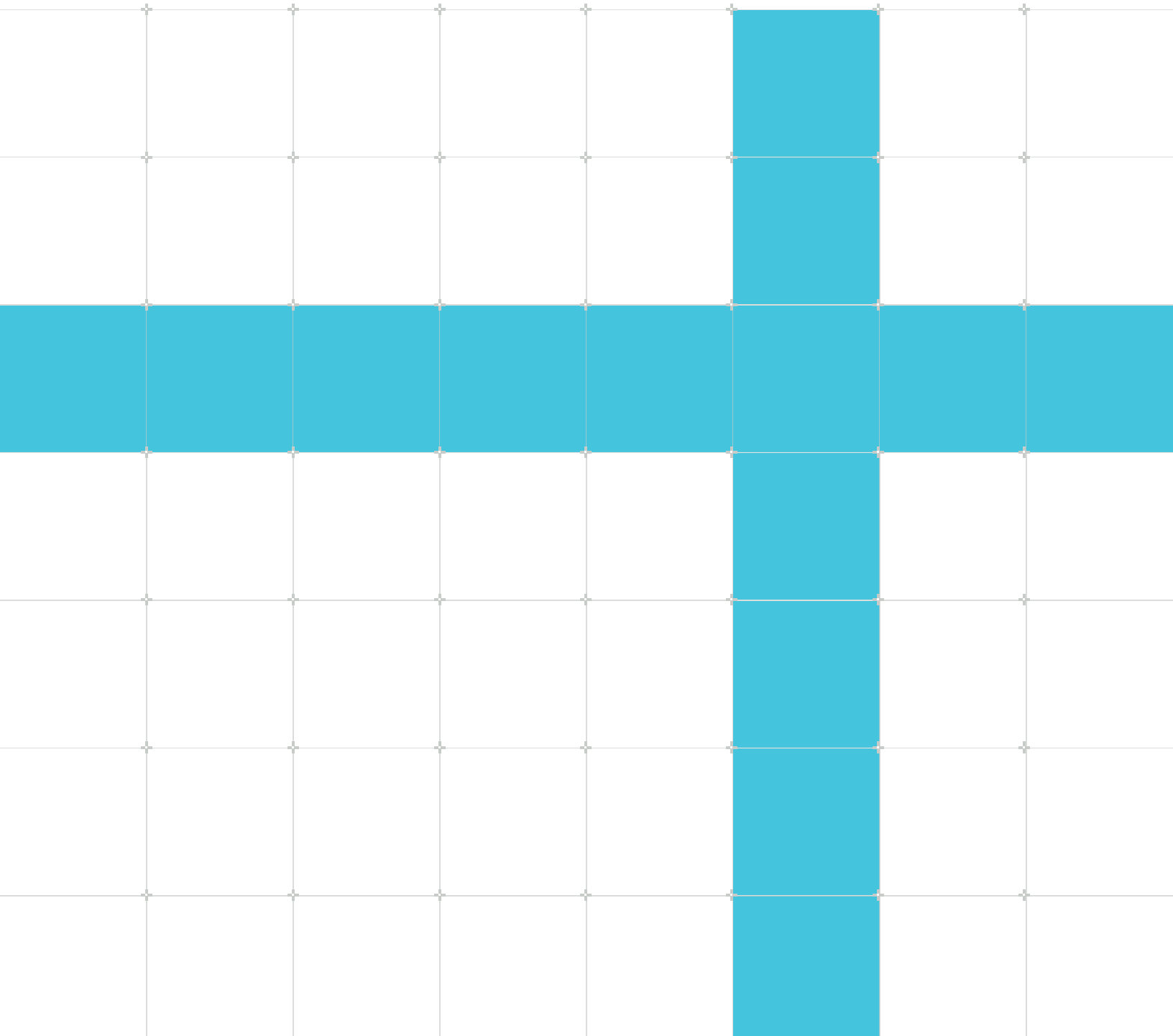
# Learn the Architecture - Booting the Cortex-R82

Revision r0p0

## Guide

**Non-Confidential**
Copyright © 2025 Arm Limited (or its affiliates).
All rights reserved.

**Issue 00**
109917_0000_00_en



# Learn the Architecture - Booting the Cortex-R82 Guide

This document is Non-Confidential.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.
Arm only permits use of this document if you have reviewed and accepted Arm's Proprietary Notice found at the end of this document.

The product described in this document complies with the requirements set out in the included conformance notice.

This document (109917_0000_00_en) was issued on 2025-05-15. There might be a later issue at https://developer.arm.com/documentation/109917

The product revision is r0p0.

See also: Proprietary notice | Conformance notices | Product and document information | Useful resources

## Start reading

If you prefer, you can skip to the start of the content.

## Intended audience

Embedded Software Developers

## 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.

## 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.

# Contents

**1. Overview** ..... **4**

**2. Before you begin** ..... **5**

**3. Architecture of the Cortex-R82** ..... **6**

**4. Cortex-R82 implementation details** ..... **9**

**5. About boot** ..... **10**

**6. MPU programming basics** ..... **13**

**7. EL2 boot steps** ..... **16**

**8. EL1 boot steps** ..... **26**

**9. Related information** ..... **30**

**10. Next steps** ..... **31**

**Proprietary notice** ..... **32**

**Conformance notices** ..... **34**

**Product and document information** ..... **36**
Product status ..... 36
Revision history ..... 36
Conventions ..... 37

**Useful resources** ..... **39**

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 1. Overview

This guide explains how to boot a Cortex-R82 from reset into EL2 to running an application at EL1. This guide sets up the Memory Protection Unit (MPU) in EL2 and EL1 to use the stage 1 translation regime.

To help explain steps you must do to boot the Cortex-R82 with the MPU enabled, short code examples are included.

The following is a high-level overview of the content in this guide:

- Architecture the Cortex-R82 implements.
- Details of the Cortex-R82 implementation.
- Other boot details not covered in this guide.
- Basics of programming the Cortex-R82 MPU.
- Boot steps for enabling the MPU in EL2 and EL1.
- Resources for this guide.
- Reading recommendations as a next step.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 2. Before you begin

This guide assumes that you are familiar with the following:

- R-Profile architecture
- The A64 Instruction Set Architecture (ISA)
- The programmer model for AArch64, particularly the following topics:
  - Exception levels
  - Security states
  - Memory attributes and properties

If you are unfamiliar with any of these listed topics, read the following guides before continuing with this guide:

- Introducing the R-Profile architecture guide: Introduces the R-Profile of the Arm architecture.
- AArch64 exception model: Introduces the exception and privilege model in AArch64.
- Introduction to security: Introduces some generic concepts about security on Arm platforms.
- A64 Instruction Set Architecture Guide: Introduces the A64 instruction set, used in the 64-bit Armv8-A architecture, also known as AArch64.
- AArch64 memory attributes and properties: Introduces the memory attributes and properties and explains the basics of memory ordering.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 3. Architecture of the Cortex-R82

The Cortex-R82 implements the Armv8-R AArch64 architecture. Previous R-profile processors implement the Armv8-R AArch32 architecture or the Armv7-R architecture. The following sections explain the different architecture and implementation aspects of the Cortex-R82.

## Cortex-R82 architecture and implementation documents

To understand the architecture used by the Cortex-R82, first read the Arm Architecture Reference Manual for A-profile architecture. This guide refers to the Arm Architecture Reference Manual for A-profile architecture as Arm Arm A-profile. Second, read the Arm Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile. This guide refers to the Arm Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile as the Arm Armv8-R AArch64 Supplement.

For duplicated information, use the information in the Arm Armv8-R AArch64 Supplement over the information in the Arm Arm A-profile. For example, both Arm Architecture reference manuals describe the `DSB` instruction, but the description relevant to the Cortex-R82 is in the Arm Armv8-R AArch64 Supplement.

For a listing of the required and optional architecture features of the Cortex-R82, read the Architecture requirements section of the Arm Cortex-R82 Processor Technical Reference Manual (TRM).

## Memory System Architecture

Cortex-R82 r0p2 and earlier supports Protected Memory System Architecture v8-64 (PMSAv8-64) at EL1/EL0 and EL2. This support means that you can use a Memory Protection Unit (MPU) for the address translation for the EL1&EL0 (EL1&0) and EL2 translation regimes.

Cortex-R82 r1p1 and later supports Virtual Memory System Architecture v8-64 (VMSAv8-64) or PMSAv8-64 at EL1/EL0. The Cortex-R82 only supports PMSAv8-64 at EL2. This support means that you can use a Memory Management Unit (MMU) or MPU for the address translation for the EL1&0 translation regime. However, only an MPU is used for the address translation for the EL2 translation regime.



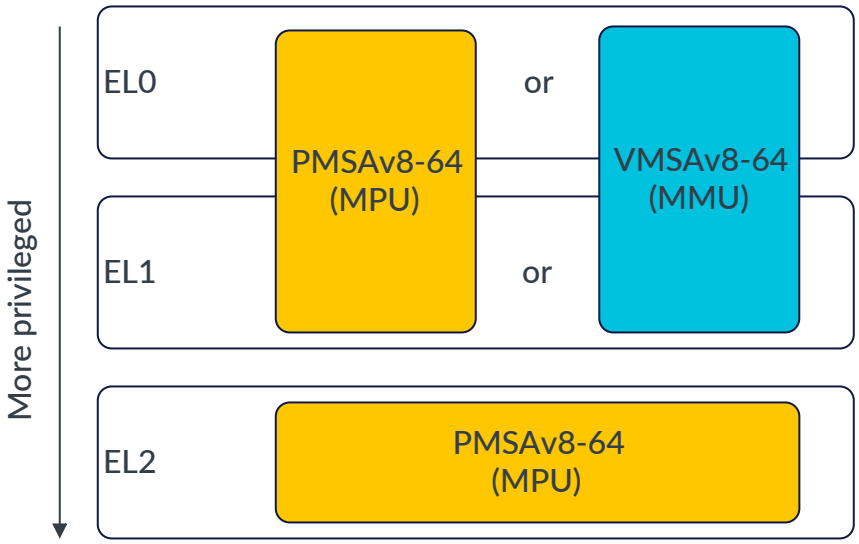
**Note**

VMSAv8-64 is a synthesis choice. Because of this choice, your system might not have VMSAv8-64 available.

The following diagram shows the Memory System Architecture (MSA) Cortex-R82 r1p1 and later uses:

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

**Figure 3-1: Diagram of Cortex-R82 memory system architecture**




The optional MMU implementation allows the Cortex-R82 to run a rich Operating System (OS) like Linux.

This guide focuses on the MSA of r1p1 and later. This guide only covers using the MPU for EL2 and EL1.

## Security state

Armv8-R AArch64 supports a single Security state, the Secure state. This Secure state is similar to the Secure state on an A-profile Arm processor.

Similar to the Armv8-A processors, when the Cortex-R82 is in Secure state, it can access 2 separate Physical Address Spaces (PAS):

- Secure PAS
- Non-secure PAS

When software accesses memory, the MPU or MMU controls which PAS is accessed. The MPU region registers and the MMU block and page descriptors include a Non-secure (NS) field that controls the output address type.

If your system has Cortex-R82 cores interacting with other cores with multiple Security states, consider carefully which PAS to use for a given area of memory.

For more information on PASs and how a memory system might use them, read Learn the architecture - TrustZone for AArch64.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Armv7-R and Armv8-R AArch32 does not allow software to control the output PAS.

## Differences between architectures

The following is a summary of some of the main differences between Armv8-R AArch64 and Armv8-A AArch64. This list is not exhaustive.

**Table 3-1: Main differences between Armv8-R AArch64 and Armv8-A AArch64**

| Topic | Armv8-R AArch64 | Armv8-A AArch64 |
|---|---|---|
| MSA | • EL1&0 translation regime: PMSAv8-64 or VMSAv8-64<br>• EL2 translation regime: PMSAv8-64 only | VMSAv8-64 in all translation regimes |
| Security state | Secure state. Can access Secure and Non-secure memory. | The number of Security states available is IMPLEMENTATION DEFINED |
| Barrier instructions | Same as ARMv8-A AArch64 except for redefines `DMB` and `DSB` instructions and adds `DFB` instruction | - |
| Exception levels | • EL2 is mandatory<br>• No EL3 | EL2 and EL3 are optional |

The following is a summary of some of the main differences between Armv8-R AArch64, Armv8-R AArch32, and Armv7-R. This list is not exhaustive.

**Table 3-2: Main differences between Armv8-R AArch64, Armv8-R AArch32, and Armv7-R**

| Topic | Armv8-R AArch64 | Armv8-R AArch32 | Armv7-R |
|---|---|---|---|
| Memory Management | • EL2: MPU<br>• EL1 and EL0: MMU or MPU | EL2, EL1, and EL0: MPU | PL1 and PL0: MPU |
| Security state | Secure state but can access Non-secure memory | Non-Secure state | NA |
| Register size | 64-bits | 32-bits | 32-bits |
| ISA | A64 | Arm (A32) and Thumb (T32) | Arm (32-bits) and Thumb (16-bits) |
| MPU region sizes | With FEAT_LPA, configurable to 52-bit address. Without FEAT_LPA, configurable to 40-bit address. | Configurable to 32-bit address | Power of 2 |

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 4. Cortex-R82 implementation details

In the previous section, you learned about the architecture of the Cortex-R82. In this section, we focus on the implementation details of the Cortex-R82.

To understand how the Cortex-R82 implements the Armv8-R AArch64 architecture, read Arm Cortex-R82 Processor Technical Reference Manual (TRM).

The Arm Cortex-R82 Processor Datasheet describes the features of the Cortex-R82.

For a comparison of the Cortex-R82 features to other Cortex-R processors, read Arm Cortex-R Processor Comparison Table.

The following lists some of the features of the Cortex-R82 and where you can find the relevant documentation:

- Architecture
  - A-profile architecture: Arm Architecture Reference Manual for A-profile architecture (Arm Arm A-profile)
  - R-profile AArch64 architecture: Arm Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile (Arm Armv8-R AArch64 supplement)
- TRM: Arm Cortex-R82 Processor Technical Reference Manual (Cortex-R82 TRM)
- AArch64 ISA
  - A-profile description: “AArch64 Instruction Set” section of the Arm Arm A-profile
  - R-profile AArch64 description: “A64 Instruction Set for Armv8-R AArch64” section of the Arm Armv8-R AArch64 Supplement
- Memory model
  - VSMAv8-64: “AArch64 Virtual Memory System Architecture” section of the Arm Arm A-profile
  - PSMAv8-64: “Armv8-R AArch64 Protected Memory System Architecture” section of the Arm Armv8-R AArch64 Supplement
- GIC
  - Architecture: Arm Generic Interrupt Controller (GIC) Architecture Specification
  - Implementation: “GIC CPU Interface” section of the Cortex-R82 TRM
- Generic Timer
  - Architecture: “The Generic Timer in AArch64 state” section of the Arm Arm A-profile
  - Implementation: “Generic Timer” section of the Cortex-R82 TRM

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 5. About boot

Depending on what operating environment you are trying to create, the boot process might involve many steps. This guide focuses on booting a Cortex-R82 from reset into EL2 to EL1 using the MPU at EL2 and EL1. This guide does not include other possible boot requirements like setting up:

- Virtualization
- MMU
- Generic Interrupt Controller (GIC)
- Generic Timer
- Advanced SIMD and floating-point
- Tightly Coupled Memory (TCM)
- Low-latency RAM (LLRAM)

This section considers other possible boot requirements in the context of a Cortex-R82. The boot requirement list is not exhaustive.

## Virtualization

Virtualization is a technology that allows a single machine to run multiple OSes. A hypervisor is a piece of software that is responsible for creating, managing, and scheduling of Virtual Machines (VMs). Virtualization environments usually use hypervisors to allow processors to run multiple VMs potentially running difference OSes at the same time.

Rich OSes like Linux require an MMU to provide Virtual to Physical Address mapping. Bare-metal OSes like FreeRTOS require an MPU to provide memory protection and attributes.

The Cortex-R82 supports either VMSAv8-64 or PMSAv8-64 for the EL1&0 translation regime and PMSAv8-64 for EL2. VMSAv8-64 means that an MMU is used. PMSAv8-64 means that an MPU is used. These features enable an environment where a hypervisor at EL2 can control VMs at EL1 running either Rich or bare-metal OSes.

What makes the Cortex-R82 different from Armv8-A processors is it uses a single Security state model. For A-profile processor environments, the hypervisor and the OSes often run in Non-secure state. On the Cortex-R82, the hypervisor and OSes run in Secure state.

If you compare Cortex-R82 and Cortex-R52 virtualization, the main difference is the Cortex-R82 supports either MMU or MPU for the EL1&0 translation regime. The Cortex-R52 only supports MPU for the EL1&0 translation regime.

Armv8-R virtualization describes Cortex-R52 virtualization and is a good reference for Cortex-R82 virtualization.

Learn the architecture - AArch64 virtualization Guide describes Armv8-A AArch64 virtualization and is also a good reference for Cortex-R82 virtualization.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

## MMU

To take advantage of a richer operating environment, you might want to use the MMU for the EL1&0 translation regime. If you are running bare-metal code, setup and enable the MMU before executing application code.

Learn the architecture - AArch64 memory management Guide describes the A-profile memory management and is a good reference for the Cortex-R82 MMU.

## GIC

The Generic Interrupt Controller (GIC) is responsible for routing interrupts to one or more cores and managing them throughout their lifecycle. Usually the GIC is setup by a hypervisor in a virtualized environment or host software in a bare-metal environment. The Cortex-R82 GIC conforms to the GICv3.2 architecture.

The Cortex-R82 implements a single Security state, Secure, so if you are familiar with A-profile processors, the GIC might appear somewhat different. For A-profile processors, most environments hypervisors present to VMs only have a single Security state. This presentation means VMs see the GIC exactly the same way as the Cortex-R82.

The GIC is different from bare-metal A-profile environments as follows:

- The A-profile has 3 groups, Group0, Non-secure Group1, and Secure Group1. The Cortex-R82 only has 2 groups, Group0 and Group1.
- In the A-profile, Secure software can only allocate the top half of the priority space. In the Cortex-R82, software can allocate all the priority space.
- Single Security state implementations like the Cortex-R82 have Group0 as FIQ interrupts and Group1 as IRQ interrupts.

The GIC is different from Cortex-R52 as follows:

- On the Cortex-R52, the GIC is integrated into the processor. On the Cortex-R82, the GIC is a separate component. To learn about the Cortex-R82 GIC, read the separate Arm Generic Interrupt Controller (GIC) Architecture Specification.
- The Cortex-R52 does not have Locality-specific Peripheral Interrupts (LPIs). The Cortex-R82 optionally has LPIs.

The “Generic Interrupt Controller” section of Armv8-R virtualization describes the Cortex-R52 GIC and is a good reference for the Cortex-R82 GIC.

Learn the architecture - Generic Interrupt Controller v3 and v4, Overview describes the A-profile GIC and is also a good reference for the Cortex-R82 GIC.

## Generic Timer

The Generic Timer provides the timer framework for on-chip processors. The Generic Timer for the Cortex-R82 is very similar to the Generic Timer A-profile processors use. The main difference between the Cortex-R82 and the A-profile Generic Timer is the timers implemented.

The Cortex-R82 implementation of the Generic Timer only has the following timers:

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- An EL1 physical timer
- A Secure EL2 physical timer
- An EL1 virtual timer

The Cortex-R82 does not include the system counter.

Learn the architecture - Generic Timer describes the A-profile Generic Timer and is a good reference for the Cortex-R82 Generic Timer.

### Advanced SIMD and floating-point

Advanced SIMD, also known as NEON, is an extension to the ISA that adds instructions to perform mathematical operations in parallel on multiple data streams.

Cortex-R82 supports half-precision, single-precision, and double-precision floating-point operations. The Cortex-R82 floating-point implementation includes several features from Armv8.2 and Armv8.3.

To use NEON or floating-point, you must enable them before executing any NEON or floating-point instructions.

How do I enable the Advanced SIMD (NEON) and Floating-point Unit on Armv8-A processors? provides more information about enabling NEON and floating-point.

Learn the architecture - Introducing Neon provides more information about NEON.

### TCM

Each core within the Cortex-R82 processor optionally has 2 TCMs: an Instruction Tightly Coupled Memory (ITCM) and a Data Tightly Coupled Memory (DTCM). The ITCM can contain instructions and data and the DTCM contains data.

You can configure the Cortex-R82 to enable the ITCM out of reset so the core boots from the ITCM.

The "Initializing TCMs" section of the Arm Cortex-R82 Processor TRM provides more information about using the TCMs.

### LLRAM

The Cortex-R82 processor has an optional LLRAM manager interface. The cores within the cluster share this interface.

You can configure the Cortex-R82 to enable the LLRAM out of reset so the core boots from the LLRAM.

The "Initializing LLRAM" section of the Arm Cortex-R82 Processor TRM provides more information about using the LLRAM.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 6. MPU programming basics

To setup memory regions and their attributes, program the MPU registers. Enable or disable the regions by programming the MPU registers.

Depending on the synthesis configuration, the Cortex-R82 can have 0, 16, or 32 MPU regions.


Note

x in this section is 1 or 2 to define either EL1 or EL2.

## MPU Background region

If SCTLR_ELx.BR is enabled and the MPU is disabled, the MPU uses the default memory map as the Background region for generating memory attributes. For the Cortex-R82, the following Arm Cortex-R82 Processor Technical Reference Manual sections define the default memory map:

- EL1-controlled MPU Background region
- EL2-controlled MPU Background region


Note

The default memory map is IMPLEMENTED DEFINED, not architectural defined. Different Armv8-R implementations might use different default memory maps. For example, the Cortex-R82 and Cortex-R52 use different default memory maps.

If the MPU and the Background region are not enabled, PMSAv8-64 uses the same memory attribute as defined by VMSAv8-64 when stage 1 translation is disabled.

## MPU programming

The MPU sets up memory regions using PRSELR_ELx, PRBABR_ELx, and PRLAR_ELx.

Use PRSELR_ELx.REGION[7:0] to configure the number of current MPU regions visible to PRBAR_ELx and PRLAR_ELx. On the Cortex-R82, PRSELR_ELx.REGION[7:4] makes 16 MPU regions visible to PRBAR_ELx and PRLAR_ELx. If 32 MPU regions are implemented, setting PRSELR_ELx.REGION[7:4] to `0b0000` and `0b0001` allows the configuration of all MPU regions.

PRBAR_ELx defines the base address of the MPU region and some attributes of the region. The following fields are available in PRBAR_ELx:

- bits [47:6] = BASE[47:6], Base address of MPU memory region
- bits [5:4] = SH[1:0], Shareability attribute
- bits [3:2] = AP[2:1], Access Permission attributes
- For PRBAR_EL2:
  - bit [1:0] = XN, Execute Never

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For PRBAR_EL1:
  - bit [1] = XN, Execute Never

For stage 2 EL1&0 translation regime when FEAT_XNX is implemented, the behavior of XN[1:0] is the same as that defined by VMSAv8-64 for EL1&0 stage 2 translation table XN[1:0], bits[54:53] field in the Armv8-A architecture.


Note

Cortex-R82 does not support FEAT_LPA which extensions the address space from 48 bits to 52 bits. Because FEAT_LPA is not supported, PRBAR_ELx.BASE[51:48] is not implemented.

PRLAR_ELx defines the following:

- The limit address of the region.
- If the memory region is in Secure or Non-secure memory.
- Additional attributes in the MAIR_ELx register.
- If the region is enabled.

The following fields are available in PRLAR_ELx:

- bits [47:6] = LIMIT[47:6], Limit address of MPU memory region.
- bit [4] = NS, Non-secure bit.
- bits [3:1] = AttrIndx[2:0], Selects attributes from within the MAIR_ELx register.
- bit [0] = EN, Region enable bit.


Note

Cortex-R82 does not support FEAT_LPA which extensions the address space from 48 bits to 52 bits. Because FEAT_LPA is not supported, PRLAR_ELx.LIMIT[51:48] is not implemented.

The MAIR_ELx register contains additional region attributes. Set the following attributes in MAIR_ELx:

- Memory type
  - If set to Device memory, also setups the Gathering, Reordering, and Early Write Acknowledgement attributes.
- Write-Through or Write-Back attribute

Cortex-R82 supports 2 methods for programming the MPU regions: an indirect method and a direct method.

The indirect method requires setting the PRSELR_ELx register to select which PRBAR_ELx and PRLAR_ELx pair is written before setting PRBAR_ELx and PRLAR_ELx. So, you must modify PRSELR_ELx each time you want to program an MPU region.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The direct method uses PRSELR_ELx to select a block of 16 regions to write. Then uses PRBAR<n>_ELx and PRLAR<n>_ELx notation to program the region, where <n> is an integer between 1 to 15. This method allows you to set the first 16 regions without modifying PRSELR_ELx each time. If you are setting up more than 16 regions, you must select the next 16 regions using PRSELR_ELx.REGION[7:0].

This guide uses the direct method to program the MPU regions.



**Note**

Cortex-R52 also supports a direct method for programming the MPU regions. The direct method used by Cortex-R52 and Cortex-R82 are not the same.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

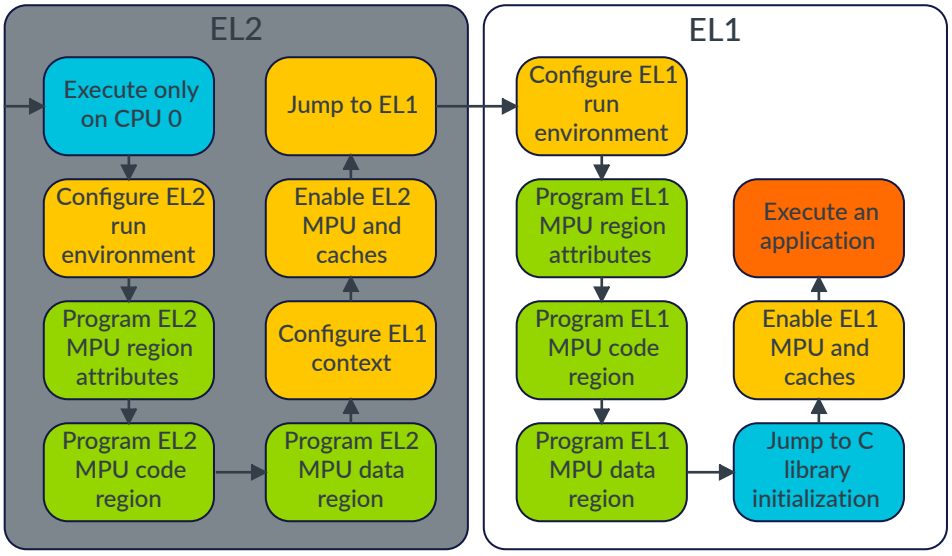
# 7. EL2 boot steps

To show the boot process for EL2 with a focus on programming the MPU, this section uses short code examples. The guide assumes:

- The code examples are built with Arm Compiler for Embedded 6.
- Your application code is written in the C programming language.
- You are running in a system with multiple central processing units (CPUs). The code examples are assumed to run on the first CPU, CPU 0, and all other CPUs in the system are put to sleep.
- After the MPU setup is complete, the `__main` function is used to start C library initialization to setup the C library.
- The application code is run after C library initialization finishes.

The following diagram shows the high-level boot sequence for EL2 and EL1 documented in this guide:

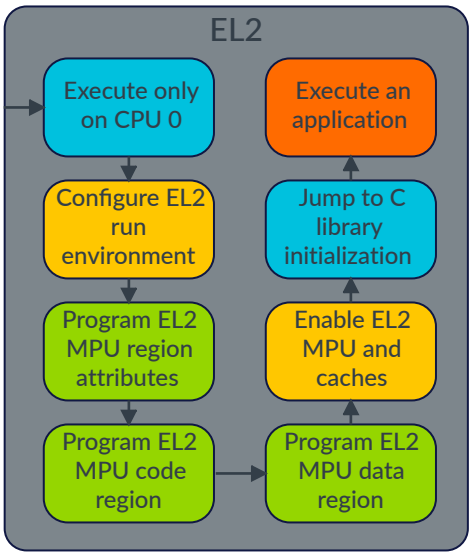
**Figure 7-1: Diagram of boot sequence for EL2 and EL1**




This guide also documents an alternative boot sequence. The alternative boot sequence is where you only boot EL2 and remain in EL2 to execution application code. The following diagram shows the high-level boot sequence for EL2 only:

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

**Figure 7-2: Diagram of boot sequence for EL2**




This section only covers the EL2 part of the boot sequence, highlighted in gray in the previous diagrams.

To understand the entire boot sequence, read this section and the following section in order from start to finish.

Note

The code examples in this guide show a basic MPU setup. Other MPU settings and attributes are available.

Note

If a code example line uses a variable instead of a value, the value of the variable is included in the comments for that line of code.

## Placement of code and data in memory

The code examples assume the target has the following memory areas:

- 1 code area
- 1 data area
- 1 stack area

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- 1 heap area

The following values are used in the code examples to represent certain memory addresses:

- `el1_vectors` = vector base address for EL1
- `el2_vectors` = vector base address for EL2
- `Image$$ARM_LIB_STACK$$ZI$$Base` = Beginning address of the stack area
- `Image$$CODE$$Base` = Beginning address of the code area
- `Image$$CODE$$Limit` = End address of the code area
- `Image$$DATA$$Base` = Beginning address of the data area
- `Image$$ARM_LIB_STACK$$ZI$$Limit` = End address of the stack area and data area

In Arm Compiler for Embedded 6, some of the previous values can map onto area names or special symbols in a scatter-loading file. For more information, read:

- "Image symbols" section of the ARM Compiler armlink User Guide Verion 6.00
- "Placing the stack and heap with a scatter file" section of the Arm Compiler for Embedded

## Execute only on CPU 0

At the start of the boot sequence, you get the ID for the currently executing CPU. To determine the ID, read and analyze the MIDR_EL1 and MPIDR_EL1 registers. If the CPU ID is not for CPU 0, put the CPU to sleep using an infinite `WFI` instruction loop.

The following code is the start of the boot sequence and puts any CPU other than CPU 0 to sleep:

```
    .global start64
    .type start64, "function"
start64:
    // Extract the core number from MPIDR_EL1 and store it in x19
    // (defined by the AAPCS as callee-saved), so we can re-use it later
    bl GetCPUID
    mov x19, x0

    // If run on a multi-core system, put any secondary cores to sleep
    cbz x19, core0_only
loop_wfi:
    dsb SY      // Clear all pending data accesses
    wfi         // Go to sleep
    b loop_wfi
```

The following code gets the CPU ID:

```
    .type GetCPUID, "function"
    .cfi_startproc
GetCPUID:

    mrs x0, MIDR_EL1
    ubfx    x0, x0, #4, #12 // extract PartNum
    b Others

Others:
    mrs x0, MPIDR_EL1
    ubfx    x1, x0, #MPIDR_EL1_AFF0_LSB, #MPIDR_EL1_AFF_WIDTH   //
 MPIDR_EL1_AFF0_LSB = 0, MPIDR_EL1_AFF_WIDTH = 8
```

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ubfx    x2, x0, #MPIDR_EL1_AFF1_LSB, #MPIDR_EL1_AFF_WIDTH   //
MPIDR_EL1_AFF1_LSB = 8, MPIDR_EL1_AFF_WIDTH = 8
    add     x0, x1, x2, LSL #2
    ret
    .cfi_endproc
```

## Configure the EL2 run environment

Create a useful run environment by programming various System registers and the Stack Pointer, SP.



Note

In this section and throughout this guide, after writing to System registers, execute an `ISB` instruction to ensure that you have a synchronized context.

1. In HCR_EL2, to configure all the architectural unknown values, except for the **RES1** bits, set HCR_EL2 to `0x0`.

```
// Set architectural unknown values in HCR to zero
ldr x1, =HCR_EL2_zeroed  // HCR_EL2_zeroed = 0x80000002
msr HCR_EL2, x1
isb
```

2. Except for the **RES1** bits, set SCTLR_EL2 and SCTLR_El1 to `0x0`.

```
// Program the SCTLRs
ldr x1, =SCTLR_EL2_zeroed  // SCTLR_EL2_zeroed = 0x30C50830
msr SCTLR_EL2, x1

ldr x1, =SCTLR_EL1_zeroed  // SCTLR_EL1_zeroed = 0x30500980
msr SCTLR_EL1, x1
isb
```

   The SCTLR registers manage the top level controls for the system including the memory system. In Armv8-R AArch64, most System registers do not have a default value. Setting the SCTLR registers to `0x0` creates a defined value for these registers.

3. Setup VBAR_EL1 and VBAR_EL2 so the correct vector address and function is used if an exception occurs during execution. On CPU 0, setup VBAR_EL1 and VBAR_EL2 to the vector base address for EL1 and EL2 respectively.

```
//
// Program the VBARs
//
ldr x1, =el1_vectors  // el1_vectors = vector base address for EL1
msr VBAR_EL1, x1

ldr x1, =el2_vectors  // el2_vectors = vector base address for EL2
msr VBAR_EL2, x1
```

4. To configure the EL2 SP, store the beginning address of the stack to SP.

```
// Configure the EL2 SP
```

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
ldr x1, =Image$$ARM_LIB_STACK$$ZI$$Base  // Image$$ARM_LIB_STACK$$ZI$$Base =
 Beginning address of the stack area
mov SP, x1
```

5. To ensure all stage 2 translations for the Non-secure Intermediate Physical Address space (IPAS) from Secure EL1 and EL0 use the Non-secure PAS, set the VCTR_EL2.NSA bit[30] to `0b1`.

```
// Configuring security state
mrs x1, VTCR_EL2
orr x1, x1, #VTCR_EL2_NSA  // VTCR_EL2_NSA = (1 << 30)
msr VTCR_EL2, x1
```

6. Further setup Non-secure memory accesses using the following bits in the VSTCR_EL2 register:
   - Ensure all stage 2 translations for the Secure IPAS access the Secure PAS by setting SA bit [30] to `0b0`.
   - If the stage 1 and 2 NS configuration do not match for the EL1 and EL0 translation regime, generate a fault by setting VSTCR_EL2.SC bit [20] to `0b1`.

```
mrs x1, VSTCR_EL2
orr x1, x1, #VSTCR_EL2_SC  // VSTCR_EL2_SC = (1 << 20)
bic x1, x1, #VSTCR_EL2_SA  // VSTCR_EL2_SA = (1 << 30)
msr VSTCR_EL2, x1
```

7. To avoid trapping accesses under the following conditions, in CPTR_EL2, set the following bits to `0x0`:
   - Disable trapping for accesses to CPACR_EL1 and CPACR using TCPAC bit [31].
   - Disable trapping System register accesses to implemented trace registers using TTA bit [20].
   - Disable trapping instructions that access Advanced SIMD and floating-point functionality using TFP bit [10].

```
// Disable trapping
ldr x1, =CPTR_EL2_zeroed  // CPTR_EL2_zeroed = 0x33FF
msr CPTR_EL2, x1
isb
```

## Program EL2 MPU region attributes

Because there is a code area and a data area that require different attributes, you program 2 MPU regions. Program 1 region for the code area and 1 region for the data area. In this guide, the region for the code area is referred to as the code region. The region for the data area is referred to as the data region.

In the following sections, PRLAR_EL2 and PRLAR1_EL2 were associated with the MAIR_EL2.Attr0 and MAIR_EL2.Attr1 respectively. MAIR_EL2.Attr0 has the attributes for the code region. MAIR_EL2.Attr1 has the attributes for the data region.

Now, you program MAIR_EL2 with the following:

- Attr0[7:0] = Normal memory, inner and outer read/write cacheable, and Write-Through

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Attr1[15:8] = Device memory that is non-Gathering (nG), non-Reordering (nR), and no Early Write Acknowledgement (nE)

```
// MAIR_EL2 configuration
mrs x0, MAIR_EL2
ldr x1, =0xBB                   // Normal inner/outer RW cacheable, write-through
bfi x0, x1, #0, #8              // Update Attr0
ldr x1, =0x04                   // Device nGnRnE
bfi x0, x1, #8, #8              // Update Attr1
msr MAIR_EL2, x0
isb
```

You must set all the attributes before enabling the MPU regions by programming MAIR_EL2 before PRLAR_EL2 and PRLAR1_EL2.

### Program EL2 MPU code region

After setting up MAIR_EL2, program the MPU to enable the EL2 stage 1 translation regime.

In this guide, you program a code region then program a data region.

To select the first set of 16 MPU regions to program, set PRSELR_EL2.REGION to `0b00000000`.

```
// Select first 16 MPU regions to program
ldr x1, =Region  // Region = 0x00
msr PRSELR_EL2, x1
isb
```

After programming PRSELR_EL2, execute an `ISB` instruction. The `ISB` ensures that the correct MPU regions are used for PRBAR_EL2 and PRLAR_EL2.

Program PRBAR_EL2 for the code region with the following:

- Retrieve the beginning address of the code region and set BASE[47:6].
- Set the Shareability attribute as Non-Shareable, SH[1:0] = `0b00`.
- Set the region access to read-only for EL2, EL1, and EL0, AP[2:1] = `0b11`.
- Instruction fetch from the region is permitted, XN[1:0] = `0b00`.

```
// Region 0 - Code
ldr x1, =Image$$CODE$$Base  // Image$$CODE$$Base = Beginning address of the code
 area
ldr x2, =((Non_Shareable << 4) | (RO_this_EL_and_lower << 2))  // Non_Shareable
 = 0x0, RO_this_EL_and_lower = 0x3
orr x1, x1, x2
msr PRBAR_EL2, x1
```

Program the PRLAR_EL2 for the code region with the following:

- Retrieve the end address of the code region, clear the bottom 6-bits, and set LIMIT[47:6].
- Specify the output address is in the Non-secure address space, NS[4] = `0b1`.
- Set the associated MAIR_EL2 to Attr0, AttrIndx[3:1] = `0b000`.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Enable the region, EN[0] = `0b1`.

```
ldr x1, =Image$$CODE$$Limit     // Image$$CODE$$Limit = End address of the code area
sub x1, x1, #1                  // Convert limit from exclusive to inclusive
bfc x1, #0, #6                  // and clear the lower 6 bits
ldr x2, =((RegionNS << 4) | (AttrIndx0 << 1) | (ENable))  // RegionNS = 0x1,
 AttrIndx0 = 0x0, ENable = 0x1
orr x1, x1, x2
msr PRLAR_EL2, x1
```

### Program EL2 MPU data region

Programming the EL2 MPU data region is very similar to programming the EL2 MPU code region. The differences in programming the regions are as follows:

- You are programming the second MPU region, so use PRBAR1_EL2 and PRLAR1_EL2.
- Instead of using the code region beginning and end addresses, you use the data addresses.
- Because the region is for data values, set PRBAR1_EL2 to read/write at EL2, EL1, and EL0.
- Associates PRLAR1_EL2 with MAIR_EL2.Attr1 instead of MAIR_EL2.Attr0.
- After programming and enabling the last MPU region, execute a `DSB SY` instruction to enable PMSAv8-64 for address translation when executing in EL2.

The following is the code to program the EL2 MPU data region:

```
// Region 1 - Data
ldr x1, =Image$$DATA$$Base  // Image$$DATA$$Base = Beginning address of the data
 area
ldr x2, =((Non_Shareable << 4) | (RW_this_EL_and_lower << 2))  // Non_Shareable
 = 0x0, RW_this_EL_and_lower = 0x1
orr x1, x1, x2
msr PRBAR1_EL2, x1

ldr x1, =Image$$ARM_LIB_STACK$$ZI$$Limit  // Image$$ARM_LIB_STACK$$ZI$$Limit = End
 address of the data area
sub x1, x1, #1
bfc x1, #0, #6
ldr x2, =((RegionNS << 4) | (AttrIndx1 << 1) | (ENable))  // RegionNS = 0x1,
 AttrIndx1 = 0x1, ENable = 0x1
orr x1, x1, x2
msr PRLAR1_EL2, x1
dsb sy
isb
```

### Configure the EL1 context

Now that the EL2 MPU programming is complete, to program the EL1 MPU, set up the EL1 context before jumping to EL1. If you do not want to execute in EL1, skip to the Enable the EL2 MPU and caches section of this guide.

To set up the EL1 context, perform the following steps:

1. Set the VMID for the EL1 context. In this guide, VMID is set to 0 by setting VSCLTR_EL2.VMID[15:8] and [7:0] to 0.

```
// Setting up EL1 context
```

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
// Initialize VMID for the EL1 context
msr VSCTLR_EL2, xzr
isb
```

2. Propagate PE identification information so EL1 can access it. MPIDR_EL1 and MIDR_EL1 contains this information. Propagate the information into VMPIDR_EL2 and VPIDR_EL2 respectively.

```
// Propagating MPIDR_EL1 and MIDR_EL1 to EL1
mrs x0, MPIDR_EL1
msr VMPIDR_EL2, x0
mrs x0, MIDR_EL1
msr VPIDR_EL2, x0
```

3. Configure EL1 and EL0 to use PMSAv8-64 by setting the VTCR_EL2.MSA bit[31] to `0b0`.

```
// Configuring EL1 in PMSA
mrs x1, VTCR_EL2
bic x1, x1, #(1U << 31)    //  VTCR_EL2.MSA [31]
msr VTCR_EL2, x1
```

   To avoid any unexpected behavior when performing the left shift, use an unsigned value when clearing bit[31].

4. To prevent trapping from self-hosted debug and the Performance Monitors Extension, set MDCR_EL2 to 0.

```
//Configure for uninterrupted execution in EL1
msr MDCR_EL2, xzr
```

5. Use CNTVOFF_EL2 to set the 64-bit virtual offset between the physical and virtual count values to 0.

```
// Allow predictable values from CNTVCT_EL0
msr CNTVOFF_EL2, xzr
```

   Setting CNTVOFF_EL2 to 0 grants predictability to the virtual count value in CNTVCT_EL0.

6. Set CNTFRQ_EL0 so software can discover the System Counter frequency. In this guide, CNTFRQ_EL0 is set to 50MHz.

```
// Set CNTFRQ_EL0 so software can discover the system counter frequency
ldr x2, =50000000  // Set to 50MHz
msr CNTFRQ_EL0, x2
```

   For advice on setting the System Counter frequency, read What is the frequency requirement for the System Counter?.

7. To avoid trapping accesses related to virtualization, set the following bits to `0b1` in HCR_EL2:
   - Do not trap accesses to SCXTNUM_EL1 and SCXTNUM_EL0 using EnSCXT[55]. These registers are associated with FEAT_CSV2_2 and FEAT_CVS2_1p2.
   - Do not trap accesses to ERXPFGCDN_EL1, ERXPRGCTL_EL1, and ERXPFGF_EL1 using FIEN[47]. These registers are associated with FEAT_RASv1p1.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Do not trap accesses when executing instructions related to Pointer Authentication using API[41]. This bit is associated with FEAT_PAuth.
- Do not trap accesses to registers holding “key” values for Pointer Authentication using APK[40]. This bit is associated with FEAT_PAuth.

```
// Avoid trapping accesses related to virtualization
mrs x2, HCR_EL2
orr x2, x2, #HCR_EL2_EnSCXT  // To allow accesses to SCXTNUM_EL1 and SCXTNUM_EL0
 for FEAT_CSV2_2 and FEAT_CVS2_1p2, HCR_EL2_EnSCXT = (1 << 53)
orr x2, x2, #HCR_EL2_FIEN    // To allow RAS in EL1, HCR_EL2_FIEN = (1 << 47)
orr x2, x2, #HCR_EL2_API     // To allow pointer authentication in EL1,
 HCR_EL2_API = (1 << 41)
orr x2, x2, #HCR_EL2_APK     // HCR_EL2_APK = (1 << 40)
msr HCR_EL2, x2
isb
```

## Enable the EL2 MPU and caches

Now that the MPU setup for EL2 is finished, turn on the MPU and caches. For this guide, to prevent exceptions because of unaligned accesses done by executed instructions, you also disable alignment checking. All these settings are in the SCTLR_EL2 register.

For this section, you use the following SCTLR_EL2 bits:

- If M[0] = `0b1`, enable MPU.
- If C[2] = `0b1`, enable the Instruction cache (I-cache).
- If I[12] = `0b1` , enable the Data cache (D-cache).
- If A[1] = `0b0`, no alignment checking on accesses done by executed instructions.

```
//------------------------------------------------------------------
// Enable EL2 MPU and caches
//------------------------------------------------------------------

mrs x1, SCTLR_EL2
orr x1, x1, #SCTLR_ELx_M  // SCTLR_ELx_M = (1 << 0)
orr x1, x1, #SCTLR_ELx_C  // SCTLR_ELx_C = (1 << 2)
orr x1, x1, #SCTLR_ELx_I  // SCTLR_ELx_I = (1 << 12)
bic x1, x1, #SCTLR_ELx_A  // Disable alignment fault checking.  To enable, change
 bic to orr, SCTLR_ELx_A = (1 << 1)
msr SCTLR_EL2, x1
isb
```

## EL2 configuration complete

Now the EL2 setup is complete, you have the choice of:

- Staying in EL2, calling the C library initialization code, and then running your application in EL2, or
- Jumping to EL1, configuring EL1, calling the C library initialization code, and then running your application code in EL1.

If you do not want to jump to EL1, call the C library initialization using `__main()`.

```
.global __main
```

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
b       __main
```

When the C library initialization is complete, you are ready to run application code in EL2. Unless you are interested in the content, do not read the remaining EL1 boot sections in this guide.

If you want to jump to EL1, continue reading.

### Jump to EL1

To continue the boot sequence by programming the EL1 MPU, you jump to EL1.

The following code jumps to EL1 by:

1. Storing the address of the first EL1 function in ELR_EL2.
2. Putting placeholder return information for EL2 in SPSR_EL2. You do not return to EL2 in these code examples.
    a. M[3:0] = `0b0101`, set Exception level and selected Stack Pointer to EL1.
    b. F[6] = `0b1`, enable FIQ interrupt mask.
    c. I[7] = `0b1`, enable IRQ interrupt mask.
    d. A[8] = `0b1`, enable SError exception mask.
3. Jump to EL1 using the `ERET` instruction.

```
    .global drop_to_el1
drop_to_el1:
    adr x1, el1_entry_aarch64
    msr ELR_EL2, x1
    mov x1, #(AARCH64_SPSR_EL1h | \  // AARCH64_SPSR_EL1h = 0x5
              AARCH64_SPSR_F  | \    // AARCH64_SPSR_F = (1 << 6)
              AARCH64_SPSR_I  | \    // AARCH64_SPSR_I = (1 << 7)
              AARCH64_SPSR_A)        // AARCH64_SPSR_A = (1 << 8)
    msr SPSR_EL2, x1
    eret
```

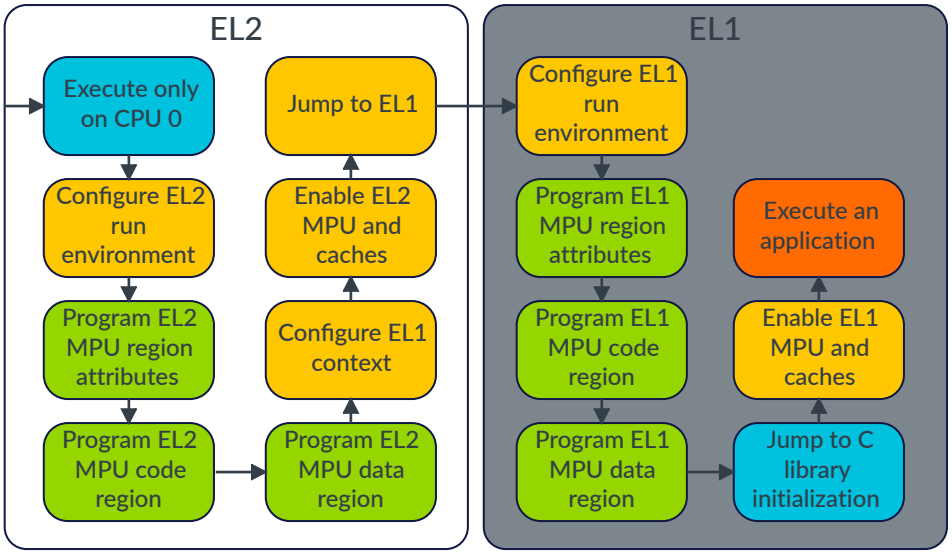
Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 8. EL1 boot steps

To show the boot process for EL1 with a focus on programming the MPU, this section uses short code examples.

The following diagram shows the high-level boot sequence for EL2 and EL1 documented in this guide:

**Figure 8-1: Diagram of boot sequence for EL1**




This section only covers the EL1 part of the boot sequence, highlighted in gray in the previous diagram.

Note

The code examples in this guide show a basic MPU setup. Other MPU settings and attributes are available.

Note

If a code example line uses a variable instead of a value, the value of the variable is included in the comments for that line of code.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

## Requirements

Before using the code in this section, you must use all the code in the previous section, EL2 boot steps.

To understand the boot sequence documented in this section, you must read the EL2 boot steps section first before reading this section.

## Configure the EL1 run environment

Now you are in EL1, set up EL1.



**Note**

In this section and throughout this guide, after writing to System registers, execute an `ISB` instruction to ensure that you have a synchronized context.

First, you setup the stack for EL1 and disable exception trapping for any floating-point instructions. Setting up the stack allows you to use the stack space when executing in EL1.

Disabling floating-point trapping prevents exceptions occurring because of executing floating-point instructions. In this case, trapping is disabled to prevent any called C library functions that include floating-point instructions from causing exceptions. To disabled floating-point trapping, set the CPACR_EL1.FPEN bits [21:20] to `0b11` to prevent trapping any floating-point instructions.

```
//
// Now we're in EL1, setup the application stack
// Allocate 2^14 (=0x4000) bytes for the application stack per core
// x19 contains the CPU number
//
ldr x0, =Image$$ARM_LIB_STACK$$ZI$$Limit  // Image$$ARM_LIB_STACK$$ZI$$Limit = End
 address of the stack area
sub x0, x0, x19, lsl #14
mov sp, x0

//
// Enable floating point
//
mov x0, #CPACR_EL1_FPEN  // CPACR_EL1_FPEN = (3 << 20)
msr CPACR_EL1, x0
isb
```

## Configure the EL1 MPU

Because these code examples uses the same code and data memory spaces for EL2 and EL1, the MPU programming for both Exception levels is very similar.

The differences between the EL2 and EL1 MPU programming sequence are as follows:

- EL1 uses the PRSELR_EL1, PRBAR_EL1, PRLAR_EL1, PRBAR1_EL1, PRLAR1_EL1, and MAIR_EL1 registers.
- For PRBAR_EL1, set the region access to read-only at EL1 or EL0, AP[2:1] = `0b11`.
- For PRBAR1_EL1, set the region access to read/write at EL1 or EL0, AP[2:1] = `0b01`.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Unlike PRBAR_EL2 and PRBAR1_EL2, PRBAR_EL1.XN and PRBAR1_EL1.XN is a single bit at bit [1]. EL2 and EL1 use the same XN[1] value.

To learn more about the MPU boot steps, refer to the following previous sections in this guide:

- Program EL2 MPU region attributes
- Program EL2 MPU code region
- Program EL2 MPU data region

The following code programs the EL1 MPU:

```
//-----------------------------------------------------------------
// EL1 MPU Configuration
//-----------------------------------------------------------------

// MAIR_EL1 configuration
mrs x0, MAIR_EL1
ldr x1, =0xBB                   // Normal inner/outer RW cacheable, write-through
bfi x0, x1, #0, #8              // Update Attr0
ldr x1, =0x04                   // Device nGnRnE
bfi x0, x1, #8, #8              // Update Attr1
msr MAIR_EL1, x0
isb

// Select first 16 MPU regions to program
ldr x1, =Region    // Region = 0x00
msr PRSELR_EL1, x1
isb


// Region 0 - Code
ldr x1, =Image$$CODE$$Base      // Image$$CODE$$Base = Beginning address of the code
 area
ldr x2, =((Non_Shareable << 4) | (RO_this_EL_and_lower << 2))  // Non_Shareable
 = 0x0, RO_this_EL_and_lower 0x3
orr x1, x1, x2
msr PRBAR_EL1, x1

ldr x1, =Image$$CODE$$Limit     // Image$$CODE$$Limit = End address of the code area
sub x1, x1, #1                  // convert limit from exclusive to inclusive
bfc x1, #0, #6                  // and clear the lower 6 bits
ldr x2, =((RegionNS <<4) | (AttrIndx0 << 1) | (ENable))   // RegionNS = 0x1,
 AttrIndx0 = 0x0, ENable = 0x1
orr x1, x1, x2
msr PRLAR_EL1, x1

// Region 1 - Data
ldr x1, =Image$$DATA$$Base  // Image$$DATA$$Base = Beginning address of the data
 area
ldr x2, =((Non_Shareable << 4) | (RW_this_EL_and_lower << 2))  // Non_Shareable
 = 0x0, RW_this_EL_and_lower 0x1
orr x1, x1, x2
msr PRBAR1_EL1, x1

ldr x1, =Image$$ARM_LIB_STACK$$ZI$$Limit  // Image$$ARM_LIB_STACK$$ZI$$Limit = End
 address of the data area
sub x1, x1, #1
bfc x1, #0, #6
ldr x2, =((RegionNS << 4) | (AttrIndx1 << 1) | (ENable))  // RegionNS= 0x1,
 AttrIndx1 = 0x1, ENable = 0x1
orr x1, x1, x2
msr PRLAR1_EL1, x1
dsb sy
```

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
isb
```

### Jump to C library initialization

Now, the EL1 MPU programming is complete. In this guide, instead of enabling the EL1 MPU and caches now, the next step is to call the `__main` function to initialize the C library. We enable the EL1 MPU and caches after the C library initialization finishes.

To start the C library initialization, call `__main()`.

```
.global __main
b       __main
```

### Enable the EL1 MPU and caches

Now the C library initialization is complete, enable the EL1 MPU and caches.

The same settings are used for enabling the EL1 MPU and caches as the EL2. The only difference is the EL1 enable uses the SCTLR_EL1 register instead of the SCTLR_EL2 register.

The following code enables the EL1 MPU and caches:

```
enable_mpu_and_caches:
    mrs x1, SCTLR_EL1          // Read System Control Register
    orr x1, x1, #SCTLR_ELx_M   // Set M bit to enable MPU, SCTLR_ELx_M = (1 << 0)
    orr x1, x1, #SCTLR_ELx_C   // Enable D-cache, SCTLR_ELx_C = (1 << 2)
    orr x1, x1, #SCTLR_ELx_I   // Enable I-cache, SCTLR_ELx_I = (1 << 12)
    bic x1, x1, #SCTLR_ELx_A   // Disable alignment fault checking.  To enable,
 change bic to orr, SCTLR_ELx_A = (1 << 1)
    msr SCTLR_EL1, x1          // Write System Control Register
    isb                        // Ensure subsequent insts execute wrt new MPU
 settings
    ret
```

### Execute an application

After enabling the EL1 MPU and caches, you are finished with the boot process and are ready to run application code.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 9. Related information

Here are some resources that relate to the material in this guide:

- Arm Architecture Reference Manual for A-profile architecture
- Arm Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile
- Arm Cortex-R82 Processor Technical Reference Manual

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# 10. Next steps

This section provides next steps you can perform now that you have finished reading this guide.

To learn about software for the Cortex-R82, read:

- Software Migration Guide from Armv8-R AArch32 to Armv8-R AArch64
- Arm Cortex-R82 Processor Software Optimization Guide

To learn about other aspects of Cortex-R82 boot, read the following:

Virtualization:

- Armv8-R virtualization
- Learn the architecture - AArch64 virtualization Guide

MMU:

- Learn the architecture - AArch64 memory management Guide

GIC:

- Arm Generic Interrupt Controller (GIC) Architecture Specification
- “Generic Interrupt Controller” section of Armv8-R virtualization
- Learn the architecture - Generic Interrupt Controller v3 and v4, Overview

Generic Timer: Learn the architecture - Generic Timer

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# Proprietary Notice

This document is protected by copyright and other related rights and the use 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 Limited ("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 the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

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.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

## Conformance notices

This section contains conformance notices.

### Federal Communications Commission Notice

This device is test equipment and consequently is exempt from part 15 of the FCC Rules under section 15.103 (c).

### Class A

Important: This is a Class A device. In residential areas, this device may cause radio interference. The user should take the necessary precautions, if appropriate.

### CE/UKCA Conformity

These marks indicate that this product meets all essential health, safety and environmental requirements. The CE mark indicates conformity within EU member states and the UKCA mark indicates conformity within the UK.

The Declarations of Conformity are available on request.



The *Waste Electrical and Electronic Equipment* (WEEE) marking, that is, the crossed out wheelie-bin figure, indicates that this product must not be disposed of with general waste within the European Union. To prevent possible harm to the environment from uncontrolled waste disposal, the user is required to recycle the product responsibly to promote reuse of material resources. To comply with EU law, you must dispose of the product in one of the following ways:

- Return it to the distributor where it was purchased. The distributor is required to arrange free collection when requested.
- Recycle it using local WEEE recycling facilities. These facilities are now very common and might provide free collection.
- If purchased directly from Arm, Arm provides free collection. Please e-mail weee@arm.com for instructions.
- End-of-Life products can be disposed of safely using an *Approved Authorized Treatment Facility* (AATF). To support safe disposal, Arm has partnered with B2B Compliance. B2B can be contacted at the following weblink: https://b2bcompliance.org.uk

During the lifetime of the product, you are advised to:

- Inspect the product regularly to ensure that it is in good working order.
- Ensure that the product is free from dust and debris that might cause damage.
- Clean the product with an air duster when necessary.
- Power down the system when not in use.
- Observe ESD precautions when handling the product.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The product can radiate Radio Frequency Interference (RFI) or Electromagnetic Interference (EMI) and might cause harmful interference to radio communications. There is no guarantee that interference cannot occur in a particular installation. If you suspect that this equipment is causing interference to other equipment, you are encouraged to try to correct the interference by one or more of the following measures:

- Ensure attached cables do not lie across any sensitive equipment.
- Increase the distance between the product and the receiver.
- Connect the equipment to an outlet on a circuit different from that to which the product is connected.
- Consult Arm for help.

The product can be sensitive to Radio Frequency Interference (RFI) or Electromagnetic Interference (EMI) which might cause incorrect operation of the product:

- Avoid using the product near sources of EMI.
- Never use the product in *Safety-Critical-Systems* (SCS), or *Life-Critical-Systems* (LCS).


Note

Arm recommends that, wherever possible, shielded interface cables be used.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

## Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

### Product completeness status

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

## Revision history

These sections can help you understand how the document has changed over time.

### Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

**Document history**

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0000-00 | 15 May 2025 | Non-Confidential | Initial release |

### Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in Document release information on page 36.

**Table 2: Issue 01-01**

| Change | Location |
|---|---|
| First Development release for r0p0 | - |

**Table 3: Issue 01**

| Change | Location |
|---|---|
| First non-confidential release for RelB | - |

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# Conventions

The following subsections describe conventions used in Arm documents.

## Glossary

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

See the Arm Glossary for more information: developer.arm.com/glossary.

## Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

| Convention | Use |
|---|---|
| *italic* | Citations. |
| **bold** | Interface elements, such as menu names.<br><br>Terms in descriptive lists, where appropriate. |
| `monospace` | Text that you can enter at the keyboard, such as commands, file and program names, and source code. |
| `monospace underline` | A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name. |
| `<and>` | Encloses replaceable terms for assembler syntax where they appear in code or code fragments.<br><br>For example:<br><br>`MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>` |
| SMALL CAPITALS | Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**. |




We recommend the following. If you do not follow these recommendations your system might not work.




Your system requires the following. If you do not follow these requirements your system will not work.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential




You are at risk of causing permanent damage to your system or your equipment, or harming yourself.




This information is important and needs your attention.




A useful tip that might make it easier, better or faster to perform a task.




A reminder of something important that relates to the information you are reading.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

# Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential