Armv8-A memory model

Version 1.0

Armv8-A memory model guide

Release Information

Document History

Version	Date	Confidentiality	Change
1.0	26 April 2019	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

Product Status

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

Web Address

http://www.arm.com

Contents

1 Overview

This guide introduces the memory model in Armv8-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, such as boot code or drivers. It is particularly relevant to anyone writing code to setup or manage the *Memory Management Unit* (MMU).

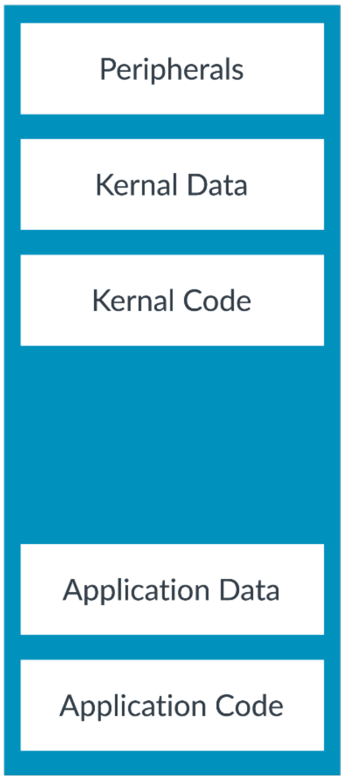
At the end of this guide, you can check your knowledge. You will have learned about the different memory types and their key differences. You will be able to describe the memory ordering rules for Normal and Device memory types. And you will also be able to list the memory attributes that can be applied to a given address.

2 What is a memory model, and why is it needed?

A memory model is a way of organizing and defining how memory behaves. It provides a structure and a set of rules for you to follow as you configure how addresses, or regions of addresses, are accessed and used in your system.

The memory model provides attributes that you can apply to an address and it defines the rules associated with memory access ordering.

Consider a simple system with the address space:

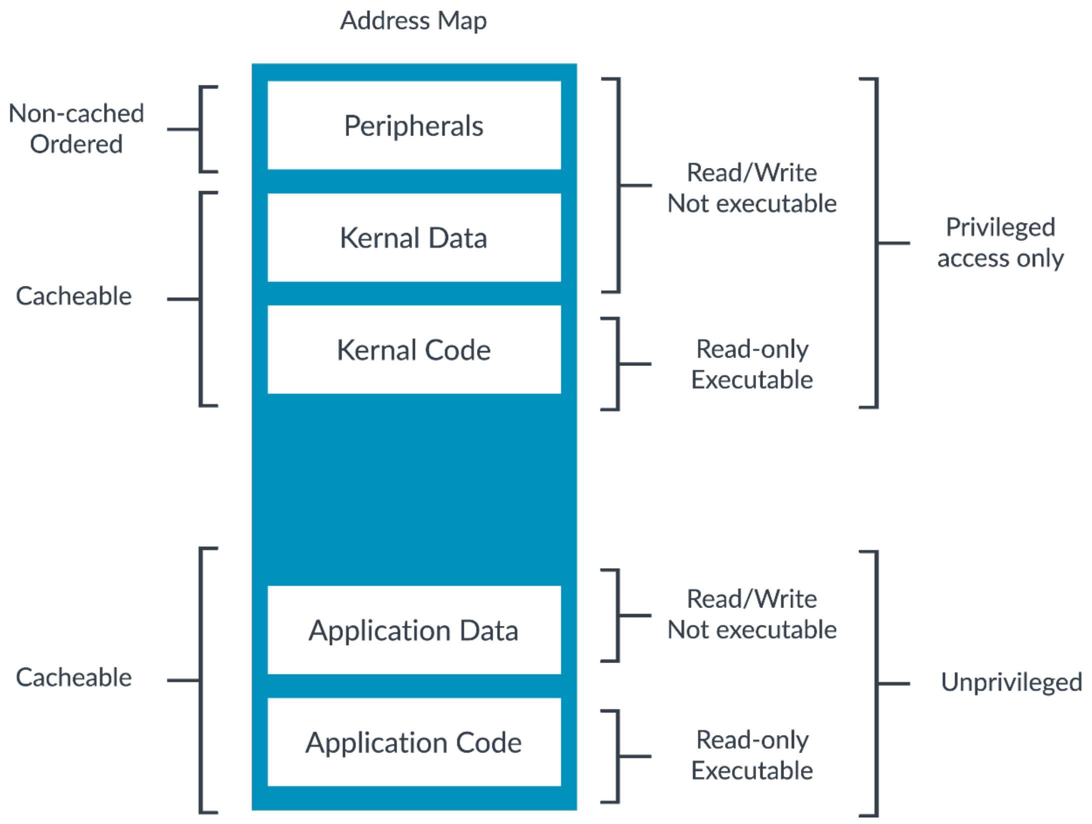


The arrangement of memory regions in the address space is known as an address map. Here, the map contains:

- Memory and peripherals.
- In the memories, code and data.
- Resources belonging to the OS and resources belonging to user applications.

The way that you want the processor to interact with a peripheral is different to how it should interact with a memory. You will usually want to cache memories but not peripherals. Caching is where you store a copy of information from memory into a location (known as a cache) that's closer to the core and therefore faster for the core to access. Similarly, you will usually want the processor to block user access to kernel resources.

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



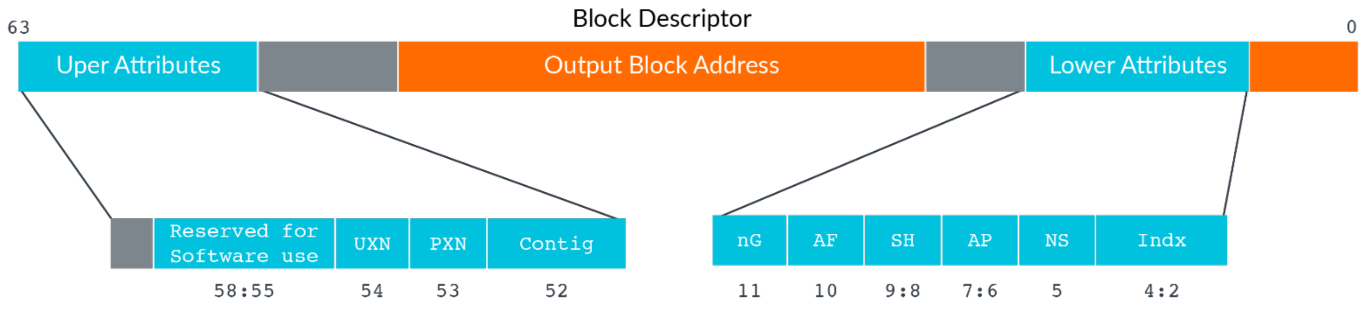
You need to be able to describe these different attributes to the processor, so that the processor accesses each location appropriately.

3 Describing memory in Armv8-A

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 provides the corresponding physical address and the attributes for accessing that page.

Each translation table entry is known as 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:



Important attributes are:

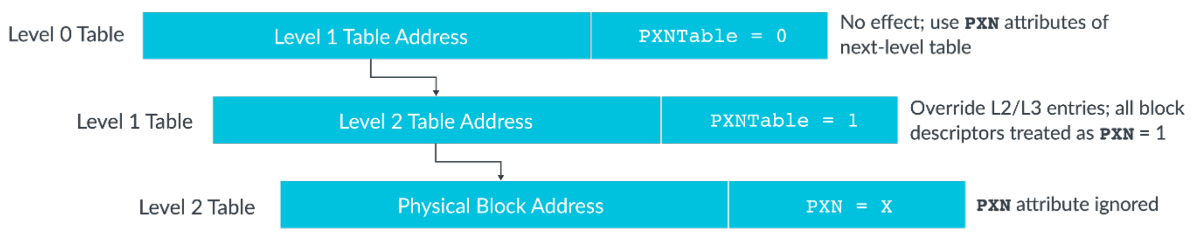
- SH - The shareable attribute.
- AP - The access permission.
- UXN and PXN – Execution permissions.

Make a note of these attributes as we will revisit them further on 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:



From Armv8.1-A, support for setting the Access Permission and Execution Permissions using the hierarchical attributes in a table descriptor can be disabled. This is controlled via the relevant TCR_ELx register. When disabled, the bits previously used for the hierarchical controls are available to software to use for other things.

3.2. MMU disabled

To recap, 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).

So 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 (we will explain this later on this guide).
- All instruction fetches are treated as Cacheable.
- 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 access ordering

In the instruction set architecture guide, we introduce *Simple Sequential Execution* (SSE) which is a conceptual model for instruction ordering. Memory access ordering and instructions ordering are two different, but related, concepts. It is important that you understand the difference.

SSE describes the order instructions appear to executed by the processor. To recap, modern processors have long and complex pipelines. Often able to re-order instructions or execute multiple instructions in parallel to maximize performance. SSE means that the processor must behave as if the processor is executing the instructions one at a time, in the order that they are given in the program code. This means that any re-ordering or multi-issuing of instructions by hardware must be invisible to software.

Memory ordering is about the order memory accesses appear in the memory system. Because of mechanisms such as write-buffers and caches, even when instructions are executed in order the related memory accesses may not be executed in order. This is why memory ordering is an important thing to consider even though the processor follows the SSE model for instructions.

5 Memory types

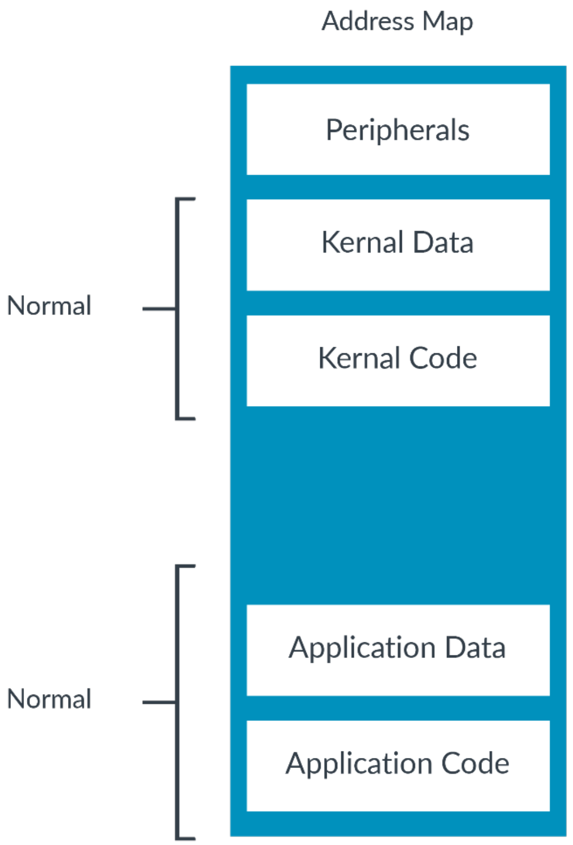
All addresses in a system, that are not marked as faulting, are assigned a memory type. There are two memory types in Armv8-A; Normal and Device. The memory type is a high-level description of how the processor should interact with the address region.

Note: In Armv6 and Armv7 there was a third memory type, Strongly Ordered. In Armv8, this maps to Device_nGnRnE.

6 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 here:



6.1. 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 known as 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. We'll introduce some of these here.

A location marked as Normal has no direct side-effects when it is accessed. What this means is that reading the location just returns us the data, it does not cause the data to change or directly trigger another process. Because of this, for locations marked as Normal a processor may:

- Merge accesses. Code can access a location multiple times, or access multiple consecutive locations. For efficiency, the processor is permitted to detect and merge these accesses into a single access. For example, if software writes to a variable multiple times the processor might only present the last write to the memory system.
- Perform accesses speculatively. The processor is permitted to read a location marked as Normal without it being specifically requested by software. For example, the processor might use pattern recognition to prefetch data before software has requested it, based on the patterns of previous accesses. This technique is used to expedite accesses by predicting behavior.

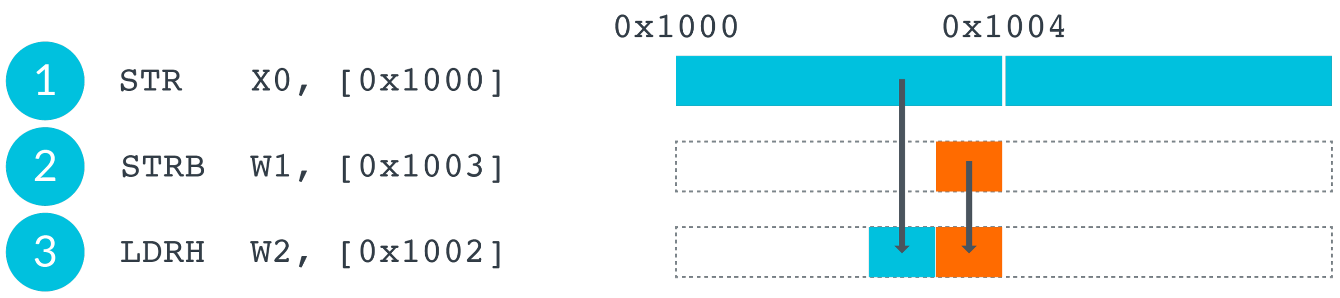
- Re-order accesses. The order that accesses are seen in the memory system might not match the order that the accesses were issued in by software. For example, a processor might re-order two reads to allow it to generate a more efficient bus access. Accesses to the same location cannot be re-ordered but might be merged.

These optimizations can be considered as freedoms that allow the processor to employ techniques to speed up performance and improve power efficiency. This means that the Normal memory type usually gives the best performance.

Note: The processor is permitted to optimize in these ways, but that does not mean it always will. How much use a given processor will make of these freedoms comes down to its micro-architecture. From a software perspective you should assume that the processor might do any or all of them.

6.2. Limits on re-ordering

To recap, accesses to locations marked as Normal can be re-ordered. Let's consider this example code with a sequence of three memory accesses, two stores and then a load:



If the processor were to re-order these accesses, then we might end up with the wrong value in memory, and this is not allowed.

For accesses to the same byte(s), ordering must be maintained. The processor needs to detect the hazard and ensure that the accesses are ordered correctly for the intended outcome.

This does not mean that there is no possibility of optimization with this example. The processor could merge the two stores together, presenting a single combined store to the memory system. It could also detect that the load operation is from the bytes written by the store instructions so it could return the new value without re-reading it from memory.

Note: The sequence given in the example is deliberately contrived to make the point. In practice, these kinds of hazard tend to be subtler.

There are other cases where ordering is enforced, for example Address Dependencies. An Address Dependency is when a load or store uses the result of a previous load as an address. In this example, the second instruction is dependent on the outcome of the first instruction:

```
LDR X0,[X1]
STR X2,[X0] // The result of the previous load is the address in this store.
```

This example also shows an address dependency where the second instruction is dependent on the outcome of the first instruction:

```
LDR X0,[X1]
STR X2,[X5, X0] // The result of the previous load is used to calculate the address.
```

Where there is an Address Dependency between two memory accesses, the processor must maintain the order.

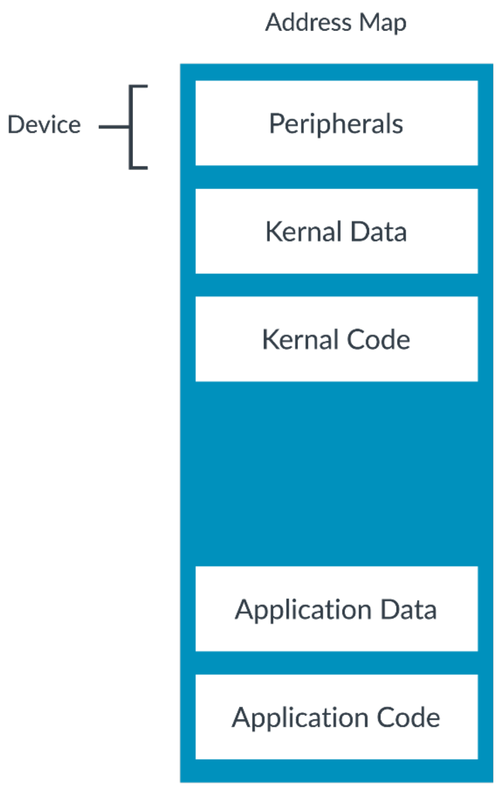
This rule does not apply to control dependencies. A control dependency is where the value from a previous load is used to make a decision. This example shows a load followed by a Compare and Branch on Zero operation that relies on the value from the load:

```
LDRX0, [X1]
CBZ X0, <somewhere_else>
STRX2, [X5][Symbol] // There is a control dependency on X0, this does not guarantee ordering.
```

There are cases where ordering needs to be enforced between accesses to Normal memory, or accesses to Normal and Device memory. This can be achieved using Barrier instructions.

7 Device memory

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



To recap, the Normal memory type means that there are no side-effects to the access. For 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, as it is very unlikely that you would ever 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, and Arm recommends 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.

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

Here are two examples:

1. Device_GRE. This allows gathering, re-ordering, and early write acknowledgement.

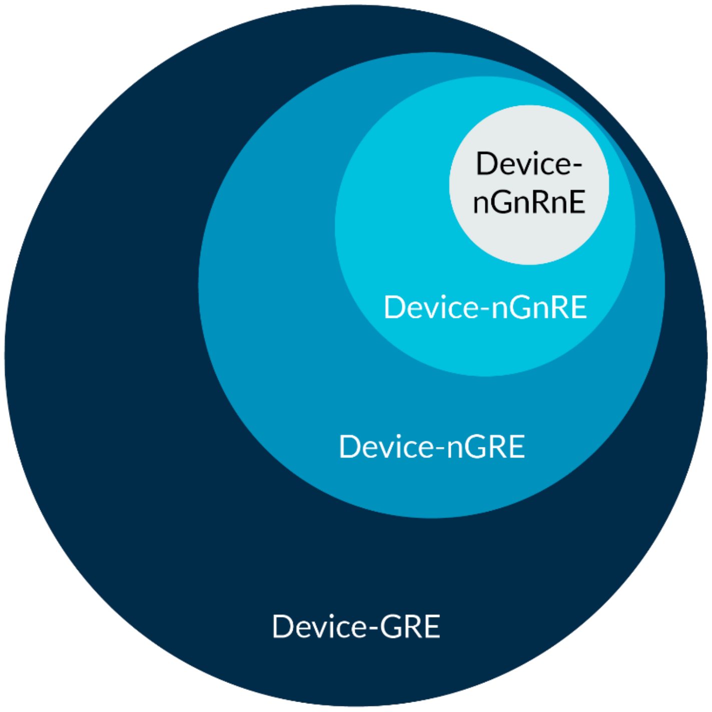
2. Device_nGnRnE. This does not allow gathering, re-ordering, and early write acknowledgement.

Of these optimizatons, we've already seen how re-ordering work but we've not yet 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 any point on the bus between the core and the peripheral at the address.

Note: Normal Non-cacheable and Device_GRE might appear at first glance to be the same, but this is not the case. Normal Non-cacheable still allows speculative data accesses, Device_GRE does not.

7.1. Does the processor really do something different for each type?

The memory type describes the set of allowable behaviors for a location. Taking just the Device type, we can represent the allowable behaviors as shown here:



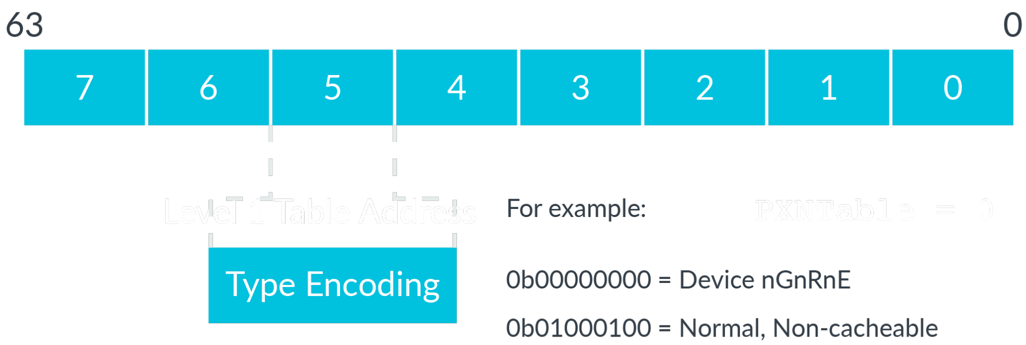
Device_nGnRnE is the most restrictive sub-type, and has the fewest allowed behaviors. Device_GRE is the least restrictive, and therefore has the most allowed behaviors.

Importantly, all the behaviors allowed for Device_nGnRnE are also permitted for Device_GRE. As, 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 specific example is extreme and unlikely for A-class processors. But 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.

8 Describing the memory type

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



For example:

0b00000000 = Device nGnRnE

0b01000100 = Normal, Non-cacheable

0b11111111 = Normal, Inner/Outer Write-Back Cacheable

The selected field determines the memory type and cacheability information.

So why is an index to a register used instead of encoding the memory type directly into the translation table entries? The reason is because the number of bits in the translation table entries is limited. It requires 8 bits to encode the memory type, but only 3 bits to encode the index into `MAIR_ELx`. This way the architecture efficiently uses less bits in the table entries.

9 Cacheability and Shareability attributes

Locations marked as Normal also have Cacheability and Shareability attributes. These attributes control whether a location can be cached, and if so, 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.

10 Permissions attributes

The *Access Permissions* (AP) attribute controls whether a location can be read and written, and what privilege is necessary. This 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

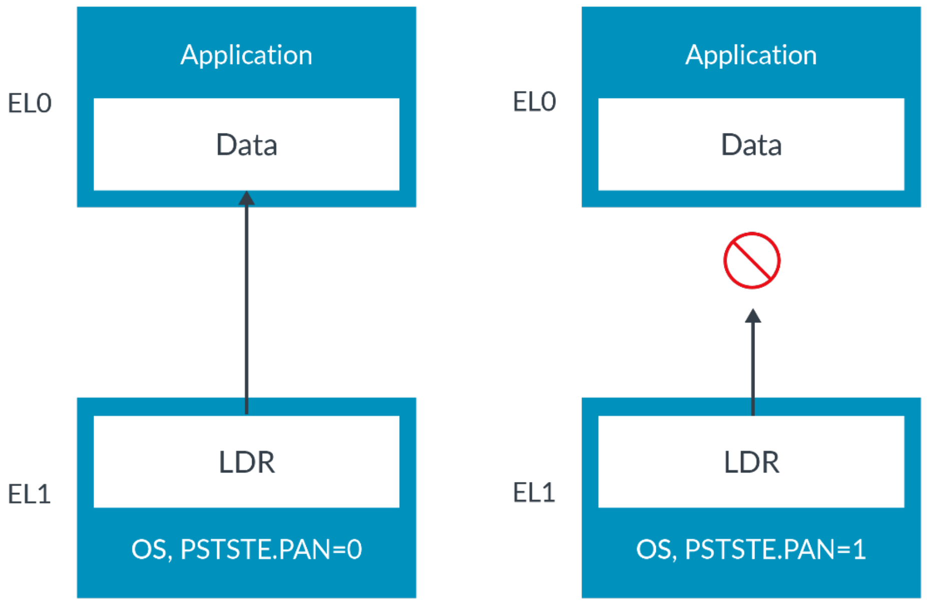
If an access breaks the specified permissions, for example a write to a read-only region, an exception (labelled as a permission fault) is generated.

10.1. Privileged accesses to unprivileged data

The standard permission model is that a more privileged entity can access anything belonging to a less privileged entity. Or to put it another way, an *Operating System* (OS) can see all the resources allocated to an application. A hypervisor can see all the resources allocated to a virtual machine. 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 use to try to trick an OS into accessing data on the application's behalf, 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), as this diagram illustrates:

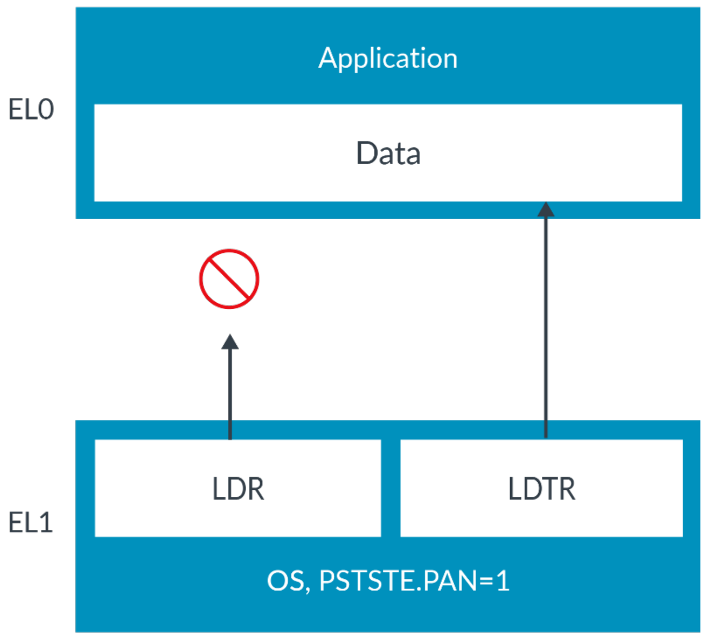


Note: PAN was added in Armv8.1-A.

`PAN` allows unintended accesses to unprivileged data to be trapped. That is, when an instruction accesses a privileged region and the OS thinks the instruction should be accessing an unprivileged region.

There are cases where 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. As these are explicitly unprivileged accesses, they are not blocked by `PAN`, as this diagram shows:



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: Some historical trivia. 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.

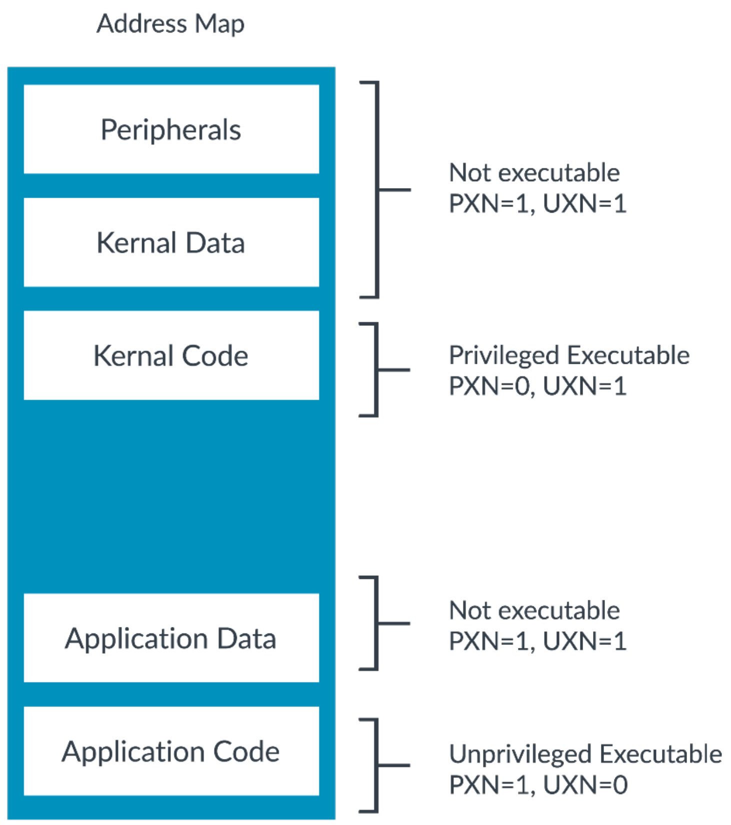
10.2. Execution permissions

As well as access permissions, there are also execution permissions. These attributes let you specify that instructions cannot be fetched 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, as application code needs to be executable in user space (EL0) but should never be executed with kernel permissions (EL1/EL2), as this diagram shows:


Address Map
Peripherals
Kernal Data
Kernal Code
Application Data
Application Code
Not executable PXN=1, UXN=1
Privileged Executable PXN=0, UXN=1
Not executable PXN=1, UXN=1
Unprivileged Executable PXN=1, UXN=0

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`).

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.`

This is useful for operating systems, as you can use it to identify which pages are not currently being used and as a result could be paged-out (removed from RAM).

Note: 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. As a page is accessed its AF bit is set. Software can parse the tables checking 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:

1. 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.

2. 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 and 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, as if it is paged-out it 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 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 access permission they record dirty state. This means that when the DBM bit is set to 1 the access permission bits do not cause access faults.

Note: 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

12.1. Alignment

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

For `LDR` and `STR` instructions this means 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 16 bits 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 loads two 64-bit values, 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 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 and Little endian, it is configured per Exception level.

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

Arm's Cortex-A processors support both Big and Little endian.

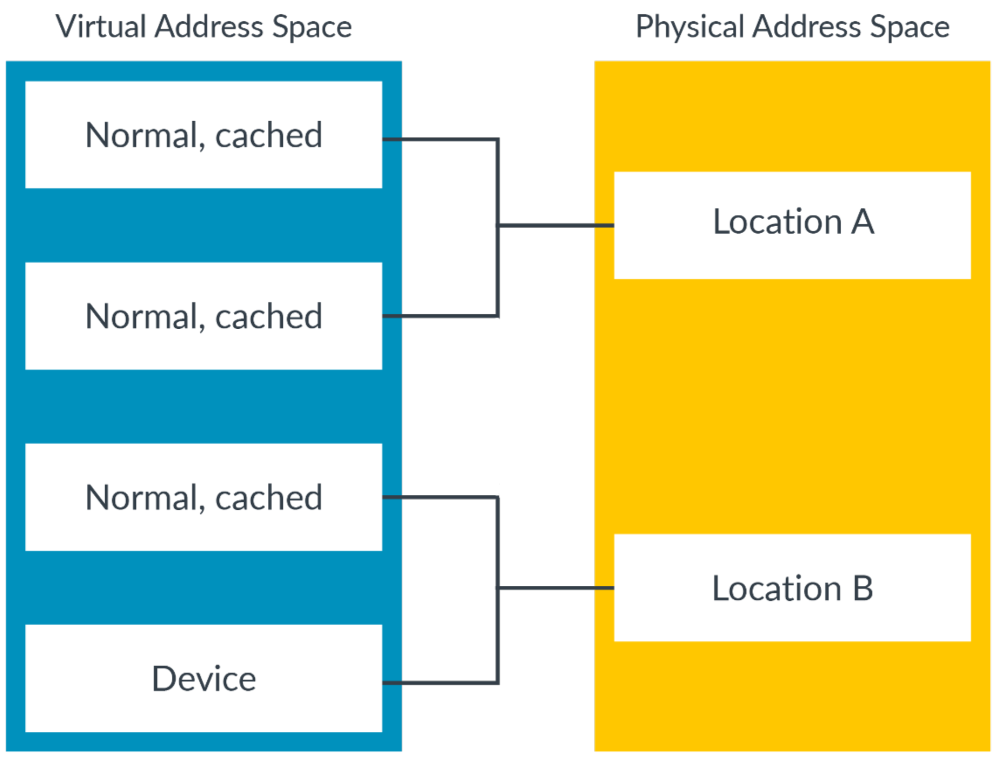
13 Memory aliasing and mismatched memory types

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

Attributes are based on virtual addresses because attributes come from the translation tables. When there are multiple aliases of a physical location, it is important that all the virtual aliases have compatible attributes. By compatible we mean:

- 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 you expect, and this can impact performance.

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), and this can negatively affect coherency and performance:

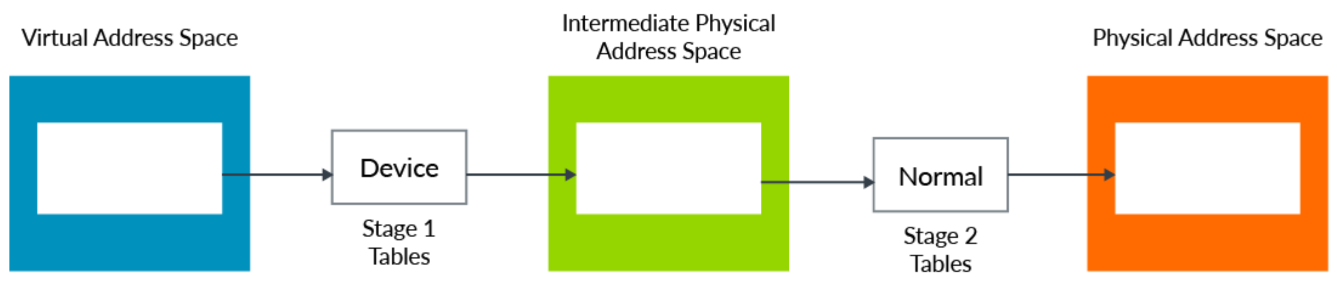


Arm strongly recommends that software does not assign incompatible attributes to different aliases of the same location.

14 Combining Stage 1 and Stage 2 attributes

When virtualization is used, a virtual address goes through two Stages of translation. One stage under control of the OS and a second stage under the control of the hypervisor. Both Stage 1 and Stage 2 tables include attributes. How are these combined?

This next diagram shows an example where Stage 1 has marked a location as Device, but the corresponding Stage 2 translation is marked as Normal. What should the resulting type be?



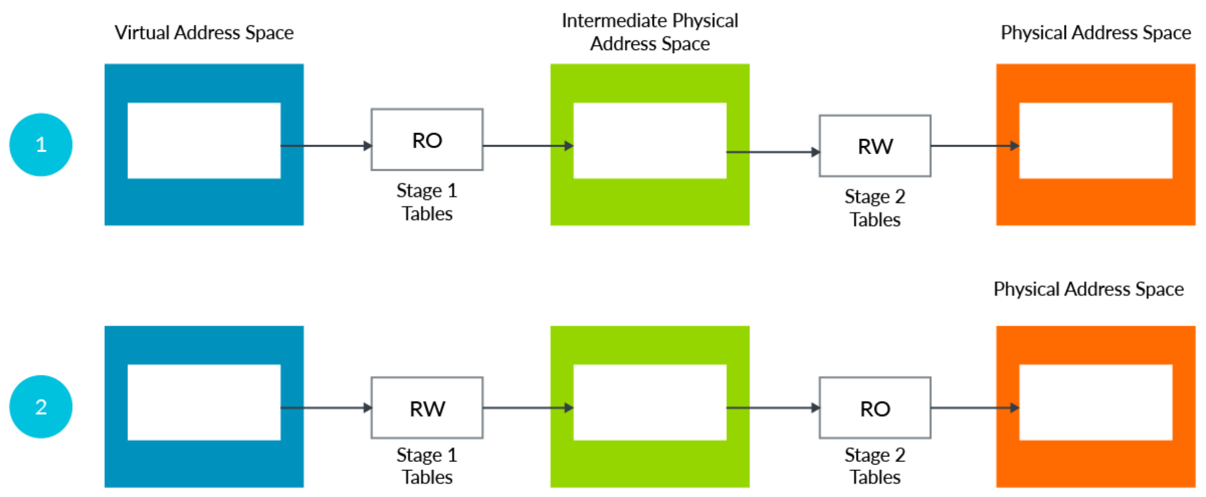
The default in the Arm architecture is to use the most restrictive type. In this example, Device is more restrictive than Normal. Therefore, the resulting type is Device.

For the type and Cacheability, an additional control (`HCR_EL2.FWB`) allows this behavior to be overridden. When `FWB` is set, Stage 2 can override the Stage 1 type and cacheability settings, instead of the combining behavior.

Note: `HCR_EL2.FWB` was introduced in Armv8.4-A.

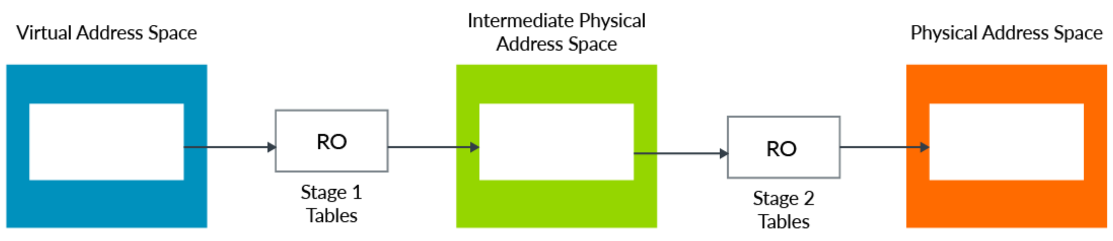
14.1. Fault handling

Consider these two cases:



In both of these examples, the resulting attribute is RO (read-only). If software were to write the location, a fault (permission fault) would be generated. But, in the first case this is a Stage 1 fault and the second case it would be a Stage 2 fault. In the example, the Stage 1 fault would have gone to the OS at EL1. While the Stage 2 fault would have gone to EL2 and be handled by a hypervisor.

Finally, let's consider the case where Stage 1 and Stage 2 attributes are the same:



Here, the resulting attribute is simple, it is R0. But if software writes to the location, is a Stage 1 or Stage 2 fault generated? The answer is Stage 1. This would also be true if Stage 1 and Stage 2 would raise different fault types. Stage 1 faults always take precedence over Stage 2 faults.

15 Check your knowledge

Q. Where do the attributes for an address location come from?

A. The translation tables, typically the Block/Page descriptor. Although hierarchal attributes can override this.

Q. What are the two memory Types in Armv8-A?

A. Normal and Device.

Q. What does _nGnRE mean in Device_nGnRE?

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

Q. Think of an example of when accesses to a region marked as Normal cannot be re-ordered.

A. Either:

- Address dependency.
- Access to the same location.
- Barrier.

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

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

Q. What is the AF bit typically used for?

A. Tracking which pages have been accessed.

Q. What endianness are instruction fetches?

A. Little endian.

Q. Before enabling the MMU some start-up code makes an unaligned access, which leads to an alignment fault. Why?

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

16 Related information

This guide introduces the basic principles of memory access ordering. Memory ordering, and the use of barriers, is a big topic. For an introduction start here:

Memory ordering guide (coming soon).

We skipped over describing the cacheability and shareability attributes. These are discussed here:

Caches for temporalness 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 are discussed in:

Security - ROP and JOP guide (coming soon).

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.

16.1. 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).

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

16.2. Cacheability and Shareability attributes

Caches and coherency guide (coming soon).

16.3. Combining Stage 1 and Stage 2 attributes

Translation, both Stage 1 and 2, are discussed in more detail in Memory Management.

For background information see Virtualization.

17 Next steps

The Armv8-A memory model provides the basis for how a processor core interacts with memories in a system. You can apply the principles of the model that you've learned through this guide as you begin to develop low level code, such as boot code or drivers, and in particular when you write code to setup 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.