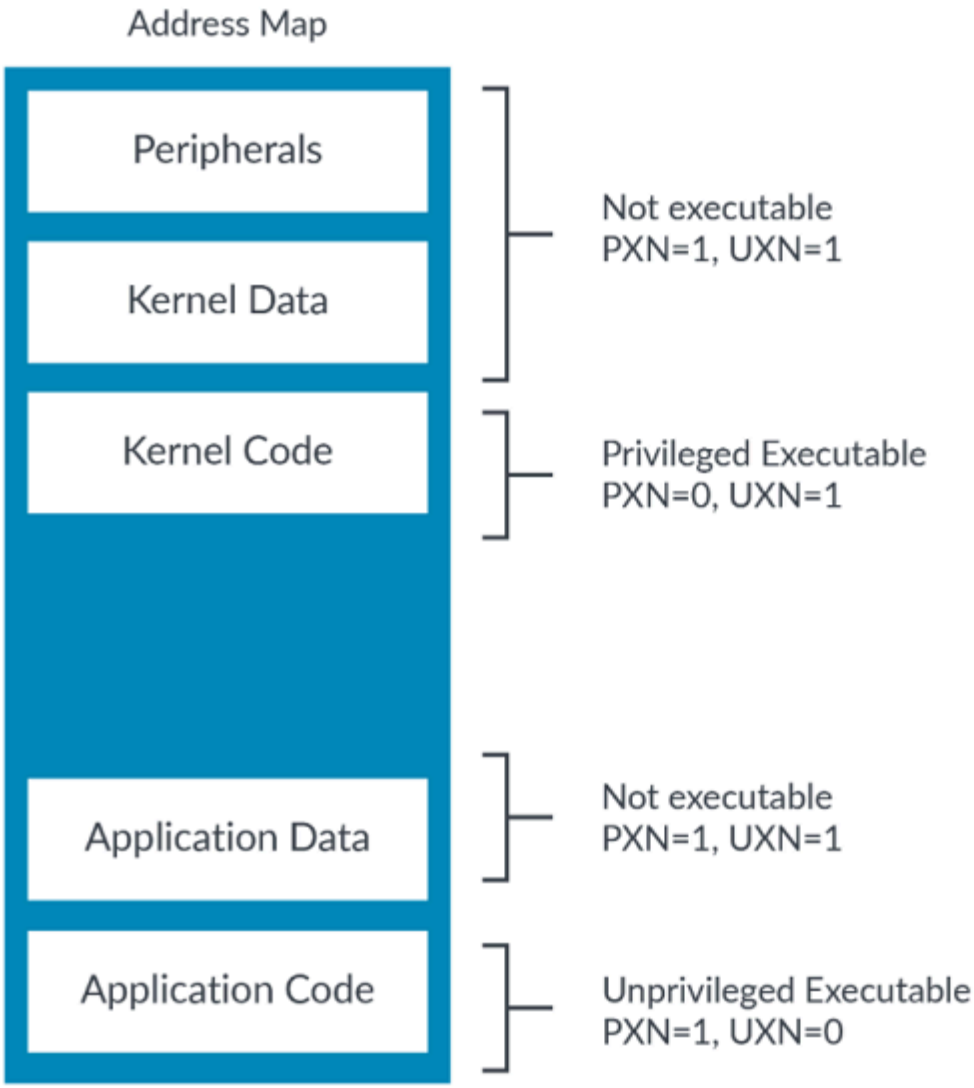
In addition to data access permissions, there are also execution permissions. These attributes let you specify that instructions cannot be executed from the address:

- UXN. User (EL0) Execute Never (Not used at EL3, or EL2 when HCR_EL2.E2H==0)
- PXN. Privileged Execute Never (Called XN at EL3, and EL2 when HCR_EL2.E2H==0)

These are Execute Never bits. This means that setting the bit makes the location not executable.

There are separate Privileged and Unprivileged bits, because application code needs to be executable in user space (EL0) but should never be executed with kernel permissions (EL1/EL2), like this diagram shows:

Figure 9-3: A diagram showing kernel permissions.



The architecture also provides controls bits in the System Control Register (SCTLR_ELx) to make all write-able addresses non-executable.

A location with EL0 write permissions is never executable at EL1.


Note

Remember, Arm recommends that Device regions are always marked as Execute Never (XN).

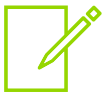
10. Permission indirection and permission overlay extensions

Access permissions can be encoded using either the direct permission scheme or the indirect permission scheme. This chapter discusses the indirect permission scheme. See the Permissions section of this guide for details of the direct permission scheme.

Permission indirection introduces a way to set permissions which allows more efficient use of the permission bits in Translation Table Descriptors (TTDs) while providing the ability to introduce new permission types.

Permission overlays allow permissions to be progressively restricted by processes running at EL0 while reducing the number of calls to processes running at more privileged exception levels (such as the OS), and without requiring costly Translation Lookaside Buffer (TLB) maintenance.

The use of permission indirection and permission overlays allows for a much more flexible approach to setting permissions on pages of memory and can reduce the performance impact of modifying access permissions.



Permission indirection and permission overlay extensions were introduced in Armv8.9-A/Armv9.4-A.

10.1 Permission indirection

When permission indirection is enabled, a Permission Indirection Index (PIIndex) field in the TTD is available. The PIIndex field indexes into the Permission Indirection Register (PIR) for the appropriate exception level, so that the OS (EL1), Hypervisor (EL2), or firmware (EL3) can set the stage 1 translation base permissions:

Register	Description
PIRE0_EL1	PIR to set unprivileged EL1&0 base permissions
PIRE0_EL2	PIR to set unprivileged EL2&0 base permissions
PIR_EL1	PIR to set privileged EL1 base permissions
PIR_EL2	PIR to set privileged EL2 base permissions
PIR_EL3	PIR to set privileged EL3 base permissions

The PIRE0_ELx, or PIR_ELx register fields indicate the base permissions to be applied to that page of memory. See the descriptions of the Permission Indirection Registers for details of the permissions available.


Note

The PIR field indexed in the PII also specifies whether permission overlay is applied to that page of memory. If the PIR entry indicates that permission overlay is not applied, the POI field in the translation table entry is ignored.

10.2 Permission overlays

The base permissions for the page of memory can be further restricted by use of the Permission Overlay Index (POIndex) field in the translation table entries. The POIndex field indexes into the Permission Overlay Register (POR) for the execution level:

Register	Description
POR_EL0	POR for unprivileged EL0 stage 1 overlay permissions
POR_EL1	POR for privileged EL1 stage 1 overlay permissions
POR_EL2	POR for privileged EL2 stage 1 overlay permissions
POR_EL3	POR for privileged EL3 stage 1 overlay permissions

The POR_ELx register fields specify the permissions overlay that is applied to the memory pages in addition to the base permissions. See the descriptions of the Permissions Overlay Registers for details of the additional permission restrictions that can be configured.


Note

The base permissions setting is allowed to be cached in the TLB, but the overlay setting is not. This means that the overlay setting can be changed without the need for a costly TLB invalidate (TLBI)

10.3 An example of using permission indirection and permission overlay features

In this example, the OS running at EL1 wants to set the base and overlay permissions of a range of memory locations that will be used by an application running at EL0.

To set the base permissions, the OS does the following:

1. Allocates a permissions index in the Permission Indirection Register (PIR_ELx). In this example the OS selects index 3.
2. Sets the permission that index 3 in the PIR_ELx represents to “Read and Write, Overlay applied”, by setting the index’s bits to `0b0101`.
3. Sets the PIIndex field in the TTDs of the memory locations in that range to `0b0011` (3).

This sets the base permissions for the memory pages in that range to “Read and Write, Overlay applied”. See the descriptions of the Permission Indirection Registers for details of the permissions available.

The application running at EL0 can request that the OS specify a permissions overlay that the application can later request be applied to a subset of the memory pages. To specify the overlay the OS does the following:

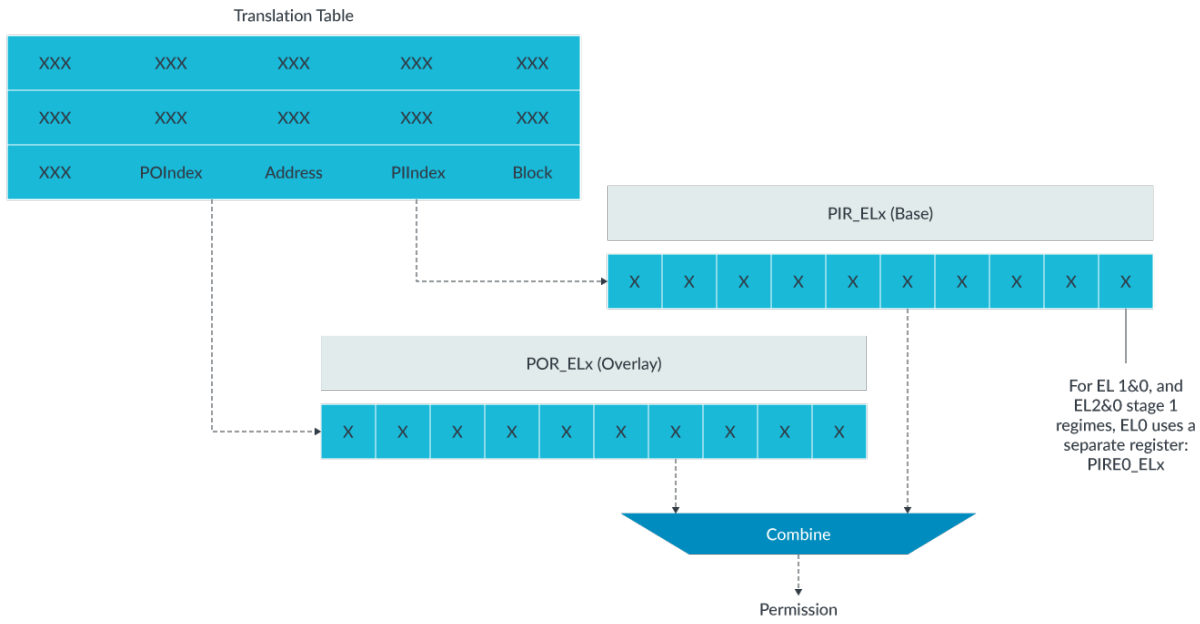
1. Allocates a permissions overlay index in the Permission Overlay Register (POR_ELx). In this example the OS selects index 6.
2. Sets the permission that index 6 in the POR_ELx represents to “Read only”, by setting index 6’s bits to `0b0001`.

This sets the available overlay permission for that memory location to “Read only”. The application running at EL0 can then request (using a syscall) that the OS apply the overlay to a subset of the memory range. The OS does this by setting the POIndex field in the TTDs of those memory pages to `0b110` (6) to have the further “Read only” permission restrictions applied to the base permissions. This effectively allows the application running at EL0 to subtract the write permission from those memory pages. See the descriptions of the Permissions Overlay Registers for details of the additional permission restrictions that can be configured.

This example shows how a greater range of base permissions can be applied to memory pages using permission indirection. It also shows how processes running at EL0 can restrict permissions without the maintenance overhead of a TLBI and while minimizing calls to the OS running at EL1. The permission indirection and overlay extensions allow much more flexibility in applying permissions to memory pages.

The following diagram shows how permission indirection and permission overlays combine to set permissions for a page of memory:

Figure 10-1: A diagram showing how permission indirection and overlays combine.



11. Access Flag

You can use the Access Flag (AF) bit to track whether a region covered by the translation table entry has been accessed. You can set the AF bit to:

- AF=0. Region not accessed.
- AF=1. Region accessed.

The AF bit is useful for operating systems, because you can use it to identify which pages are not currently being used and could be paged-out (removed from RAM).

The Access Flag is not typically used in a bare-metal environment, and you can generate your tables with the AF bit pre-set.

11.1 Updating the AF bit

When the AF bit is being used, the translation tables are created with the AF bit initially clear. When a page is accessed, its AF bit is set. Software can parse the tables to check whether the AF bits are set or clear. A page with AF==0 cannot have been accessed and is potentially a better candidate for being paged-out.

There are two ways that the AF bit can be set on access:

- Software Update: Accessing the page causes a synchronous exception (Access Flag fault). In the exception handler, software is responsible for setting the AF bit in the relevant translation table entry and returns.
- Hardware Update: Accessing the page causes hardware to automatically set the AF bit without needing to generate an exception. This behavior needs to be enabled using the hardware access update bit of the Translation Control Register (TCR_ELx.HA) that was added in Armv8.1-A.

11.2 Dirty state

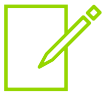
Armv8.1-A introduced the ability for the processor to manage the dirty state of a block or page. Dirty state records whether the block or page has been written to. This is useful, because if the block or page is paged-out, dirty state tells the managing software whether the contents of RAM need to be written out to the storage.

For example, let's consider a text file. The file is initially loaded from disk (Flash or hard drive) into RAM. When it is later removed from memory, the OS needs to know whether the content in RAM

is more recent than what is on disk. If the content in RAM is more recent, then the copy on disk needs to be updated. If it is not, then the copy in RAM can be dropped.

When managing dirty state is enabled, software initially creates the translation table entry with the access permission set to Read-Only and the DBM (Dirty Bit Modifier) bit set. If that page is written to, the hardware automatically updates the access permissions to Read-Write.

Setting the DBM bit to 1 changes the function of the access permission bits (AP[2] and S2AP[1]), so that instead of recording write permission, they record dirty state. This means that when the DBM bit is set to 1 the access permission bits do not cause permission faults.



The same results can be achieved without using the hardware update option. The page would be marked as Read-Only, resulting in an exception (permission fault) on the first write. The exception handler would manually mark the page as read-write and then return. This approach might still be used if software wants to do copy-on-write.

12. Alignment and endianness

This section explains alignment and endianness.

12.1 Alignment

An access is described as aligned if the address is a multiple of the element size.

For LDR and STR instructions, the element size is the size of the access. For example, a LDRH instruction loads a 16-bit value and must be from an address which is a multiple of 2 bytes to be considered aligned.

The LDP and STP instructions load and store a pair of elements, respectively. To be aligned, the address must be a multiple of the size of the elements, not the combined size of both elements. For example:

```
LDP X0, X1, [X2]
```

This example loads two 64-bit values, so 128 bits in total. The address in X2 needs to be a multiple of 64 bits to be considered aligned.

The same principle applies to vector loads and stores.

When the address is not a multiple of the element size, the access is unaligned. Unaligned accesses are allowed to addresses marked as Normal, but not to Device regions. An unaligned access to a Device region will trigger an exception (alignment fault).

Unaligned accesses to regions marked as Normal can be trapped by setting SCTLR_ELx.A. If this bit is set, unaligned accesses to Normal regions also generate alignment faults.

12.2 Endianness

In Armv8-A, instruction fetches are always treated as little-endian.

For data accesses, it is **IMPLEMENTATION DEFINED** whether both little-endian and big-endian are supported. And if only one is supported, it is **IMPLEMENTATION DEFINED** which one is supported.

For processors that support both big-endian and little-endian, endianness is configured per Exception level.


Note

If you cannot remember the definition of **IMPLEMENTATION DEFINED**, read about it in Introducing the Arm Architecture.

Arm Cortex-A processors support both big-endian and little-endian.

13. Memory aliasing and mismatched memory types

When a given location in the physical address space has multiple virtual addresses, this is called aliasing.

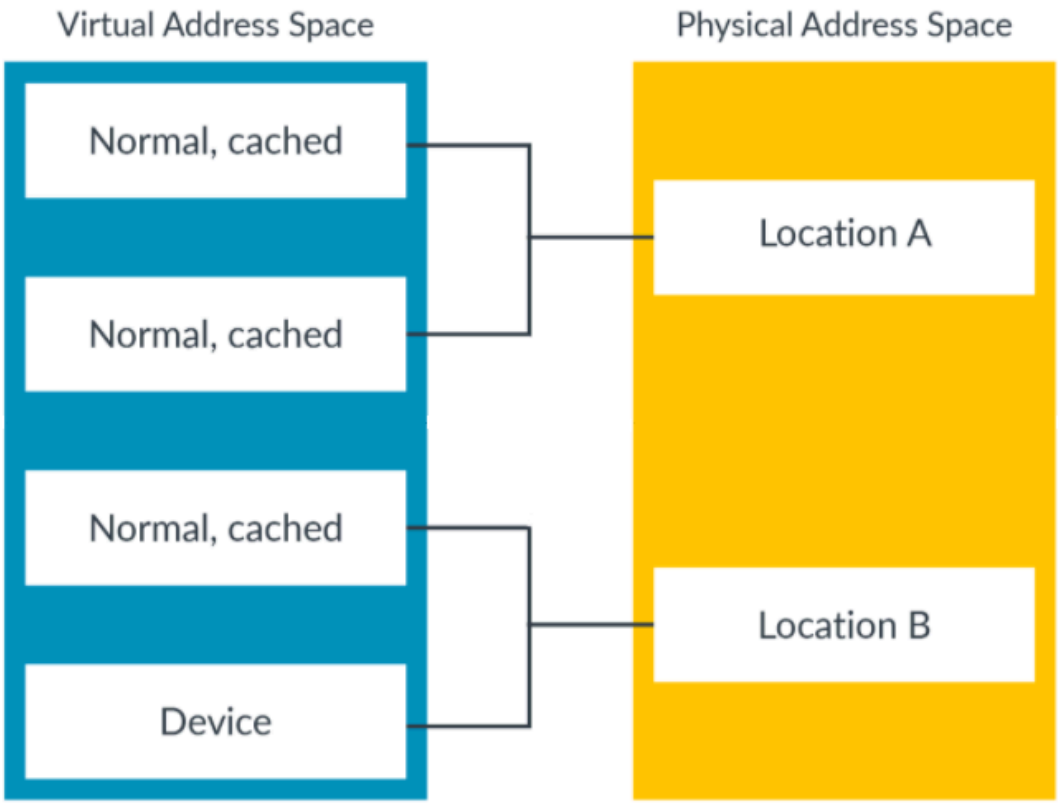
Attributes are based on virtual addresses. This is because attributes come from the translation tables. When a physical location has multiple aliases, it is important that all of the virtual aliases have compatible attributes. We describe compatible as:

- Same memory type, and for Device the same sub-type
- For Normal locations, the same cacheability and shareability

If the attributes are not compatible, the memory accesses might not behave as expected.

This diagram shows two examples of aliasing. The two aliases of location A have compatible attributes. This is the recommended approach. The two aliases of location B have incompatible attributes (Normal and Device), which can negatively affect coherency and performance:

Figure 13-1: A diagram showing virtual address space.



Arm strongly recommends that software does not assign incompatible attributes to different aliases of the same location.

14. Check your knowledge

The following questions help you test your knowledge.

Where do the attributes for an address location come from?

The translation tables, typically the Block/Page descriptor, although hierarchical attributes can override this.

What are the two memory Types in Armv8-A?

Normal and Device.

What does -nGnRE mean in Device-nGnRE?

Does not allow gathering and re-ordering, but does allow early write acknowledgement.

Why might a page be marked as PXN=1, UXN=0?

Application code needs to be executable in User space and not executable in Kernel space.

What is the AF bit typically used for?

Tracking which pages have been accessed.

What endianness are instruction fetches?

Little-endian

Before enabling the memory management unit (MMU), some start-up code makes an unaligned access, which leads to an alignment fault. Why?

When the MMU is disabled, all accesses are treated as Device. Unaligned accesses to regions marked as Device always fault.

15. Related information

Here are some resources related to material in this guide:

- Arm Community

  Ask development questions, and find articles and blogs on specific topics from Arm experts.
- Memory ordering and barriers guide

  Provides information about memory ordering, and the use of barriers.
- Security - pointer signing and landing pads guide (coming soon)

  Armv8.5-A introduced support for Branch Target Instructions (BTI). BTI support is controlled by the GP bit in the stage 1 translation tables. Branch Target Instructions will be discussed in this guide.
- Armv8-A Instruction Set Architecture

  Includes information about Simple Sequential Execution (SSE).
- Translation process: If are you are interested in the full details of translation process, it is described fully in pseudo code. The translation pseudo code is included with the XML for the instruction set. A good place to start is the `AArch64.FullTranslate()` function.

Here are some resources related to topics in this guide:

15.1 Describing memory in Armv8-A

Cacheability of instruction fetches are a bit more complicated than you might think. This topic is covered in Caches and Coherency guide (coming soon).

For EL0 and EL1, this behavior can be partly overridden using the virtualization controls. This topic is covered in the Virtualization guide.

15.2 Cacheability and shareability attributes

Caches and cache coherency guide (coming soon).

15.3 Combining stage 1 and stage 2 attributes

Stage 1 and stage 2 translation are discussed in more detail in the Memory Management guide.

For background information, see the Virtualization guide.

15.4 Training modules:

Here are links to some related training modules that may be useful:

- Introduction to Armv8-A
- Memory model overview
- What does architecture consist of?

16. Next steps

The Armv8-A memory attributes and properties provide the basis for how a processor core interacts with memories in a system. You can apply the principles you have learned throughout this guide when you begin to develop low level code, like boot code or drivers. You can also put this learning into practice when you write code to set up or manage the Memory Management Unit (MMU).

The next guide in this series discusses address translation in Memory management.

To keep learning about the Armv8-A architecture, see more in our series of guides.