Armv7-M and Armv8-R

Software Migration Guide

Non-confidential

Issue 01
109692

Armv7-M and Armv8-R
Software Migration Guide

Release information

Document history

Issue	Date	Confidentiality	Change
01	29-February-2024	Confidential/Non-Confidential	

Non-Confidential Proprietary Notice

Confidentiality Status

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 Armv7-M and Armv8-R, 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

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

Contents

1. Introduction

1.1. Intended audience

This guide is useful for partners looking to reuse their existing software based on Armv7-M and Armv8-R based designs. This is also useful for new projects using Cortex-M and/or Cortex-R based designs.

1.2. Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

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

Typographical conventions

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. 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 bold	Language keywords when used outside example code.
monospace underline	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. 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	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.

Convention	Use
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better, or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.3. Useful resources

Arm products	Document ID	Confidentiality
Arm Cortex-R Series Programmer's Guide	den0042	Non-confidential
Arm Cortex-M3 Processor Technical Reference Manual	100165	Non-confidential
Arm Cortex-M4 Processor Technical Reference Manual	100166	Non-confidential
Arm Cortex-M7 Processor Technical Reference Manual	ddi0489	Non-confidential
Arm Cortex-R52 Processor Technical Reference Manual	100026	Non-confidential

Arm architecture and specifications	Document ID	Confidentiality
Armv7-M Architecture Reference Manual	ddi0403	Non-confidential
Arm Architecture Reference Manual for A-profile architecture	ddi0487	Non-confidential
Arm Architecture Reference Manual Supplement - Armv8, for the Armv8-R AArch32 architecture profile	ddi0568	Non-confidential

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.
Adobe PDF reader products can be downloaded at http://www.adobe.com.

2. Overview

With the trend towards central compute in vehicle architecture from the existing discrete ECU architecture, in the near-term future there will be increasing interest in porting the existing matured software from multiple discrete ECUs (based on Cortex-M) onto a single Cortex-R based zone controller (with virtualization).

Scalability both in terms of hardware and software will be beneficial for partners, as partners will be able to scale across multiple products without reinvesting in development from scratch.

This is a guide for partners on the best practices for software development that can help to scale and migrate software across architectures. There are architectural differences between Cortex-M and Cortex-R cores; however, the intention is to capture the differences and suggestions on how software needs to be adapted so that it can be scaled and migrated to a different Arm architecture.

This guide presents general guidelines on how to migrate software from Cortex M to Cortex-R.   This whitepaper This whitepaper supports you to port original Cortex-M software to Cortex-R, not to enable all features of Cortex-R. We will mainly focus on Armv7-M processors (Cortex-M3, Cortex-M4, Cortex-M7), and Armv8-R processors (Cortex R52, Cortex-R52+). To be more targeted in this whitepaper, unless otherwise specified, Cortex-R mainly refers to Cortex-R52 and Cortex-R52+, not Cortex-R82 implemented with the Armv8-R architecture.

Because Cortex-R52 and Cortex-R52+ share the same instruction set and they are software compatible, references to Cortex-R52 in this document also apply to Cortex-R52+.

3. ISA

There are three different instruction sets used by Armv7-M and Armv8-R architecture.

3.1. Instruction set

- T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.
- A32 This is a fixed-length instruction set that uses 32-bit instruction encodings. Before the introduction of Armv8, it was called the Arm instruction set.
- A64 The A64 instruction set provides access to 64-bit wide integer registers and data operations. Instruction opcodes, however, are still 32-bit long, not 64-bit.

Armv7-M processors support T32 instruction set. As Figure-1 shows, different Arm v7-M processors can support a certain subset of the T32 instruction set.

Cortex-R52 and Cortex-R52+ implement T32 and A32 instruction sets.

Cortex-R82 supports the A64 instruction set, but not the A32 or T32 instruction sets.

Additionally, Arm v8-M supports a variant of the T32 instruction set, which is a superset of T32.

Figure 3-1 and Figure 3-2 show the instruction set support status among different architectures.

Figure 3-1: Instruction set of T32 in Armv7-M

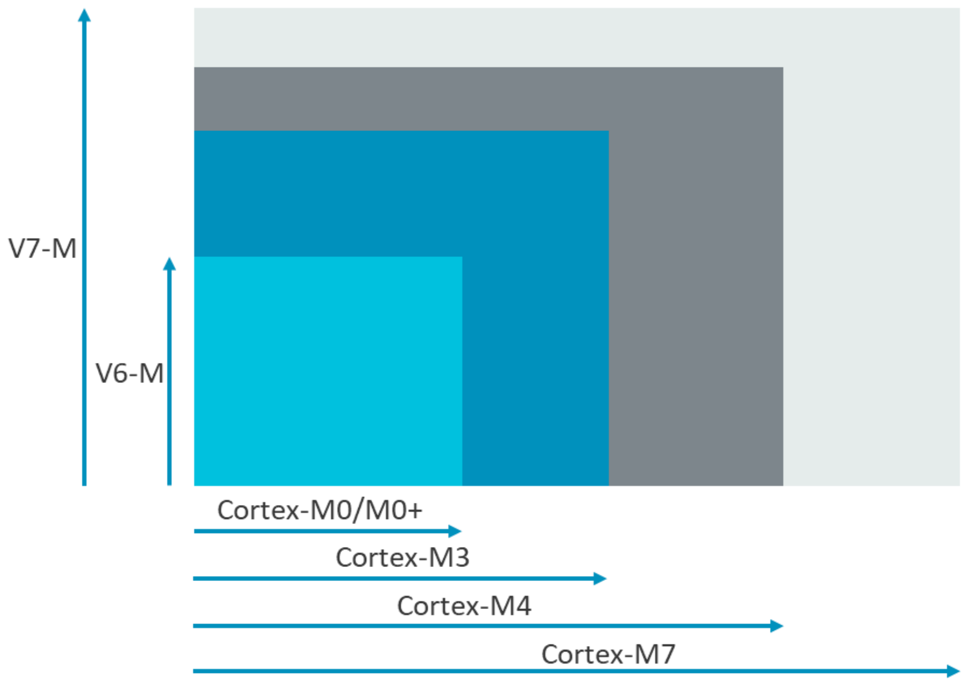


Figure 3-2: Instruction set among different architectures

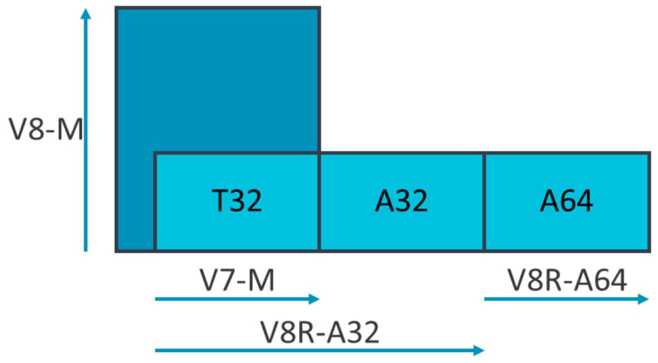


Figure 3-2 shows that Armv8-R A32 supports all instructions of Armv7-M. You can compile Armv7-M software (C language or assemble language source code) directly in Armv8-R A32 environment if the tools are compatible. There might be minor instruction differences such as SWP/SWPB, which is deprecated in Cortex-R52, so you need replace them with equivalent instructions such as `LDREX/STREX`.

A bigger gap exists if you are mitigating from Armv7-M to Armv8-R A64. If you have written any hand-coded assembler (in the T32 instruction set), replace this with equivalent instructions from the A64 instruction set and rebuild the software. You also need modify and check data types (size) and instructions that have no equivalent. For the source code implemented in C language, it might also be necessary to modify and check the data type, such as from `int` to `uint32`. Recompiling the C code with the correct compiler switches automatically generates the appropriate A64 instructions. See section 9.1 Compiler for more details. .

Note

- Armv8-M implements a superset of the T32 ISA. You might need to substitute any instructions which do not have an equivalent, and re-assemble if you mitigate from v8-M to v8-R AArch32.
- Some v8-M cores such as Cortex-M55 implement Helium technology, which is the M-profile Vector Extension (MVE). You might need additional work if you are mitigating from v8-M with MVE to v8-R. Due to the instruction difference, it is better to avoid using C intrinsic or assembler instructions in your source code. You might need to re-write key vector routines including all vector assembly or intrinsic functions.

3.2. Register set

Both Armv7-M and Armv8-R share the same register definition and size of `R0-R15`, but they have some different register banking among processor modes.

ARMV7-M only banks stack pointer as Main Stack Pointer (MSP) and Process Stack Pointer (PSP) modes.

Figure 3-3 shows that Armv8-R banks more registers (`SP`, `SPSR`, `LR`, and `R8-R12` in FIQ mode) among different processor modes.

Figure 3-3: Registers banking of Armv7-M and Armv8-R

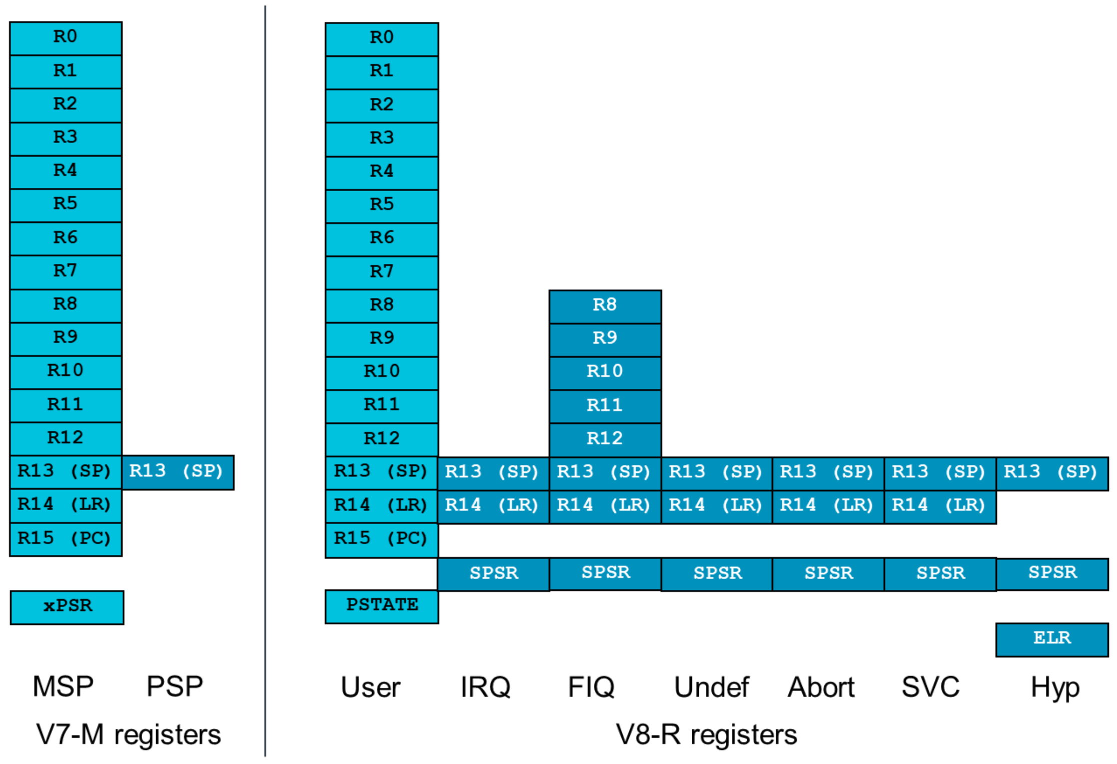


Figure 3-4 and Figure 3-5 show that there are also some differences between `xPSR` in Armv7-M and `PSTATE` in Armv8-R. You need to be aware of these differences when you migrate your software.

Figure 3-4: PSR register in Armv7-M

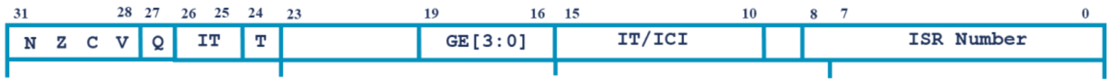
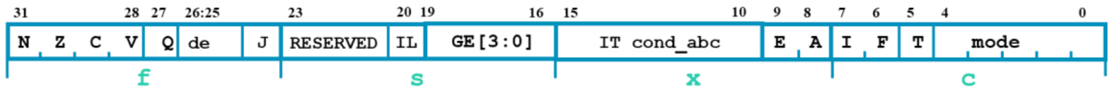


Figure 3-5: PSR register in Armv8-R



4. Exception model

The exception model focuses on the behavior of Armv7-M and Armv8-R when the core enters exception, such as interrupt.

4.1. Exception mode

Armv7-M processor has two privileged levels and two processor modes.

4.1.1. Thread Mode

Thread mode is used for application execution. It can use either Main Stack Pointer (MSP) or Process Stack Pointer (PSP), and it also can be either in privileged mode or unprivileged mode. Cortex-M is in MSP and privileged mode at reset.

4.1.1.1. Handler Mode

Handler mode is used for exception handling. This mode is entered when an exception occurs. It only uses Main Stack Pointer (MSP) and is always in privileged mode.

Table 4-1 shows the processor modes in Armv7-M.

Table 4-1: Modes of Armv7-M

Mode	Description
Handler Privileged	Used to handle exceptions
Thread Privileged	Mode under which most RTOS run
Thread Unprivileged	Mode under which most tasks run

Armv8-R processor has three exception levels (`EL0` is unprivileged level, `EL1` and EL2 are privileged level) and 8 processor modes available, as Table 4-2 shows.

Table 4-2: Modes of Armv8-R

Mode	Description	
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	

Mode	Description	
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

Although there are lots of exception mode differences between Armv7-M and Armv8-R, and mapping equivalent exception modes from Armv7-M to Armv8-R is still the simplest way to migrate software. Table 4-3 shows an example for these equivalences.

Table 4-3: Equivalent mode from Armv7-M to Armv8-R

Mode of Armv7-M	Equivalent mode of Armv8-R
(no equivalent)	Hypervisor
Handler	Supervisor
	FIQ
	IRQ
	Abort
	Undef
(no equivalent)	System
Thread Unprivileged	User

Note

After mapping equivalent exception patterns, your software might still require some modification. The complexity of modification is often related to your software architecture, such as how deeply your software incorporates unusual patterns.

4.1.2. Exception vector table

Table 4-4 shows the V7 M exception vector table has 16+N entries, and each entry consists of addresses, not instructions.

Table 4-4: Exception vector table of Armv7-M

Address	Description	Exception number
0x40+4N	External interrupt N	16+N
…	…	
0x40	External interrupt 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	(Reserved)	13
0x30	Debug Monitor	12
0x2C	SVCall	11
0x28	(Reserved)	10
0x20-0x24	(…)	…
0x1C	(Reserved)	7
0x18	UsageFault	6
0x14	BusFault	5
0x10	MemManage	4
0x0C	HardFault	3
0x08	NMI	2
0x04	Reset	1
0x00	SP_main	N/A

Table 4-5 shows that there are two vector tables in Armv8-R, one for EL1 and one for EL2. In some software implementations, each guest OS can have its own EL1 vector table, and these vector tables are managed by hypervisor. In contrast to Armv7-M, each Armv8-R vector table has only 8 entries. Each entry consists of instructions, not addresses, which means one 32-bit instruction or two 16-bit instructions, usually direct branch instruction.

Table 4-5: Exception vector table of Armv8-R

0x1C	FIQ (EL1 Vector table)	FIQ (EL2 Vector table)
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)

0x1C	FIQ (EL1 Vector table)	FIQ (EL2 Vector table)
0x08	Software Interrupt	HVC (from Hypervisor Mode)
0x04	Undefined Instruction	Undefined Instruction (From Hypervisor Mode)
0x00	(Reset)	Reset

There is no simple equitant map between the exceptions of Armv7-M and Armv8-R because they use different exception terminology. The *Arm® Architecture Reference Manual for A-profile architecture, section G1.16 - AArch32 state exception descriptions* which describes the possible reasons for each exception entry.

4.1.3. Exception handling

When an exception is received, the processor interrupts the execution of current instruction flow. To minimize interrupt latency, the processor allows exceptions during the execution of multi-cycle instruction.

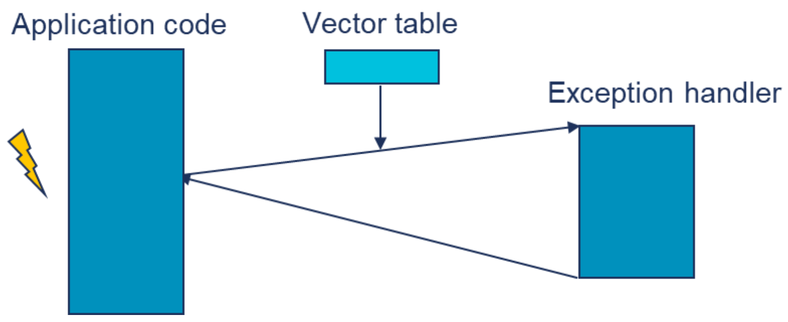
Because the Armv8-R exception architecture is very different from Armv7-M, the exception handling process (both in hardware and software) is also different between Armv7-M and Armv8-R. Armv7-M is a hardware-implemented architecture that handles the context save/restore by hardware so that there is no software overhead for exception entry and exit. However, in Armv8-R, software has the responsibility of context save and restore. Typically, software needs a wrapper, named as top-level handler, to handle them.

4.1.3.1. Exception handling in Armv7-M

Figure 4-1 shows that the processor executes the following process to handle exceptions in Armv7-M.

1. The processor state is automatically saved to the current stack, which includes `PC`, `R0-R3`, `R12`, `LR`, `xPSR`. The Link Register is modified for interrupt return.
2. The processor accesses the vector table and the address of the `ISR` is read from the vector table.
3. The exception handler routine executes in handler mode using the main stack.
4. The exception handler returns to main (assuming that there is no nesting).

Figure 4-1: Exception handling in Armv7-M

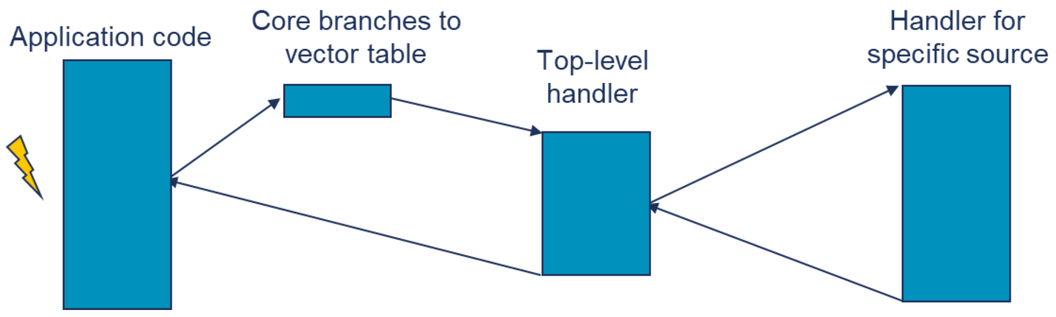


4.1.3.2. Exception handling in Armv8-R

Figure 4-2 shows that the processor executes the following process to handle exceptions in Armv8-R.

1. The processor state is automatically saved to the SPSR, and the current processor state is updated. The link register is modified for interrupt return if the exception is handled in `EL1`.
2. The processor executes from the corresponding entry of the vector table, which typically is a branch instruction to the top-level handler.
3. The top-level handler saves current context, and then calls the second-level exception handler.
4. The second-level handler routine is executed, typically in a C language environment, and then returns to the top-level handler.
5. The top-level handler restores current context, and then returns to main (assuming that there is no nesting).

Figure 4-2: Exception handling in Armv8-R



Some existing RTOS support both Cortex-M and Cortex-R cores. Typically, RTOS code includes architectural support and implements software code to support the corresponding exception model. Using these existing RTOSs in your project can significantly reduce porting effort.

If your Armv7-M software source code is bare-metal or your RTOS does not support Armv8-R, you might need to rework privileged code from Armv7-M to Armv8-R, such as SVC and return procedures. You can port the exception handling code from an existing RTOS, which supports Armv8-R, to your current software.

4.2. Reset

Reset is a special exception in both Armv7-M and Armv8-R.

4.2.1. Reset state in Armv7-M

Armv7-M always resets at privilege thread mode using *Main Stack Pointer* (MSP).

4.2.2. Reset state in Armv8-R

Armv8-R always resets at EL2 in hypervisor mode.

The software between processor reset and functional tasks execution is also called boot code. The boot code sets up the basic environment in which functional tasks run.

Because of the differences between the exception models in Armv7-M and Armv8-R, migrating from Armv7-M to Armv8-R requires rewriting most of the boot code.

Fortunately, most of the startup code is the same among different Armv8-R platforms, which means that, it only needs minor effort if you take another Armv8-R platform as reference (for example, open-source Armv8-R platform).

Furthermore, the workload decreases significantly if you are using existing RTOS which supports Armv8-R because RTOS might take care of all and makes you ignore that there is a boot.

5. Interrupt

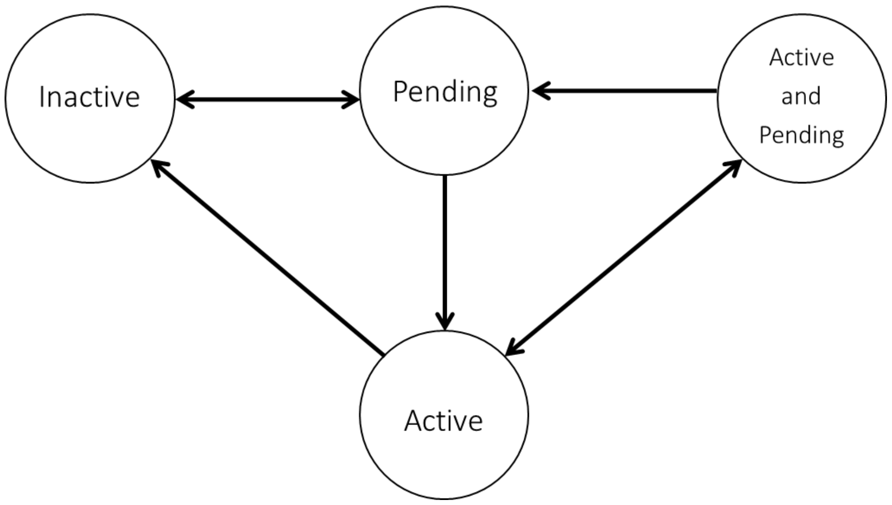
Each interrupt has four different states in both Armv7-M and Armv8-R.

- Inactive: Interrupt is not active and not pending.
- Pending: Interrupt is asserted but not yet being serviced.
- Active: Interrupt is being serviced but not yet complete.
- Active & Pending: Interrupt is both active and pending.

Figure 5-1 shows that the interrupt state can change from the current state to another state. For example:

- Inactive   Pending: When the interrupt is asserted.
- Pending   Active: When the interrupt is under processing by a core.
- Active   Inactive: When the core finished interrupt handling.

Figure 5-1: Interrupt state machine



The state machine can have more paths in a real processor. We simplified it because this simplification does not affect software migration.

5.1. Interrupt controller

Armv7-M cores integrated NVIC as its interrupt controller, but Cortex-R cores have different types of interrupt controllers:

- Cortex-R4 and Cortex-R5: VIC
- Cortex-R7 and Cortex-R8: built-in GIC-v1

- Cortex-R52: built-in GIC-v3
- Cortex-R82: external GIC-v3.2

Using an existing off-the-shelf OS to manage interrupts is the most efficient way to migrate from Armv7-M to Armv8-R.

Both Armv7-M and Armv8-R abstract the same interrupt state machine, and your OS can abstract the interrupt routing control functions and replace them with the Armv8-R interrupt controller implementation. There might be some minor feature differences, such as the number of available priority levels. In this case, you need to modify them manually.

GIC-v3 introduces more features, for example:

- Interrupt group: asserts interrupt through IRQ signal or FIQ signal.
- Interrupt virtualization: asserts vIRQ signal to the core.
- Software generated interrupt: sends interrupt to other cores.

If you want to enable the corresponding software APIs while migrating your software, you must implement them.

5.2. Interrupt handling

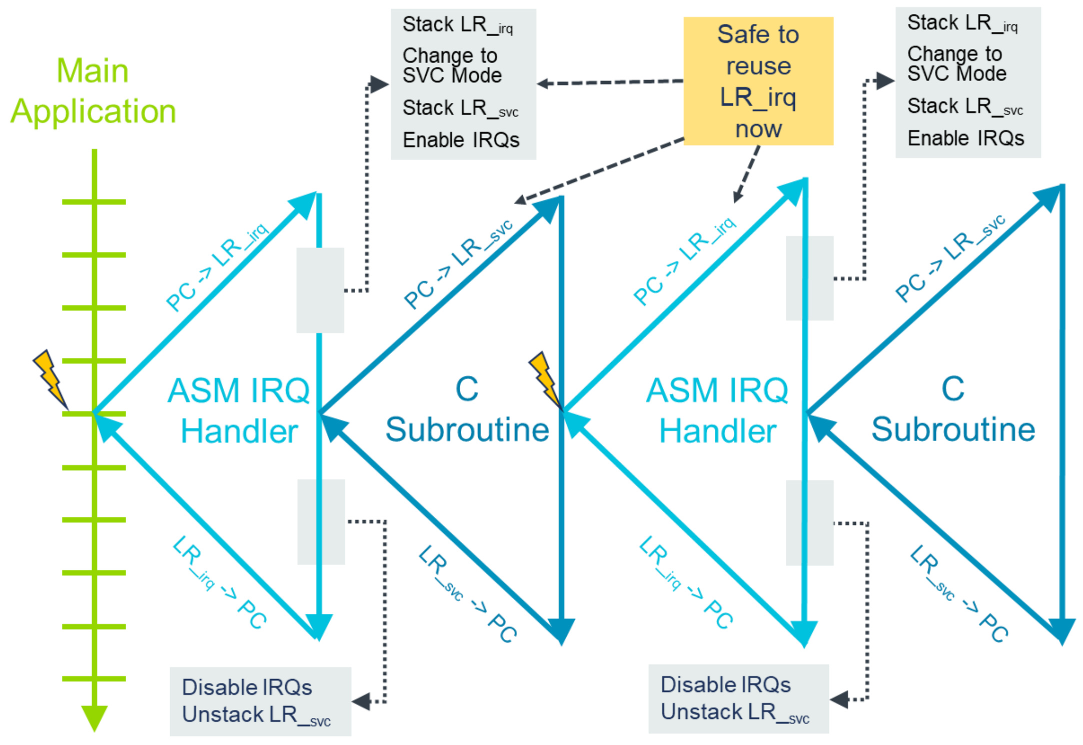
The processor enters an exception in both Armv7-M and Armv8-R. As mentioned before, Armv7-M implements a hardware-implemented exception mechanism controlled by hardware, but such work is mainly maintained by software in Armv8-R, which causes more complexity when your software handles the interrupt.

Using an existing off-the-shelf OS to manage interrupts is the most efficient way to migrate from Armv7-M to Armv8-R.

If you are using bare-metal software, your assembler stubs in Armv8-R need to recreate the Armv7-M stacking and vectoring, combined with trapping and emulation for exception return. Interrupt entry and exit routines need to be re-written.

Your software can have a nested-interrupt requirement which is important for interrupt preemption support. This feature is naturally supported by NVIC in Armv7-M, but you must enable it through software in Armv8-R, for example, as Figure 5-2 shows.

Figure 5-2: Nested interrupt support in Armv8-R



The following sequence demonstrates nested-interrupt support in Armv8-R:

1. Save `LR_irq` and `SPSR_irq` to the SVC mode stack.
2. Switch to SVC mode.
3. Store remaining AAPCS registers on to the SVC mode stack.
4. Store adjustment and `LR_svc` to stack.
5. Identify and clear interrupt source.
6. Enable interrupt.
7. Call interrupt handler routine.
8. Disable interrupt.
9. Restore `LR_svc` and unadjust the stack.
10. Restore AAPCS registers and return from the SVC mode stack.

6. Virtualization

Because Armv8-R adds virtualization support and always boots from hypervisor, there are two possible cases that can occur while you are migrating from Armv7-M to Armv8-R.

You might want to directly migrate your Armv7-M software to Armv8-R without virtualization support. In this case, the boot code of Armv8-R must be modified to ensure EL2 is never used after the boot stage.

These are the minimum steps to be completed before leaving EL2:

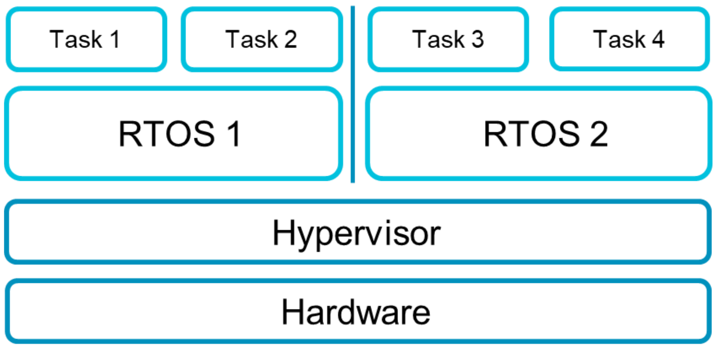
1. Configure HCR (Hypervisor Control Register) to disable HVC.
2. Configure HACTLR (Hypervisor Auxiliary Control Register) to allow EL1 access to implementation specific registers.
3. Set `VBAR` (Vector Base Address Register) to the address of EL1 vector table.
4. Set `ELR_hyp` (Exception Link Register Hyp Mode) to the entry of EL1 code.
5. Execute the `ERET` instruction.

Another possible solution is to treat the Armv7-M workload as a single guest under a hypervisor. This is suitable if your Armv8-R project has bigger scope and takes advantage of virtualization support.

If you want to enable virtualization in Armv8-R, you need to implement additional software, such as a hypervisor (VMM, Virtual Machine Manager).

Figure 6-1 shows a typical Armv8-R software architecture with virtualization enabled.

Figure 6-1: Virtualization enabled software in Armv8-R



Generally, the task or Real-Time Operating System (RTOS) might not be aware of the existence of virtualization, so most applications can run directly on the virtualization-enabled environment, but some system-related software might still require some modification, such as system controls and device drivers.

Arm also provides an example of virtualization based on Cortex-R52. You can get the example from the Armv8-R virtualization manual on the Arm Developer website.

7. System registers

Most of the system features are controlled by system registers. The system registers are different between Armv7-M and Armv8-R, not only because they have different features, but also because some features are implemented differently between Armv7-M and Armv8-R.

System registers can be accessed in one of the three ways:

- MRC/MCR: access through co-processor
- MRS/MSR: access processor state
- LDR/STR: memory mapped system registers

For example, in Cortex-R52:

- Read PAR[31:0] into Rt:
  ```
  MRC p15, 0, <Rt>, c7, c4, 0
  ```
- Write Rt to PAR[31:0]. PAR[63:32] are unchanged
  ```
  MCR p15, 0, <Rt>, c7, c4, 0
  ```
- Read 64-bit PAR into Rt (low word) and Rt2 (high word)
  ```
  MRRC p15, 0, <Rt>, <Rt2>, c7
  ```
- Write Rt (low word) and Rt2 (high word) to 64-bit PAR
  ```
  MCRR p15, 0, <Rt>, <Rt2>, c7
  ```

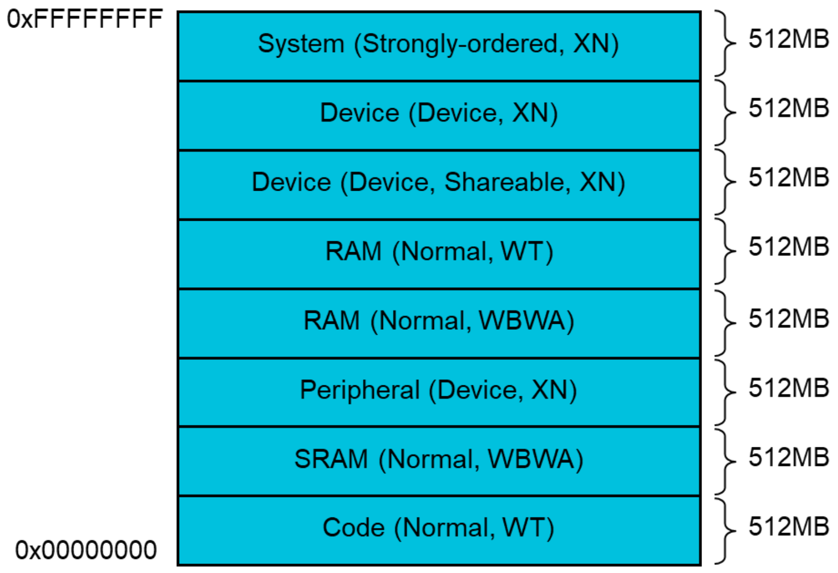
A possible way to migrate from Armv7-M to Armv8-R is that software abstracts the system control functions or register-field access into a library, so that you only need to replace the corresponding registers in Armv7-M with the equivalent registers in Armv8-R. Because of the differentiation of the features between Armv7-M and Armv8-R, sometimes you might need to update all system register access routines, or some C code in Armv7-M might need to be re-written in assembly code in Armv8-R.

Your existing Cortex-M software can use CMSIS which abstracts most of system registers accessing into feature-based functions. CMSIS is not available for Cortex-R, but you can still implement the corresponding functions with Armv8-R system registers if your Armv7-M software uses CMSIS. This will save effort in your migration.

8. Memory model

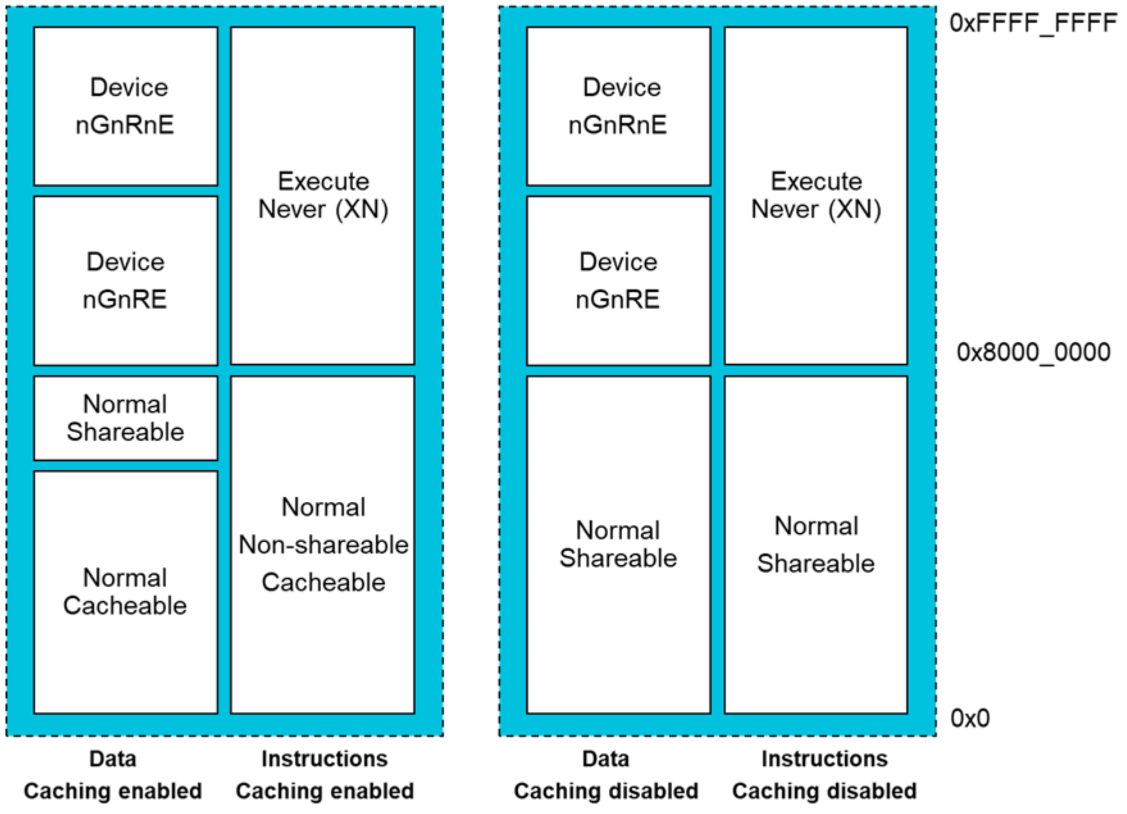
Armv7-M is a memory-mapped architecture with PMSA-v7 (Protected Memory System Architecture) support. The memory is divided into 8 x 512MB regions, as Figure 8-1 shows.

Figure 8-1: Default memory map in Armv7-M



Armv8-R has a different memory map view, which is implemented to support PMSA-v8. The memory attributes are different according to access origin and caching policy, as Figure 8-2 shows.

Figure 8-2: Default memory map in Armv8-R



8.1. Memory type

There are three mutually exclusive memory types defined in Armv7-M.

8.1.1. Normal memory

Normal memory is the most flexible memory type. It is suitable for different types of memory, for example, ROM, RAM, Flash, and SDRAM. Caches and write buffers are permitted. Accesses to normal memory can be restarted.

8.1.2. Device memory

Device memory is suitable for peripherals and I/O devices. Caches are not permitted, but write buffers are still supported. Accesses to device memory must not be restarted.

8.1.3. Strongly ordered

Strongly-ordered memory is like device memory but buffers are not supported.

There are two mutually exclusive memory types defined in Armv8-R.

8.1.4. Normal memory

Like Armv7-M, the Normal memory type is used for code and most data regions. It allows the core to re-order, repeat, and merge accesses.

8.1.5. Device memory

The device type is used for regions where accesses can have side-effects. Typically, this type is only used for peripherals.

There are four variants of device memory in Armv8-R:

- Device-nGnRnE most restrictive
- Device-nGnRE
- Device-nGRE
- Device-GRE least restrictive

Replacing the memory type in Armv7-M with the equivalent memory type in Armv8-R is the simplest way to migrate software, as Table 8-1 shows.

Table 8-1: Equivalent memory type from Armv7-M to Armv8-R

Memory type in Armv7-M	Equivalent type in Armv8-R
Normal	Normal
Device	Device - nGnRE
Strongly-ordered	Device - nGnRnE

In addition to the memory type, many other factors need to be considered during the memory model migration, such as cache policy, and access rights. See section 8.2 Memory protection unit for more details.

8.2. Memory protection unit

The Memory Protection Unit (MPU) provides basic memory management by allowing attributes to be applied to different address regions.

Armv7-M supports Protected Memory System Architecture (PMSAv7), and Armv8-R supports PMSAv8.

Armv7-M has an optional programmable MPU. It allows subdividing the 4GB memory address range into regions, and subdividing regions into subregions. Armv7-M also supports region overlapping. Each memory region is defined by a base address, size, access permissions, and memory attributes.

Armv8-R (Cortex-R52) has two programmable MPUs controlled by EL1 and EL2 . Each MPU allows subdivision of the 4GB memory address range into regions. Each memory region is defined by a base address, limit address, access permissions, and memory attributes.

Most of the memory attributes can be migrated from Armv7-M to Armv8-R easily, but the access permission is not compatible between Armv7-M and Armv8-R.

Armv7-M uses three bits to control access rights, as Table 8-2 shows.

Table 8-2: Access permission in Armv7-M

AP	Unprivileged	Privileged
000	No access	No access
001	No access	Read/Write
010	Read-only	Read/Write
011	Read/Write	Read/Write
100	UNPREDICTABLE	UNPREDICTABLE
101	No access	Read-only
110	Read-only	Read-only
111	Read-only	Read-only

Armv8-R only uses two bits to control access rights, as Table 8-3 shows.

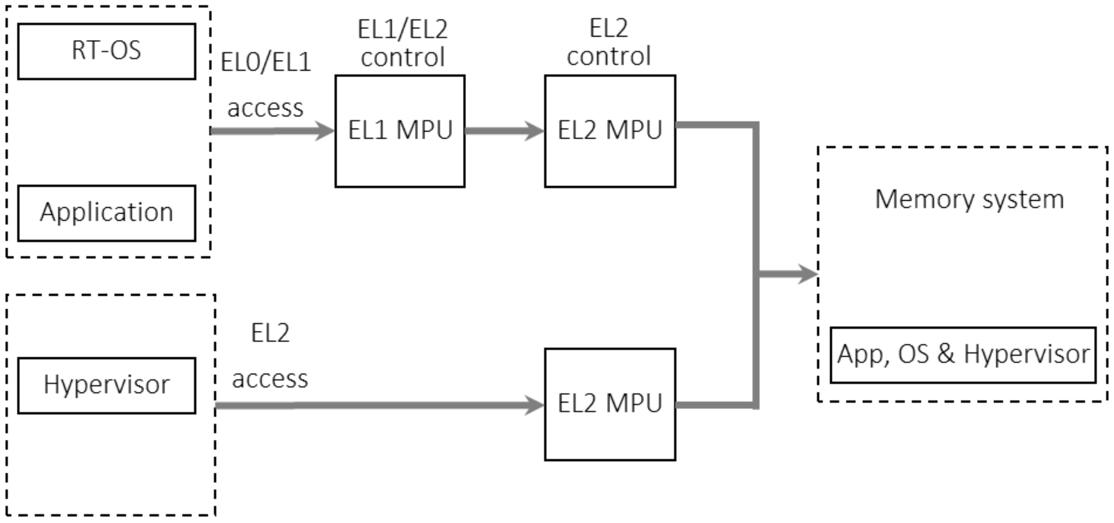
Table 8-3: Access permission in Armv8-R (Cortex-R52)

AP	Unprivileged (EL0)	Privileged (EL1)
00	No access	Read/write
01	Read/write	Read/write
10	No access	Read-only
11	Read-only	Read-only

Some combinations do not exist in the Armv8-R configuration such as read-only in Unprivileged, and read/write in Privileged.

Furthermore, to support virtualization, Armv8-R implements a two-level MPU structure. MPU1 can be used for RTOS and application, MPU2 can be used for Hypervisor. The final attributes are combined from MPU1 and MPU2, as Figure 8-3 shows.

Figure 8-3: Two-level MPU in Armv8-R



The two-level MPU structure decouples the memory management of RTOS and hypervisor. RTOS manages its memory configuration only in MPU1 and hypervisor switches active RTOS only through reconfiguration of MPU2. This mechanism greatly facilitates the implementation of virtualization in your system.

9. Tools

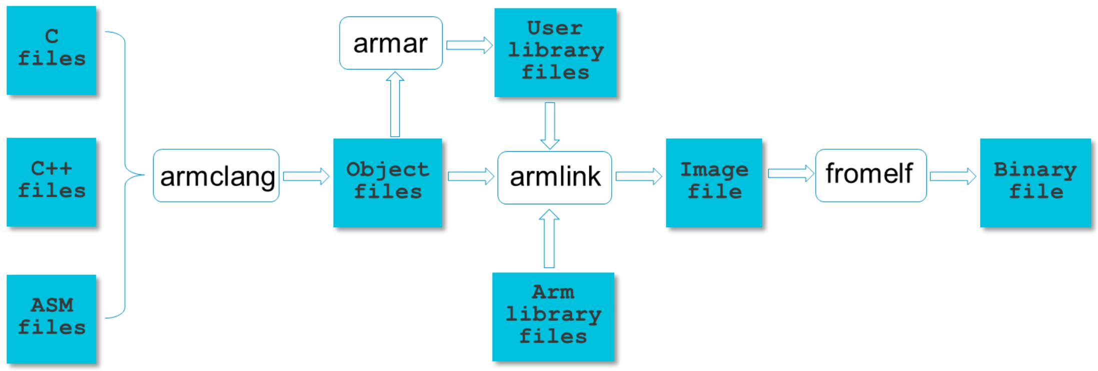
There are many development tools and environments that support both Armv7-M and Armv8-R, such as Arm DS. Using an integrated development environment that supports both Armv7-M and Armv8-R can reduce the workload of migration significantly.

Arm recommends using the Arm Development Studio integrated development environment, which is designed specifically for the Arm architecture with support of multicore debug for Cortex-A, Cortex-R, Cortex–M, and Neoverse Arm CPUs.

In this section, we use the components in Arm DS to demonstrate the compile and link differences of Armv7-M and Armv8-R.

Figure 9-1 shows the embedded application build process of Arm DS.

Figure 9-1: Arm DS embedded application build process



9.1. Compiler

Armv8-R is supported by Arm Compiler for Embedded (Arm Compiler 6), which is based on LLVM Clang. Other third-party compilers might also support Armv8-R such as GCC, IAR, and CLANG.

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 Armv7-M to Armv8-R, you need to change some compiler parameters such as CPU architecture and CPU type to adopt current architecture and core, as Table 9-1 shows.

Table 9-1: Example of compile options migration

Parameter	Cortex-M4	Cortex-R52
`-march`	`armv7-m`	`armv8-r`
`-mcpu`	`cortex-m4`	`cortex-r52`
`-mfpu`	`fpv4-sp-d16`	`neon-fp-armv8 (if neon is present)`
-mfloat-abi	`hard`	`hard`

9.2. Linker

Arm Compiler for Embedded uses scatter-files to maintain its configurations. Because Armv7-M and Armv8-R have different memory maps, you need to modify the corresponding section of scatter file to reflect the Armv8-R memory map.

This example shows a scatter file for Cortex-M4:

```
LOAD_REGION 0x00000000 0x00005000
{
  VECTORS +0 0x400
  {
    exceptions.o (vectors, +FIRST)
  }
  CODE +0 0x4000-0x400
  {
    * (+RO)
  }
  DATA 0x20004000 0x2000
  {
    * (+RW, +ZI)
  }
  ARM_LIB_STACKHEAP 0x20006000 EMPTY 0x1000  {  }
  PROCESS_STACK 0x20007000 EMPTY 0x1000  {  }
  SCS_REGION 0xE000E000 UNINIT 0x1000
  {
    scs.o(.bss.scs_registers)
  }
}
```

This example shows a scatter file for Cortex-R52:

```
LOAD 0x0
{
    CODE +0 0x8000
    {
        startup.o (VECTORS, +First)
        startup.o (RESET)
        * (InRoot$$Sections)
        * (+RO)
    }
    DATA 0x8000 0x4000
    {
        * (+RW,+ZI)
    }
    ARM_LIB_HEAP 0xC000 ALIGN 64 EMPTY 0x1000 {}
    ARM_LIB_STACK    +0 ALIGN 64 EMPTY 4 * 0x4000 {}
    ATCM 0xB0000000 0x4000
    {
    sorts.o (+RO)
    }
}
```

10. Startup

This section summarizes the basic startup flow of Cortex-R52.

10.1. EL2 Vector Table initialization

After reset, program execution starts at the reset vector of EL2 that is defined by the `CFGVECTABLEx[31:5]` signals. Your linker script links the EL2 reset vector to this address. The A32/T32 status is defined by the `CFGTHUMBEXCEPTIONSx pin`, and data endianness configured by `CFGENDIANESSx`. Your software build option must match these configurations. The EL2 vector table initialization code looks like this:

```
EL2_Start:
      B      EL2_Reset_Handler
      B      EL2_Undefined_Handler
      B      EL2_HVC_Handler
      B      EL2_Prefetch_Handler
      B      EL2_Data_Handler
      B      EL2_HypModeEntry_Handler
      B      EL2_IRQ_Handler
      // EL2_FIQ_Handler may follow immediately to avoid branching
```

10.2. EL2 initialization

Before entering EL1, the following EL2 registers might need to be properly configured. Some of these steps are optional. It depends on the type of software residing in EL2. These steps show the typical effort which needs to be done in EL2:

```
Configure Hypervisor System Control Register(HSCTLR)
Configure Hypervisor Stack Pointer(SP_HYP)
Configure Hypervisor Configuration Register(HCR)
Configure Hypervisor Auxiliary Control Register(HACTLR)
Configure and enable EL2 MPU
Enable caches
Configure VBAR
Configure SPSR_HYP to target EL1 SVC_Mode
Configure ELR_HYP to point to the EL1 “reset” code(EL1_Reset_Handler)
Execute ERET
```

You can use the following process to disable EL2 if the software does not employ EL2 during your migration.

```
Configure HCR to disable HVC
Configure HACTLR to enable EL1 access to IMP DEF registers
Configure VBAR to target EL1 vector table
Configure SPSR_HYP to target EL1 SVC_Mode
Configure ELR_HYP to point to the EL1 “reset” code(EL1_Reset_Handler)
Execute ERET
```

10.3. EL1 Vector Table initialization

Typically, the core enters EL1 reset vector entry after executing the instruction `ERET`. If your software supports virtualization, there can be many VMs that exist, and then it is likely that many EL1 vector tables are present. This is normally under the supervision of the hypervisor. This code is an example of the EL1 vector table:

```
EL1_Start:
       B      EL1_Reset_Handler
       B      EL1_Undefined_Handler
       B      EL1_SVC_Handler
       B      EL1_Prefetch_Handler
       B      EL1_Data_Handler
       NOP    ; Reserved vector
       B      EL1_IRQ_Handler
       // EL1_FIQ_Handler may follow immediately to avoid branching
```

10.4. Stack initialization

In EL1_Reset_Handler, it is usually necessary to initialize the stack for each exception mode to enable the software stack. To setup the stack pointer, you can simply enter each mode and assign the appropriate value to the stack pointer. For example, the sample below allocates 512 bytes of stack for Abort, IRQ, and SVC modes, and you can do the same for any other exception mode.

```
// Stack location & size (in bytes)
.equ StackBase,      0x20000000    // Stack at top of memory
.equ ExcStackSize,   512           // 512 bytes per exception type
              LDR    r0, =StackBase      // Base address of all stacks
              CPS    #AbortMode          // Change to Abort mode
              MOV    sp, r0              // Set sp_abt
              CPS    #IRQMode
              SUB    r0, r0, #ExcStackSize
              MOV    sp, r0              // Set sp_irq
              CPS    #SVCMode
              SUB    r0, r0, #ExcStackSize
              MOV    sp, r0              // Set sp_svc
```

10.5. MPU initialization

Cortex-R52 supports 16 MPU regions for EL1 and EL2. EL2 MPU can only be initialized in EL2. EL1 MPU can be initialized in either EL1 or EL2. If your software supports virtualization, there can be many VMs, and then there can be many EL1 MPU configurations that are usually maintained in the hypervisor. The following example shows a typical MPU initialization process, assuming the MPU configuration data is referenced by register `r5`:

```
       LDM r5!, {r1-r4}    // r5 points to MPU configuration
       MCR (H)PRBAR0, r1   // write (H)PRBAR0
       MCR (H)PRLAR0, r2   // write (H)PRLAR0
       MCR (H)PRBAR1, r3   // write (H)PRBAR1
       MCR (H)PRLAR1, r4   // write (H)PRLAR1
       … (8 times)
       LDM r5!, {r1-r2}    // load (H)MAIR0/1 setting
       MCR (H)MAIR0, r1    // write (H)MAIR0
       MCR (H)MAIR1, r2    // write (H)MAIR1
```

10.6. VFP/NEON initialization

You need to enable the coprocessors CP10 and CP11 to enable VFP/NEON by configuring the register Coprocessor Access Control Register (CPACR) located in `CP15`. In many software design, NEON/VFP does not have to be turned on at reset, because software might need to delay enabling NEON/VFP to save power, such as enabling VFP/NEON through an undefined exception handler to execute NEON or VFP instructions.

The following example shows the process of enabling VFP/NEON:

```
       MRC   p15, 0x0, r0, c1, c0, 2     // Read CP15 CPACR
       ORR   r0, r0, #(0x0f << 20)        // Full access rights
       MCR   p15, 0x0, r0, c1, c0, 2     // Write CP15 CPACR
       ISB
       MOV   r0, #0x40000000             // Enable Advanced SIMD & VFP
       VMSR  FPEXC, r0                   // Write FPEXC
```

10.7. Cache/MPU enabling

The following example shows the process of enabling the cache and the MPU by configuring the register System Control Register (SCR):

```
       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
       ORR     r0, r0, #0x1              // enable MPU
       MCR     p15, 0, r0, c1, c0, 0     // write System Control Register
```

10.8. Interrupt controller initialization

The following example shows how to initialize GIC and enable a vTimer interrupt in Cortex-R52:

```
       GICD.CTLR = 0x13;                 // enable Group 0, Group 1
       while ((GICD.CTLR&0x80000000) != 0x0);// poll RWP is cleared
       GICR_RD0.WAKER &= 0xFFFFFFFD;     // clear ProcessorSleep
       while ((GICR_RD0.WAKER&0x4) == 0x4);// poll ChildrenAsleep
       enableGroup0Ints();               // Enable group 0 interrupts
       enableGroup1Ints();               // Enable group 1 interrupts
       setPriorityMask(0xFF);                   // Set priority to 0xFF
       setBinaryPoint(0x0);                     // Set Group0 binary point
       setAliasedBinaryPoint(0x0);              // Set Group1 binary point
       GICR_SGI.IPRIORITYR[27] = 0x7F;   // set priority vTimer
       GICR_SGI.IGROUPR0 = 0xFFFFFFFF;   // Set SGI/PPI to group 1
       GICR_SGI.ICFGR[1] = 0x0;          // PPI level sensitive     GICR_SGI.ISENABLER0
= 0x08000000;// Enable vTimer
```

10.9. Nested Interrupt Handling

As mentioned in section 5 Interrupt, the Cortex-R52 must save and restore interruptible context to support nested interrupt. The following example shows the procedure for this process:

```
Nested_IRQ_Handler:
       SUB     lr, lr, #4
       SRSFD   sp!, #0x13
       CPS      #0x13
       PUSH     {r0-r3, r12}
       AND      r1, sp, #4
       SUB      sp, sp, r1
       PUSH      {r1, lr}
       BL       identify_and_clear_source
       CPSIE    i
       BL       C_irq_handler
       CPSID    i
       POP      {r1, lr}
       ADD      sp, sp, r1
       POP      {r0-r3, r12}
       RFEFD    sp!
```

10.10. Cluster Startup

There are two possible approaches to cluster booting in cortex-R52.

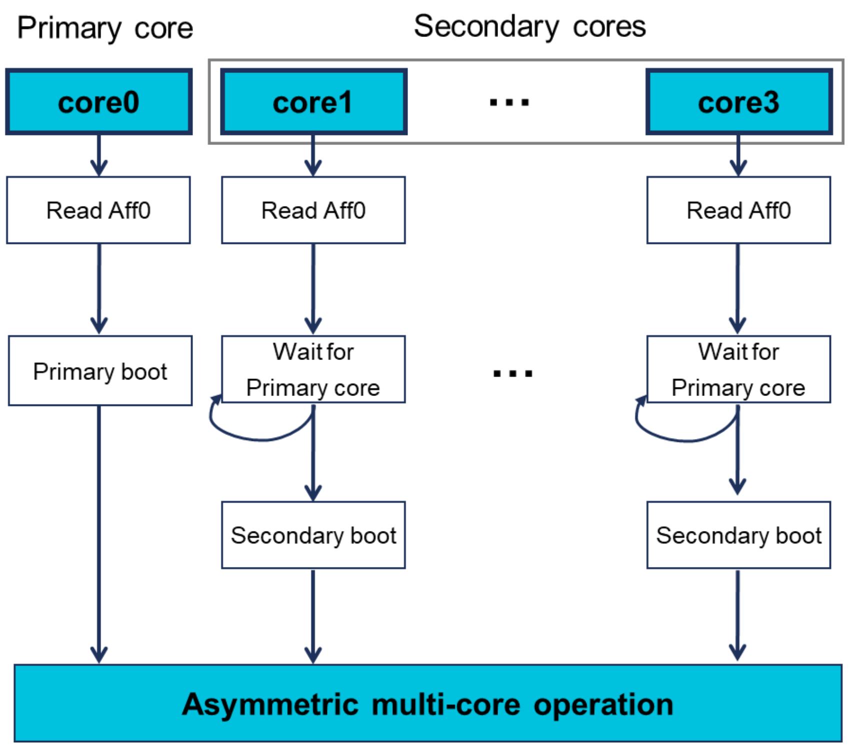
Approach 1

Hardware brings up a primary core after reset, and all other secondary cores are held in reset by the hardware. After the primary core finishes its startup process the primary core brings other cores out of reset.

Approach 2

All cores in cortex-R52 cluster come out of reset and software is used to stall secondary cores. Secondary cores wait until the primary core instructs them to proceed, as Figure 10-1 shows.

Figure 10-1: Cluster startup



Below is the pseudocode for Figure 10-1:

```
cluster_boot()
{
    core_id = Read_reg(MPIDR) & 0xFF;
    if((core_id == 0)) {
        do_primary_boot();
        set_primary_boot_done();
    }
    else {
        wait (PRIMARY_BOOT_DONE);
        do_secondary_boot();
    }
    next_boot_stage();
}
```