arm

Armv8-R AArch64 Architecture
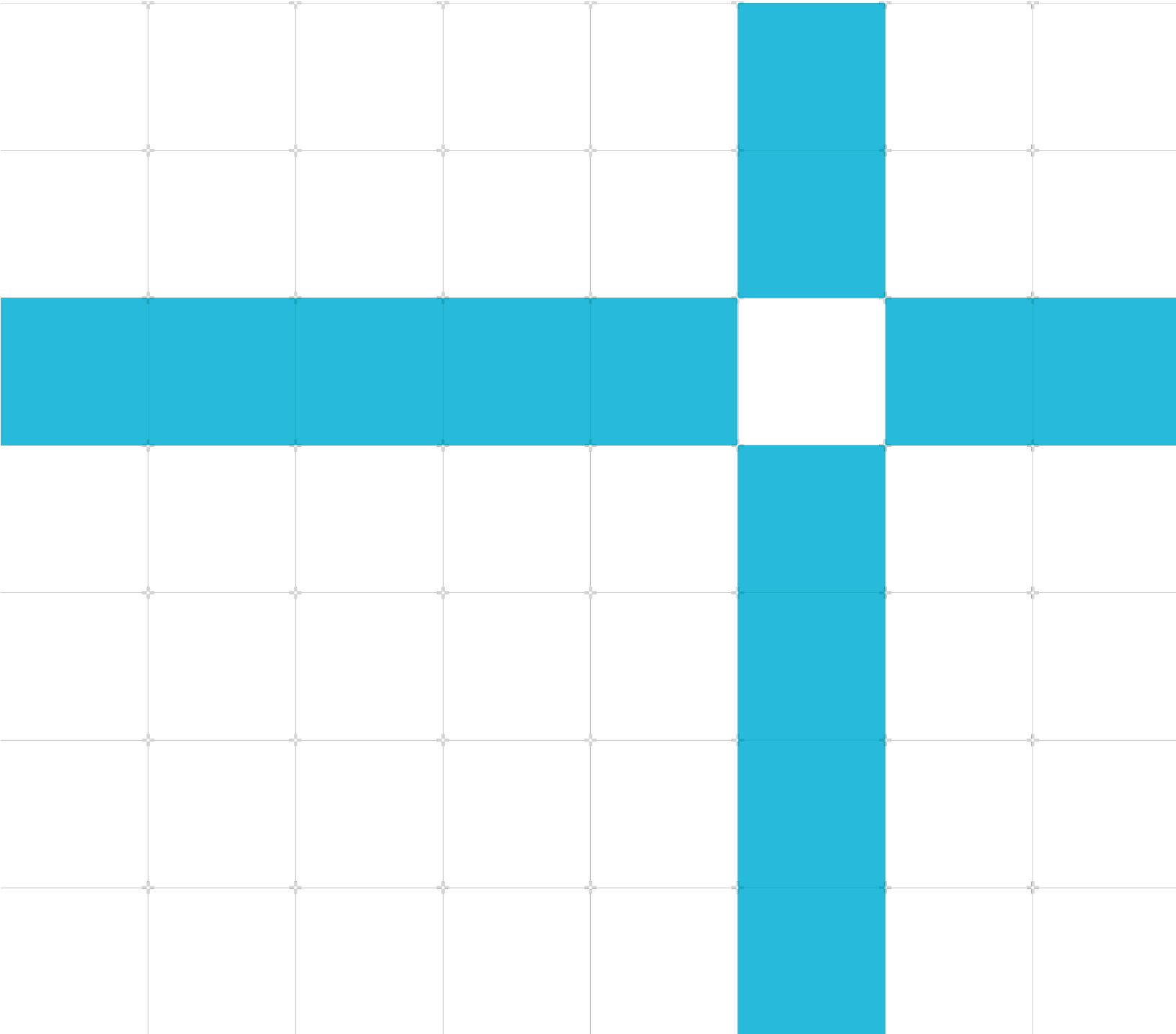
Software Migration Guide from Armv8-R AArch32 to Armv8-R AArch64

Issue 1.0

109860_1.0_en

Armv8-R AArch64 Architecture Software Migration Guide from Armv8-R AArch32 to Armv8-R AArch64

Start reading

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

Intended audience

This guide is intended for software developers who want to migrate the software from Armv8-R AArch32
processor to Armv8-R AArch64 processor.

This document assumes that you are familiar with the following:

- Armv8-R AArch32 software development

- Armv8-A AArch64 Architecture

- Armv8-R AArch32 Architecture

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 Armv8-R AArch64
Architecture, 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

Currently, Armv8-R AArch32 based CPUs including Cortex-R52 and Cortex-R52+ are widely used in diverse markets including automotive and industrial. With the introduction of Cortex-R82 and Cortex-R82AE, many partners will be interested in using these higher performance cores. Such partners can minimize their development time and cost, by reusing their existing software based on Cortex-R52 or Cortex-R52+ to the new 64-bit cores.

The purpose of this guide is to help developers to migrate software from Armv8-R AArch32 processors to Armv8-R AArch64 processors. This guide presents the difference and suggestions on how software needs to be modified so that it can be migrated to different Arm processors.

Figure 1-1 shows the comparison between Armv8-R AArch32, Armv8-R AArch64 and Armv8-A architecture.

Armv8-R AArch32 is a 32-bit architecture, which is backward compatible with Armv7-R. For example, Cortex-R52 is an implementation of Armv8-R AArch32 architecture. Armv8-R AArch64 is a 64-bit architecture, which is based on Armv8-A AArch64. For example, Cortex-R82 is an implementation of Armv8-R AArch64 architecture. Unlike Armv8-A, which can support both AArch32 and AArch64 on one CPU implementation.

This Guide will cover different parts of the architecture implementation including ISA, Exception model, Memory model, System register, Security, Virtualization, Compile, and others.

Figure 1-1: Architecture comparation
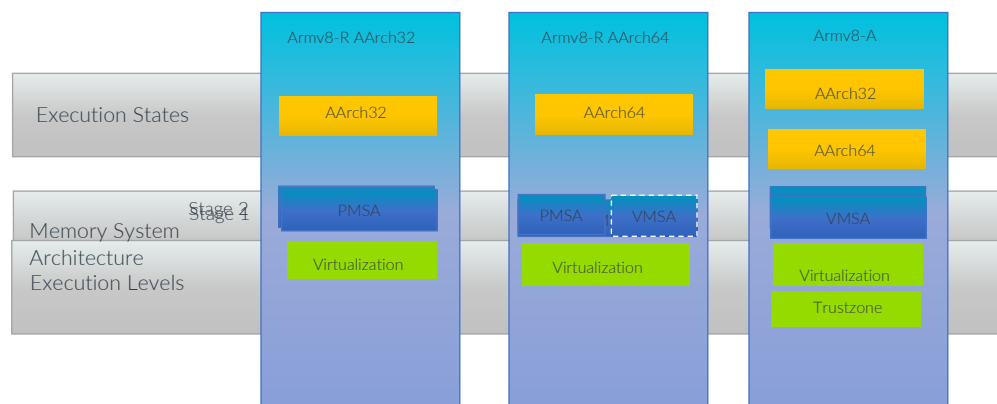
1 ISA

1.1. Base Instruction Set

There are three different Instruction Sets used by Armv8-R AArch32 and Armv8-R AArch64.

Table 1-1: Instruction Sets Used in different Architectures

Armv8-R AArch32	Armv8-R AArch64
A32(Arm)	A64
T32(Thumb)	

Most Armv8-R A32 instructions are not changed with respect to Armv8-A A32. Some new or changed instructions are as follows:

Table 1-2: Updated instructions in Armv8-R AArch32

Armv8-R A32 Instructions	Status with respect to Armv8-A A32 Instructions
DCPS3	Unused
DFB	New
DMB	Redefined
DSB	Redefined
SMC	Unused

Similar, most of Armv8-R A64 instructions are not changed with respect to Armv8-A A64 instruction. Some new or changed instruction are as follows:

Table 1-3: Updated instructions in Armv8-R AArch64

Armv8-R A64 Instructions	Status with respect to Armv8-A 64 Instructions
DCPS3	Unused
DFB	New
DMB	Redefined
DSB	Redefined
SMC	Unused

For specific instructions please refer to the Armv8-R AArch32 Supplement and the Armv8-R AArch64_supplement documents.

If you are migrating the hand-coded assembler in T32 or A32, please replace the instructions with equivalent instructions from the A64 Instruction Set and rebuild the software. You also need to modify and check the size of data types and instructions that have no equivalent.

Note, for the source code implemented in C/C++ language, it might also be necessary to modify and check the data type, such as from int to uint32, long type data size is extended to 64-bit. Recompile the C/C++ code with the correct compiler option to generate the appropriate A64 instructions. See Section 9.1 for more details.

1.2. Advanced SIMD and Floating-point support

Armv8-R AArch32 has the option to support single-precision floating-point only. But Armv8-R AArch64 does not have this option. The following table shows some Armv8-R AArch32 CPUs and Armv8-R AArch64 CPU configurations for Advanced SIMD and Floating-point.

Table 1-4: Advanced SIMD and Floating-point configuration support

	Floating-point	Advanced SIMD
Cortex-R52 Cortex-R52+	Single-precision only	Not implemented
	Single-precision + Double-precision	Implemented
Cortex-R82	Half-precision + Single-precision + Double-precision	Implemented
	Not implemented	Not implemented

1.3. Register Set

Armv8-R AArch64 architecture provides 31 general purpose registers, each register has 64-bit(Xn) and 32-bit(Wn) form. It's different with Armv8-R AArch32 architecture definition, which follow the AAPCS-AArch32 definition refer Table 1-6: AArch32 register definition for AAPCS-AArch32.

A separated set of 32 registers are used for floating point and vector operations called V0-V31. These registers have 128-bit(Qn), 64-bit(Dn), 32-bit(Sn), 16-bit(Hn), and 8-bit(Bn) form.

For developers writing the software code for Armv8-R AArch64 CPUs, the code should follow the **Procedure Call Standard for the AArch64**. It is different from writing the software code for Armv8-R AArch32 CPUs. For Armv8-R AArch32 CPUs, the code should follow the **Procedure Call Standard for AArch32**.

Table 1-5: AArch64 register definition for AAPCS-AArch64

X0-X7	X8-	X16-x18	X19-X30
Parameter and Result Registers	X8(XR)	X16(IP0)	X19-X28(Callee-saved Registers)
		X17(IP1)	X29(FP)

	X9-X15(Temporary registers)	X18(PR)	X30(LR)

- IP0-IP1: Intra-procedure-call temporary registers

- XR: Indirect result location parameter

- PR: Platform registers, reserved for use by platform ABIs

- FP: Frame pointer

Table 1-6: AArch32 register definition for AAPCS-AArch32

| R0-R3	R4-R12	R13-R14	R15
Parameter and Result Registers	R4-R11 (Callee-saved Registers)	R13(SP)	PC
	Corruptible Register(R12)	R14(LR)	

2 Exception model

2.1. Exception model

Armv8-R AArch32 exception model is similar to the Armv8-A AArch32 with Non-Secure State. It has 8 Modes on 3 Exception Levels.

Table 2-1: Armv8-R AArch32 Exception Model

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

Armv8-R AArch64 only supports AArch64 on 3 Exception levels (EL0-EL2) with Secure state. EL0 is the least privileged exception level. EL2 is the most privileged exception level. EL3 is not supported. EL0 is for application space, EL1 is for Kernel space, and EL2 is for Hypervisor or firmware space.

Figure 2-1: Secure and Privilege
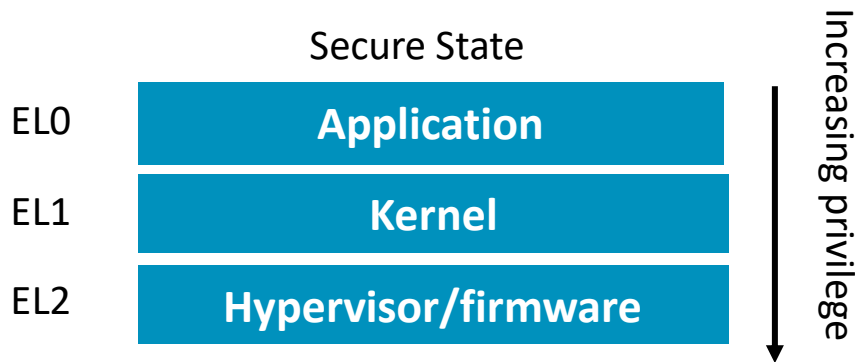
Exception level can only be changed when taking or returning from an exception.

Figure 2-2: Exception level switch
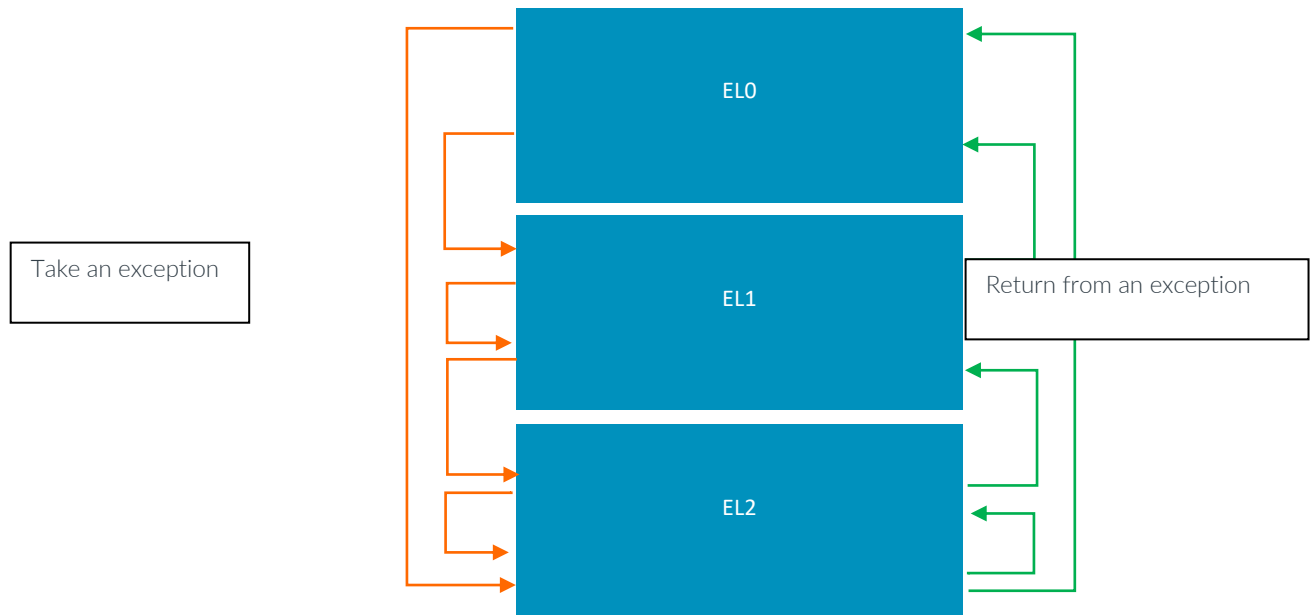
On taking an exception, the exception level can stay the same or increase. On returning from an exception, the exception level can stay the same or decrease.

2.2. Exception Vector Table

There are two vector tables in Armv8-R AArch32, one for EL1 and another for EL2. In some software implementations each guest OS may have its own EL1 vector table. These vector tables are managed by the hypervisor. Each Armv8-R AArch32 vector table has only 8 entries, as shown in Table 2-2: Armv8-R AArch32 Vector table. Each entry has 4bytes which consists of instructions not addresses. This means there is one 32-bit instruction or two 16-bit instructions, usually direct branch instruction or LDR instruction that loads the exception handler address to PC register. The following table shows the Armv8-R AArch32 vectors.

Table 2-2: Armv8-R AArch32 Vector table

Offset	EL1 Vector table	EL2 Vector table
0x1C	FIQ	FIQ
0x18	IRQ	IRQ
0x14	(Reserved)	Hypervisor Trap / Hypervisor mode entry
0x10	Data Abort	Data Abort(From Hypervisor mode)
0x0C	Prefetch Abort	Prefetch Abort(From Hypervisor mode)
0x08	Software Interrupt	HVC(From Hypervisor mode)
0x04	Undefined Instruction	Undefined Instruction(From Hypervisor mode)
0x00	(Reset)	Reset

Armv8-R AArch64 also has 2 vector tables, one is for EL1, another is for EL2. The format of the2 tables is the same. However, the Armv8-R AArch64 vector table size is larger. It has 16 entries, each entry has 128 bytes, 32 instructions. The developer could prepare more instructions to handle the exception. It does not need to branch outside if the exception handler is short. It is better than 1 or 2 instructions allowed in Armv8-R AArch32 Architecture. The vector table can be split to 4 sections as Table 2-3: Armv8-R AArch64 Vector table. Each section has 4 entries. The top section is not used on Armv8-R AArch64 processors. The location of the vector table is set in the register VBAR_ELx.

The Armv8-R AArch64 vector table is same as Armv8-A AArch64 vector table. The selection of one entry depends on where the exception comes from and which SP register is used.

Table 2-3: Armv8-R AArch64 Vector table

Offset	EL1/EL2 Vector table	
0x780		Not used on Armv8-R AArch64
0x700		
0x680		
0x600		
0x580	SError	Exception from a lower Exception Level
0x500	FIQ	
0x480	IRQ	
0x400	Synchronous	
0x380	SError	Exception from the current Exception Level while using the SP_ELx
0x300	FIQ	
0x280	IRQ	

0x200	Synchronous	
0x180	SError	Exception from the current Exception Level while using the SP_EL0
0x100	FIQ	
0x080	IRQ	
0x000	Synchronous	

2.3. Exception handling

Armv8-R AArch64 exception handling keeps the same with Armv8-A AArch64, a little different with Armv8-R AArch32 handling. The Table below doesn't include all differences, please refer to the Arm Architecture Reference Manual Section D1.3 for details.

Table 2-4: Exception handling difference

	Armv8-R AArch32	Armv8-R AArch64
ISA in ISR	T32/A32	A64
Exception return address	LR in EL1, ELR in EL2	ELR
Vector table address	VBAR is at EL1, HVBAR is at EL2	VBAR is at EL1, HVBAR is at EL2
LR register adjustment	Yes in EL1, No in EL2	No
Exception information	IFSR/IFAR/DFSR/DFAR at EL1 HSR/HDFAR/HIFAR/HPFAR at EL2	ESR_EL1 FAR_EL1 at EL1 ESR_EL2 FAR_EL2 at EL2
...	...	...

3 Memory model

3.1. Memory System Architecture

Armv8-R AArch32 CPUs apply the Protected Memory System Architecture (PMSAv8-32). Armv8-R AArch64 implementation supports both the PMSAv8-64(Protected Memory System Architecture) and the VMSAv8-64(Virtual Memory System Architecture). Both PMSAv8-64 and VMSAv8-64 are compliant with Armv8-R AArch64 architecture.

The memory system architecture might be different at different ELs.

Table 3-1: Memory system implementation

	V8-R AArch32	V8-R AArch64
Secure EL0&EL1	N/A	VMSAv8-64 or PMSAv8-64
Secure EL2	N/A	PMSAv8-64
Non-Secure EL0&EL1	PMSAv8-32	N/A
Non-Secure EL2	PMSAv8-32	N/A

PMSA defines the memory protection region using MPU. The PMSAv8-32 defines maximum size of region as 4 GB. The maximum supported address bit size of PMSAv8-64 is 48, or 52 if FEAT_LPA is enabled.

VMSAv8-64 provides an MMU that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE.

Armv8-R AArch64 supports VMSAv8-64 as an optional memory system architecture at the stage1 of the Secure EL1&EL0 translation regime, and supports general purpose operating systems, such as Linux and Android at EL1.

3.2. Protected Memory System Architecture

An MPU defines protection regions in the address map. A protection region is a contiguous memory region for which the MPU defines the memory attributes and the access permissions.

PMSAv8-64 defines the protection region by a pair of registers, a Base Address Register, and a Limit Address Register with the following constraints.

- Have a minimum of 64 bytes.

- Have a maximum of the entire address map.

- Must not overlap.

- Have access permission and memory attributes.

The definition of memory region protection is the same for both Armv8-R AArch32 PMSAv8-32 and Armv8-R AArch64 PMSAv8-64. The maximum supported address bit size in the PMSAv8-64 region is 48, or 52 if FEAT_LPA is enabled. A PE can implement a smaller PA range and the actual implemented physical address range which is provided by the ID_AA64MMFR0_EL1.PARange field. Any access to physical memory address outside the address range results in a memory fault.

3.2.1. Default Memory Map

The PMSAv8-32 MPU has an associated default memory map which is used when the MPU is not enabled.

When the MPU is enabled and Background region checking is enabled, privileged access that does not hit defined protection regions undergoes a second check. SCTLR.BR controls the Background region checking for the EL1 MPU. HSCTLR.BR controls the Background region checking for the EL2 MPU. The EL1/EL2 Default Memory Maps are as Figure 3-1.

Figure 3-1: PMSAv8-32 Default Memory Maps
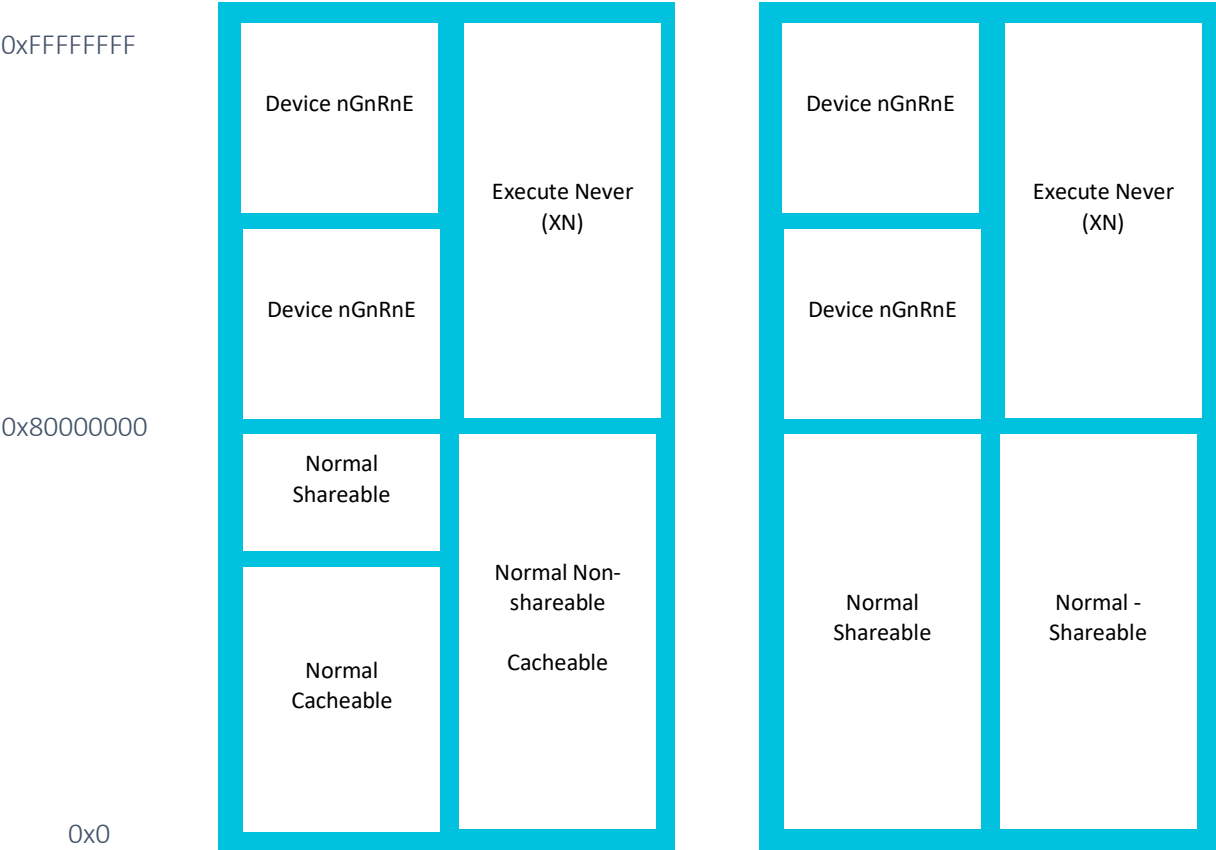
For PMSAv8-64, the Background region is enabled and disabled using SCTLR_ELx.BR, which is set as 0 by PE reset. If MPU is disabled and the Background region is enabled, the MPU uses the default memory map as the Background region to generate the memory attribute. If MPU and Background region are both disabled, it uses Armv8-A AArch64 memory view to generate the

memory attribute. This means data access is treated as Device-nGnRnE memory attribute, instruction access is treated as Normal memory attribute.

Different from PMSAv8-32, the default memory map of the Armv8-R AArch64 architecture is **IMPLEMENTATION DEFINED**. Therefore, the Armv8-R AArch64 architecture defines only the condition to access the default memory map, but not the memory map itself. The memory attributes, access permissions, and Security state of the memory regions in the default memory map are also **IMPLEMENTATION DEFINED**. For example, the R82 default memory map is as follows:

Figure 3-2: Cortex-R82 Default Memory Map
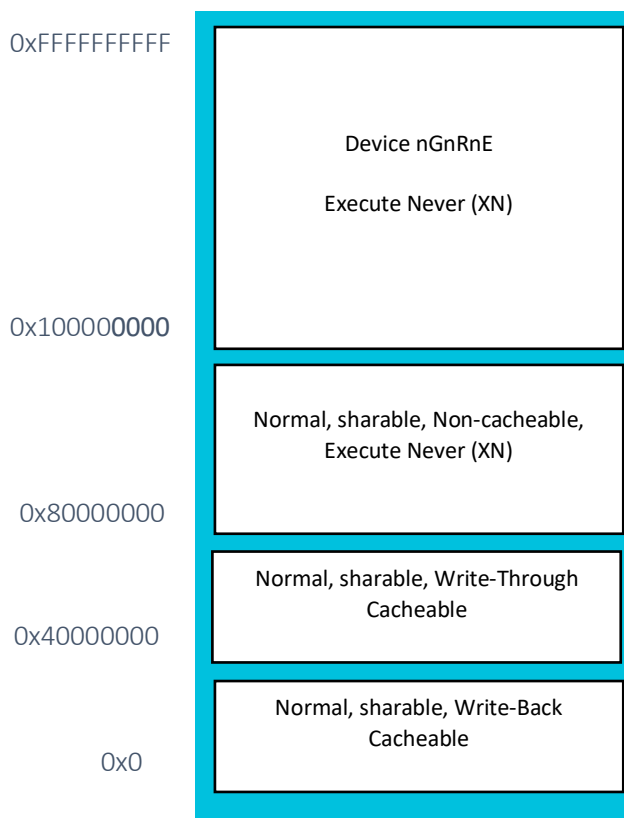
Any access outside the implemented physical address range in the default memory map results in a memory fault. The default memory map is the same for EL1 MPU and EL2 MPU.

3.3. Virtual Memory System Architecture

Unlike Armv8-R AArch32, Armv8-R AArch64 also supports the VMSAv8-64 as an optional memory system architecture at the stage 1 of the EL0&1 translation regime. The VMSAv8-64 provides a MMU that controls address translation, access permissions and memory attributes determination and checking, for memory accesses made by the PE. The process of address translation maps the virtual address (VA) used by the PE onto the physical address (PA) of the physical memory system.

With VMSAv8-64 supported at EL1, the Armv8-R AArch64 architecture profile can have the following memory system configurations:

- PMSAv8-64 at EL1 and EL2.

- PMSAv8-64 or VMSAv8-64 at EL1, and PMSAv8-64 at EL2. VMSA is used at EL1 if VCTR_EL2.MSA=1. PMSA is used at EL2 if VCTR_EL2.MSA=0.

For more information, see The AArch64 Virtual Memory System Architecture chapter of the Arm Architecture Reference Manual for A-profile architecture.

4 System Registers

Most Armv8-R AArch32 system registers are unchanged with respect to ARMv8-A AArch32 system registers. The Table 4-1: Armv8-R AArch32 System Registers lists the differences:

Table 4-1: Armv8-R AArch32 System Registers

Armv8-R AArch32 system Register	Status with respect to Armv8-A AArch32	Armv8-R AArch32 system Register	Status with respect to Armv8-A AArch32
DBGAUTHSTATUS	Redefined	HMPUIR	New
DBGDSCRext	Redefined	HPRBAR	New
DLR	Redefined	HPRBAR<n>	New
FPSCR	Redefined	HPRLAR	New
HCPTR	Redefined	HPRLAR<n>	New
HCR	Redefined	HPRENR	New
HCR2	Redefined	HPRSELR	New
HDCR	Redefined	MPUIR	New
HSR	Redefined	PRBAR	New
HSCTLR	Redefined	PRBAR<n>	New
ID_MMFR0	Redefined	PRLAR	New
ID_MMFR2	Redefined	PRLAR<n>	New
IFSR	Redefined	PRSELR	New
IFSR	Redefined	VSCTLR	New
PMCR	Redefined	NMRR	Unused
SCTLR	Redefined	PRRR	Unused
		RMR	Unused

For each specific new or redefined register information please refer Armv8-R AArch32 Supplement Section E1.1.

Armv8-R AArch64 system registers are also modified with respect to Armv8-R AArch32 profile. The Table 4-2: Armv8-R AArch64 System Registers lists most of the different registers but not all, some debug and PMU registers are not included.

Table 4-2: Armv8-R AArch64 System Registers

Armv8-R AArch64 System Register	Respect to Armv8-R AArch32	Armv8-R AArch64 System Register	Respect to Armv8-R AArch32
MPUIR_ELx	MPUIR/HPUIR	PRBAR_ELx	PRBAR/HPRBAR

PRBAR<n>_ELx	PRBAR<n>/HPRBAR<n>		PRLAR_ELx	PRLAR/HPRLAR
PRLAR<n>_ELx	PRLAR<n>/HPRLAR<n>		PRENR_ELx	HPRENR
PRSELR_EL1	PRSELR		PRSELR_EL2	HPRSELR
HCR_EL2	HCR/HCR2		SCTLR_ELx	SCTLR/HCTLR
TCR_ELx	New		VSTCR_EL2	New
VTCR_EL2	New		VSCTLR_EL2	VSCTLR
TTBR0_EL1	New		MAIR_EL1	MAIR0/MAIR1
			MAIR_EL2	HMAIR0/HMAIR1

Armv8-R AArch64 has the option to implement the VMSAv8-64 at stage 1 of EL1&0 translation
regime. So, the TTBR0_EL1 is added to hold the base address of the translation table for the initial
lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0
translation regime, and other information for this translation regime. TCR_ELx, VSTCR_EL2 and
VTCR_EL2 registers are added to control the translation regime.

For virtualization, instead of VSCTLR, VSCTLR_EL2 provides configuration information for
VMSAv8-64 and PMSAv8-64 virtualization using stage 2 of EL1&0 translation regime.

5 Security

Armv8-R AArch32 only supports a single Security state, Non-secure state. But Armv8-R AArch64 only supports a single Security state, Secure. Armv8-R AArch64 does not support EL3, always executes code in Secure state with EL2 as the highest Exception level. It's not possible to switch to Normal state. EL0, EL1 and EL2 all run in Secure state.

Table 5-1: Secure State Difference

	Armv8-R AArch32	Armv8-R AArch64
EL0	Normal	Secure
EL1		
EL2		

Armv8-R AArch32 could initiate a Normal memory access, couldn't initiate a Secure memory access.

Armv8-R AArch64 could initiate a Normal memory access or Secure memory access by setting the MMU or MPU for memory attribute.

Armv8-R AArch64 adopts FEAT_SEL2 feature from Armv8.4-A architecture extension and introduces it as the PMSAv8-64 based architecture features. The protection region configuration registers for EL1 and EL2 MPUs, PRLAR_EL1 and PRLAR_EL2, are extended to include the NS bit that has the same behavior as the NS bit in the translation table descriptor of VMSAv8-64. Security controls for translation table walks are not supported in PMSAv8-64. The NS bit in the PRLAR_EL1 and PRLAR_EL2 specifies whether the output address is in the Secure or Non-secure address space. Each protection region can be independently configured to the Secure or Non-secure address space. PMSAv8-32 does not have the capability to be configured to access the Secure address space.

Armv8-R AArch64 also supports the VMSAv8-64 memory system architecture. The following table shows the configuration bits required to implement Secure and Normal state in VMSAv8-64 and PMSAv8-64:

**Table 5-2: Armv8-R AArch64 PMSA&VMSA configuration**

VMSAv8-64	PMSAv8-R AArch64
NS bit in translation table	PRLAR_EL1.NS/PRLAR_EL2.NS
NSTable bit in translation table	NA
VSTCR_EL2.RW	NA
VSTCR_EL2.SA	VSTCR_EL2.SA
VTCR_EL2.NSW	NA
VTCR_EL2.NSA	VTCR_EL2.NSA

For each specific register description please refer to Armv8-R AArch64 Supplement Section
G1&G2.

6 Virtualization

Armv8-R AArch32 implements a two level MPU structure for virtualization. For RTOS or Application memory accesses from EL0 or EL1 are protected by the EL1 MPU stage1 checking and EL2 MPU stage2 checking.

Figure 6-1: Two level MPU in Armv8-R AArch32
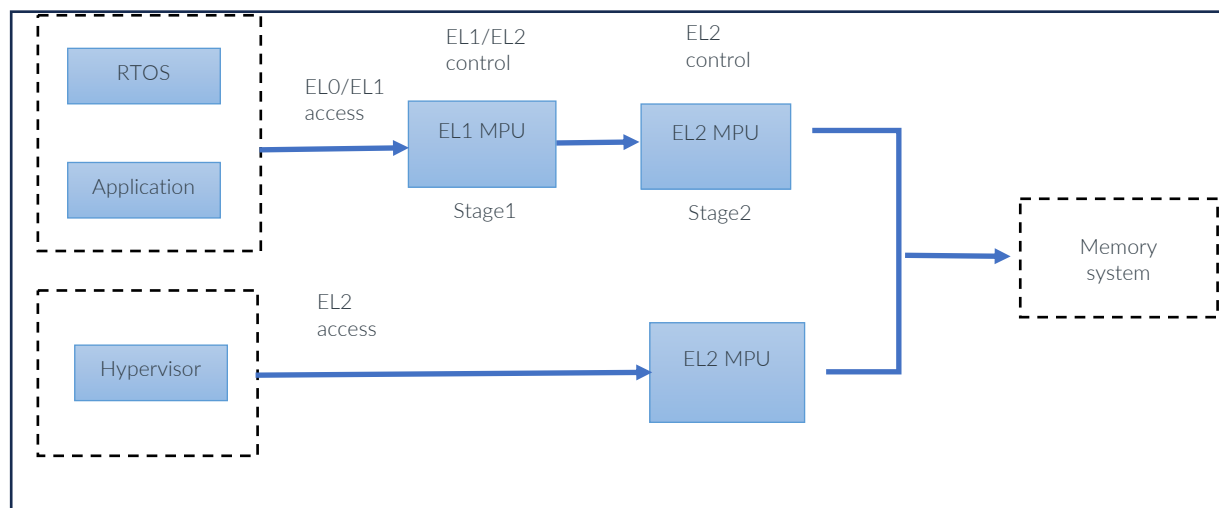
Same with Armv8-R AArch32, Armv8-R AArch64 also implements a permission-based containerization by introducing a stage 2 translation using an MPU controlled by a hypervisor running at EL2. The MPU does not perform address mapping and only checks permissions. But Armv8-R AArch64 also optionally supports the VMSAv8-64 at EL0/1. The guest OS could make use of the MMU to perform address mapping at the stage 1 of the Secure EL1&0 translation regime.

Figure 6-2: Two level MPU/MMU in Armv8-R AArch64
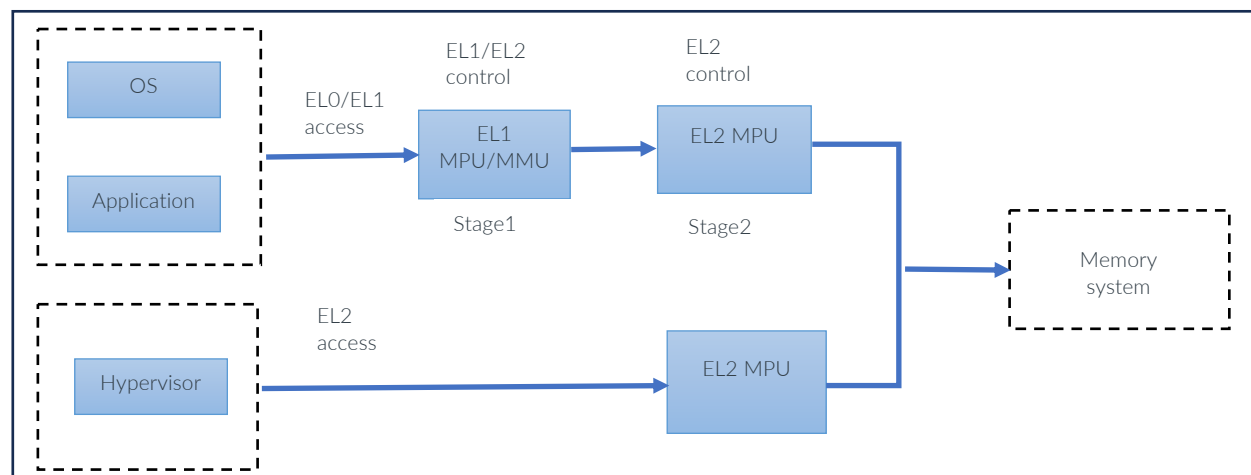
7 Compiler and Optimization

Armv8-R AArch32 and Armv8-R AArch64 are both supported by Arm Compiler for Embedded (Armclang), which is based on LLVM Clang. Other third-party compilers may also support Arm v8-R32/R64, such as GCC, IAR, CLANG, etc.

7.1. Compiler

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 Operating System (RTOS) applications. During the migration from Arm v8-R AArch32 to Arm v8-R AArch64, you need to change some compiler parameters such as CPU architecture and CPU type to adopt current architecture/core, for example:

Table 7-1: Example of Compiler options migration

Compiler option	Cortex-R52	Cortex-R82
--target	arm-arm-none-eabi	aarch64-arm-none-eabi
-march	armv8-r	armv8-r
-mcpu	cortex-r52	cortex-r82
-mfloat-abi	hard/soft	N/A

The following table shows the compiler options to generate different instruction sets.

Table 7-2: Compiler options for ISAs

Instruction Set	Compiler option for Architecture	
	Armv8-R AArch32	Armv8-R AArch64
A32	--target=arm-arm-none-eabi -marm	N/A
T32	--target=arm-arm-none-eabi -mthumb	N/A
A64	N/A	--target=aarch64-arm-none-eabi

7.2. Building for Armv8-R AArch64 target without hardware FP

AArch64 ABI assumes that Hardware Floating-Point is always available on AArch64. It means the compiler or library can make use of vector instructions to optimize memory manipulation. But Armv8-R AArch64 CPUs have the RTL configuration without Floating-Point, which is different with Armv8-A AArch64 CPUs. Since Arm Compiler 6.22, it enables the feature support for Armv8-R

AArch64 CPUs without FPU. To build an application for an Armv8-R AArch64 target without
hardware floating-point support, you must:

- Compile with an -march or -mcpu option for Armv8-R AArch64 that specifies the +nofp
  feature. For example, either -march=armv8-r+nofp or -mcpu=cortex-r82+nofp.

- Compile with -mabi=aapcs-soft.

- Link with –cpu=8-R.64 –fpu=SoftVFP.

If your application includes assembly code, assembling with +nofp reports an error if your assembly
code contains floating-point instructions. Therefore, we recommend that you assemble with both
+nofp and -mabi=aapcs-soft. The -mabi=aapcs-soft is not supported for C++ source language
modes.

In summary:

- Armv8-A/R with or without hardware floating-point is supported in AArch32 state.

- Armv8-A without hardware floating-point is not supported in AArch64 state.

- Armv8-R AArch64 state implementations with hardware floating-point are fully supported.

- Armv8-R AArch64 state implementations without hardware floating-point are only supported
  in C and assembly source language modes. C++ source language modes are not supported for
  such targets.

There is an example to build for an Armv8-R AArch64 target without hardware floating-point:

1. Create the file main.c containing the following C Code:

```c
#include <stdio.h>
#include <math.h>
__attribute__((noinline)) void test_nofp(float a, float b)
{
    printf("%1.1f + %1.1f = %1.f\n", a, b, a + b);
    printf("floorf(%1.1f) = %1.f\n", a, floorf(a));
    printf("floorf(%1.1f) = %1.f\n", b, floorf(b));
}
int main(void)
{
    puts("Hello, world!");
    test_nofp(2.7f, -2.3f);
    return 0;
}
```

2. Compile and link with the following command:

```
armclang –target=aarch64-arm-none-eabi -march-armv8-r+nofp –
mabi=aapcs-soft -O1 -Wl, --cpu=8-R.64 -Wl, --fpu=SpftVFP main.c -o
aarch-r.axf
```

3. Run the image on a suitable target. The image displays:

```
Hello, world!
2.7 + -2.3 = 0.4
floorf(2.7) = 2.0
floorf(-2.3) = -3.0
```

4. Run fromelf to display the disassembly:

```
fromelf –disassmeble aarch640r.axf
```

In the disassembly, you can see that no floating-point is used because:

- There are no FP registers for the test_nofp() function or main() function.

- There are no FP registers for the floorf() library function.

Proprietary notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and Arm Limited ("Arm") or the terms of the agreement between you and the party authorized by Arm to disclose this document to you.

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. **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, without obtaining Arm's prior written consent, you will not use or permit others to use the information: **(i)** for the purposes of determining whether the subject matter of this document infringes any third party patents; **(ii)** for developing technology or products which avoid any of Arm's intellectual property; **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party.

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.

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 the 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.

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
01	June 5, 2024	Non-Confidential	First 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**.

Table 1: Issue 1.0-01

Change	Location
First release for version 1.0	-

Conventions

Document-specific conventions

- In this document wherever only Cortex-R52 is mentioned, it also means that it refers to both Cortex-R52 and Cortex-R52+.
- In this document wherever only Cortex-R82 is mentioned, it also means that it refers to both Cortex-R82 and Cortex-R82AE.

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: **https://developer.arm.com/glossary**.

Typographical conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	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` <u>underline</u>	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. For example: `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.
⚠️ Caution	We recommend the following. If you do not follow these recommendations your system might not work.
⚠️ Warning	Your system requires the following. If you do not follow these requirements your system will not work.

Danger	You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.
Note	This information is important and needs your attention.
Tip	This information might help you perform a task in an easier, better, or faster way.
Remember	This information reminds you of something important relating to the current content.

Useful resources

This document contains information that is specific to this product. 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 architecture and specifications	Document ID	Confidentiality
Arm Architecture Reference Manual for A profile Architecture	DDI0487J_a	Non-Confidential
Armv8-R AArch64 Supplement	DDI0600A_c	Non-Confidential
Armv8-R AArch32 Supplement	DDI0568A_b	Non-Confidential