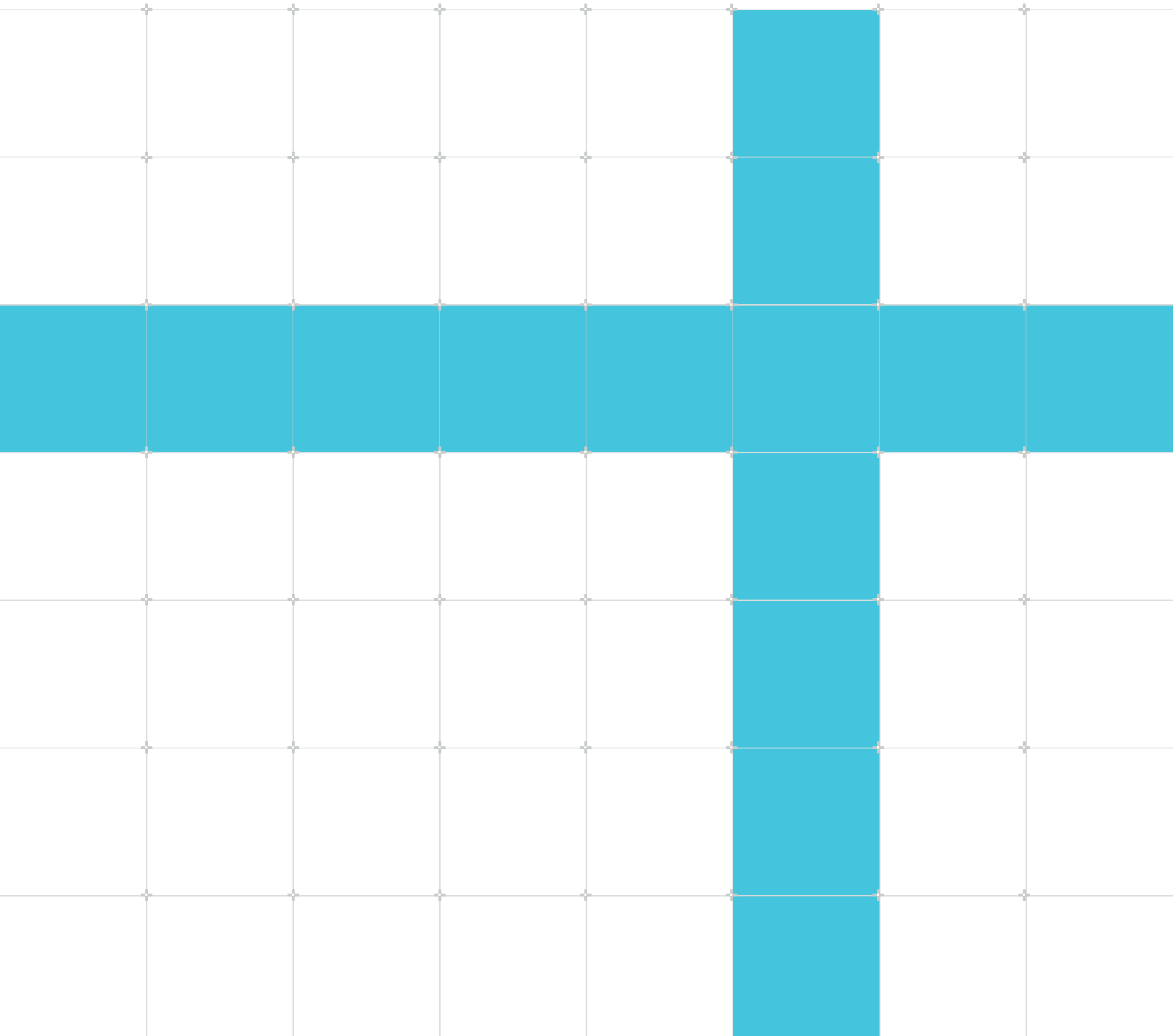
Cortex-R52 and Cortex-R52+ Programmer's Guide

Version 1.0

Issue 01
109997_100_01_en



Cortex-R52 and Cortex-R52+ Programmer's Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	4 June 2025	Non-Confidential	First release

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 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.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

This guide is intended for software developers who want to develop software or booting software for the Cortex-R52/R52+ processors. This document assumes that the developers are familiar with the developing for Armv7-R processors.

1.1 Useful resources

This document contains information that is specific to these products. See the following resources for other relevant information.

- Arm Non-Confidential documents are available on developer.arm.com/documentation. Each document link in the tables below provides direct access to the online version of the document.
- Arm Confidential documents are available to licensees only through the product package

Arm products	Document ID	Confidentiality
Arm Cortex-R52 Processor Technical Reference Manual	100026	Non-Confidential
Arm Cortex-R52+ Processor Technical Reference Manual	102199	Non-Confidential
ARM Cortex-R Series Programmer's Guide	DEN0042A	Non-Confidential

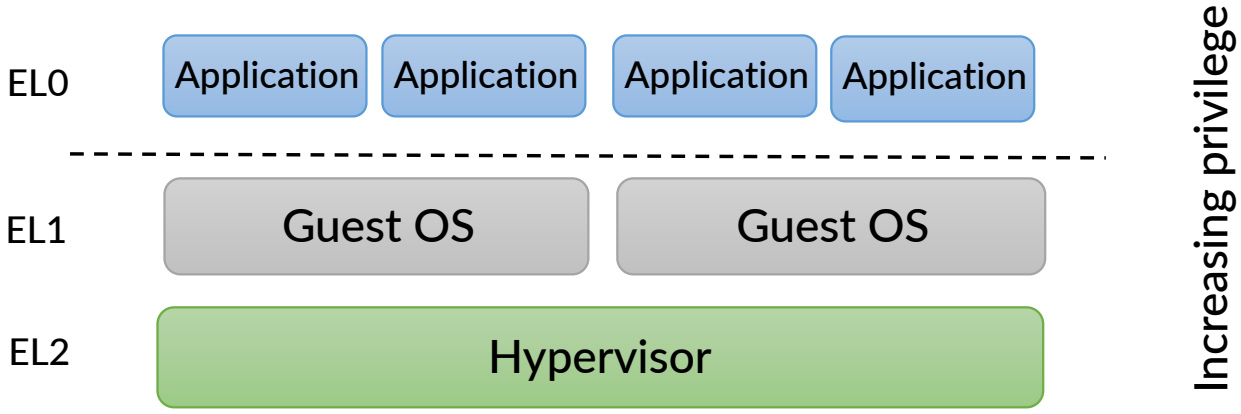
Arm architecture and specifications	Document ID	Confidentiality
Arm Architecture Reference Manual for A-profile architecture	DDI0487K_a	Non-Confidential
Armv8-R AArch32 Supplement	DDI0568A_c	Non-Confidential

2. Arm Architecture and Processors

The Armv8 architecture defines two Execution states, AArch64 and AArch32. Each state describes execution using 64-bit wide general-purpose registers or 32-bit wide general-purpose registers, respectively. Generally, the Armv8 AArch32 state retains the ARMv7 definitions of privilege. When in AArch32 state, the processor can execute either the A32 (ARM) or the T32 (Thumb) instruction set.

Cortex-R52/R52+ are Armv8-R AArch32 processors, so the ARM Cortex-R Series Programmer guide, which complies with the Armv7-R architecture, is also a good reference guide for developers. The following Figure 2-1: Armv8-R AArch32 Exception Levels on page 8 shows the organization of Armv8-R Exception levels in AArch32 state.

Figure 2-1: Armv8-R AArch32 Exception Levels



Unlike the Armv8-A processors, Armv8-R processors only support a single Secure state. For Cortex-R52/R52+ processors, the single Secure state is Non-secure state. Cortex-R52/R52+ do not implement the EL3. Armv7 architecture and Armv8 architecture define three separate profiles as the following Table 2-1: Armv8 Architecture Profiles on page 8 shows. These are variants of architectures describing processors targeting different markets and users.

Table 2-1: Armv8 Architecture Profiles

Profile	Description
A	The A-profile architecture processors provide solutions for devices undertaking complex computational tasks, such as hosting a rich Operating System (OS) platform, and supporting multiple software applications.
R	The R-profile architecture processors deliver fast and deterministic processing, while meeting challenging real-time constraints in a range of situations. They combine these features in a Performance, Power and Area (PPA) optimized package, making them the trusted choice in reliable systems demanding high error-resistance.
M	The M-profile architecture processors are optimized for cost and energy-efficient microcontrollers. These processors are found in a various of applications, including IoT, industrial and everyday consumer devices. The processor family is based on the Microcontroller profile architecture that provides low-latency and a highly deterministic operation, for deeply embedded systems.

Cortex-R52 is the first 32-bit Armv8-R processor and Cortex-R52+ is the second 32-bit Armv8-R processor. They both comply the Arm Architecture Reference Manual for A-profile architecture and ARM Architecture Reference Manual Supplement - ARMv8, for the ARMv8-R AArch32 architecture profile. Cortex-R52/Cortex-R52+ are the next generation of Cortex-R4 and Cortex-R5, which are ARM7-R processors. The following Table 2-2: Cortex-R family comparison on page 9 shows the difference between the cores.

Table 2-2: Cortex-R family comparison

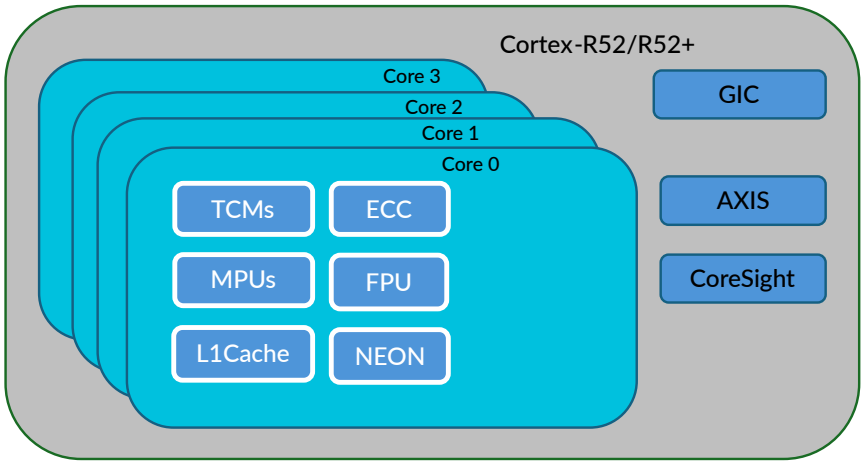
Feature	Cortex-R4	Cortex-R5	Cortex-R52	Cortex-R52+
Architecture	Armv7-R	Armv7-R	Armv8-R AArch32	Armv8-R AArch32
Instruction Set	A32(ARM), T32(Thumb)	A32(ARM), T32(Thumb)	A32(ARM), T32(Thumb)	A32(ARM), T32(Thumb)
Pipeline Depth	8 stage, in-order, dual issue	8 stage, in-order, dual issue	8 stage, in-order, superscalar	8 stage, in-order, superscalar
Address Bits	32	32	32	32
Addressable Memory	4GB	4GB	4GB	4GB
ECC on Memories	Optional	Optional	Optional	Optional
MPU or MMU	MPU	MPU	MPU	MPU
Maximum MPU Regions	12	16	24+24	24+24
Multi-core Support	1 core, No coherency	2 core, IO coherency	up to UP4, no coherency	up to UP4, no coherency
Floating-Point Unit (FPU)	Optional	Optional	Optional	Optional
Advanced SIMD (NEON)	No	No	Optional	Optional
Maximum External Interrupts	Up to 480(GIC)	Up to 480(GIC)	Up to 960	Up to 960
Bus Protocol	AXI3	AXI3	AXI4	AXI4
Dual Core Lock-Step	Optional	Optional	Optional	Optional
Safety Package	No	Yes	Yes	Yes
Software Test Library	No	Yes	Yes	Yes
Virtualization	No	No	Yes	Yes

Cortex-R52/R52+ are mid-performance, in-order, superscalar processors primarily for use in automotive and industrial applications. They are also suited to a wide variety of other embedded applications such as communication and storage devices.

Cortex-R52+ is software compatible to Cortex-R52. Compared to Cortex-R52, Cortex-R52+ provides improved configurability for real-time applications with function safety requirement. It is transparent from the software view. So in this guide Cortex-R52 and Cortex-R52+ are referred to as Cortex-R52/R52+ for convenience.

As the following Figure 2-2: Cortex-R52/R52+ Structure on page 10 shows, Cortex-R52/R52+ support up to 4 cores, each with an 8-stage in-order, superscalar pipeline with branch prediction. There is no cache coherency logic implemented in these cores. So, Cortex-R52/R52+ do not support the Symmetric Multi-Processing (SMP) if the shared data is cacheable. Developers should keep the data coherency with cache maintenance instruction. And Cortex-R52/R52+ also do not support the synchronization between cores if no external global exclusive monitor is implemented.

Figure 2-2: Cortex-R52/R52+ Structure



Each core supports floating-point instructions, optional TCMs, Harvard Caches, Advanced SIMD (NEON) and bus interfaces, including AXI-M, LLPP and flash interfaces. These cores share the GIC and AXIS interface. To understand how the Cortex-R52/R52+ implements the Armv8-R AArch32 architecture, see Arm Cortex-R52 Processor Technical Reference Manual (TRM) and Arm Cortex-R52+ Processor Technical Reference Manual.

- The exception return address is saved in ELR register not LR_mode register.
- The ELR register value does not need the adjustment.
- The exception return instruction is ERET.
- There is only one exception mode (Hyp mode) for exceptions handling at EL2.
- EL2 vector table has a standalone entry for Hypervisor trap or Hyp mode entry.
- The exception informations are kept in HSR/HDFSR/HIFAR/HDFAR.

3. Instructions

Cortex-R52/R52+ are Armv8-R AArch32 processors, which use the AArch32 Execution state. Cortex-R52/R52+ support the A32(ARM) and T32(Thumb) instruction set. The following Table 3-1: Instruction update on page 11 lists the new or redefined instructions for Armv8-R AArch32 processors compared to Armv8-A instructions.

Table 3-1: Instruction update

Instruction	Respect to Armv8-A	Note
DMB	Redefined	The ordering requirements are relaxed compared to the Armv8-A behavior of DMB
DSB	Redefined	The ordering requirements are relaxed compared to the Armv8-A behavior of DSB
DFB	New	If executed at EL2, the instruction orders memory accesses irrespective of the Exception level or associated VIMD. If executed at EL1 or EL0, the instruction behaves as DSB SY

The following Table 3-2: Armclang options to generate ISA on page 11 shows how developer can use the Armclang compiler to generate the basic instructions. But the instructions of DMB, DSB and DFB must be written manually.

Table 3-2: Armclang options to generate ISA

Options	Description
–target=arm-arm-none-eabi	Generate A32/T32 instructions for AArch32 state
-mcpu=cortex-r52/cortex-r52plus -march=armv8-r	Always use -march or -mcpu when compiling with –target=arm-arm-none-eabi
-mthumb	Generate T32 instructions for Thumb state
-marm	Generate A32 instructions for AArch32 state

3.1 Advanced SIMD and floating-point support

The Cortex-R52/R52+ processors support single-precision floating-point instructions and can optionally support the double-precision floating-point and Advanced SIMD instructions. Refer to the following Table 3-3: MVFR0 register Specification on page 11, the register MVFR0 specifies the features of Advanced SIMD and floating-point instruction support. Cortex-R52/R52+ do not implement all the architectural combinations of Advanced SIMD and floating-point support.

Table 3-3: MVFR0 register Specification

Register field	Description
MVFR0.FPDP	`0x0`: No double-precision support `0x2`: Double-precision floating-point and Advanced SIMD technology support
MVFR0.FPSP	`0x2`: Single-precision floating-point support

The following Table 3-4: Armclang options to generate floating-point and Advanced SIMD instructions on page 12 shows developers could use compiler options to generate the floating-point instructions or Advanced SIMD instructions.

Table 3-4: Armclang options to generate floating-point and Advanced SIMD instructions

Options	Description
-mfpu=fp-armv8 -mcpu=cortex-r52+fp -mcpu=cortex-r52plus+fp	Generate floating-point instructions
-mfpu=neon-fp-armv8 -mcpu=cortex-r52+fp +simd -mcpu=cortex-r52plus+fp+simd	Generate floating-point instructions and Advanced-SIMD instructions
-mfloat-abi=soft	Software library functions for floating-point operations and software floating-point linkage The -mfpu option doesn't have an effect if -mfloat-abi is specified
-mcpu-cortex-r52/cortex-r52plus+nofp +nosimd	Prevent the compiler generating the floating-point instructions or Advanced-SIMD instructions

3.2 Reigsters

Similar to Armv7-A/R processors, Cortex-R52/R52+ also have 16x32-bit general-purpose registers. The register definition follows the AAPCS-AArch32 requirement as the following Table 3-5: General register define for AAPCS-AArch32 on page 12 shows.

Table 3-5: General register define for AAPCS-AArch32

R0-R3	R4-R12	R13-R14	R15
Parameter and result registers	R4-R11 (Callee-saved Registersregisters) Corruptible register (R12)	R13(SP) R14(LR)	PC

The number of Advanced SIMD and floating-point registers depends on the configuration parameter, and it is indicated in the register MVFR0.SIMDReg as shown in the following Table 3-6: MVFR0 for the Advanced SIMD implementation on page 12.

Table 3-6: MVFR0 for the Advanced SIMD implementation

NVFR0.SIMDReg	Description
0x1	16x64-bit registers
0x2	32x64-bit registers

3.3 Coprocessor

Backwards compatible with ARMv7-R processors, Cortex-R52/R52+ implement some Coprocessors as the following Table 3-7: Coprocessor implementation on page 12 shows

Table 3-7: Coprocessor implementation

Coprocessor		Example
CP10&CP11	Used together, for control and configuration of the floating-point and Advanced-SIMD architecture extension	CPACR.{cp10,cp11} defines the access right for the Advanced SIMD and floating-point features
CP14	Access debug and trace registers	MRC p14, 0, R0, c0, c0, 0; read DBGDIDR into R0
CP15	Access system control registers	MRC p15, 0, R0, c0, c0, 0; read MIDR into R0

See the ARM Cortex-R Series Programmer's Guide Section 5.8.1 for a brief introduction to the Coprocessor instructions.

4. Exceptions

This section is a supplement to the topic of exceptions. It covers the PE mode and exception vector table.

The ARM Cortex-R Series Programmer's Guide section Exception and Interrupts introduces the exception type, exception priorities and exception handing.

4.1 processing Element(PE) mode

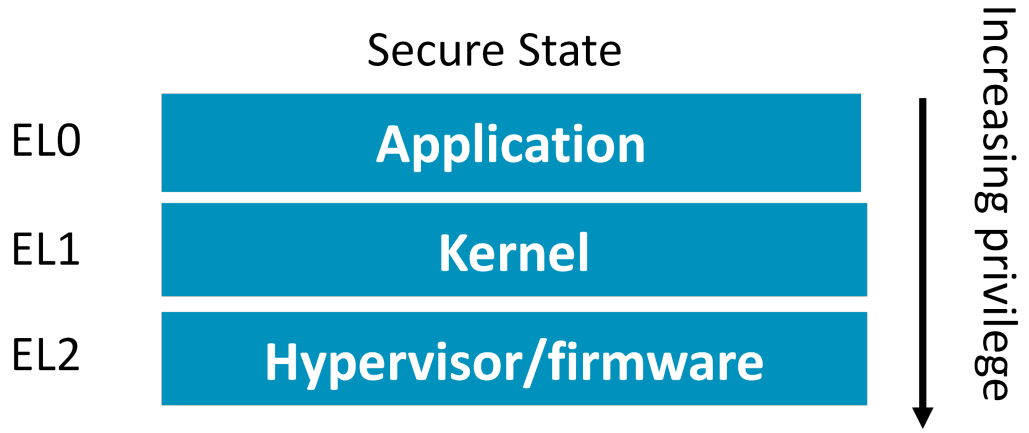
Armv8-R AArch32 PE mode is similar to the Armv8-A AArch32 with Non-secure state. The following Table 4-1: Mode and Exception level on page 14 shows 3 Exception levels and 8 modes. The PE modes are backwards compatible with Armv7-R Exception Mode.

Table 4-1: Mode and Exception level

Mode	Description	Exception level
Hyp	Entered on reset or when a Hypervisor Call instruction (HVC) is executed, used for virtualization	EL2
Supervisor(SVC)	Entered when a Supervisor Call instruction (SVC) is executed	EL1
FIQ	Entered when a high priority (fast) interrupt is raised	EL1
IRQ	Entered when a normal priority interrupt is raised	EL1
Abort	Handles memory access violations	EL1
Undefined	Handles undefined instructions	EL1
System	Privileged mode using the same registers as User mode	EL1
User	Mode under which most tasks run	EL0

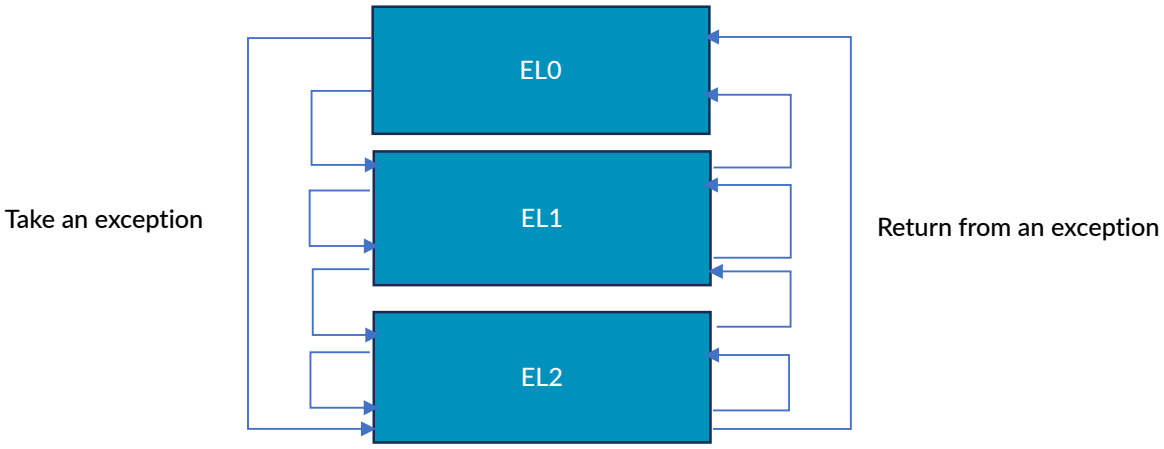
Cortex-R52/R52+ implement the Armv8-R AArch32 state at all Exception levels (EL0-EL2). As shown in the Figure 4-1: Exception level privilege on page 15, EL0 is the least privileged Exception level, typically used for application space. EL2 is the most privileged Exception level, typically used for Hypervisor or firmware space. EL1 is typically used for kernel space running RTOS.

Figure 4-1: Exception level privilege



The Figure 4-2: Exception level switch on page 15 shows that the Exception level can only be changed when taking or returning from an exception. On taking an exception, the Exception level can stay the same or increase. On returning from an exception, the Exception level can stay the same or decrease.

Figure 4-2: Exception level switch



4.2 Exception vector table

Exception handling on the core is controlled by using an area of memory. On Cortex-R52/R52+ processors there are two vector tables, which is different with Armv7-R vector table. One is for EL0/EL1 exception handling, and the another is for EL2 exception handling. In some software implementations each guest OS has its own EL1 vector table. Both vector tables are managed by the hypervisor.

The formats of the 2 tables are similar. Each vector table has 8 entries, as the following Table 4-2: EL1/EL2 Vector table on page 16 shows. EL1 vector table offset `0x14` entry is reserved. EL2 vector table offset `0x14` entry is for Hypervisor trap or Hypervisor mode entry. Each entry has 4 bytes which consist of instructions not the address. It means there is one 32-bit instruction or two 16-bit instructions, usually direct branch instruction or LDR instruction to load the exception handler address to the PC register. Do not use the BL or BLX instructions on EL1 Vector table entry because it will corrupt the LR register value.

Table 4-2: EL1/EL2 Vector table

Offset	EL1 vector table	EL2 vector table
`0x1C`	FIQ	FIQ
`0x18`	IRQ	IRQ
`0x14`	Reserved	Hyp trap, or Hyp mode entry
`0x10`	Data Abort	Data Abort, from Hyp mode
`0x0C`	Prefetch Abort	Prefetch Abort, from Hyp mode
`0x08`	Supervisor Call	Hypervisor Call, from Hyp mode
`0x04`	Undefined instruction	Undefined instruction, from Hyp mode
`0x00`	(Reset)	Reset

When the Core is out of reset, the EL2 vector table base address is set by the input signals CFGVECTABLEx[31:5]. There are 3 vector base address registers as the following Table 4-3: Vector base address registers on page 16 shows.

Table 4-3: Vector base address registers

Register	Description
RVBAR	Read-only and reflects the value of CFGVECTABLEx[31:5]
HVBAR	The vector base address for EL2 exception handing, reset value is determined by signal CFGVECTABLEx[31:5]
VBAR	The vector base address for EL1 exception handing, reset value is `0x0`

Developers can program the HVBAR or VBAR to set the vector table base address during the initialization or runtime at EL1 or EL2. Cortex-R52/R52+ EL1 exception handling remains the same as the Armv7-R Exception handling. The difference of EL2 exception handing include:

- The exception return address is saved in ELR register not LR_mode register.
- The ELR register value does not need the adjustment.
- The exception return instruction is ERET.
- There is only one exception mode (Hyp mode) for exceptions handling at EL2.
- EL2 vector table has a standalone entry for Hypervisor trap or Hyp mode entry.
- The exception informations are kept in HSR/HDFSR/HIFAR/HDFAR

5. Cache and TCM

Cache and TCM are two important components of the Cortex-R52/R52+ processor. The cache is a small, fast memory that stores frequently used data and instructions. TCM is a type of memory closely integrated with the CPU to provide fast and predictable access to critical data. The cache and TCM are designed to improve the performance of the processor by reducing the latency of accessing data and instructions. The cache and TCM are designed to work together to provide a high-speed, low-latency memory system for the processor.

The Caches and Tightly Coupled Memory chapters in the Cortex-R Series Programmer's Guide describe how the cache and TCM works. Cortex-R52/R52+ optionally implement the instruction and data cache in the core. The cache size can be configured during the SoC integration. You can get the cache implementation information such as cache level number, cache line size, sets number and way number from the registers CTR, CLIDR and CCSIDR. For example,

```
MRC p15,0,<Rt>,c0,c0,1 ; Read CTR into Rt
MRC p15,1,<Rt>,c0,c0,1 ; Read CLIDR into Rt
MRC p15, 1, <Rt>, c0, c0, 0 ; Read CCSIDR into Rt
```

As the following Table 5-1: Cache control registers on page 17 shows, developers can enable or disable the cache by programming the SCTLR or HSCTLR for EL0/EL1 and EL2 Normal memory access.

Table 5-1: Cache control registers

Register	Description
SCTLR.C	0: Data cache disable for EL0/EL1 memory access. 1: Data cache enable for EL0/EL1 memory access
SCTLR.I	0: Instruction cache disable for EL0/EL1 memory access. 1: Instruction cache enable for EL0/EL1 memory access
HSCTLR.C	0: Data cache disable for EL2 memory access. 1: Data cache enable for EL2 memory access
HSCTLR.I	0: Instruction cache disable for EL2 memory access. 1: Instruction cache enable for EL2 memory access

Cortex-R52/R52+ processors do not implement the cache coherency logic between the cores in a cluster. Cortex-R52/R52+ processors do not cache data that is marked as sharable. All cache maintenance instructions are performed locally, which means the cache maintenance operations are not broadcast to any other core. The write behavior for the data cache is always write-through whatever the region attribute is write-back or write-through.

L1 instruction and data caches segregate allocations from the Flash or AXIM interface into separate cache ways. Developers can set IMP_CSCTLR.IFLW and IMP_CSCTLR.DFLW to control the number of ways that are allocated to each interface.

Developers can dump the content of L1 instruction and L1 data cache through Cortex-R52/R52+ specific system registers. See Direct access to internal memory of the Cortex-R52+ TRM. Generally, external debugger or RTOS self-hosted debugger can provide the feature to dump the cache content, including the data RAM and tag RAM.

Each core in Cortex-R52/R52+ has unified TCMs, which provide fast access and have the deterministic memory access timing. This is important for the real-time applications. TCM

size depends on the implementation. The base address and enablement of each TCM can be programmed by setting the TCM region registers, including IMP_ATCMREGIONR, IMP_BTCMREGIONR and IMP_CTCMREGIONR. They can be enabled separately for software running at EL2 and software running at EL1 or EL0. The Figure 5-1: TCMs region register on page 18 shows the IMP_ATCMREGIONR, IMP_ATCMREGIONR, and IMP_ATCMREGIONR bit assignments.

Figure 5-1: TCMs region register

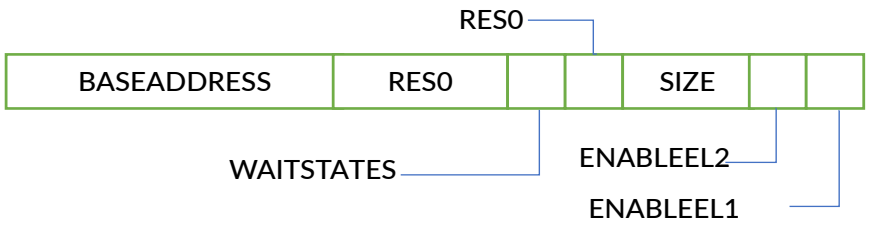


Table 5-2: TCMs region register bit assignments

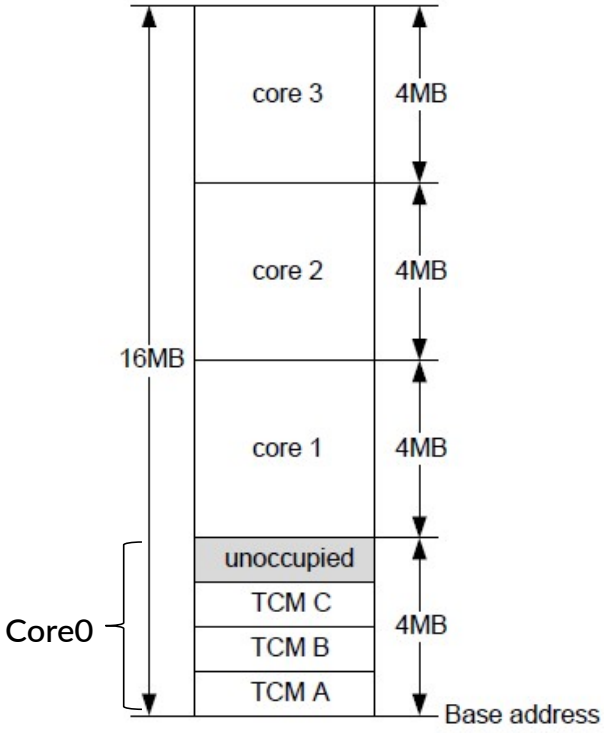
Bits	Name	Description
[31:13]	BASEADDRESS	TCM base address
[12:9]	-	Reserved as 0
[8]	WAITSTATES	Indicate if the TCM is implemented with the optional wait state
[7]	-	Reserved as 0
[6:2]	SIZE	TCM size
[1]	ENABLEEL2	Enable TCM at EL2
[0]	ENABLEEL1	Enable TCM at EL1 and EL0

The TCM base address must be size-aligned and developers can change the TCM base address during the initialization or runtime. Ensure that running code is not in the TCM and does not overlap with the TCM address space after changing. The TCM size depends on the hardware IMPLEMENTATION, read-only. TCM has a higher priority if the TCM region is overlapped with another memory region.

If virtualization is enabled, Hypervisor can manage the Virtual Machine's TCM setting. Hypervisor can trap the write access to TCM region registers by setting the register bit HCR.TIDCP as 1.

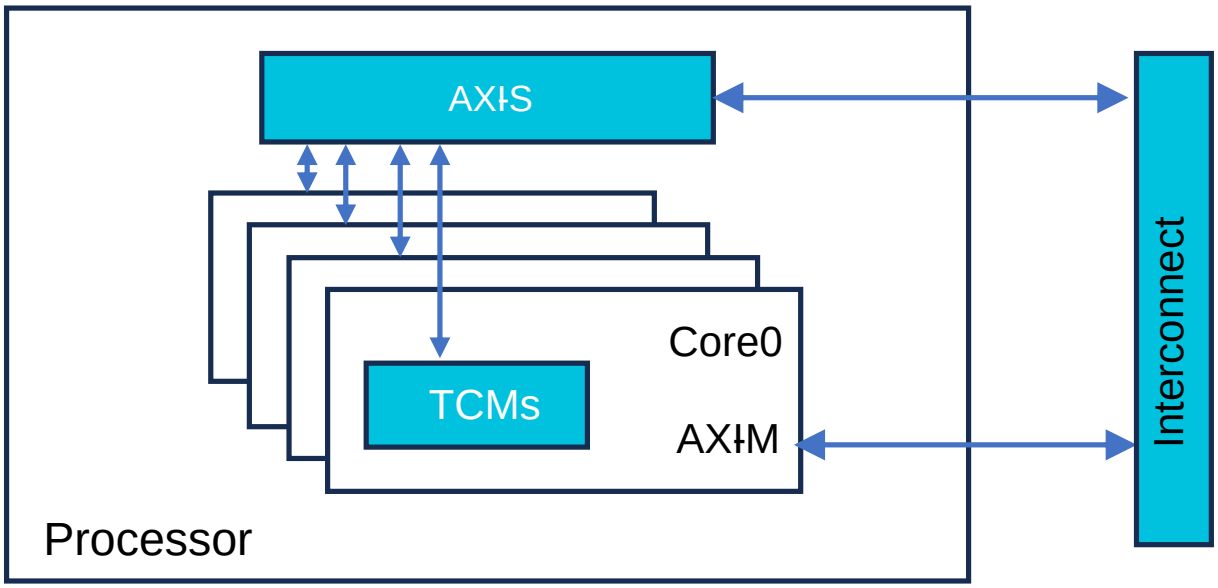
If the Core is booting from the TCM, the image should be prepared in the TCM in advance by external debugger or DMA. The processor TCMs can be accessed through the AXIS interface. The Figure 5-2: AXIS TCM mapping on page 19 shows how each TCM is mapped into AXIS address space.

Figure 5-2: AXIS TCM mapping



The AXIS interface can access all the TCMs on each core, as the Figure 5-3: Example TCM access via AXIS on page 20 shows. Each core TCM region size is 4MB. The AXIS base address is configured by input signals CFGAXISTCMBASEADDR[31:24], which must match the address of AXIS port in the system memory map.

Figure 5-3: Example TCM access via AXIS



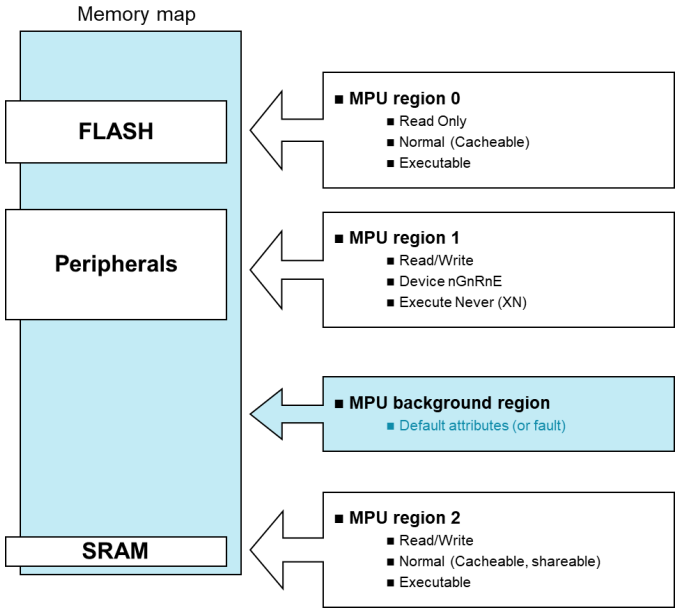
TCM is always accessed as Non-cacheable Non-shareable Normal memory, and ignores the memory attribute of the TCM memory region in the MPU. Developers can set TCM access permission on the TCM memory region in the MPU.

6. The Memory Protection Unit

Many real-time systems operate with a multitasking Operating System (OS). The OS provides a facility to ensure that the task currently executing does not disrupt other tasks. System resources, the code, and data of other tasks are protected. The protection system typically relies on both hardware and software.

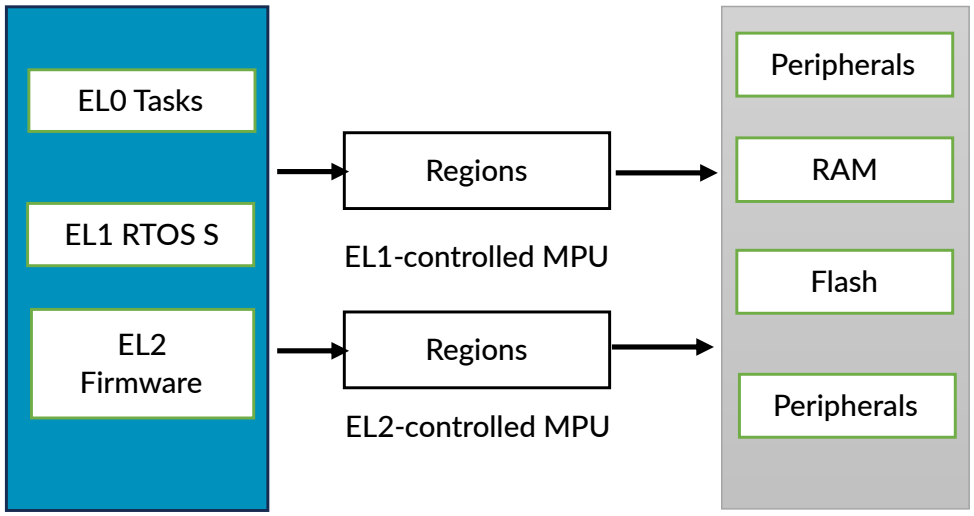
Cortex-R52/R52+ processors implement the Physical Memory System Architecture (PMSA), providing a Memory Protection Unit (MPU) to control memory protection. The MPU uses registers to define regions with associated properties such as read/write permissions and cache policies. When the processor accesses a memory location, the MPU checks the access against the properties for the region the address is in. This model is known as the PMSA. The Figure 6-1: Example MPU system mapping on page 21 shows a simple set of regions that can be defined in the MPU.

Figure 6-1: Example MPU system mapping



The Figure 6-2: Armv8-R PMSA on page 22 shows the PMSA architecture if virtualization is not enabled. Virtualization will be explained in Chapter 8.

Figure 6-2: Armv8-R PMSA



Each processor core has an EL1-controlled MPU with 16, 20 or 24 programmable regions, and an EL2-controlled MPU which optionally supports 0, 16, or 24 programmable regions. A region is a contiguous range of addresses starting at a base address, extending up to and including a limit address. The region can not overlap other regions. The region has a minimum size of 64 bytes and a maximum size of the entire address map, 4GB.

ARMv7- R processors differ in that a region is defined by its base address and size. The region can overlap other regions. The region's base address must always be aligned to its size. The region size can be any power of two between 32 Bytes and 4 GB.

For EL1-controlled MPU regions, the base address is configured by the register PRBAR and the limit address is configured by the register PRLAR. For EL2-controlled MPU regions, the base address is configured by the register HPRBAR and the limit address is configured by the register HPRLAR. The base address is aligned on a 64-byte boundary and the limit address is aligned to the byte below a 64-byte boundary. The minimum size for a region is 64 bytes. Developer can program the PRBAR/PRLAR and HHPBAR/HPRLAR register by selecting the region n registers via PRSELR/HPRSELF. For example, set the region 0 PRBAR/PRLAR registers:

```
MOV R0, #0     // Select region 0 register
MCR p15, 0, R0, c6, c2, 1 //Write R0 to PRSELR register
MCR p15, 0, <Rt>, c6, c3, 0 // Write Rt to PRBAR register
MCR p15, 0, <Rt>, c6, c3, 1 //Write Rt to PRLAR register
```

Developer also can program the MPU region n directly via access PRBARn/PRLARn or HPRBARn/HPRLARn registers. For example, read/write PRBAR0/PRLAR0.

```
MCR p15, 0, <Rt>, c6, c8+n[3:1], 4 *n[0]  //Read PRBARn into Rt
MRC p15, 0, <Rt>, c6, c8+n[3:1], 4 *n[0]  //Write Rt into PRBARn
MCR p15, 0, <Rt>, c6, c8+n[3:1], 4 *n[0]+1 //Read PRLARn into Rt
```

```
MRC p15, 0, <Rt>, c6, c8+n[3:1], 4 *n[0]+1  //Write Rt into PRLARn
```

For EL1-controlled MPU regions, PRBAR and PRLAR define

- The access permission (PRBAR.AP)
- The shareability (PRBAR.SH)
- The Executed-never bit (PRBAR.XN)
- The memory attribute index (HPRLAR.AttrIndex)

The memory attributes are determined by indexing the HMAIRx with HPRLAR.AttrIndx.
The memory region can be enabled or disabled by setting or clearing the region enable bit (HPRLAR.EN). It also can be enabled or disabled by setting the HPRENR register. The related MPU registers are listed as the following Table 6-1: MPU registers on page 23 shows.

Table 6-1: MPU registers

EL1-controlled MPU	EL2-controlled MPU	Description
PRBAR.BASE	HPRBAR.BASE	Region base address
PRLAR.LIMIT	HPRLAR.LIMIT	Region limit address
PRBAR.AP	HPRBAR.AP	Access permission
PRBAR.SH	HPRBAR.SH	Shareability
PRBAR.XN	HPRBAR.XN	Executed-never
PRLAR.AttrIndex	HPRLAR.AttrIndex	Memory attribute index of MAIRx
PRLAR.EN	HPRLAR.EN	Region enable or disable
MAIRx	HMAIRx	Memory attribute encodings
MPUIR	HMPUIR	Region information

When developers want to set up or reprogram a region, ensure that the programming code is not within this region. Execute the DSB instruction before making changes to ensure all memory accesses are completed. Execute the ISB instruction to flush the pipeline and guarantee that the changes take effect.

6.1 Default Memory Map

For Armv8-R AArch32 architecture, each PMSAv8-32 MPU has an associated default memory map which is used when the MPU is not enabled (SCTLR.M=0 or HSCRLR.M=0). When the MPU is enabled and Background region checking is enabled, privileged access that does not hit defined protection regions undergoes a second check:

- For the EL1 MPU, Background region checking is enabled for privileged access when the value of SCTLR.BR is 1.
- For the EL2 MPU, Background region checking is enabled for accesses from EL2 when the value of HSCTLR.BR is 1.

- When the EL2-controlled MPU is enabled, accesses from the EL0/EL1 translation regime that do not hit in the EL2 programmable regions generate a translation fault. This is a result of two-stage translation.

6.2 EL1/2-controlled MPU background region

The following Table 6-2: EL1/EL2-controlled MPU background region for instruction access on page 24 shows the Cortex-R52/R52+ EL1/EL2-controlled MPU background region for instruction access.

Table 6-2: EL1/EL2-controlled MPU background region for instruction access

Address range	Instruction caching enabled (SCTLR.I=1)	Instruction caching disabled (SCTLR.I=0)	Execute-never(XN)
0x00000000-0x7FFFFFFF	Normal, Non-shareable, Write-Through Cacheable	Normal, Shareable, Non-cacheable	Execution permitted
0x80000000-0xFFFFFFFF	-	-	Execute-never

The following Table 6-3: EL1/EL2-controlled MPU background region for data access on page 24 shows the Cortex-R52/R52+ EL1/2-Controlled MPU background region for data access.

Table 6-3: EL1/EL2-controlled MPU background region for data access

Address range	Data caching enabled (SCTLR.C=1)	Data caching disabled (SCTLR.C=0)
0x00000000-0x3FFFFFFF	Normal, Non-shareable, Write-Back, Write-Allocate Cacheable	Normal, Shareable, Non-cacheable
0x40000000-0x5FFFFFFF	Normal, Non-shareable, Write-Through Cacheable	Normal, Shareable, Non-cacheable
0x60000000-0x7FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Shareable, Non-cacheable
0x80000000-0xBFFFFFFF	Device-nGnRE	Device-nGnRE
0xC0000000-0xFFFFFFFF	Device-nGnRnE	Device-nGnRnE

6.3 Default cacheability

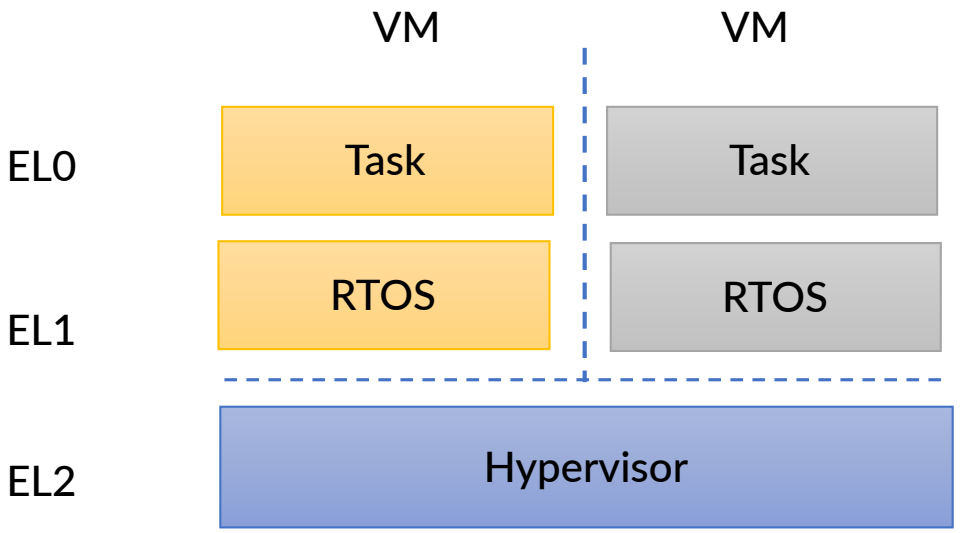
When default cacheability is enabled (HCR.DC=1), transactions using the EL1-controlled MPU background region have Normal, Inner Write-Back, Outer Write-Back, Non-shareable attributes applied with both Read-Allocate and Write-Allocate hints enabled. Instruction accesses that hit in the background region when HCR.DC=1 are always executable.

The default attributes are the most permissive, meaning that when combined with any attribute from the EL2-controlled MPU the resulting attribute is the same as the EL2-controlled MPU attribute. This allows the EL2-controlled MPU to effectively make the EL1-controlled MPU transparent to transactions from the EL1 translation regime that hit in the background region. When HCR.DC=1, all translations from the EL0/EL1 translation regime perform a two-stage MPU look up and the processor behavior as if HCR.VM is set.

7. Virtualization

Virtualization is a technology that allows a single hardware machine to run different OSs. Virtualization is possible on Armv8-R processor because code can run at EI2 above the guest OS EI1. This privilege level enables a hypervisor to sit above one or more guest OSs. A hypervisor is a complex of software running at EL2(Hyp mode) responsible for creating, managing and scheduling of Virtual Machines. The Figure 7-1: Virtualization example on page 25 shows a Hypervisor running at Exception level 2 to allow 2 VMs to run at EL0/EL1:

Figure 7-1: Virtualization example



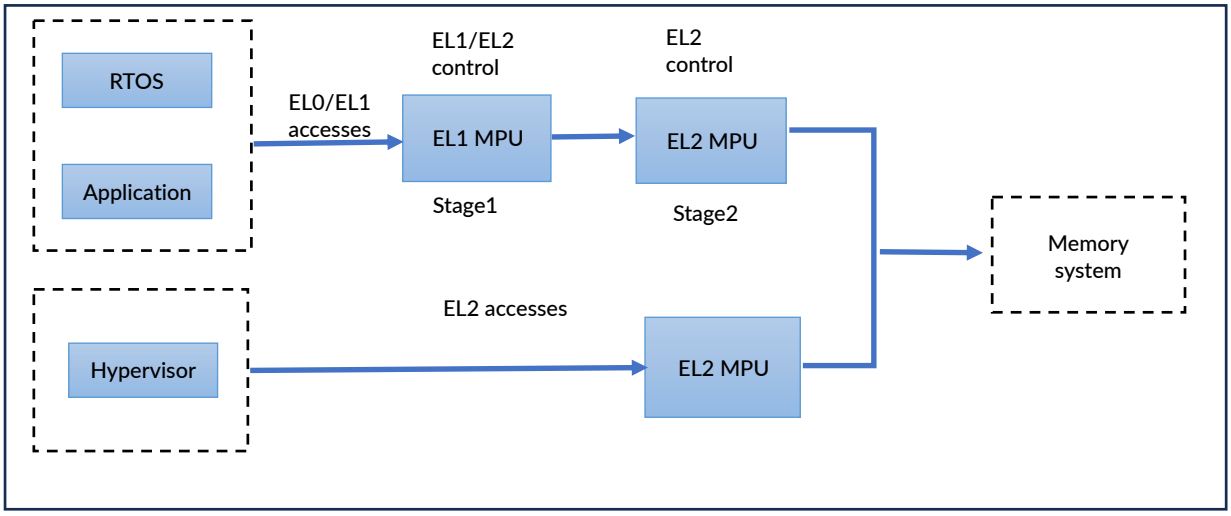
The virtualization support in the processor includes the following:

- Traps on EL1 accesses to control registers. This allows the hypervisor to emulate some operations with a VM.
- Stage 2 MPU. This allows the hypervisor to control which system resources are visible to a VM.
- Virtual interrupt support. The hypervisor can control which interrupts are presented to each VM.

See the Figure 7-2: 2-level MPU structure for virtualization on page 26 Cortex-R52/R52+ processors implement a two-level MPU structure for virtualization. For RTOS or Application memory accesses from EL0/EL1 are protected by the EL1 MPU stage 1 checking and EL2 MPU stage 2 checking. All transactions using the EL0/EL1 transaction regime perform a lookup in both MPUs. The resulting attributes are combined so that the least permissive attributes are taken. These two stages of protection allow the hypervisor to retain control over the EL0/EL1 translation

regime and therefore enables support for virtualization. When software executes using the EL2 translation regime, only the EL2-controlled MPU is used.

Figure 7-2: 2-level MPU structure for virtualization



Armv8-R virtualization describes how to implement the virtualization on Cortex-R52/R52+ processors and provides several examples as the following Table 7-1: Virtualization examples on page 26

Table 7-1: Virtualization examples

Example	Description
Simple guest OS switcher example	This project demonstrates virtualization using two simple guest OSs, register backups, using a physical countdown timer, trapping exceptions, and GIC Fast Interrupt Request (FIQ) and Private Peripheral Interrupts (PPI) configurations.
OS monitor example	This project demonstrates how to monitor a simple RTOS, using a physical countdown timer to drive a simple clock task, trapping exceptions, how to generate a virtual interrupt using Hyp Configuration Register (HCR) flags, and GIC configuration.
Guest OS switcher with virtual interrupts example	This project demonstrates virtualization with two simple guest OSs that both run a clock task, register backups, using a virtual countdown timer to drive the clock tasks in the guest OSs, trapping exceptions, how to generate a virtual system timer, virtual machine IDs, and GIC configuration.
SPIs and SGIs example	This project demonstrates how to use multiple cores, using a countdown timer from a module that is external to the core, and GIC configuration with Shared Peripheral Interrupts (SPIs) and Software-generated Interrupts (SGIs).

These examples are distributed as zipped Arm Development projects. You can download and import into the Arm DS. Build the projects and run them on the Cortex-R52/R52plus FVP.

8. Interrupts

This section is a supplement to the topic of interrupt. It provides more details on the Cortex-R52/R52+ interrupt controller and how it interacts with the system.

Cortex-R Series Programmer's Guide section The Generic Interrupt Controller introduces some basic terminology, such as functional block and interrupt type. This is for GICv1 or GICv2. Cortex-R52/R52+ processors implement the Generic Interrupt Controller v3. For GICv3 and GICv4 see the following documents:

- Learn the architecture – Generic Interrupt Controller v3 and v4, Overview
- Arm Generic Interrupt Controller (GIC) Architecture Specification for GICv3 and GICv4

This section describes some highlights for Cortex-R52/R52+ GIC implementation from the software view. Cortex-R52/R52+ GIC Distributor registers are memory-mapped, with a physical base address specified by input signals CFGPERIPHBASE[31:21], which can be read from the IMP_CBAR register.

The GIC Distributor registers are grouped into 64KB pages. One page for the GIC Distributor (GICD_) *registers and two pages per core for the GICR registers. All the GIC Redistributor (GICR_)* registers are accessible to all the Cortex-R52+ cores. GIC Distributor and Redistributor registers can be accessed through the memory-mapped interface. CPU interface registers can only be accessed through the system register interface.

Cortex-R52/R52+ GIC Distributor provides an optional export port to allow an external device to accept the GIC interrupts like a core. If the interrupt export interface is implemented, it's the last target in the below Table 8-1: Cortex-R52/R52+ GIC memory map on page 27 Cortex-R52/R52+ GIC memory map.

Table 8-1: Cortex-R52/R52+ GIC memory map

Offset from the GIC base address	Description
+64K	Distributor registers
+64K	PMC-R52/R52+ registers
+64K+128K*n	Redistributor registers for Control target n
+64K+128K*n	Redistributor registers for SGI and PPI target n

n is the Cortex-R52/R52+ target ID. If the GIC Distributor export port is included, the ID mapping is as the following Table 8-2: Cortex-R52/R52+ GIC target ID mapping on page 27. X indicates the number of cores.

Table 8-2: Cortex-R52/R52+ GIC target ID mapping

GIC target	X=1	X=2	X=3	X=4
0	Core0	Core0	Core0	Core0
1	Export port	Core1	Core1	Core1
2	-	Export port	Core2	Core2
3	-	-	Export port	Core3

GIC target	X=1	X=2	X=3	X=4
4	-	-	-	Export port

Cortex-R52/R52+ processors have 16 SGIs, INTID0-INTID15, that can be generated for each core. The SGIs have edge-triggered semantics. Software can trigger the SGIs by writing the ICC_SGI0R or ICC_SGI1R registers in the interrupt controller. See Sending and receiving Software Generated Interrupts.

Cortex-R52/R52+ processors have 16 PPIs, INTID16-INTID31 for each core. The export port doesn't support PPIs. Assigned PPIs from the core peripherals are fixed configured as level-sensitive. Unallocated PPIs can be configured to be either rising-edge triggered or active-LOW level-sensitive. The PPI INTID assignment is as the following Table 8-3.

Table 8-3: Cortex-R52/R52+ PPI INTID assignment

PPIs	INTID
Assigned PPIs	22-27, 30
Unallocated PPIs	16-21, 28, 29, 31

Cortex-R52/R52+ processors have 960 SPIs identified as INTID32-INTID991. Each SPI can be configured as rising-edge triggered or active-HIGH level-sensitive. For each core, 32 unique SPIs are routed to the core using low latency hardware. The same SPIs route to all other cores using normal latency hardware. The low latency INTID mapping is as the following Table 8-4: Cortex-R52/R52+ Low latency INTID assignment on page 28.

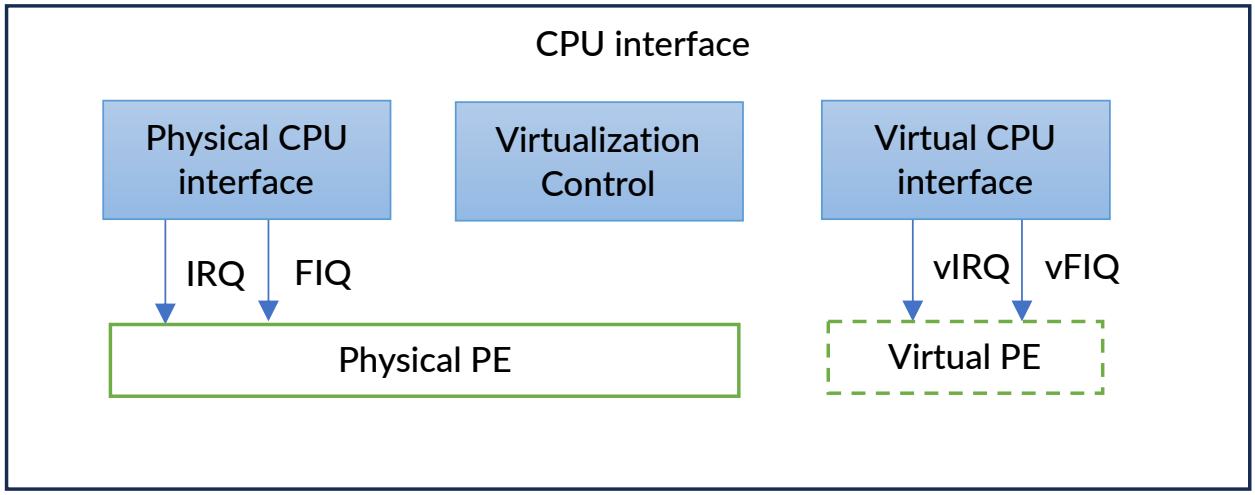
Table 8-4: Cortex-R52/R52+ Low latency INTID assignment

	Core0	Core1	Core2	Core3
Low latency INITID	32-63	64-95	96-127	128-159

On Cortex-R52/R52+ processors there is only one secure state: Non-secure state. So the GIC implements a single Secure state, which is indicated from the GICD_CTLR.DS. Each SPI can be controlled in Group 0 or Group 1 by the GICD_IGROUPRn registers. Group 0 SPIs are signaled as FIQ and Group 1 SPIs are signaled as IRQ. Cortex-R52/R52+ processors support the virtualization. The Figure 8-1: Cortex-R52/R52+ GIC virtualization on page 29 shows how the CPU interface registers are split into three groups:

- ICC: Physical CPU interface registers
- ICH: Virtualization control registers
- ICV: Virtual CPU interface registers

Figure 8-1: Cortex-R52/R52+ GIC virtualization



The physical CPU interface registers have names with the format ICC_. *The hypervisor executing at EL2 uses these registers to handle physical interrupts. Virtualization control registers have names with the format ICH_*. The hypervisor has access to these registers to control the virtualization features provided by the architecture. These features are as follows:

- Enabling and disabling the virtual CPU interface.
- Accessing virtual register state to enable context switching.
- Configuring maintenance interrupts.
- Controlling virtual interrupts for the currently schedule vPE.

These registers control the virtualization feature of the physical PE from which they are accessed. It is not possible to access the state of another PE. That is, software on a PE cannot access state for another PE.

Virtual CPU interface registers have names with the format ICV_. *Virtual machine uses these registers to handle virtual interrupts. These registers have the same format and function as the corresponding ICC_* registers.

- Accesses to group 0 registers at EL1 are virtual when HCR.FMO == 1. Virtual accesses to the following Table 8-5: GIC virtual register mapping on page 30 Group 0 ICC_* registers access the ICV_* equivalents.
- Accesses to group 1 registers at EL1 are virtual when HCR.IMO == 1. Virtual accesses to the following Table 8-5: GIC virtual register mapping on page 30 Group 1 ICC_* registers access the ICV_* equivalents.
- Accesses at EL1 to the common registers are virtual when either HCR.IMO == 1 or HCR.FMO == 1, or both. Virtual accesses to the following Table 8-5: GIC virtual register mapping on page 30 Common ICC_* registers access the ICV_* equivalents.

Table 8-5: GIC virtual register mapping

Group 0 registers	Group 0 registers	Group 1 registers	Group 1 registers	Common registers	Common registers
ICC_* registers	ICV_* equivalents	ICC_* registers	ICV_* equivalents	ICC_* registers	ICV_* equivalents
ICC_AP0R0	ICV_AP0R0	ICC_AP0R1	ICV_AP1R0	ICC_RPR	ICV_RPR
ICC_BPR0	ICV_BRP0	ICC_BPR1	ICV_BRP1	ICC_CTLR	ICV_CTLR
ICC_EOIR0	ICV_EOIR0	ICC_EOIR1	ICV_EOIR1	ICC_DIR	ICV_DIR
ICC_HPPIR0	ICV_HPPIR0	ICC_HPPIR1	ICV_HPPIR1	ICC_PMP	ICV_PMR
ICC_IAR0	ICV_IAR0	ICC_IAR1	ICV_IAR1		
ICC_IGRPEN0	ICV_IGPREN0	ICC_IGRPEN1	ICV_IGPREN1		

Software executing at EL2 can access some ICV_* registers state using ICH_VMCR and ICH_VTR as follows:

- ICV_PMR.Priority aliases ICH_VMCR.VPMR.
- ICV_BPR0.BinaryPoint aliases ICH_VMCR.VBPR0.
- ICV_BPR1.BinaryPoint aliases ICH_VMCR.VBPR1.
- ICV_CTLR.EOImode aliases ICH_VMCR.VEOIM.
- ICV_CTLR.CBPR aliases ICH_VMCR.VCBPR.
- ICV_IGRPEN0 aliases ICH_VMCR.VENG0.
- ICV_IGRPEN1 aliases ICH_VMCR.VENG1.
- ICV_CTLR.PRIbits aliases ICH_VTR.PRIbits.

The document GICv3 and v4, Virtualization describes the support for virtualization in the GICv3 and GICv4 architecture. It introduces how to maintain interrupt and context switching.

9. Memory interface

There are several memory interfaces on the Cortex-R52/R52+ processor as the following Table 9-1: Cortex-R52/R52+ memory interfaces on page 31 shows.

Table 9-1: Cortex-R52/R52+ memory interfaces

Memory interface	Notes	Private or share
AXIM	Main interface to external memory and peripherals.	Private
LLPP	Dedicated port to access private peripherals, also applied to memory without cache.	Private
Flash	The interface provides low-latency access to external read-only memory.	Private
AXIS	The interface provides external access to TCM memory.	Share

The AXIM, LLPP and Flash interfaces include a timeout mechanism to detect transactions that fail to complete within a programmable time limit. Developer can set the timeout duration by setting the IMP_BUSTIMEOUTR register. The Flash, LLPP and AXIM interface timeout value in cycles (MAXCYCLESBYFLASH16FLASH, MAXCYCLESBYFLASH16LLPP, and MAXCYCLESBYFLASH16AXIM) cannot be zero when the timeout counter is enabled.

9.1 AXIM/LLPP interface

The AXIM interface implements the AXI4 protocol with 128-bit data width. For Normal-cacheable memory accesses, the VMID is carried with the user signals.

The LLPP interface implements the AXI4 protocol with 32-bit data width. It is not optimal for Normal memory accesses. Accesses are always treated as non-Gathering, non-Recording, with no Early Write Acknowledgement. The LLPP base address and size is indicated in the IMP_PERIPHPREGIONR register, which is set from the configuration signals CFGLLPPBASEADDR and CFGLLPPSIZE. And it is the same for all cores in a cluster. CFGLLPPIMP indicates if the LLPP interface is implemented. This signal is tied LOW if the system does not use the LLPP, otherwise it must be tie HIGH.

LLPP is disabled out of reset. Developer can configure the register IMP_PERIPHPREGIONR to control whether the LLPP is enabled for access at EL2 and whether the LLPP is enabled for access at EL0 and EL1.

9.2 Flash interface

The Flash interface implements the AXI4 protocol with 128-bit data width. It is a read-only interface, receives request from the instruction side and data side. Developer can configure whether the data or instruction access have the highest priority by setting CPUACTLR.FLASHARBCTL. The Flash interface base address and size is indicated in the IMP_FLASHIFREGIONR register. The Flash interface base address is set from the CFGFLASHBASEADDR[31:27] configuration signals. The Flash region size is 128MB. Developer

can enable or disable the Flash interface by setting the IMP_FLASHIFREGIONR register. Whether the Flash interface is enabled out of reset is implemented individually for each core, and set by the CFGFLASHENx configuration signals.

9.3 AXIS interface

The AXIS implements the AXI4 protocol with 128-bit data width, but only supports a subset of transactions. It is shared between all cores in the processor. It provides external access to TCM memory.

As Figure 5-2: AXIS TCM mapping on page 19 the AXIS address space is 16MB region located at a base address set by configuration input signals CFGAXISTCMBASEADDR[31:24]. Each core has 4MB block and TCMs with in a core are mapped to the first three 1MB regions. Developer can control the AXIS access privilege by programming IMP_SLAVEPCTLR.TCMACCLVL. When the TCM implements ECC, the AXIS access also generates the ECC code.

9.4 Bus protection

The AXIM interface, the LLPP interface, the Flash interface and the AXIS interface optional implement ECC or parity protection for signal integrity, and data integrity. The ECC or parity protection protects the payload data, payload address signals, control signals, response signals, handshake signals, and interconnect protection.

All single-bit errors in outgoing data from the AXIM interface, the LLPP interface, the Flash interface, and the AXIS bus interface are correctable. It is transparent to the software function, although the timing of operation might be affected. All double-bit errors in outgoing data from each interface are uncorrectable.

10. Compiler and Optimization

This section provides information about the compiler and optimization options for Cortex-R52/R52+ processors.

The Arm Compiler for Embedded is an advanced embedded C/C++ compilation toolchain from Arm for the development of bare-metal software, firmware, and Real-Time Operation System (RTOS) applications. It supports the Cortex-R52/R52+ processors. Other third-party compilers also support Cortex-R52/R52+ processors, such as GCC, IAR, CLANG, etc. Developer needs input the appropriate options to the compiler to optimize the code as much as possible. Some important options are listed in the following Table 10-1: Highlight Compiler Options for Cortex-R52/R52+ on page 33.

Table 10-1: Highlight Compiler Options for Cortex-R52/R52+

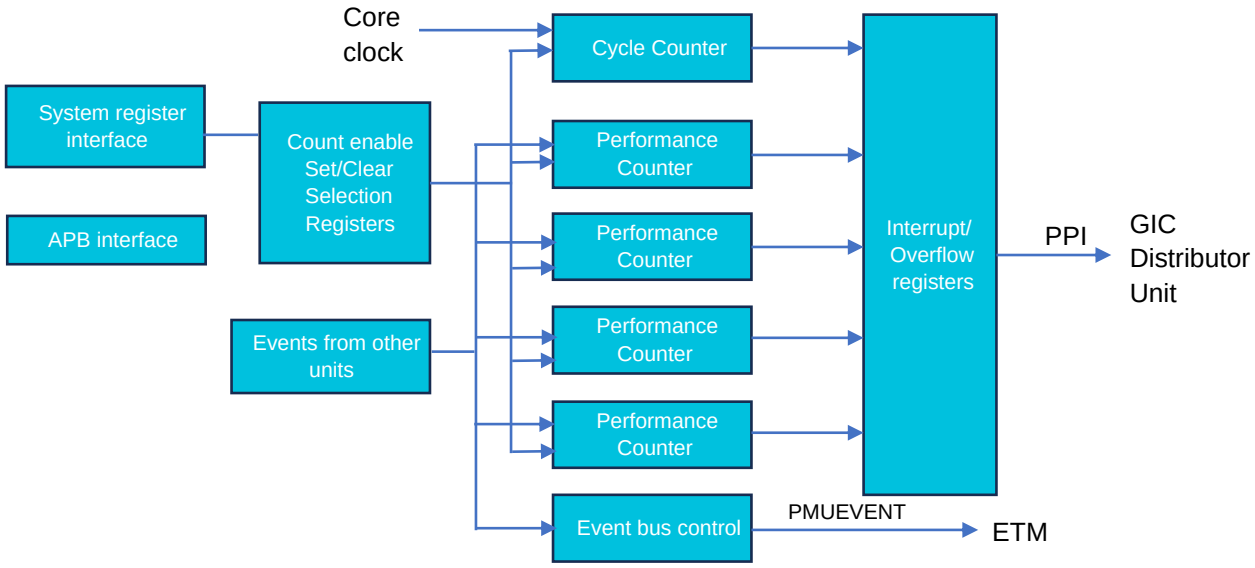
Option	Value	Notes
–target	arm-arm-none-eabi	Generate A32/T32 instructions for AArch32 state. Must be used in conjunction with -march or -mcpu
-march	armv8-r	Armv8-R, be used with –target=arm-arm-none-eabi to specify AArch32 real-time architecture profile
-marm		Compiler generates the A32 instructions, default
-mthumb		Compiler generates the T32 instructions
-mcpu	cortex-r52 [+[no]<feature>+...], cortex-r52plus [+[no]<feature>+...]	To specify a target processor. Use +<feature> or +no<feature> to specify enable or disable an option architecture feature. For example, fp and simd
-mfloat-abi	softfp	Hardware floating-point instructions and software floating-point linkage
-mfloat-abi	hard	Hardware floating-point instructions and hardware floating-point linkage

For example, armclang –target=arm-arm-none-eabi, -mcpu=cortex-r52plus+nosimd -mfloat-abi=softfp

The Cortex-R Series Programmer's guide section Profiling describes the profiling for Armv7-R processors. It is similar for Cortex-R52/R52+ processors, which implement a performance monitor in each core. The PMU provides four programmable counters and a Cycle counter. Developer can setup the programmable counter to select appropriate events and capture the counter value during the software runtime to locate the problem area or bottleneck.

Cortex-R52/R52+ PMU events are exported from the core for use by the ETM and counter overflows are exported for use by the GIC distributor. The Figure 10-1: Cortex-R52/R52+ PMU Blocks on page 34 shows the major blocks inside the PMU.

Figure 10-1: Cortex-R52/R52+ PMU Blocks



Arm Streamline tool can collect the PMU counter on bare-metal target, See Profiling with the bare-metal agent. Developer integrates the barman files (barman.c, barman.h) generated from the Streamline to the existing software project to do the following steps:

- Initialize Barman at runtime.
- Periodically call the data collection routines that Barman provides.
- Optionally, stop the capture.
- Optionally, extract the raw data that Barman collects and provide it to Streamline for analysis. Streamline provides some examples to demonstrate how to use the Barman.

11. Booting

This section provides information on how to initialize the Cortex-R52/R52+ processors after reset. It covers the hardware initialization signals, the booting process, and the EL2 boot code.

The Cortex-R Series Programmer's Guide Chapter Booting code introduces what need to do for the Armv7-R processors booting after reset. It is similar for Cortex-R52/R52+ processor. This section lists some highlights to help the developer to do the booting initialization.

11.1 Hardware initialization signals

Cortex-R52/R52+ processors implement some input signals to set up the hardware logic out of reset. Software developer should do the initialization according to the signal configuration. This section describes some important signals related with the booting as below:

- Signal CFGINITREG initializes the programmer visible registers to fixed value out of reset.
- Signals CPUHALTx let the cores wait out of reset before taking reset exception and fetching instructions. Generally used to prepare the boot image.
- Signals CFGL1CACHEINVDISx enable or disable post-reset L1 cache invalidate.
- Signals CFGVECTABLEx[31:5] set the vector table base address out of reset. Developer needs to preload the vector table in the correct location of memory.
- Signals CFGTHUMBEXCEPTIONSx determine the instruction set state (A32 or T32) and the value of HSCTLR.TE out of reset. And it's indicated in the HSCTLR.TE. Developer should guarantee the instructions in the exception vector table and exception handler are matched with the signal configuration.
- Signals CFGENDIANESSx determine the data endianness (little-endian or big-endian) out of reset and indicated in the HSCTLR.EE.
- Signals CFGPERIPHBASE[31:21] set the base address of GIC Distributor.
- Signals CFGLLPPBASEADDR[31:12] and CFGLLPPSIZE[3:0] set the base address and size of LLPP interface. And this information is indicated in the register IMP_PERIPHPREGIONR. Signal

CFGLLPPIMP is used to indicate if the LLPP region is implemented in memory map. If boot is from the TCM:

- External debugger, DMA, or other Cores can preload the boot image into the TCM through the AXIS interface. The base address of the TCMs on the AXIS interface is set by the CFGAXISTCMBASEADDR[31:14] signals. The address is indicated in the register IMP_PINOPTR.
- Signals CFGTCMBOOTx enable the ATCM from reset and set the ATCM base address at `0x0`, other TCMs are disabled, and the base address are **UNKNOWN**.

If boot is from the Flash:

- The signal CFGFLASHIMP indicates if the FLASH is implemented. The CFGFLASHENx signals enable the Flash interface out of reset. Developer can enable or disable it by setting register IMP_FLASHIFREGIONR.ENABLE.
- The signals CFGFLASHBASEADDR[31:27] set the Flash interface base address. Developers must preload the boot image in the correct range. The base address is indicated in the register IMP_FLASHIFREGIONR.

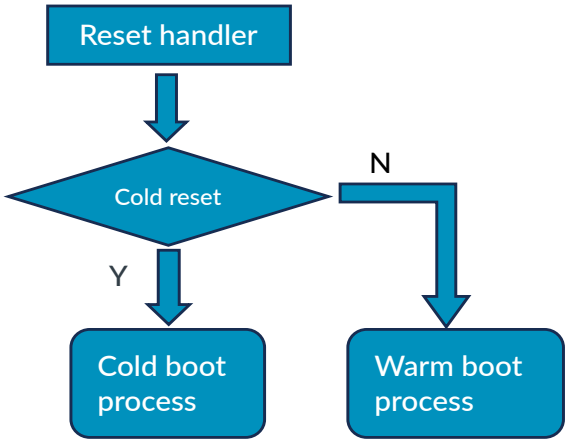
Form more signal specification see the Cortex-R52/R52+ Signal Description. Some hardware resources are disabled out of reset. Developers must enable them before using them.

- MPUs are disabled out of reset. The default memory map is used. The reset vector table and reset handler of boot image must be kept in the low 2GB region (`0x0-0x7FFFFFFF`).
- Caches are disabled out of reset. Developers can enable them by programming the (H)SCTLR registers.
- LLPP is disabled out of reset. Developers can enable the accesses at EL1 or EL2 by programming the register IMP_PERIPHPREGIONR.
- GIC is disabled.
- Advanced SIMD and VFP are disabled.
- CNTFRQ register is set to `0x0`. Developers must set the clock frequency according to the hardware implementation. Writable only at EL2.

11.2 Booting process

There are 2 boot types as the Figure 11-1: Booting type on page 36 according to the CPU reset difference. Cold boot is related with the cold reset (hard reset) and warm boot is related with the warm reset (soft reset). The cold reset is triggered by the power on reset. The warm reset is triggered by the recovering from power saving mode. The boot code needs to be aware of which type and enters the different process. It can get the information from the power management controller.

Figure 11-1: Booting type



According to the software stack running on the CPU, the boot type includes bare-metal boot as the following Figure 15, OS boot as the following Figure 16, and hypervisor boot as the Figure 11-2: Bare-metal boot on page 37.

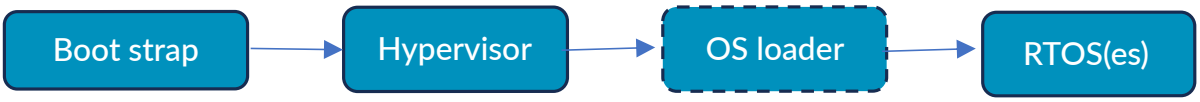
Figure 11-2: Bare-metal boot

During the bare-metal boot process, the processor starts executing from the reset handler out of reset and initializes the environment such as setup the MPU region, peripheral and so on. It enters the C library initialization from the __main symbol. After that the library calls the main() function of the bare-metal application.

Figure 11-3: OS boot

During the OS boot process, CPU runs the boot strap first. If the OS loader is available, CPU runs the OS loader to load the RTOS to the appropriate place, and then the CPU jumps to the RTOS entry point to run the RTOS. If the OS load is not available, developers must manage the bridge from the boot strap to RTOS or boot from the RTOS directly.

Figure 11-4: Hypervisor boot



If the virtualization is implemented on the platform, the boot strap brings the Hypervisor. If the OS loader is available, it loads the RTOSes and lets the Hypervisor to manage the RTOS context switching. The following code sections will show the boot steps at EL2 and EL1 from a bare-metal example code. In this example there are four Cortex-R52 cores. Each core is named by the core ID, which is read from the register MPIDR:

```
MRC p15, 0, r0, c0, c0, 5 //read MPIDR
AND r0, r0, 0xF
```

Core 0 is set as the primary core, and initializes the global resources for the cluster. All other secondary cores are held in the reset by the hardware. The primary core can bring other cores out of reset by programming the reset controller or power controller. The secondary cores then initialize their local resources.

11.3 EL2 boot code

Developers must load the boot image in the appropriate memory such as TCM, Flash, or RAM. Put the vector table at the address matched with the signal CFGVECTABLEx[31:5] configuration. For example, the EL2 vector table looks as follows:

```
EL2_Vectors:
    LDR PC, EL2_Reset_Addr
    LDR PC, EL2_Undefined_Addr
    LDR PC, EL2_HVC_Addr
    LDR PC, EL2_Prefetch_Addr
    LDR PC, EL2_Abort_Addr
    LDR PC, EL2_HypModeEntry_Addr
    LDR PC, EL2_IRQ_Addr
    LDR PC, EL2_FIQ_Addr
```

The CPU executes the first instruction LDR PC, EL2_Reset_Addr to jump to the EL2_Reset_handler.

```
EL2_Reset_Addr:         .word    EL2_Reset_Handler
```

1. In the function EL2_Reset_handler, check the CPU ID, CPU 0 continues doing the initialization. If other cores are also out of reset they enter the WFI loop to sleep.

```
EL2_Reset_Handler:
    MRC p15, 0, r0, c0, c0, 5       // Read MPIDR, check which CPU I am
    ANDS r0, r0, #0xF
    BEQ cpu0
// If run on a multi-core system, put any secondary cores to sleep
loop_wfi:
    DSB SY                          // Clear all pending data accesses
    WFI                             // Go to sleep
    B loop_wfi
```

1. Each core writes the EL2_Vectors to the HVBAR register to prepare the EL2 exception handling.

```
//secondary cores also do the below step after wakeup from the sleep.
cpu0:
    // Change EL2 exception base address
    LDR r0, =EL2_Vectors
    MCR p15, 4, r0, c12, c0, 0      // Write to HVBAR
```

1. Initialize the HSCTLR register according to Hardware initialization signals. In this example, signals CFGTHUMPEXCEPTIONS is configured as 0, which means the exceptions taken in EL2 are taken in A32 state. Signal CFGENDIANESSx are configured as 0, which means the CPSR.E is 0 (little endian) on an exception taken to EL2.

```
// Init HSCTLR
    LDR r0, =0x30C5180C             // See TRM for decoding
    MCR p15, 4, r0, c1, c0, 0       // Write to HSCTLR
```

1. Initialize the System counter timer frequency register according to the hardware implementation.

```
// Init System Timer CNTFRQ
    LDR r0, =CNTFRQ
    MCR p15, 0, r0, c14, c0, 0      // Write r0 to CNTFRQ
```

1. Set up the Stack pointer for the Hypervisor environment.

```
// Set HYP stack pointer
        LDR r0, =Image$$ARM_LIB_STACKHEAP$$ZI$$Limit
        MSR sp_hyp, r0
```

1. Enable the floating-point and Advanced SIMD.

```
    MRC     p15, 0, r0, c1, c0, 2   // Read Coprocessor Access Control Register
 (CPACR)
    ORR     r0, r0, #(0xF << 20)    // // Enable access to CP 10 & 11
    MCR     p15, 0, r0, c1, c0, 2   // Write Coprocessor Access Control Register
 (CPACR)
    ISB
    MOV     r0, #0x40000000
    VMSR    FPEXC, r0               // Write FPEXC register, EN bit set
```

Program the register IMP_FLASHIFREGIONR and IMP_PERIPHPREGIONR to get the base address and size of Flash interface and LLPP interface. Enable the LLPP access at EL0/EL1 and EL2.

```
MRC p15, 0, r0, c15, c0, 1          // Read from FLASHIFREGIONR
UBFX r1, r0, #2, #5                 // Get the Flash size
BFC r0, #0, #27                     //Get the Flash base address
MRC p15, 0, r2, c15, c0, 0          // Read from PERIPHPREGIONR
UBFX r3, r2, #2, #5                 // Get the Peripheral port region size
BFC r2, #0, #12                     //Get the Peripheral port region base address
ORR r2, r2, #3                      //enable the peripheral port at EL1 and EL2
MCR p15, 0, r2, c15, c0, 0          //write r2 to PERIPHPREGIONR
```

1. Setup the EL2 MPU regions and enable the EL2 MPU.

```
// EL2 MPU configuration
    BLX config_EL2_MPU              //function to setup EL2 MPU regions
    MRC p15, 4, r0, c1, c0, 0       // Read HSCTLR
    ORR     r0, r0, #1              // set M bit (EL2 MPU enable)
    MCR p15, 4, r0, c1, c0, 0       // Read HSCTLR
```

11.4 Jump to EL1

Before jump to EL1 mode, developers needs do some prepare steps. 1) Set the EL1 exception table to the VBAR to prepare the EL1 exception handling. This can be done at EL2 or EL1.

```
// Change EL1 exception base address
    LDR r0, =EL1_Vectors
```

```
    MCR p15, 0, r0, c12, c0, 0      // Write to VBAR
```

1. Program the Hyp Auxiliary Control Register to enable the EL1 access to all IMPLEMENTED DEFINED registers.

```
// Enable EL1 access to all IMP DEF registers
    LDR r0, =0x7F81
    MCR p15, 4, r0, c1, c0, 1       // Write to HACTLR
```

If the developer does not need EL2, can program the EL2 registers and switch to EL1 so that it can never return to EL2 except by means of a reset. Disable the HVC instruction by setting HCR.HCD to 1.

```
//Disable the HVC instruction execution at EL1
    MRC p15, 4, r0, c1, c1, 0       //read HCR
    ORR r0, r0, #(0x1<<29)
    MCR p15, 4, r0, c1, c1, 0       //write HCR
```

Change the SPSR register to go to EL1 SVC mode, set the EL1_Reset_Handler address to ELR_hyp register, execute the ERET instruction to jump to EL1 SVC mode and update the CPSR and PC value.

```
// Go to SVC mode
    MRS r0, cpsr
    MOV r1, #Mode_SVC
    BFI r0, r1, #0, #5
    MSR spsr_hyp, r0
    LDR r0, =EL1_Reset_Handler
    MSR elr_hyp, r0
    DSB
    ISB
    ERET
```

11.5 EL1 boot code

In the EL1_Reset_Handler, disable the MPU and caches.

```
    MRC p15, 0, r0, c1, c0, 0       // Read System Control Register
    BIC r0, r0, #0x05               // Disable MPU (M bit) and data cache (C bit)
    BIC r0, r0, #0x1000             // Disable instruction cache (I bit)
    DSB                             // Ensure all previous loads/stores have
 completed
    MCR p15, 0, r0, c1, c0, 0       // Write System Control Register
    ISB                             // Ensure subsequent instructions execute with
 respect to new MPU settings
```

Set up the stack pointer for each mode at EL1.

```
#define STACKSIZE 512
    // Setup the stack(s) for this CPU
    MRC  p15, 0, r1, c0, c0, 5      // Read CPU ID register
    AND  r1, r1, #0x03              // Mask off, leaving the CPU ID field
```

```
    LDR  r0, =Image$$ARM_LIB_STACK$$ZI$$Limit
    SUB  r0, r0, r1, lsl #14
    CPS #Mode_ABT
    MOV SP, r0
    //Setup the stacks for IRQ, FIQ, SVC modes
    //...
```

Developers can invalidate the entire instruction cache and data cache by cache operation instructions. By default, they are automatically invalidated before they are used, controlled by the input signals CFGL1CACHEINVDISx. If the TCMs are used, developers must setup the TCMs base address according to the system design and enable them.

```
#ifdef TCM
    DSB        // Ensure all previous loads/stores have completed
    LDR r0, =Image$$ATCM$$Base      // Set ATCM base address
    ORR r0, r0, #1                  // Enable it
    MCR p15, 0, r0, c9, c1, 0       // Write ATCM Region Register IMP_ATCMREGIONR
    ISB       //flush the pipeline and ensure the TCM configuration work
    //similar for the BTCM and CTCM
    //...
```

In the function configu_EL1_MPU, set up the MPU regions for the memory access at EL1, includes Code, Data, Stack/Heap, Peripherals and TCMs.

```
// EL1 MPU configuration
    BLX config_EL1_MPU              //Function to setup EL1 MPU regions
    MRC p15, 0, r0, c1, c0, 0       // Read SCTLR
    ORR     r0, r0, #1              // set M bit (EL1 MPU enable)
    MCR p15, 0, r0, c1, c0, 0       // Write SCTLR
    ISB
```

Branch to __main symbol to do the C library initialization routine.

```
.global __main
    B   __main
```

C library calls the main() function. In the main function developers can enable the caches and set up the GIC and timer or other initialization and jump to the application code at last.

```
Int main(void)
{
        Enable_caches();            //Enable the caches
        GIC_configure();            // Configure GIC
        ...                           //other initialization or branch to
 application code
}
```

The enable_caches function example is as follows:

```
enable_caches:
    MRC     p15, 0, r0, c1, c0, 0   // read System Control Register
    ORR     r0, r0, #(0x1 << 12)    // enable I Cache
    ORR     r0, r0, #(0x1 << 2)     // enable D Cache
    MCR     p15, 0, r0, c1, c0, 0   // write System Control Register
    ISB
```

```
    BX    lr
```

GIC example code is provide in Learn the architecture – Generic Interrupt Controller v3 and v4, Overview.