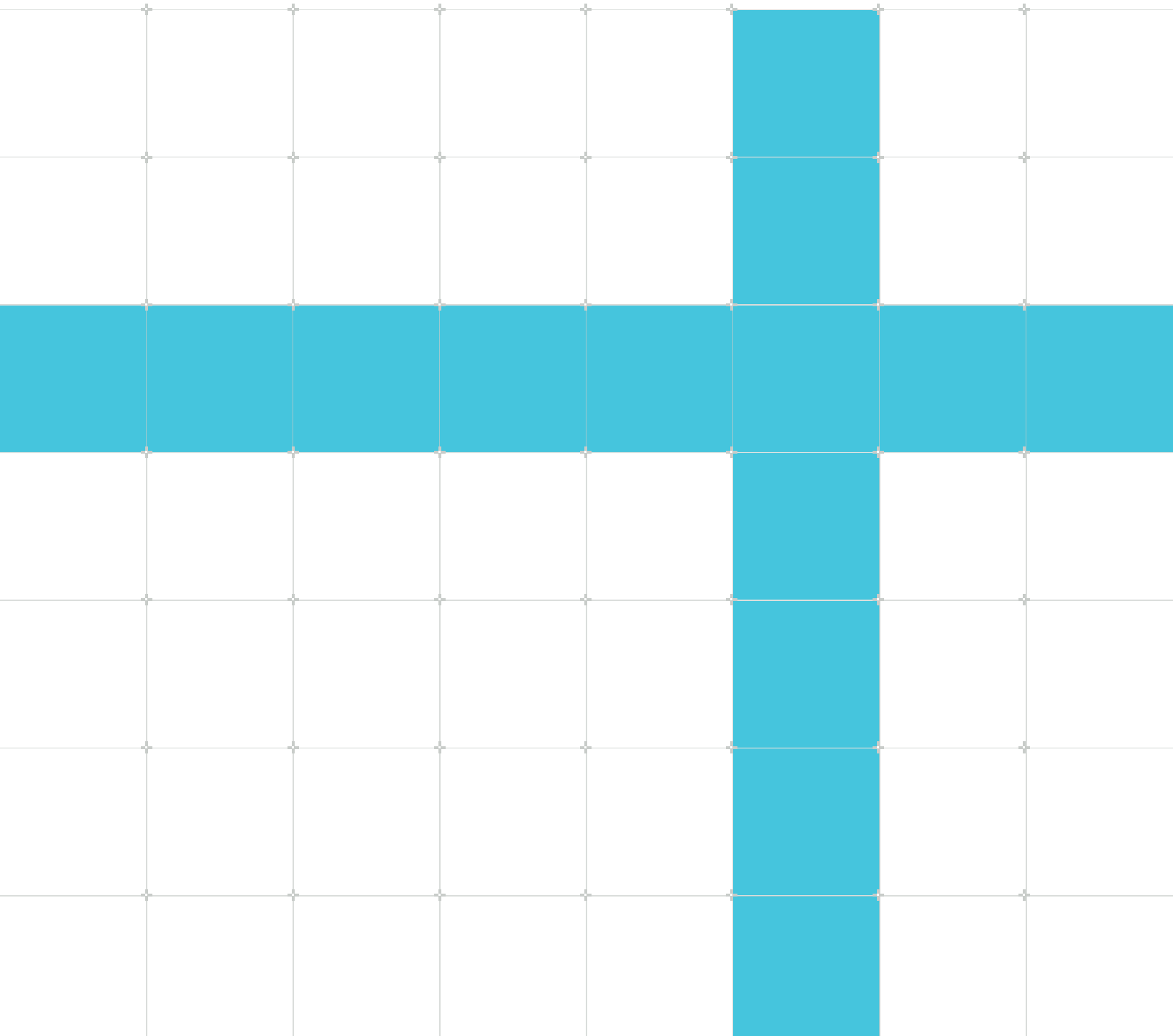
arm

Learn the architecture - AArch64 memory model

Version 1.0

Issue 02
102376_0100_02_en



Learn the architecture - AArch64 memory model

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	1 April 2019	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

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 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, 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. 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.

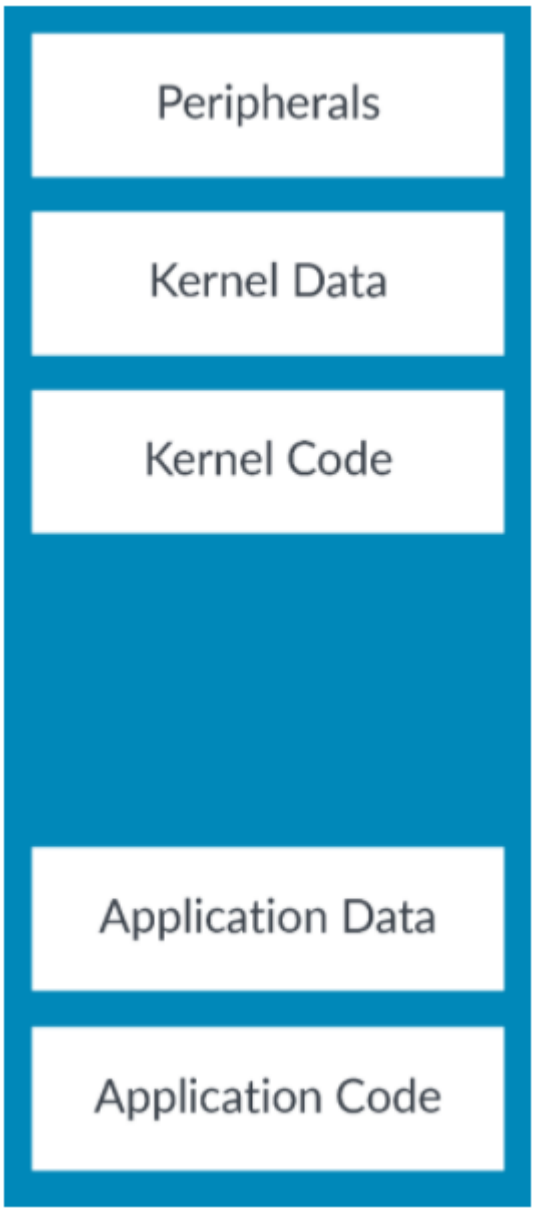
2. What is a memory model, and why is it needed

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 when 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, like you can see in this diagram:

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



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

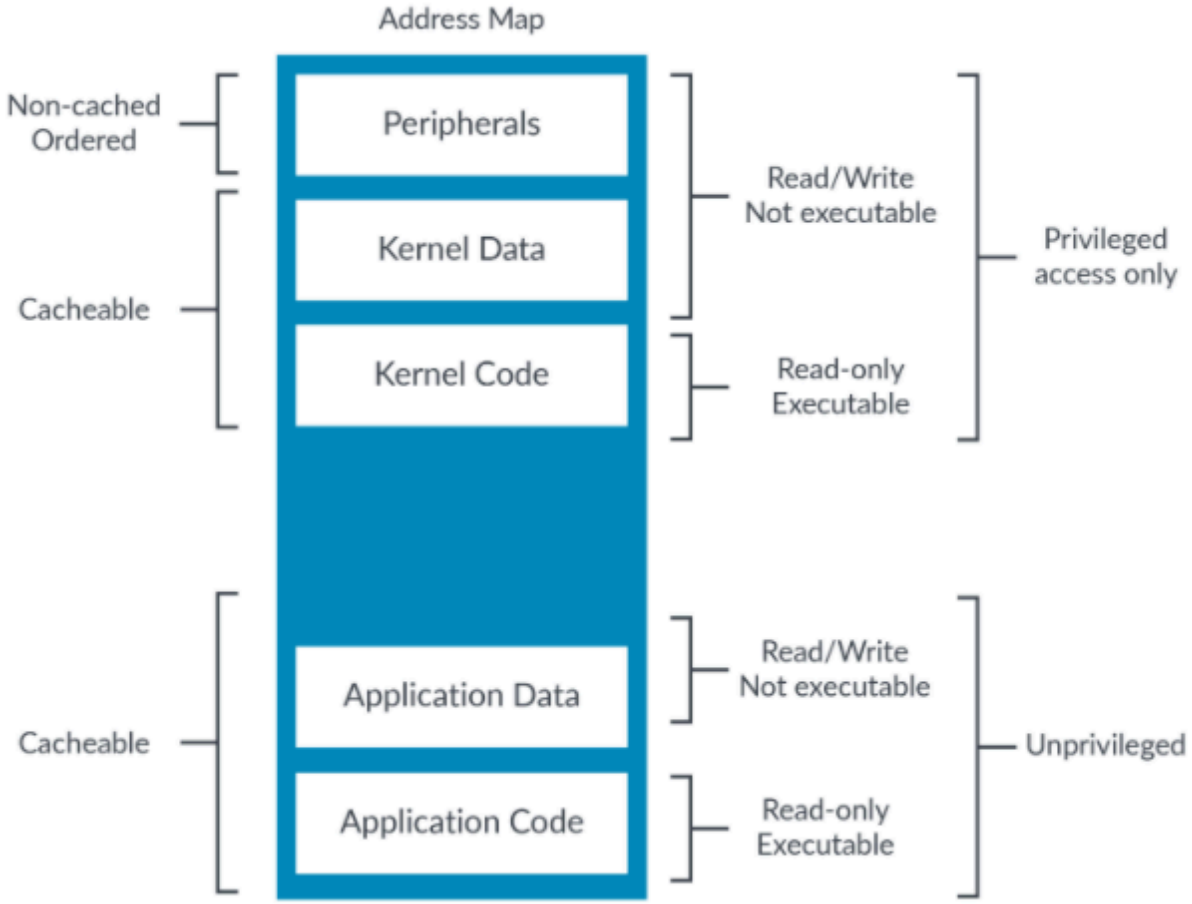
- Memory and peripherals
- In the memories, code and data
- Resources belonging to the OS and resources belonging to user applications

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 you will not want to cache peripherals. Caching is the act of storing a copy of information from memory into a location,

which is called a cache. The cache is 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:

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.

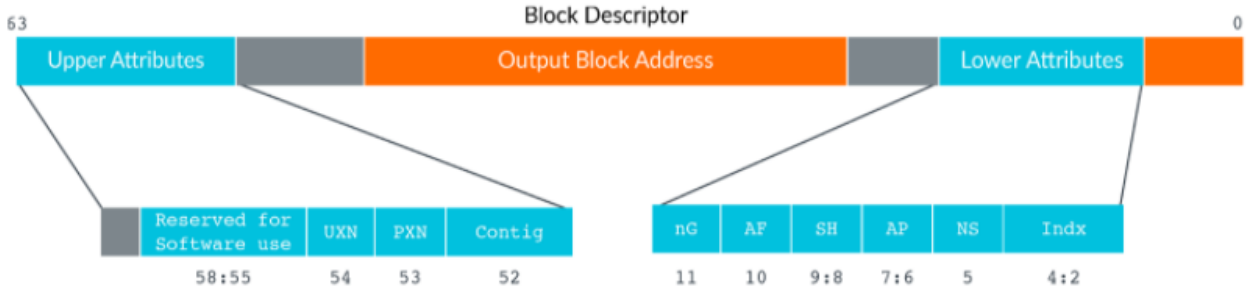
3. Describing memory in AArch64

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 are:

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

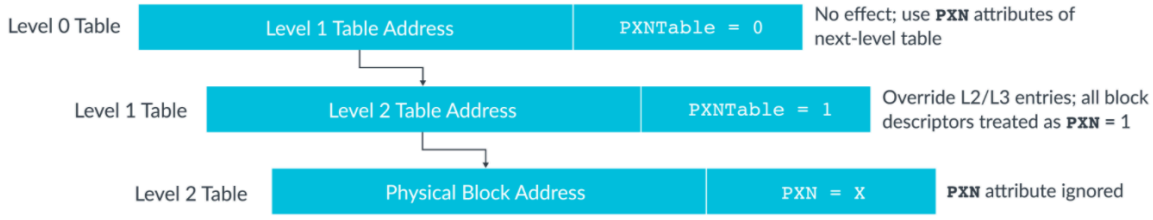
Remember these attributes, because we will revisit them later in this guide.

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 permission and execution permissions using the hierarchical attributes in a table descriptor. 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.

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. We will explain this later in 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 our guide Armv8-A Instruction Set Architecture, we introduce Simple Sequential Execution (SSE). SSE 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 between them.

SSE describes the order in which the processor appears to execute instructions. To summarize, modern processors have long and complex pipelines. These pipelines are often able to re-order instructions or execute multiple instructions in parallel, to help them maximize performance. SSE means that the processor must behave like 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 in which memory accesses appear in the memory system. Because of mechanisms like 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. 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: Normal memory and Device memory.


Note

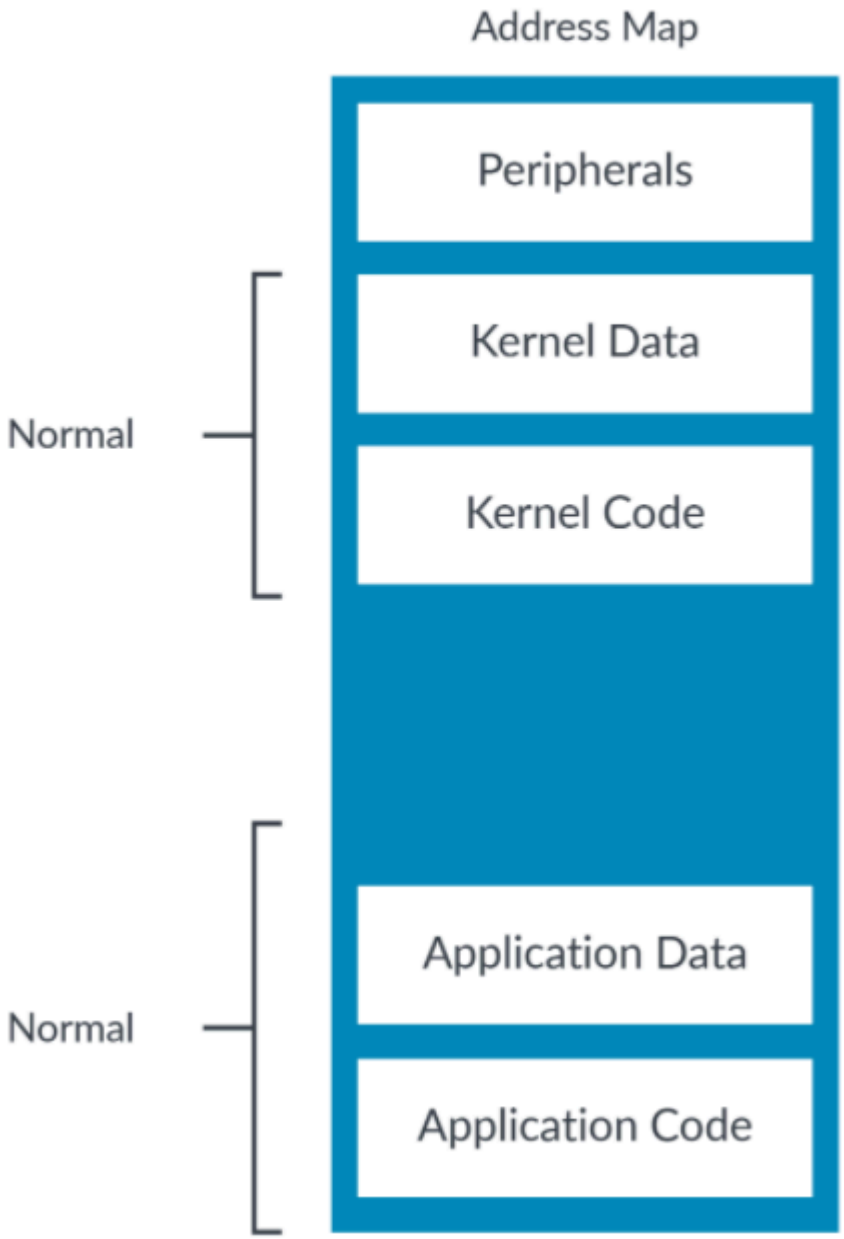
Armv6 and Armv7 include a third memory type: Strongly Ordered. In Armv8, this maps to Device_nGnRnE.

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 in this diagram:

Figure 6-1: A diagram showing the normal type.



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. We will introduce some of these optimizations here.

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

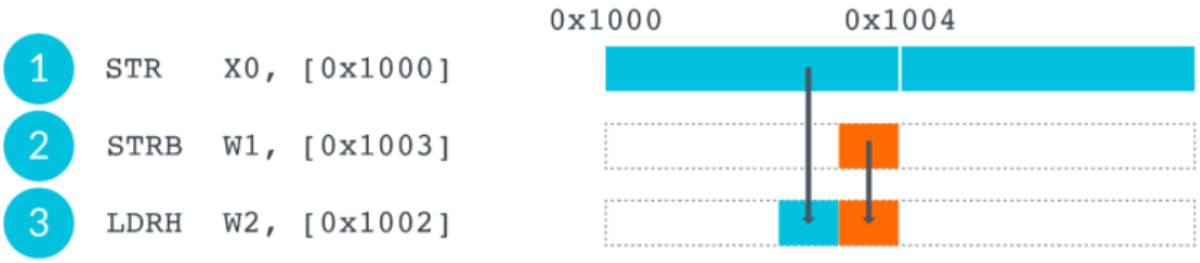
- Merge accesses. Code can access a location multiple times, or access multiple consecutive locations. For efficiency, the processor is permitted to detect and merge these accesses into a single access. For example, if software writes to a variable multiple times, the processor might only present the 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. Think about these optimizations like 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.

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 depends on its micro-architecture. From a software perspective, you should assume that the processor might do any or all of them.

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:

Figure 6-2: A diagram showing re-ordering code sequence.



If the processor were to re-order these accesses, this might result in the wrong value in memory, which is not allowed.

For accesses to the same bytes, 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 that 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 more subtle.

There are other cases in which 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 code 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 in which 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 when the value from a previous load is used to make a decision. This code example shows a load followed by a Compare and Branch on Zero operation that relies on the value from the load:

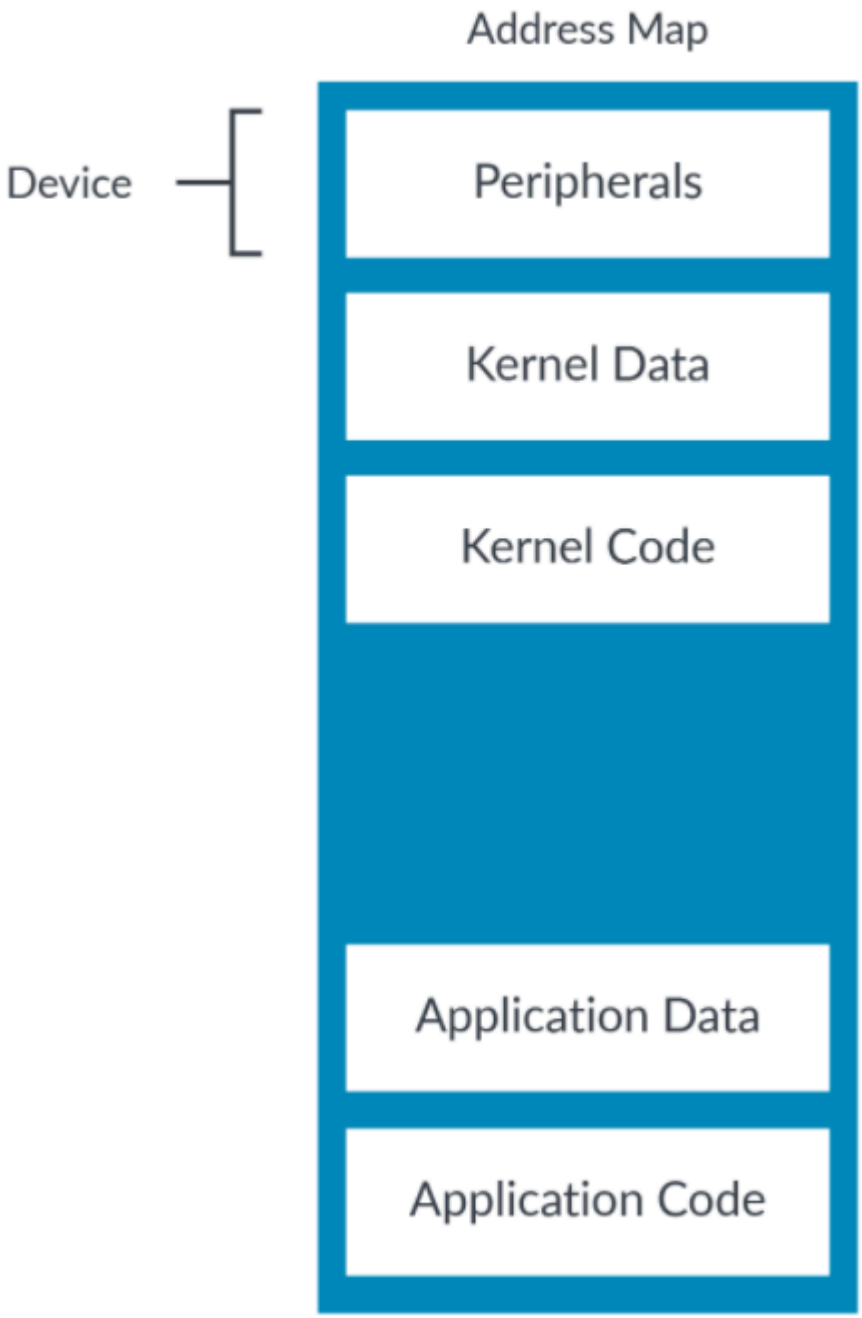
```
LDR X0, [X1]
CBZ X0, <somewhere_else>
STR X2, [X5][Symbol] // There is a control dependency on X0, this does not guarantee
 ordering.
```

There are cases in which 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. 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 7-1: 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.

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 same restrictions apply in the same way as for the Normal type.
- 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

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.

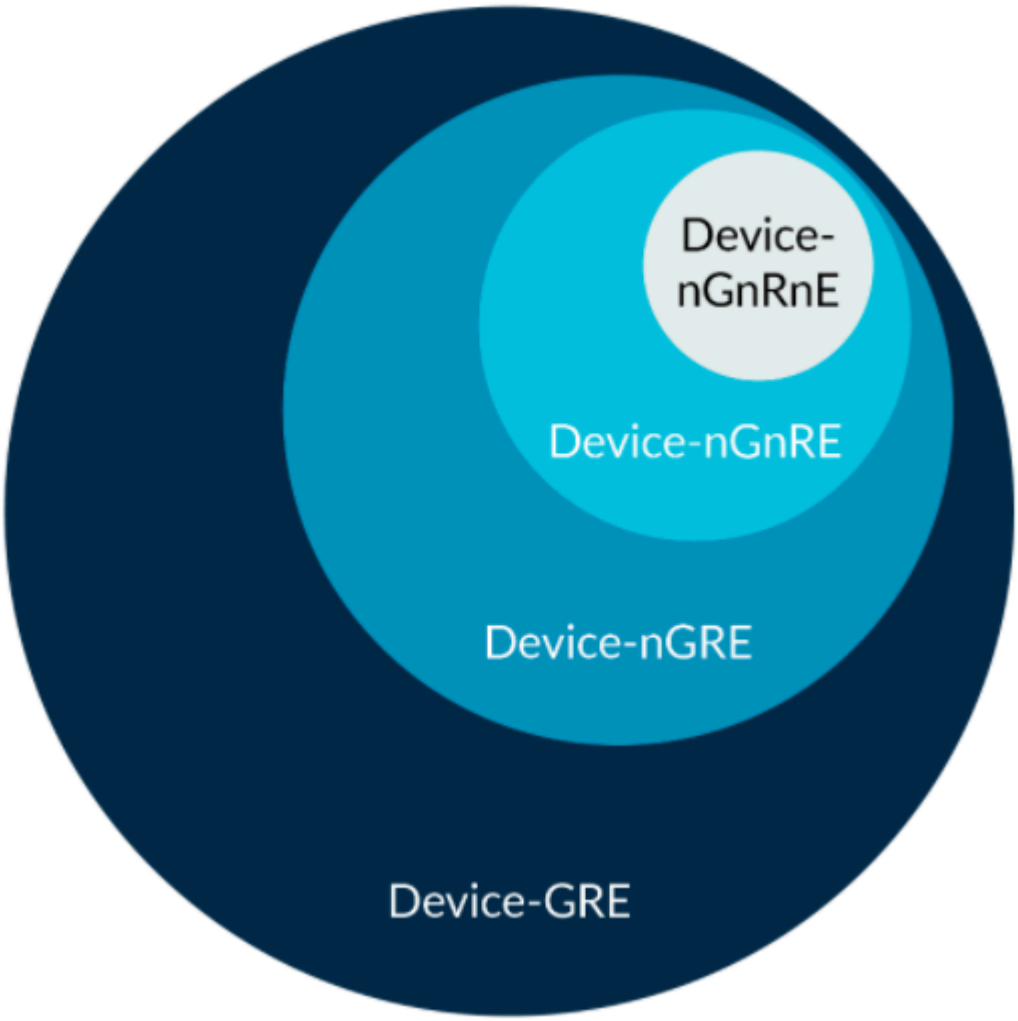
We have already seen how re-ordering work, but we have not introduced gathering or early write acknowledgement. Gathering allows memory accesses to similar locations to be merged into a single bus transaction, optimizing the access. Early write acknowledgement indicates to the memory system whether a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

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

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 7-2: 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.

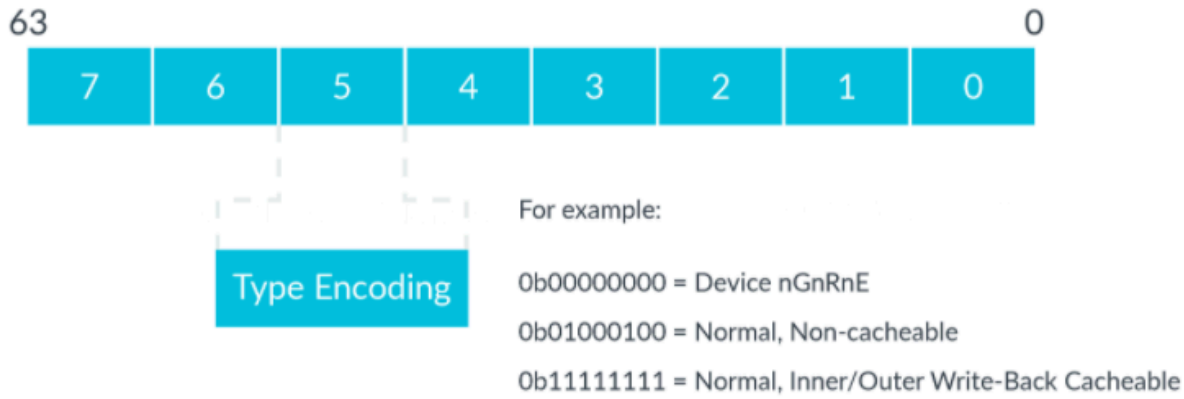
Importantly, all the behaviors allowed for Device_nGnRnE are also permitted for Device_GRE. For example, it is not a requirement for a Device_GRE memory to use Gathering - it is just allowed. Therefore, it would be permissible for the processor to treat Device_GRE as Device_nGnRnE.

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.

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).

Figure 8-1: 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.

9. Cacheability and shareability attributes

Locations marked as Normal also have cacheability and shareability attributes. These attributes control whether a location can be cached. If a location can be cached, these 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.

You can learn more about cacheability and shareability in our Caches and cache coherency guide (coming soon).

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.

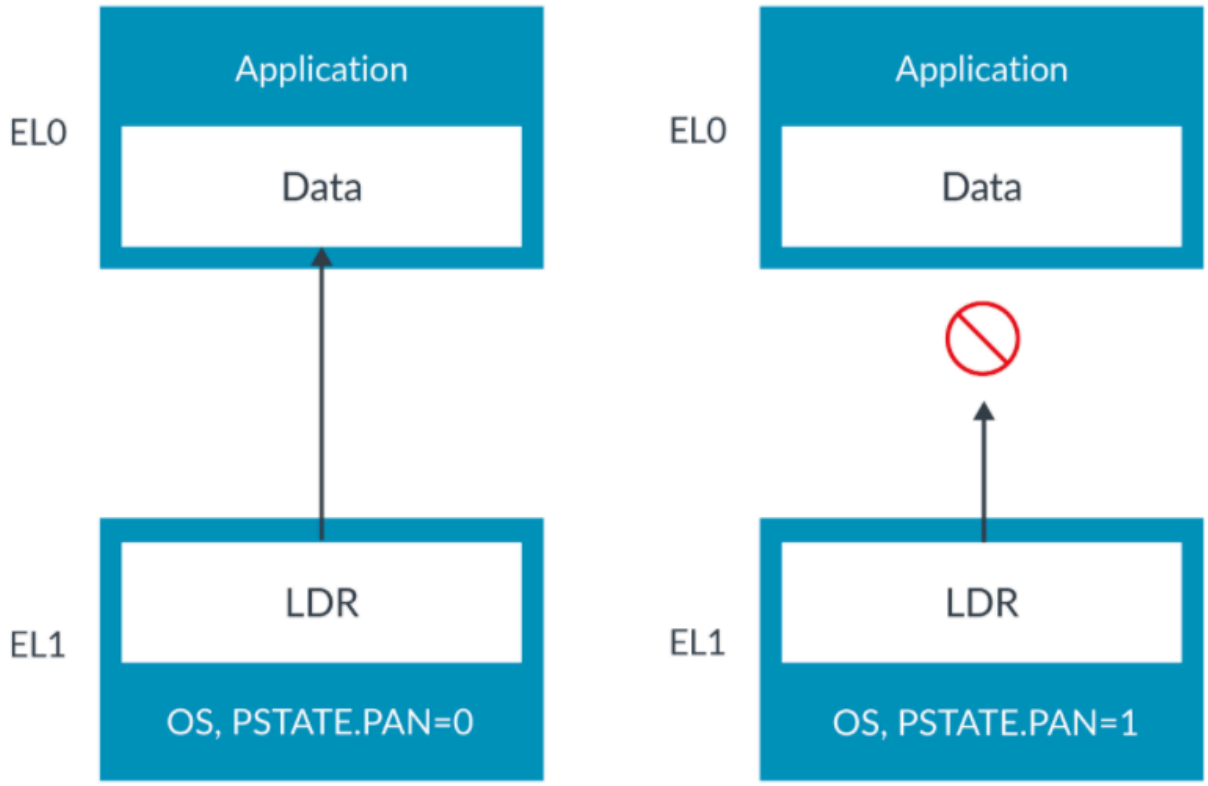
Privileged accesses to unprivileged data

The standard permission model is that a more privileged entity can access anything belonging to a less privileged entity. Explained another way, an Operating System (OS) can see all the resources that are allocated to an application. For example, a hypervisor can see all the resources that are 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 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 10-1: A diagram showing privileged access never.



Note

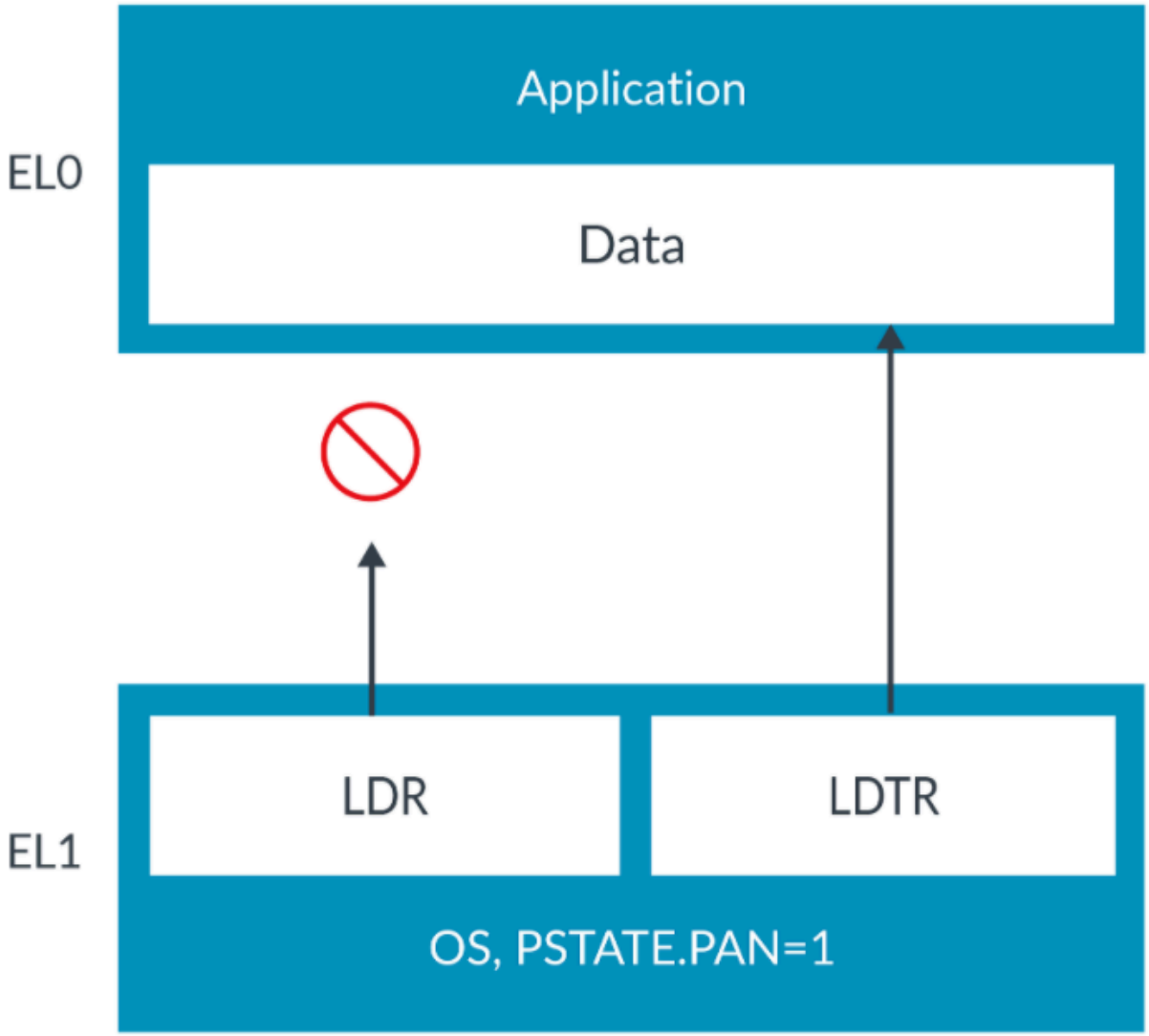
PAN was added in Armv8.1-A.

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. In either case, PAN allows us to trap the access before it occurs, ensuring safe operation.

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 10-2: 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.

Execution permissions

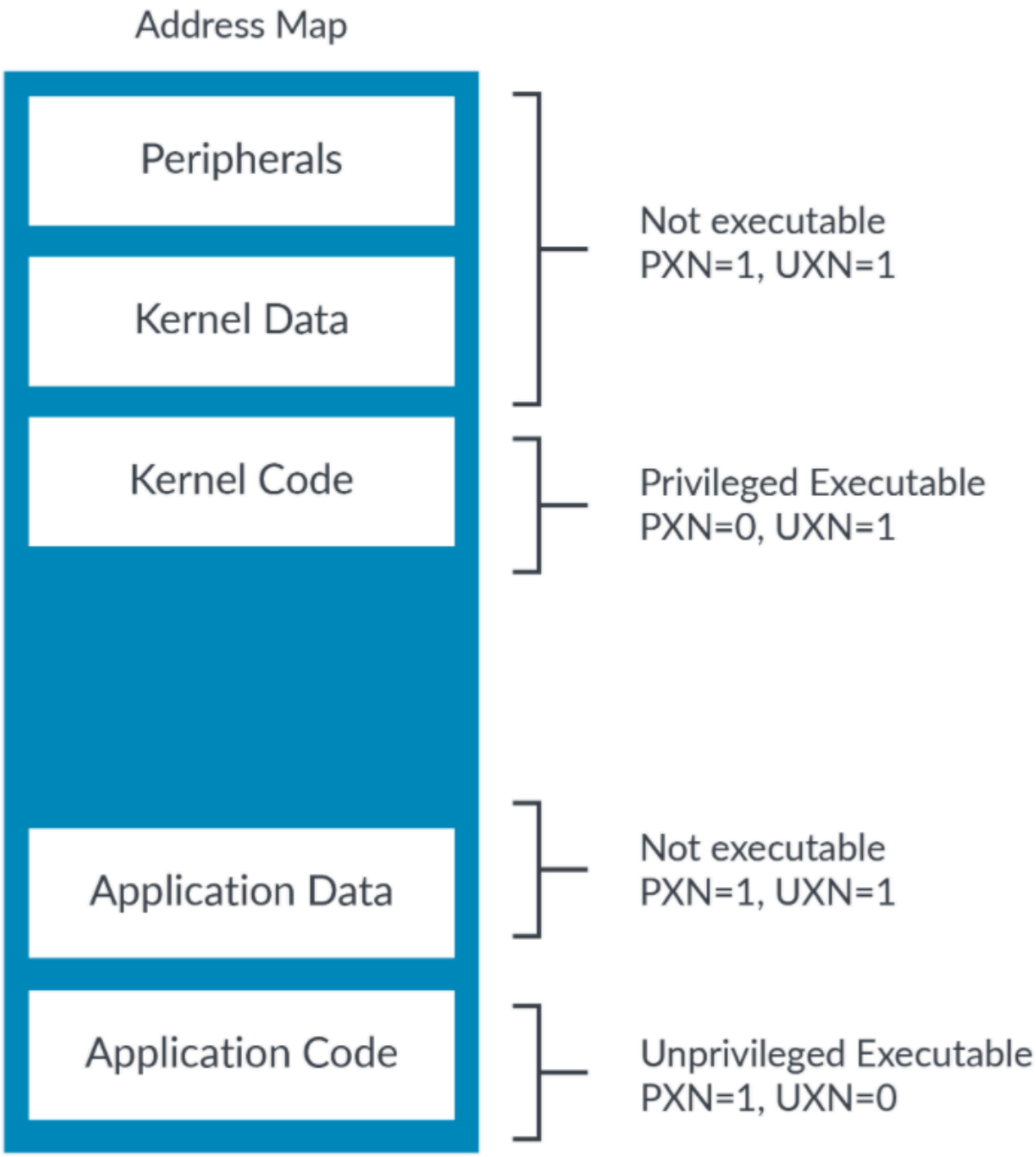
In addition to access permissions, there are also execution permissions. These attributes let you specify that instructions cannot be fetched from the address:

- 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, 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 10-3: 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.

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.

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.

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 and was added in Armv8.1-A.

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 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.

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.

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 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 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.

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](https://developer.arm.com/architectures/learn-the-architecture/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.

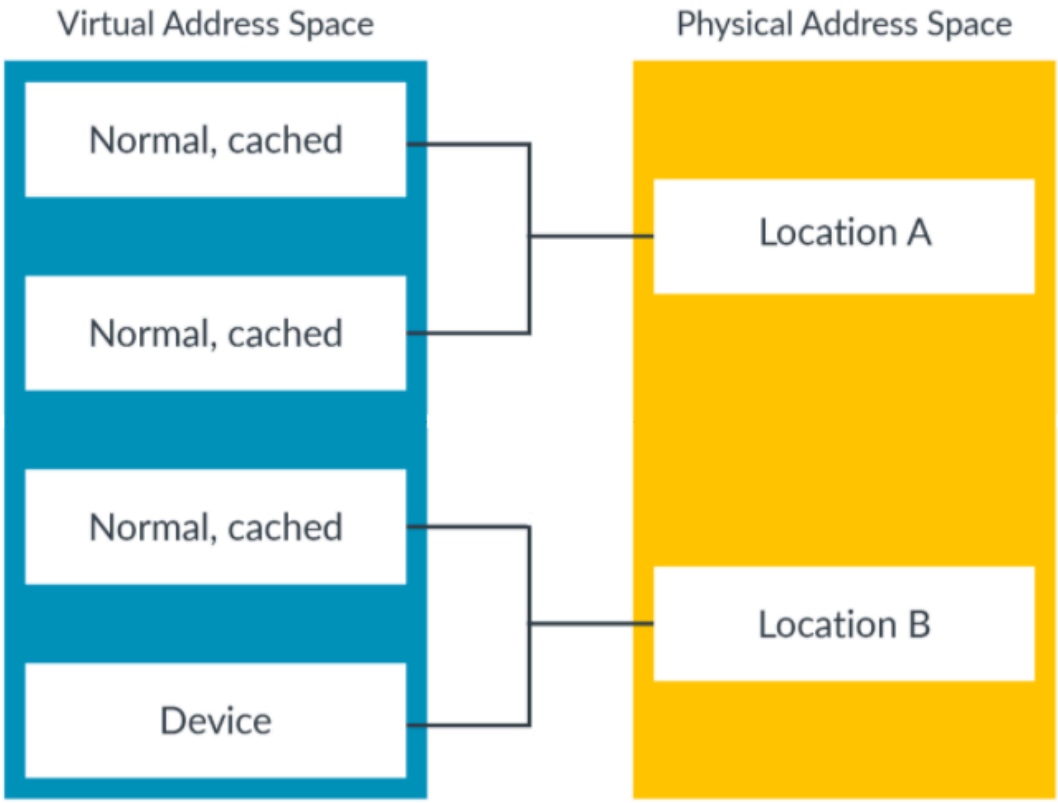
Attributes are based on virtual addresses. This is because attributes come from the translation tables. When a physical location has multiple aliases, it is important that all of the virtual aliases have compatible attributes. We describe compatible as:

- 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 like you expect, which 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), 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. Combining Stage 1 and Stage 2 attributes

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

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

Figure 14-1: A diagram showing stage 1 and 2 VAS attributes.

In the Arm architecture,the default 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.

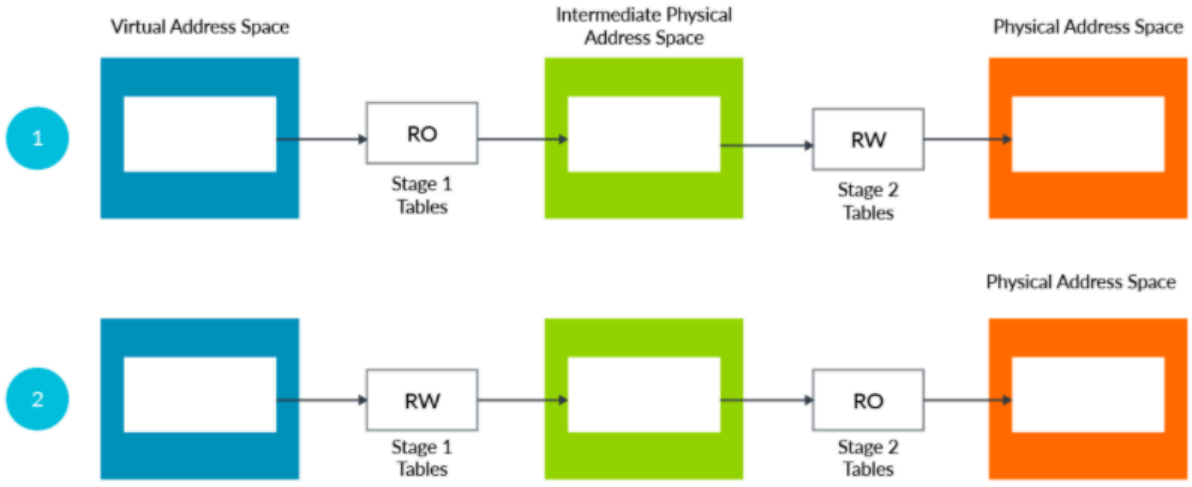
Note

HCR_EL2.FWB was introduced in Armv8.4-A.

Fault handling

Let's look at the two examples in this illustration:

Figure 14-2: A diagram showing fault handling.



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 look at an example in which Stage 1 and Stage 2 attributes are the same:

Figure 14-3: A diagram showing stage 1 and 2 VAS faults.

Here, the resulting attribute is simple. It is RO. 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

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.

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

Either:

- Address dependency
- Access to the same location
- Barrier

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.

16. 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 (coming soon) (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 are discussed in this guide.
- Armv8-A Instruction Set Architecture (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:

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 behavior can be partly overridden using the virtualization controls. This topic is covered in our Virtualization guide.

Cacheability and shareability attributes

Caches and cache coherency guide (coming soon).

Combining Stage 1 and Stage 2 attributes

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

For background information, see our Virtualization guide.

Useful links to training:

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

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 have learned through 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.