AN14618

Multi-Core Application Development on i.MX RT700

Rev. 3.0 — 21 January 2026 Application note

Document information

Information	Content
Keywords	AN14618, i.MX RT700, multi-core application
Abstract	The i.MX RT700 features five domains, with compute and sense as primary processors. This application note outlines strategies for managing memory, cache, boot sequence, and inter-core communication to develop multi-core application.

1 Introduction

The i.MX RT700 MCU includes five independent functional domains:

- Compute
- Sense
- Common
- Digital Signal Processing (DSP)
- Media

The compute and sense domains act as the primary processing domains. Each of these two domains integrates an Arm Cortex-M33 core (processor), colocated with a Cadence Tensilica HiFi4/HiFi1 DSP core, as shown in Figure 1.

The media domain also features a Reduced Instruction Set Computer-V (RISC-V) core called EZH-V, primarily implemented to provide a SmartDMA engine for postprocessing graphics data assembled by the GPU and/or CPU. The EZH-V core then passes this data to the Flexible Input/Output (FlexIO) or Mobile Industry Processor Interface Display Serial Interface (MIPI DSI) for output. Processing operations include data packing and byte and bit order adjustment.

EZH-V handles any required general-purpose tasks and supports an extensive selection of trigger inputs from General-Purpose Input/Output (GPIO) and most on-chip peripherals.

The single i.MX RT700 MCU includes the following five heterogeneous cores:

- 2x Cortex-M33
- HiFi4 DSP
- HiFi1 DSP
- EZH-V

Therefore, i.MX RT700 users must properly manage communication between each core and memory allocations to avoid resource conflict and optimize performance.

This application note explains how to manage memory, cache, boot sequence, and inter-core communication in i.MX RT700. The i.MX RT700 also includes a Neural Processing Unit (NPU) to accelerate neural network operations, but this application note does not cover the NPU.

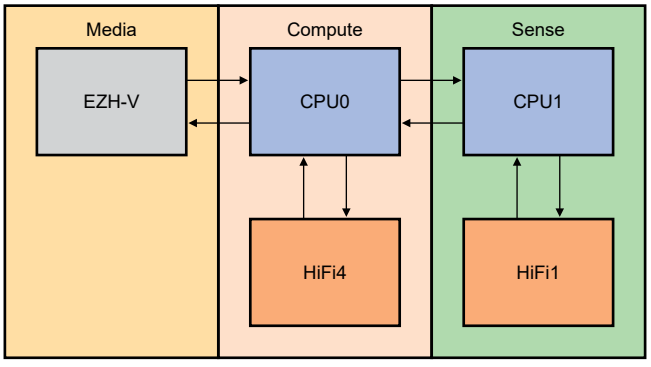


Figure 1. The compute and sense domains have an Arm Cortex-M33 core processor, colocated with a DSP and the media domain has a RISC-V core called EZH-V

2 Memory usage in i.MX RT700 multi-core applications

This section explains how i.MX RT700 MCU manages memory across different domains and cores. It also describes memory architecture, cache controllers, memory maps, and memory allocation strategies.

2.1 Memory architecture

In a multi-core application development, developers decide the physical location of code, data, and shared buffers to avoid resource conflict and optimize performance. Figure 2 shows the memory architecture of the i.MX RT700 family. The figure highlights which memory is tightly coupled to each core.

The following Random Access Memory (RAM) arbiter modules manage access to the on-chip shared RAM memory partitions:

- RAM_ARBITER0, located in the VDD2_COM power domain, manages accesses to VDD2 shared memory partitions (P0-P17).
- RAM_ARBITER1, located in the VDD1_SENSE power domain, manages accesses to VDD1 shared memory partitions (P18-P29).

This shared RAM is divided into multiple partitions with variable sizes. Some partitions are intentionally larger, for example, for frame buffers or DMA descriptors, while others are smaller for stack or heap, or control structures. Each arbiter enforces access rules for every partition. Contention occurs only when two masters target the same partition. Accesses to different partitions proceed independently. CPU0 takes a performance hit for accessing partitions, P18-P30 and CPU1 takes a performance hit for accessing partitions, P0-P17.

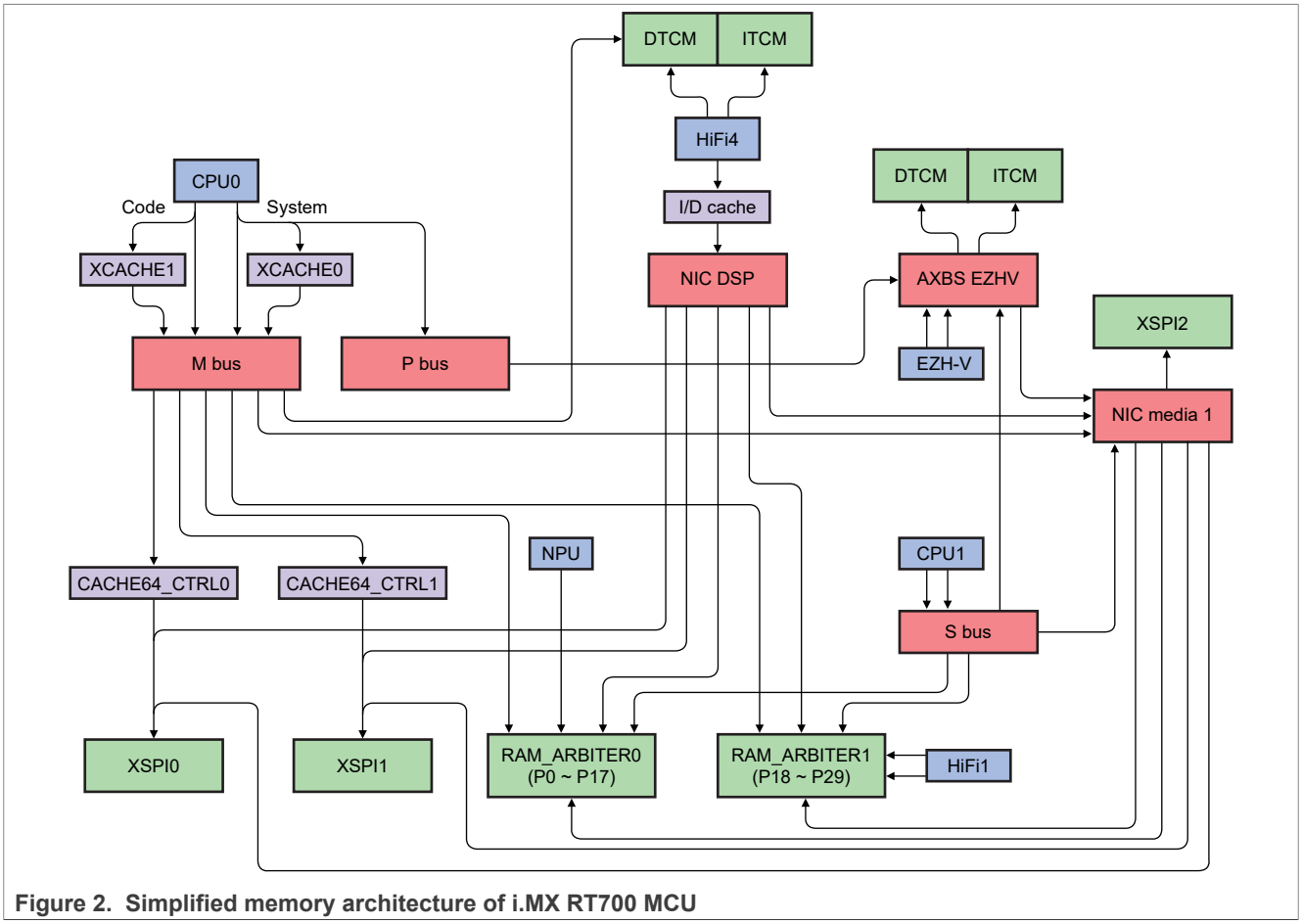


Figure 2. Simplified memory architecture of i.MX RT700 MCU

The subsections that follow explain the memory architecture of different i.MX RT700 functional domains.

2.1.1 Compute domain

The compute domain includes the following main cores and components:

- Cortex-M33 (CPU0)
- HiFi4 DSP
- NPU
- A diverse set of peripherals

One architectural enhancement in i.MX RT700 is the cache support for CPU0. In this scheme, shared Static Random Access Memory (SRAM) is cached, while CPU0 directly accesses the peripherals. There are separate cache controllers for the CPU0 code and system buses. The SRAM is shared, but it can be configured partially as cacheable, with the remainder configured as uncached.

The compute domain uses a split-bus architecture consisting of the following:

- Memory Bus (M-Bus) matrix connects the core initiators to the shared memory.
- Peripheral Bus (P-Bus) matrix interfaces to peripherals.

XSPI0 primarily executes code from off-chip Serial Peripheral Interface (SPI) flash memory. XSPI0 supports Execute-in-Place (XIP) and, if required, On-the-Fly (OTF) decryption using the PRINCE module. It also provides a mechanism to shift designated addresses to a different region of off-chip memory to support dual-image boot.

XSPI1, if used at all, primarily accesses data from pSRAM efficiently. The HiFi4 DSP and CPU0 access the XSPI1.

Both the XSPI0 and XSPI1 interfaces include a 32 KB cache with a CACHE64 cache controller.

2.1.2 Common domain

The common domain holds a shared SRAM (RAM_ARBITER0).

2.1.3 DSP domain

The HiFi4 DSP core has a private cache controller, independent of CPU0 accesses to the shared memory. It also has a dedicated 64 KB Data Tightly Coupled Memory (DTCM) and a 64 KB Instruction Tightly Coupled Memory (ITCM), which are accessible by CPU0. These memories store HiFi4 vectors and time-critical code and data.

2.1.4 Sense domain

The sense domain has its own bus, S-Bus, which connects the other Cortex-M33 core (CPU1), a HiFi1 DSP core, and two Direct Memory Access (DMA) controllers as core initiators.

CPU1 cannot execute in XIP mode and does not have cache memory. The domain has a shared SRAM that can be accessed through RAM_ARBITER1. CPU1 in the sense domain can access SRAM for code and data. The SRAM is also accessible by CPU0, but the frequency is SENSE_RAM_CLK, which is slower than the RAM_ARBITER0 clock (COMMON_RAM_CLK) in many cases.

The HiFi1 DSP does not have private cache memory. Instead, its dual-memory TCM bus interfaces connect to the shared memory. HiFi1 cannot access RAM_ARBITER0.

2.1.5 Media domain

The purpose of the media domain is to hold peripherals that require high performance. The media domain has cross-links to the compute and sense domains and direct access to all partitions of the shared memory.

EZH-V has dedicated 32 KB ITCM and 32 KB DTCM, which are appropriate areas for EZH-V to access code and data frequently. EZH-V does not have cache memory.

XSPI2 is primarily used to access data from pSRAM or HyperRAM, particularly for Graphics Processing Unit (GPU) or LCD Interface (LCDIF), efficiently.

2.2 Cache controllers

CPU0 and HiFi4 have cache memories. While cache improves performance, software issues can occur in multi-core development due to cache coherency, if not managed carefully. Therefore, it is important to determine whether a shared resource among multiple cores is cacheable.

The subsections that follow describe how to configure the cache for different i.MX RT700 cores.

2.2.1 CPU0

The CPU0 and CPU1 code buses access memory and peripherals, as shown in Figure 2. CPU0 code bus access is routed to XCACHE1. This XCACHE1 controller processes cacheable accesses as needed, while bypassing noncacheable accesses or forwarding cache write-through and cache miss accesses to downstream memories through the controller port of the cache controller.

All system bus accesses are routed to the target address in destination memories through a multilayer Advanced High-performance Bus (AHB) matrix slave port. The CPU0 system bus access is routed to XCACHE0.

The CPU0 Memory Protection Unit (MPU) defines the cache policy. In the Armv8-M architecture, memory attributes are not defined for each region differently as in the Armv7-M architecture (for example, Cortex-M3 and Cortex-M7). Instead, the Memory Attribute Indirection Registers (MAIR), specifically MPU_MAIR0 and MPU_MAIR1, define memory attributes that are indirectly referenced by the configuration of each memory region.

In the attached project (AN14618SW), MPU_MAIR0 and MPU_MAIR1 are defined according to Table 1. For a detailed explanation of the memory attributes, refer to the Arm document, MPU Memory Attribute Indirection Registers 0 and 1.

Table 1. MPU_MAIR0 and MPU_MAIR1 configuration

Attribute index	Memory attribute
0	Device-nGnRnE
1	Normal memory, noncacheable
2	Write-through transient, read-allocation

The following code snippet implements the configuration in Table 1:

```
/* Attr0: device memory. */
ARM_MPU_SetMemAttr(0U, ARM_MPU_ATTR(ARM_MPU_ATTR_DEVICE,
 ARM_MPU_ATTR_DEVICE_nGnRnE));
/* Attr1: non-cacheable. */
ARM_MPU_SetMemAttr(1U, ARM_MPU_ATTR(ARM_MPU_ATTR_NON_CACHEABLE,
 ARM_MPU_ATTR_NON_CACHEABLE));
/* Attr2: nontransient, write-through, read-allocate. */
attr = ARM_MPU_ATTR_MEMORY_(0U, 0U, 1U, 0U);
```

```
ARM_MPU_SetMemAttr(2U, ARM_MPU_ATTR(attr, attr));
```

Region Base Address Register (MPU_RBAR) and Region Limit Address Register (MPU_RLAR) are defined according to Table 2 to specify memory regions. Region 0 is accessed under the write-through and read-allocation policy, while Region 1 is noncacheable. Region 2 is accessed under nongathering, non-reordering, and non-early-write-acknowledge policy. For a detailed explanation of the memory attributes, refer to the Arm documents, MPU Region Base Address Register and MPU Region Limit Address Register.

Table 2. MPU_RBAR and MPU_RLAR register

Region index	Address	Shareable	Access permission	Execute never	Attribute index
0	0x0 - 0x1FFFFFFF	N	R/W by any privilege level	Permitted	2
1	{nonCacheStart} - {nonCacheStart + nonCacheSize - 1}	Outer	R/W by any privilege level	Permitted	1
2	0x40000000 - 0x5FFFFFFF	N	R/W by any privilege level	Permitted	0

The following code snippet implements the configuration in Table 2:

```
/* Region 0: [0x0, 0x1FFFFFFF], nonshareable, read/write, any privileged,
 executable. Attr 2 (write-through). */
ARM_MPU_SetRegion(0U, ARM_MPU_RBAR(0U, ARM_MPU_SH_NON, 0U, 1U, 0U),
 ARM_MPU_RLAR(0x1FFFFFFFU, 2U));
/* Region 1 setting : outter-shareable, read-write, nonprivileged, executable.
 Attr 1. (non-cacheable) */
ARM_MPU_SetRegion(1U, ARM_MPU_RBAR(nonCacheStart, ARM_MPU_SH_OUTER, 0U, 1U, 0U),
ARM_MPU_RLAR(nonCacheStart + nonCacheSize - 1, 1U));
/* Region 2 (Peripherals): [0x40000000, 0x5FFFFFFF], nonshareable, read/write,
 nonprivileged, executable. Attr 0
* (device). */
ARM_MPU_SetRegion(2U, ARM_MPU_RBAR(0x40000000U, ARM_MPU_SH_NON,
 0U, 1U, 0U), ARM_MPU_RLAR(0x5FFFFFFF, 0U)); ARM_MPU_SetMemAttr(1U,
 ARM_MPU_ATTR(ARM_MPU_ATTR_NON_CACHEABLE, ARM_MPU_ATTR_NON_CACHEABLE));
```

2.2.2 CPU1

CPU1 does not have cache memory.

2.2.3 HiFi4

The HiFi4 system bus accesses the system memory and peripherals, as shown in Figure 3. The HiFi4 system bus access is routed to an internal cache controller. Memory Management Unit (MMU) defines the cache policy.

In the HiFi4 core of the i.MX RT700 MCU, the MMU includes a region-protection feature that divides the 32-bit address space into eight segments of 512 MB each. Table 3 shows the example of MMU configuration. The software can control protection settings depending on the requirements, as discussed in Section 6.1.

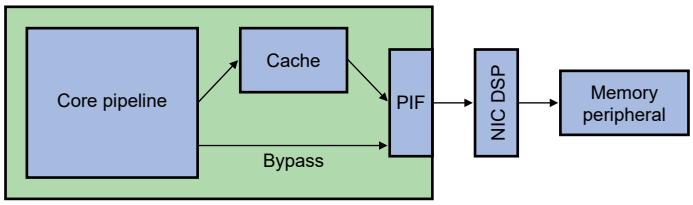


Figure 3. HiFi4 cache controller

Table 3. HiFi4 MMU configuration (example)

Segment	Start address	Policy
0	0x00000000	Write-through
1	0x20000000	Bypass
2	0x40000000	Bypass
3	0x60000000	Write-back
4	0x80000000	Illegal
5	0xA0000000	Illegal
6	0xC0000000	Illegal
7	0xE0000000	Illegal

2.2.4 HiFi1

HiFi1 does not have cache memory.

2.3 Memory map

Table 4 shows the memory map. The system bus is defined as an alias for the code bus and uses the 28th bit for identification.

Table 4. Memory map and accessibility

Address	Allocation	CPU0	CPU1	HiFi4	HiFi1	NPU	eDMA0	eDMA1	eDMA2	eDMA3	Media domain controllers
0x00000000	SRAM P0-P17 (system)	✓									
0x00580000	SRAM P18-P30 (system)	✓	✓		✓						
0x00780000	Reserved										
0x08000000	XSPI1	✓		✓			✓	✓			✓
0x20000000	SRAM P0-P17 (code)	✓	✓	✓		✓	✓	✓	✓	✓	✓
0x20540000	SRAM P18-P30 (code)	✓	✓	✓		✓	✓	✓	✓	✓	✓
0x20780000	Reserved										
0x24000000	HiFi4_DTCM	✓					✓	✓			
0x24010000	Reserved										

Table 4. Memory map and accessibility*...continued*

Address	Allocation	CPU0	CPU1	HiFi4	HiFi1	NPU	eDMA0	eDMA1	eDMA2	eDMA3	Media domain controllers
0x24020000	HiFi4_ITCM	✓					✓	✓			
0x24030000	Reserved										
0x24100000	EZHV_ITCM	✓	✓								✓
0x24108000	EZHV_DTCM	✓	✓								✓
0x24112000	Reserved										
0x28000000	XSPI0	✓		✓			✓	✓			✓
0x40000000	Peripheral										
0x40520000	NPU	✓	✓				✓	✓	✓	✓	✓
0x60000000	XSPI2	✓	✓	✓			✓	✓	✓	✓	✓
0xE0000000	PPB_INT	✓	✓								
0xE0040000	PPB_EXT	✓	✓								

2.4 Memory allocation

The software on each core needs individual and shared resources. Each object must reside in the correct memory area to ensure that multiple cores boot properly and compute efficiently.

In the attached project (AN14618SW), each object is placed according to Table 5. You can change the start address and the size depending on your requirements, as discussed in Section 6. The shared region is used for inter-core communication, as discussed in Section 4.

Table 5. Memory allocation for multi-core applications

Address range	Allocation	Objects
0x00000000-0x0003FFFF	SRAM P0-P5	CPU0 noncache data (secure)
0x00040000-0x0005FFFF	SRAM P6	CPU0 noncache data (nonsecure)
0x00060000-0x00067FFF	SRAM P7	Shared by CPU0 (secure) and CPU1
0x00068000-0x0006FFFF	SRAM P7	Shared by CPU0 (secure) and EZH-V
0x00070000-0x0007FFFF	SRAM P7	Shared by CPU0 (secure) and HiFi4
0x000C0000-0x000FFFFF	SRAM P9	CPU0 data/code/stack (secure)
0x00100000-0x0013FFFF	SRAM P10	CPU0 vector table/code (nonsecure)
0x00140000-0x0017FFFF	SRAM P10	CPU0 data/stack (nonsecure)
0x00400000-0x0057FFFF	SRAM P14-P17	HiFi4 data/code/stack
0x00580000-0x00587FFF	SRAM P18	HiFi1 vector table
0x005B0000-0x005BFFFF	SRAM P23	Shared by CPU1 and HiFi1
0x005C0000-0x005FFFFF	SRAM P24-25	CPU1 data/stack
0x00600000-0x0067FFFF	SRAM P26	CPU1 boot Image
0x00680000-0x006FFFFF	SRAM P27	HiFi1 code
0x00700000-0x0077FFFF	SRAM P28-P29	HiFi1 data/stack

Table 5. Memory allocation for multi-core applications...*continued*

Address range	Allocation	Objects
0x24000000-0x2400FFFF	HiFi4_DTCM	HiFi4 data
0x24020000-0x2402FFFF	HiFi4_ITCM	HiFi4 code
0x24100000-0x24107FFF	EZHV_ITCM	EZH-V code
0x24108000-0x2410FFFF	EZHV_DTCM	EZH-V data
0x28000000-0x2FFFFFFF	XSPI0	Flash configuration CPU0 vector table (secure) CPU0 boot image (secure) Veneer table

3 Boot sequence

In a multi-core application, the cores must be booted following the correct procedure.

3.1 Boot sequence overview

After a power-on reset or warm reset, CPU0 executes the ROM bootloader from internal Read-only Memory (ROM). The ROM bootloader loads the CPU0 boot image from eMMC, NOR flash, or serial communication, depending on the boot configuration.

At reset, CPU1, HiFi1, HiFi4, and EZH-V are clock gated. To boot each core, follow the steps below:

1. Enable the clock and power for SRAM or TCM and the core.
2. Load the image into an appropriate RAM area.
3. Set the offset of the vector table, if needed.
4. Send the reset signal to the core.
5. Clear the stall status bit.

Figure 4 illustrates the boot sequence in the attached project (AN14618SW):

1. CPU0 boots from the QSPI NOR flash memory at reset.
2. CPU0 boots CPU1, HiFi4, and EZH-V.
3. CPU1 boots HiFi1.

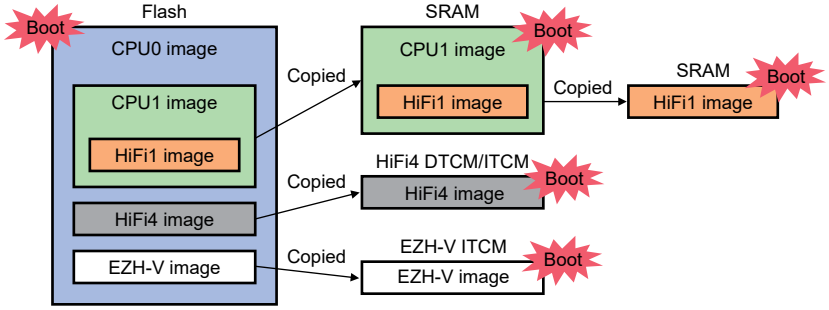


Figure 4. Boot sequence

3.2 Clock root

Table 6 shows how each core works under a different clock root. Each clock root operates independently and supports individual configuration, as shown in Figure 5.

Table 6. Clock root of each core

Processor	Clock root	CLKCTL instance
CPU0	COMPUTE_MAIN_CLK	CLKCTL0
HiFi4	DSP_CLK	CLKCTL0
HiFi1	SENSE_DSP_CLK	CLKCTL1
CPU1	SENSE_MAIN_CLK	CLKCTL3
EZH-V	MEDIA_MAIN_CLK	CLKCTL4

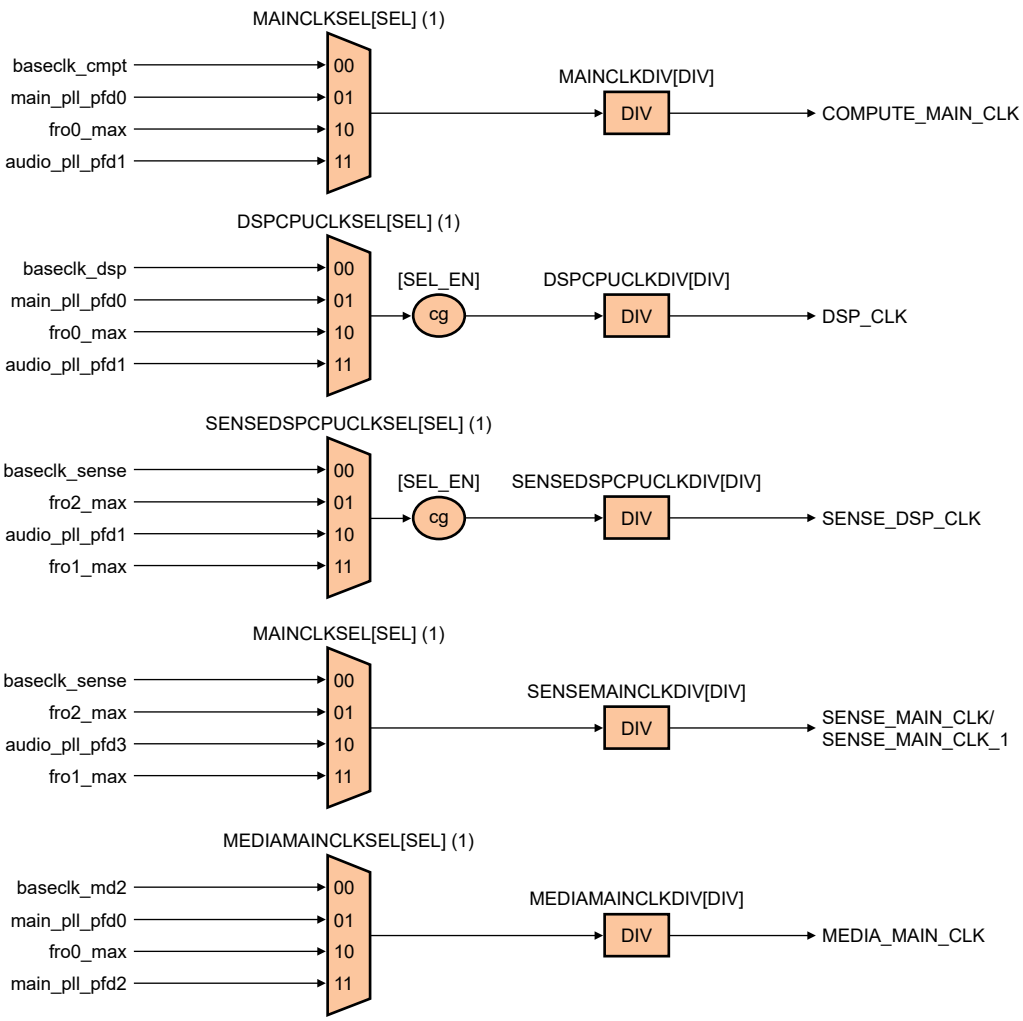


Figure 5. Clock root selector and divider for each core

3.3 Software implementation

The CPU0 source files implement the HiFi4 boot sequence. The following code snippet boots HiFi4. As listed in Table 5, it is important to note that the binary is copied to three isolated regions.

```
/* Initialize PMIC */
/* BOARD_InitPmic(); */
PMC0->PDRUNCFG2 &= ~0x0003C000; /* power up dsp used SRAM. */
```

```
PMC0->PDRUNCFG3 &= ~0x0003C000;
POWER_DisablePD(kPDRUNCFG_PD_VDD2_DSP);
POWER_ApplyPD();
/* Let's DSP run on PLL clock. */
CLOCK_AttachClk(kMAIN_PLL_PFD0_to_DSP);
CLOCK_SetClkDiv(kCLOCK_DivDspClk, 2U);
/* Initializing DSP core */
DSP_Init();
/* Copy literals to DSP DTCM */
DSP_CopyImage(&literal_image);
/* Copy vectors to DSP ITCM */
DSP_CopyImage(&text_image);
/* Copy application from RAM to DSP_RAM */
DSP_CopyImage(&data_image);
/* Only for HiFi4 */
XCACHE_CleanInvalidateCacheByRange((uint32_t)data_image.destAddr,
 data_image.size);
/* Copy ncache section to DSP_RAM */
DSP_CopyImage(&ncache_image);
/* Run DSP core */
DSP_Start();
```

The CPU1 source files implement the HiFi1 boot sequence. The following code snippet boots HiFi1. As listed in Table 5, it is important to note that the binary is copied to three isolated regions.

```
CLOCK_AttachClk(kSENSE_BASE_to_SENSE_DSP);
CLOCK_EnableClock(kCLOCK_SenseAccessRamArbiter0);
CLOCK_EnableClock(kCLOCK_Syscon1);
CLOCK_EnableClock(kCLOCK_Sleepcon1);
/* Enable SENSE private clock. */
POWER_DisablePD(kPDRUNCFG_SHUT_SENSEP_MAINCLK);
/* Initializing DSP core */
DSP_Init();
/* Copy literals to DSP RAM */
DSP_CopyImage(&vector_image);
/* Copy vectors to DSP ITCM */
DSP_CopyImage(&text_image);
/* Copy application from RAM to DSP_RAM */
DSP_CopyImage(&data_image);
/* Copy ncache section to DSP_RAM */
DSP_CopyImage(&ncache_image);
/* Run DSP core */
DSP_Start();
```

The CPU0 source files implement the EZH-V boot sequence. The following code snippet boots EZH-V.

```
CLOCK_EnableClock(kCLOCK_Ezhv);
CLOCK_EnableClock(kCLOCK_AxbsEzh);
POWER_DisablePD(kPDRUNCFG_APD_EZHV_TCM);
POWER_DisablePD(kPDRUNCFG_PPD_EZHV_TCM);
POWER_ApplyPD();
EZHV_InstallFirmware(&ezhv_image));
XCACHE_CleanInvalidateCacheByRange(ezhv_image.destAddr, ezhv_image.size);
EZHV_Boot(EZHV_ITCM_ADDRESSESS);
```

The CPU0 source files implement the CPU1 boot sequence. The following code snippet boots CPU1.

```
BOARD_CopyCore1Image(CORE1_BOOT_ADDRESS);
BOARD_ReleaseCore1Power();
```

```
BOARD_BootCore1((CORE1_BOOT_ADDRESS & 0x0FFFFFFF), (CORE1_BOOT_ADDRESS &
 0x0FFFFFFF));
```

4 Inter-core communication

In a multi-core application, the different cores need to interact with each other. This section explains how inter-core communication happens.

RPMsg-Lite is a middleware, that enables one-to-one communication between cores. In this model, one core acts as the master and the other as the remote. Furthermore, RPMsg endpoints provide logical connections on top of the RPMsg channel, allowing you to bind multiple callbacks on the same channel. This mechanism works like a port number used in the TCP/UDP protocol. You can register callbacks for each endpoint. The endpoint address can be announced dynamically to the name service endpoint.

Here, the software implementation is discussed first, and the hardware implementation is explained later.

4.1 Software implementation

The master first initializes the Virtual Input/Output (VirtIO) queue, which is shared with the remote. Each VirtIO queue contains a *Used buffer* and an *Avail buffer*. These buffers hold elements that point to the buffer descriptor index, as shown in Figure 6. Each buffer descriptor includes the address and length of the buffer, which is used to exchange data between master and remote.

The two VirtIO queues have initially been created.

- Master TVQ (= Remote RVQ from the perspective of remote):
  - The *Used buffer* contains elements with buffer descriptor index.
  - The *Avail buffer* remains empty.
- Remote TVQ (= Master RVQ from the perspective of master):
  - The *Avail buffer* contains elements with a buffer descriptor index.
  - The *Used buffer* remains empty.

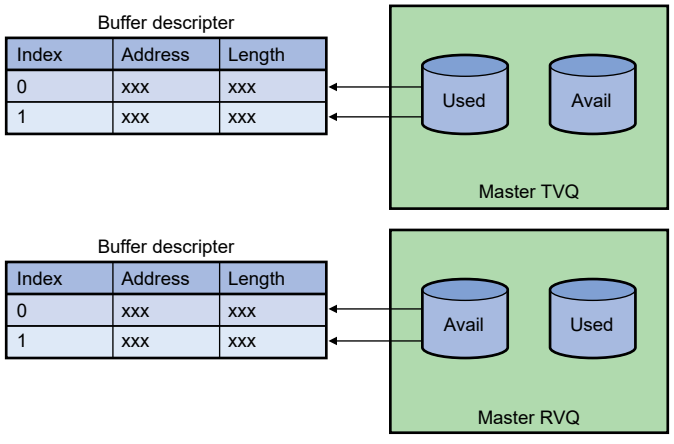


Figure 6. Two VirtIO queues initialized by the master

When the master sends data to the remote, the master and remote processors follow the below procedure, as shown in Figure 7:

1. The master dequeues the descriptor index from the *Used buffer* in the master Transmit VirtIO Queue (TVQ).
2. The master updates the buffer.
3. The master enqueues the descriptor index into the *Avail buffer* in the master TVQ.

4. The master kicks the remote.
5. The remote dequeues the descriptor index from the *Avail buffer* in the remote Receive VirtIO Queue (RVQ).
6. The remote reads the buffer.
7. The remote enqueues the descriptor index from the *Avail buffer* in the remote RVQ.

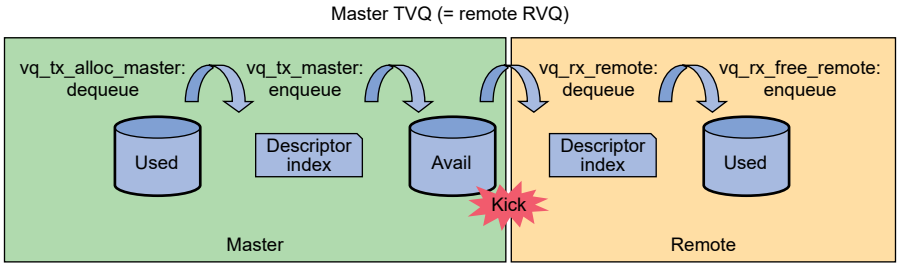


Figure 7. The master sends data to the remote

When the remote sends data to the master, each processor follows the procedure shown in Figure 8. When the remote sends a message to the master, the roles of the *Avail* and *Used* buffers are swapped.

1. The remote dequeues the descriptor index from the Avail buffer in the remote TVQ.
2. The remote updates the buffer.
3. The remote enqueues the descriptor index into the Used buffer in the remote TVQ.
4. The remote kicks the master.
5. The master dequeues the descriptor index from the Used buffer in the master RVQ.
6. The master reads the buffer.
7. The master enqueues the descriptor index from the Used buffer in the master RVQ.

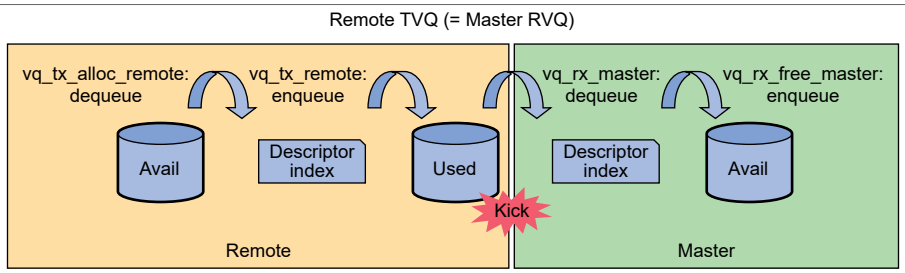


Figure 8. The remote sends data to the master

4.2 Hardware implementation

RPMsg-Lite requires the following two basic components in hardware perspective:

- Shared memory
- Inter-core interrupt

This section explains how the shared memory and inter-core interrupt are implemented in the i.MX RT700 MCU.

4.2.1 Shared memory

RPMsg-Lite uses the VirtIO queue to exchange data, and this queue resides in the shared memory. The start address and size of the shared memory must remain fixed during initialization time on both the master and remote sides. In the attached project (AN14618SW), the shared memory is allocated as listed in Table 7.

In RPMsg-Lite v5.1.3 or higher, the shared regions is either cacheable or noncacheable. If it is cacheable, the library maintains cache coherency.

Table 7. Shared memory placement

Master and remote	Address	Size
CPU0 and CPU1	0x20060000	0x8000
CPU0 and EZH-V	0x20068000	0x8000
CPU0 and HiFi4	0x20070000	0x8000
CPU1 and HiFi1	0x205B0000	0x8000

4.2.2 Inter-core interrupt

RPMsg-Lite uses inter-core interrupts to notify target core (such as CPU1, HiFi4, HiFi1, or EZH-V) about VirtIO queue updates. The subsections that follow explain how inter-core communication can happen using interrupts.

4.2.2.1 Messaging unit (MU)

The MU enables one processor to signal the other processor using interrupts. The MU consists of the following two interfaces as shown in Figure 9:

- MUA: MUA can generate interrupts to processor A when interrupts are enabled.
- MUB: MUB can generate interrupts to processor B when interrupts are enabled.

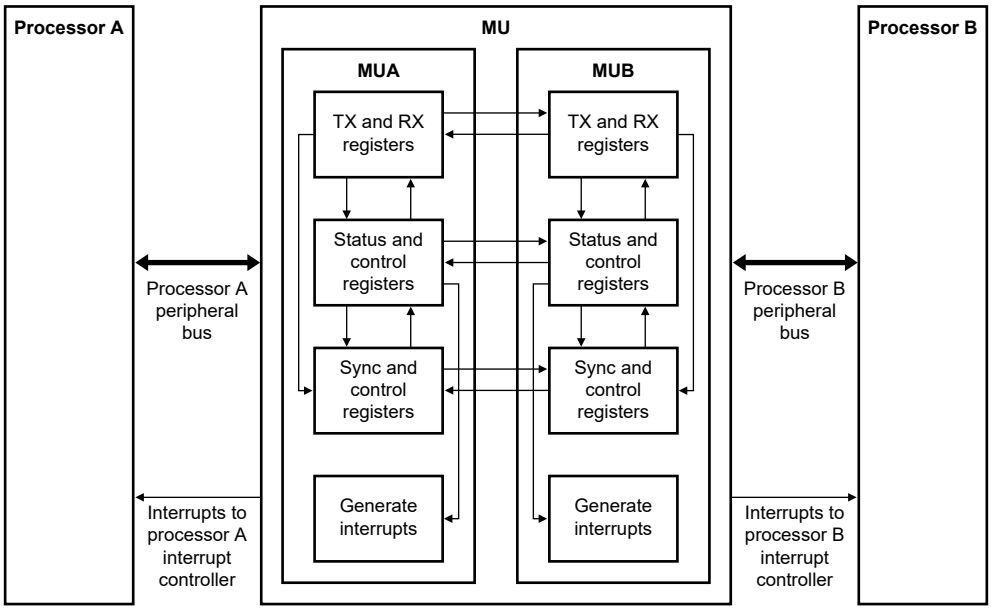


Figure 9. MU block diagram

The i.MX RT700 MCU includes five MU instances, with processor pairings, as listed in Table 8.

Table 8. Processor A and processor B for each MU instance

MU instance	Processor A	Processor B
MU0	CPU0	HiFi1
MU1	CPU0	CPU1

Table 8. Processor A and processor B for each MU instance*...continued*

MU instance	Processor A	Processor B
MU2	HiFi4	CPU1
MU3	CPU1	HiFi1
MU4	CPU0	HiFi4

4.2.2.2 Arm to EZH-V and EZH-V to Arm interrupt

The Arm core can generate an interrupt to EZH-V core directly without using the MU, and EZH-V core can also signal the Arm core in the same way.

The following mechanisms are used for the two types of processors:

- For the Arm core: The Arm core uses the EZHV2ARM_INT_CHAN register to identify which VirtIO queue has been updated.
- For EZH-V: EZH-V uses the machine software input interrupt and machine external input interrupt to determine which VirtIO queue has been updated.

5 Running the demo

This section explains how to build and run the i.MX RT700 multi-core application project in IAR Embedded Workbench for Arm.The same basic procedure applies to other toolchains. The i.MX RT700 multi-core application demonstrates exchanging message by RPMsg Lite between each core.

5.1 Apply the patch to the MCUXpresso SDK

RPMsg-Lite does not support EZH-V in MCUXpresso SDK 24.12.00. Therefore, a software patch must be applied to the MCUXpresso SDK. To apply the patch, follow the steps below:

1. Download the MCUXpresso SDK 24.12.00 and unzip it to a path of your choice.
2. Copy `rpmsg-lite_ezhv.patch` to the SDKroot directory.
3. Change the current directory to the SDK root directory.
4. Apply the patch using the following command:

```
patch -p1 < rpmsg-lite_ezhv.patch
```

5.2 Hardware and PC setup

To set up the hardware and PC for the i.MX RT700 multi-core application demo, follow the steps below:

1. On the MIMXRT700-EVK board, ensure that the jumper JP18 is open if you use the onboard debugger; otherwise, short JP18.
2. To see two COM ports from the PC, open the jumper JP27.
3. Open two terminal consoles:
   - One for CPU0, HiFi4, and EZH-V
   - The other for CPU1 and HiFi1
4. Configure both consoles with the following settings:
   - 115,200 baud rate
   - 8 data bits
   - No parity
   - One stop bit

- No flow control

5.3 Build the i.MX RT700 multi-core application project

To build all components of the i.MX RT700 multi-core application, follow the step-by-step procedures mentioned in the following subsections. The order of building is important because some images are embedded into other images (for example, the CPU1 image includes the HiFi1 image).

5.3.1 Build HiFi4 and HiFi1

To build the HiFi4 and HiFi1 DSP using Xtensa Xplorer, follow the steps below:

1. To set up Xtensa Xplorer, follow the Getting Started with Xplorer for MIMXRT700-EVK.
2. Download the attached project (AN14618SW) and unzip it to *boards\mimxrt700evk\demo_apps*.
3. Import the project from the file system, as shown in Figure 10.

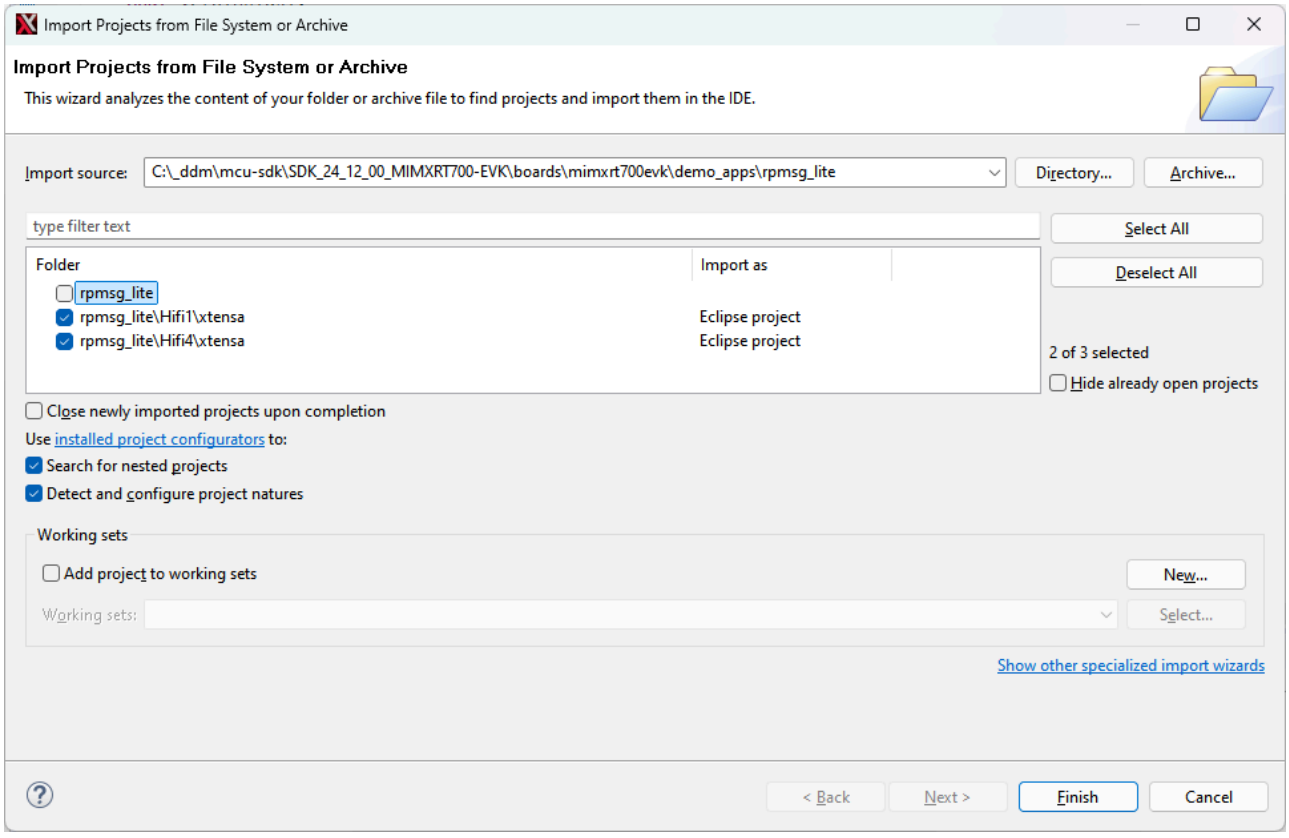


Figure 10. Import projects from file system

4. Build the project, as shown in Figure 11 and Figure 12. Use correct configurations for each project in the pulldown as shown in Figure 13:
   - rt700_hifi4_RI23_11_nlib for HiFi4
   - rt700_hifi1_RI23_11_nlib for HiFi1

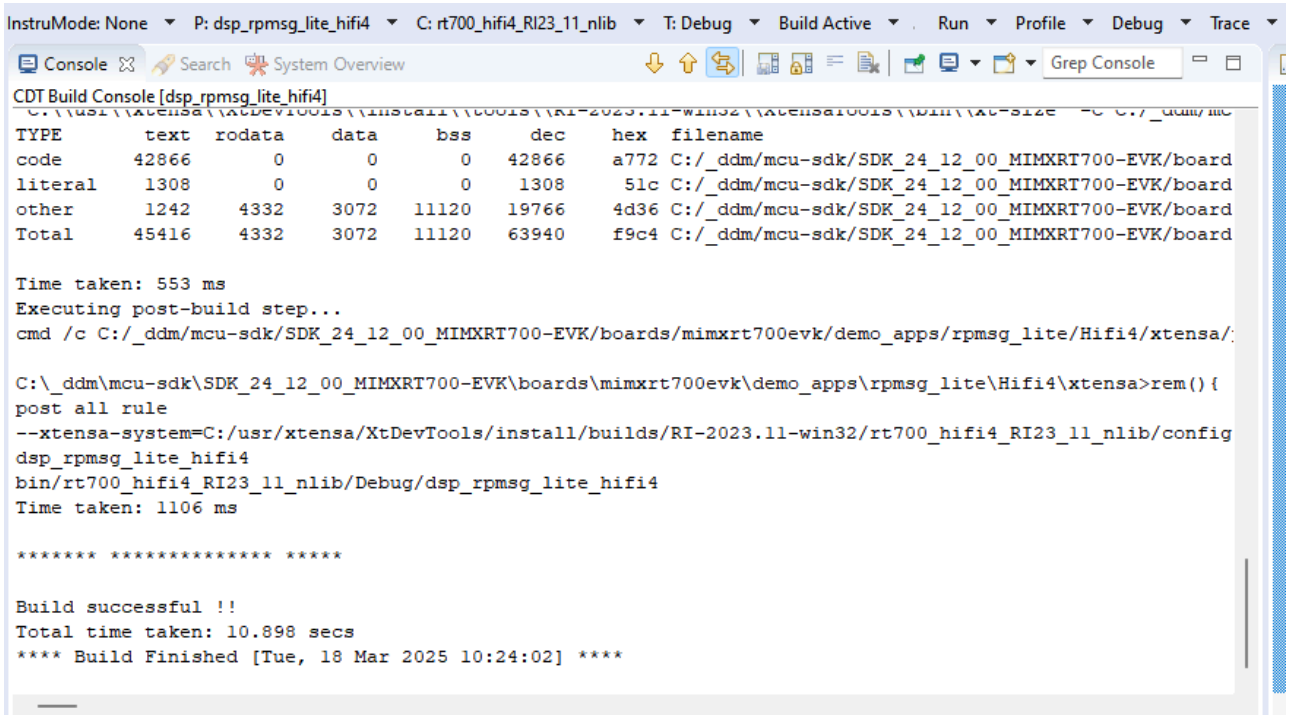


Figure 11. Build HiFi4 project in Xtensa Xplorer

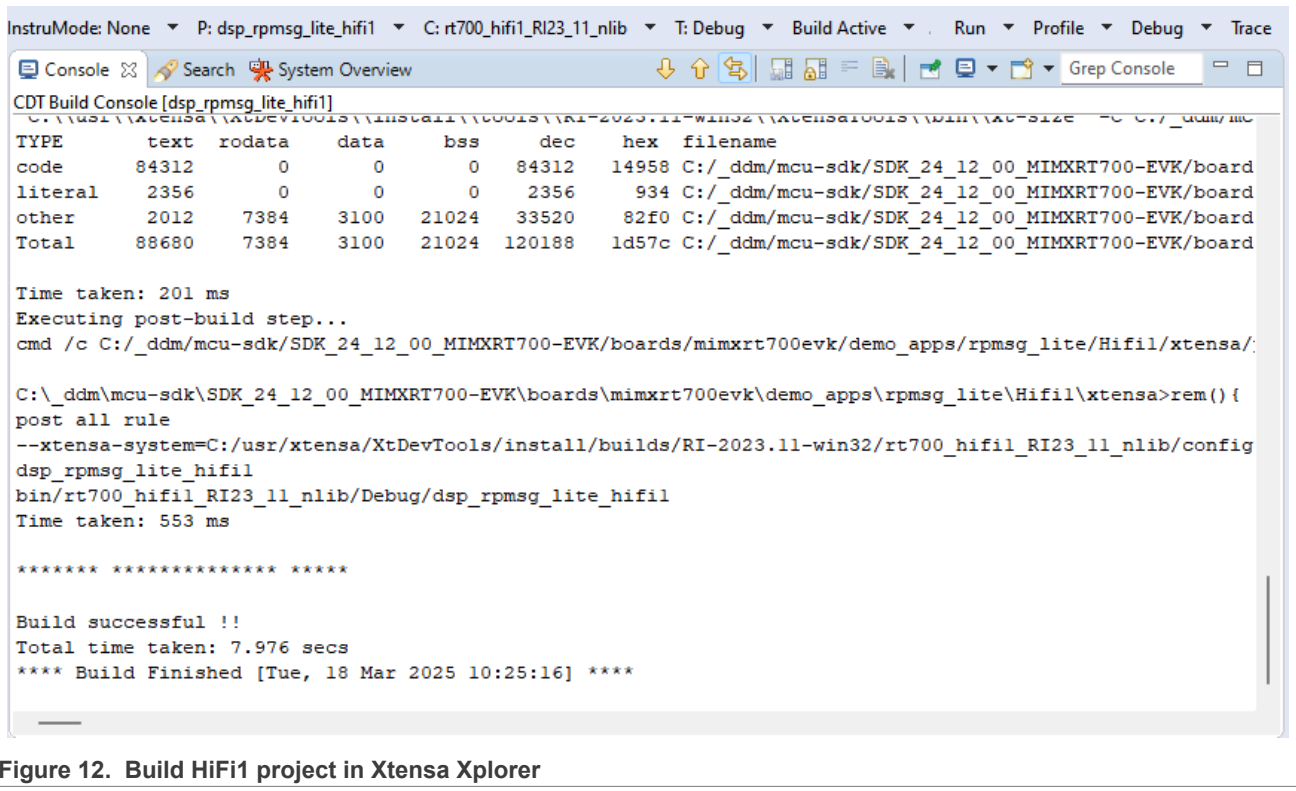


Figure 12. Build HiFi1 project in Xtensa Xplorer

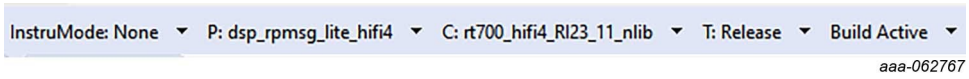


Figure 13. Use the correct configuration depending on the project

5. Find the generated binary images in:
   - rpmsg_lite\Hifi4\binary
   - rpmsg_lite\Hifi1\binary

5.3.2 Build EZH-V

To build the EZH-V project using LLVM/Clang toolchains, follow the steps below:

1. To set up the LLVM/Clang toolchains, follow *Developing Environment Setup for i.MX RT700 EZH-V* (document AN14614).
2. In the command prompt, open the *ezhv/riscvllvm* directory.
3. Run `build_debug.bat` or `build_release.bat`, as shown in Figure 14.
4. Find the generated binary images in `rpmsg_lite\ezhv\binary`. The build log is saved in `build_log.txt`.

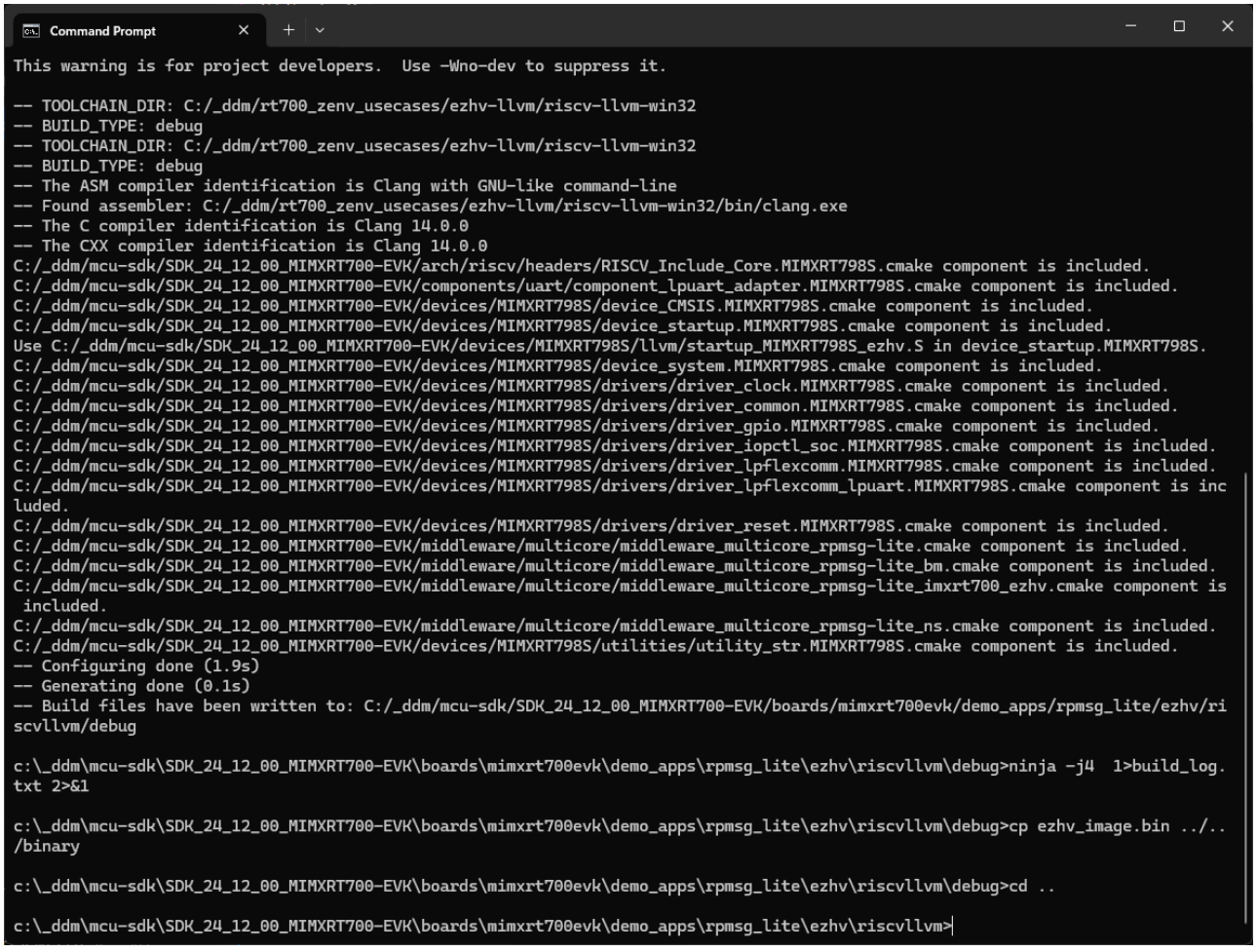


Figure 14. Running build script in command prompt

5.3.3 Building CPU1 project

There are several IDEs available for building the project. This section explains IAR and MCUXpresso for Visual Studio Code.

5.3.3.1 IAR Embedded Workbench for Arm

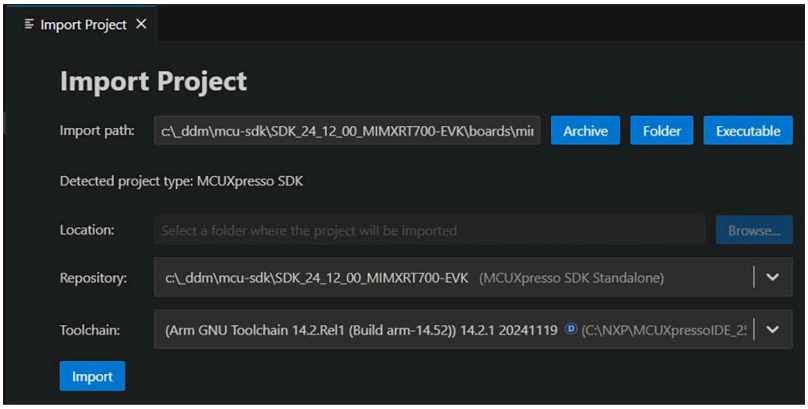
To build the CPU1 project using IAR Embedded Workbench, perform the following steps:

1. Install IAR Embedded Workbench 9.60.3 or later.
2. Open `rpmsg_lite_cm33_core1.eww`.
3. Build the project at release configuration.

5.3.3.2 MCUXpresso for Visual Studio Code

To build the CPU1 project using MCUXpresso for Visual Studio Code, perform the following steps:

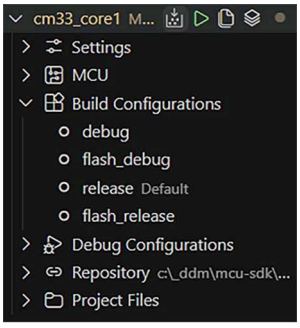
1. To install MCUXpressso for Visual Studio Code, follow Getting Started with MCUXpresso for Visual Studio Code.
2. Import *cm33_core1* folder, as shown in Figure 15.



aaa-064227

Figure 15. Import project from folder

3. Build the project at release configuration, as shown in Figure 16.



aaa-064228

Figure 16. Imported project for CPU1

5.3.4 Building CPU0 project

There are several IDEs available for building the CPU0 project. This section explains IAR and MCUXpresso for Visual Studio Code.

5.3.4.1 IAR Embedded Workbench for Arm

To build the CPU0 project using IAR Embedded Workbench, perform the following steps:

1. Open `rpmsg_lite_cm33_core0.eww` in EWARM.
2. Build the project at *flash_debug* configuration.
3. Download the project to MIMXRT700-EVK.

5.3.4.2 MCUXpresso for Visual Studio Code

To build the CPU0 project using MCUXpresso for Visual Studio Code, perform the following steps:

1. Import *cm33_core0_s* folder in the same way as CPU1.
2. Build the project at *flash_debug* configuration, as shown in Figure 17.

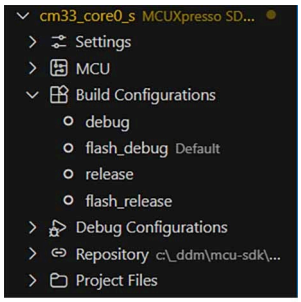


Figure 17. Imported project for CPU0

5.4 Result

After building and downloading all projects, verify that the demo runs successfully by checking the serial console output. The console output displays messages exchanged between the following pairs of cores as shown in Figure 18 and Figure 19:

- CPU0 ↔ CPU1
- CPU0 ↔ EZH-V
- CPU0 ↔ HiFi4
- CPU1 ↔ HiFi1

Note:

- *CPU0, EZH-V, and HiFi4 share the same serial COM port.*
- *CPU1 and HiFi1 share another COM port.*

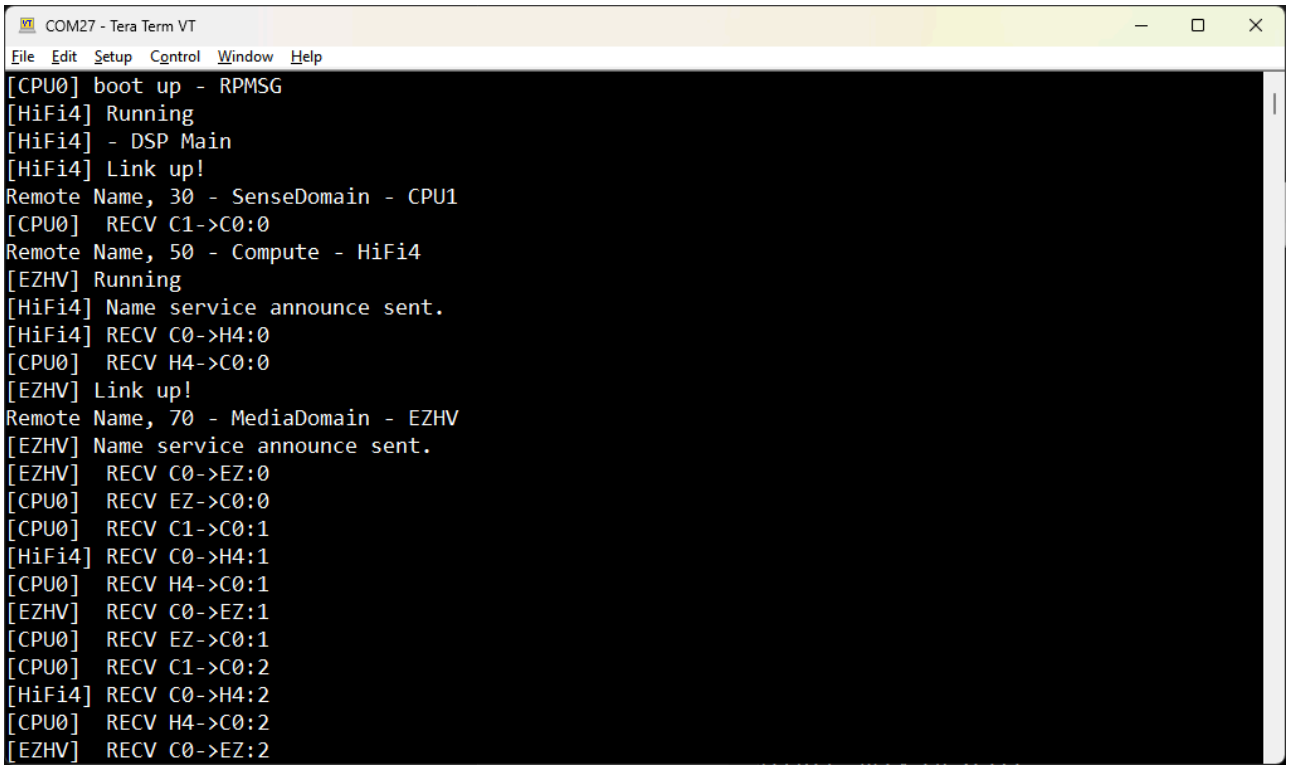


Figure 18. The log of CPU0, HiFi4, and EZH-V

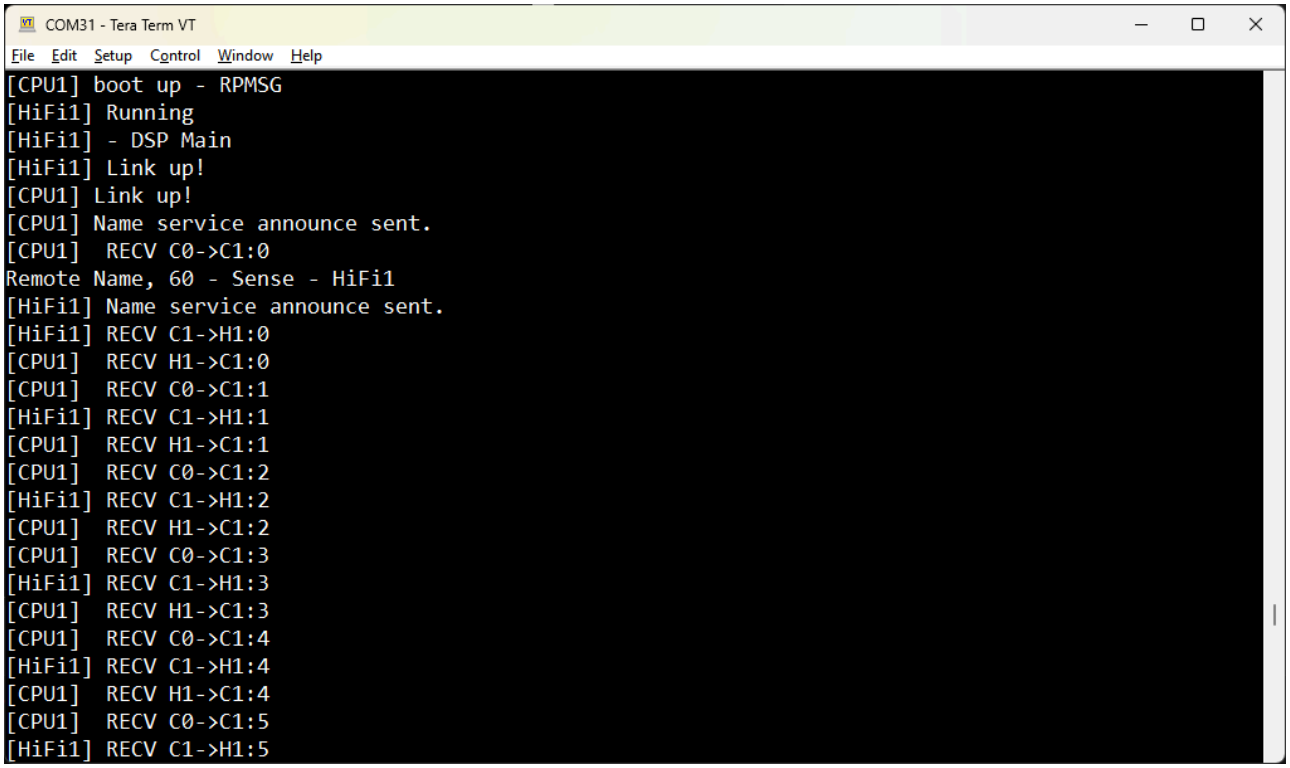


Figure 19. The log of CPU1 and HiFi1

5.5 Debug the i.MX RT700 multi-core application project

To debug each core of the i.MX RT700 multi-core application, use the tools and methods described in the following subsections:

5.5.1 Debug CPU0/CPU1

Use one of the following IDEs to debug CPU0 and CPU1 cores:

- IAR Embedded Workbench for Arm
- Arm Keil MDK
- MCUXpresso IDE

5.5.2 Debug HiFi4/HiFi1

You can debug HiFi4 and HiFi1 cores using Xtensa Xplorer. To set up Xtensa Xplorer and learn the debugging steps, including updating the EVK debug drivers to J-Link, refer to Getting Started with Xplorer for MIMXRT700-EVK.

5.5.3 Debug EZH-V

You can debug EZH-V cores using J-Link Ozone. For detailed debugging instructions, refer to *Developing Environment Setup for i.MX RT700 EZH-V* (document AN14614).

6 Linker customization

The linker is configured properly to realize the memory allocation according to Table 5. The configuration method varies depending on the toolchains.

6.1 Configure linker for HiFi4/HiFi1

The Xtensa toolchain, called Linker Support Package (LSP), defines the memory allocation, so you do not need to modify the linker scripts manually.

LSP consists of three major components:

- memmap.xmm defines which object resides in which region memory. It also provides memory attributes for the objects (for example, executable, writable, uncached). MMU is initialized automatically according to this file.
- The specification file defines the standard object files and libraries to include in the linker command line for the final application executable.
- The specification file references the object files and libraries.

You can access Xtensa tool documentation at the below location after the installation of Xtensa Xplorer:

C:/usr/xtensa/XtDevTools/downloads/RI-2023.11/docs/index.html

The memmap.xmm file for HiFi4 defines memory allocation as shown below. Compare it to Table 5 to understand how memmap.xmm defines the memory allocation. You can also use the GUI tool called memory map editor to edit the memory description, as shown in Figure 20.

```
BEGIN dsp_core
0x20400000: sysram : dsp_core : 0x180000 : executable, writable ;
 dsp_core : C : 0x20400000 - 0x2057ffff : dsp_core.data dsp_core.bss;
END dsp_core
```

```
BEGIN dram0
0x24000000: dataRam : dram0 : 0x10000 : writable ;
 dram0_0 : C : 0x24000000 - 0x2400ffff :  STACK :  HEAP :
 __llvm_prf_names .data .rodata .literal .dram0.rodata .ResetVector.literal
 .Level2InterruptVector.literal .Level3InterruptVector.literal
 .DebugExceptionVector.literal .NMIExceptionVector.literal
 .KernelExceptionVector.literal .UserExceptionVector.literal
 .DoubleExceptionVector.literal .iram0.literal .dram0.data .dram0.bss .bss;
END dram0

BEGIN iram0
0x24020000: instRam : iram0 : 0x10000 : executable, writable ;
 iram0_0 : F : 0x24020000 -
 0x240203ff : .ResetVector.text .ResetHandler.literal .ResetHandler.text;
 iram0_1 : F : 0x24020400 - 0x2402057b : .WindowVectors.text;
 iram0_2 : F : 0x2402057c - 0x2402059b : .Level2InterruptVector.text;
 iram0_3 : F : 0x2402059c - 0x240205bb : .Level3InterruptVector.text;
 iram0_4 : F : 0x240205bc - 0x240205db : .DebugExceptionVector.text;
 iram0_5 : F : 0x240205dc - 0x240205fb : .NMIExceptionVector.text;
 iram0_6 : F : 0x240205fc - 0x2402061b : .KernelExceptionVector.text;
 iram0_7 : F : 0x2402061c - 0x2402063b : .UserExceptionVector.text;
 iram0_8 : F : 0x2402063c -
 0x2402ffff : .DoubleExceptionVector.text .iram0.text .text;
END iram0

BEGIN iocached
0x70000000: io : iocached : 0xda00000 : executable, writable ;
END iocached

BEGIN rpmsg_sh_mem
0x20070000: sysram : rpmsg_sh_mem : 0x8000 : executable, writable ;
END rpmsg_sh_mem

BEGIN rambypass
0x80000000: sysram : rambypass : 0x10000000 : device, executable, writable ;
END rambypass

BEGIN iobypass
0x90000000: io : iobypass : 0xda00000 : device, executable, writable ;
END iobypass
```

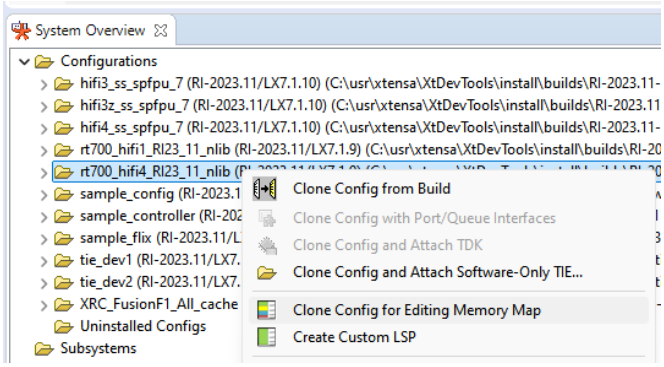


Figure 20. Running a GUI tool to edit memory description

There are some standard LSPs in Xtensa Xplorer, for example:

- gdbio builds an application that uses a connected debugger (gdb), if any, for all console and file I/Os.

- min-rt builds an application to run in most systems or simulations, but without any board-specific support, such as the character I/O.

For more details, refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

After editing memmap.xmm based on standard LSPs, to generate linker scripts from it, follow the steps below:

1. Set the configuration to `rt700_hifi4_RI23_11_nlib`.
2. Right-click the project and select **Open Command Shell** as shown in Figure 21.
3. Execute the following command under the HiFi4 directory, as shown in Figure 22:

```
xt-genldscripts -b gdbio
```

4. The linker script (`*.ld`) is generated automatically as shown in Figure 22.

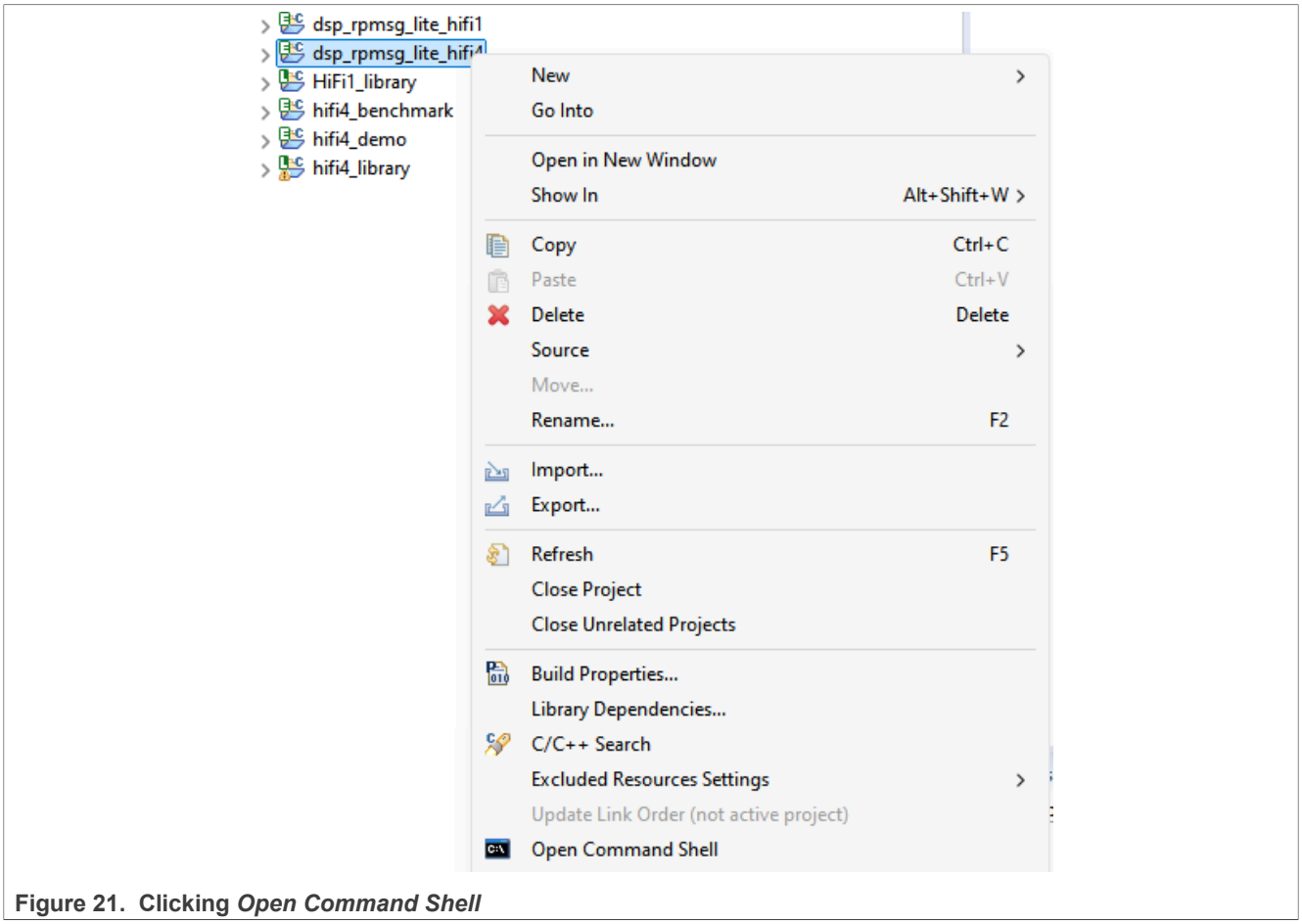


Figure 21. Clicking *Open Command Shell*

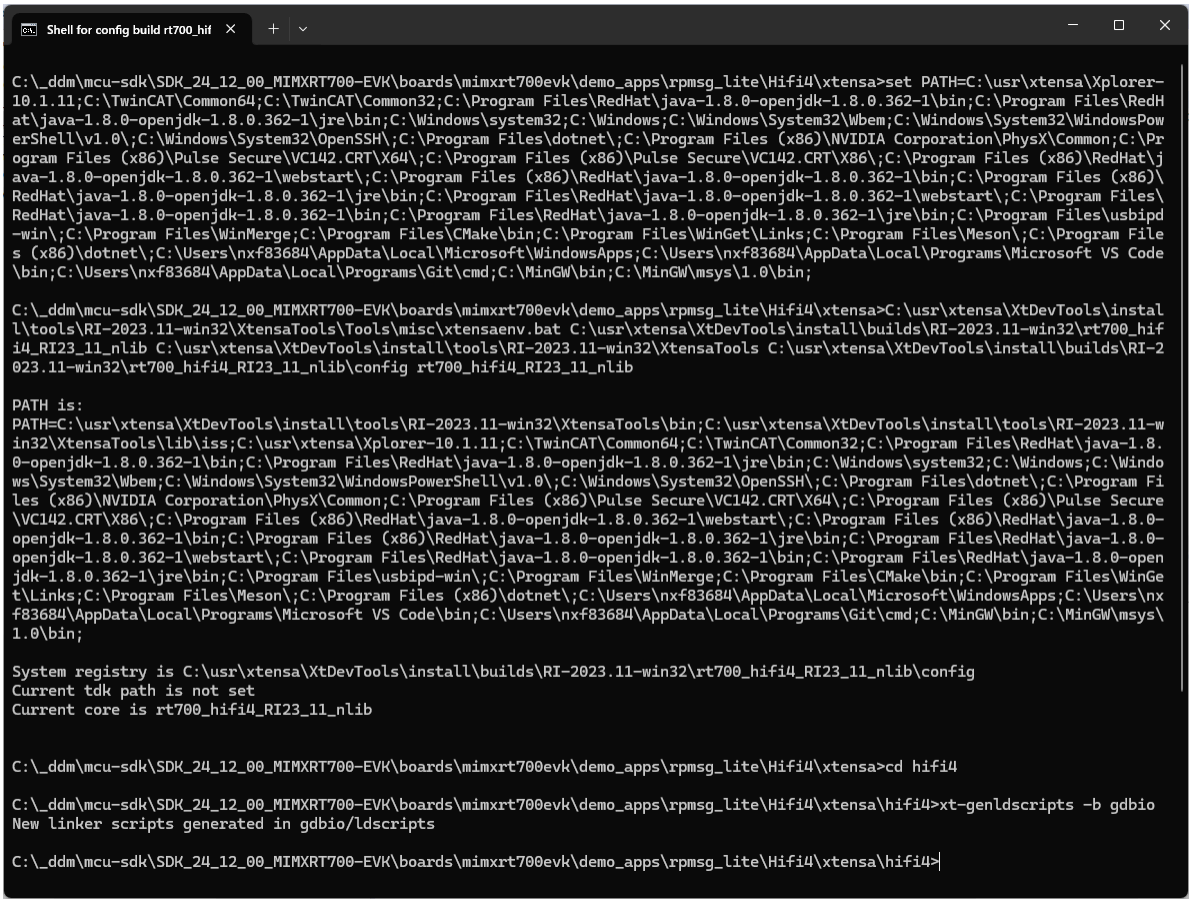


Figure 22. xt-genldscripts -b gdbio command generates linker scripts

To implement the memory allocation according to Table 5, HiFi4 image must be copied into three different areas, as specified in Table 9. Use the post-build.bat file to generate these three separate images. If you modify the memmap.xmm file, update the post-build.bat file to match your memmap.xmm file.

Table 9. Image destination

Sections	Destination address	Binary name
dsp_core.data	0x20400000	dsp_data_release.bin
.data .rodata .literal .rtos.rodata .Level3InterruptVector.literal .DebugExceptionVector.literal .NMIExceptionVector.literal .rtos.percpu.data	0x24000000	dsp_literal_release.bin
.ResetVector.text .WindowVectors.text .Level2InterruptVector.text .Level3InterruptVector.text .DebugExceptionVector.text .NMIExceptionVector.text	0x24200000	dsp_text_release.bin

Table 9. Image destination*...continued*

Sections	Destination address	Binary name
.KernelExceptionVector.text .UserExceptionVector.text .DoubleExceptionVector.text.text		

6.2 Configure linker for EZH-V

LLVM/Clang toolchain is compatible with the GNU ID. For example, the shared region for RPMsg Lite is defined as an uninitialized data section using the following code snippet.

```
/* Uninitialized data section */
/* NOINIT section for rpmsg_sh_mem */
.noinit_rpmsg_sh_mem (NOLOAD) : ALIGN(4)
{
        __RPMSG_SH_MEM_START__ = .;
        *(.noinit.$rpmsg_sh_mem*)
        . = ALIGN(4) ;
        __RPMSG_SH_MEM_END__ = .;
} > rpmsg_sh_mem
```

6.3 Configure linker for CPU0/CPU1

For CPU0 and CPU1, configure the linker to generate a monolith image for downloading into the QSPI NOR flash memory. The linker configuration must ensure that each image contains the required components:

1. The CPU1 image includes the HiFi1 image.
2. The CPU0 image includes the CPU1, HiFi4, and EZH-V images.

Note: *The configuration method varies depending on the toolchain.*

6.3.1 IAR Embedded Workbench for Arm

In the IAR linker, use the `--image_input` option to embed a binary into an image. For example, in the CPU0 configuration, the following option is used. This option defines `cm33_core1.bin` as the `__core1_bin` symbol in the `__core1_image` section with 4-byte alignment. Include the `--keep` option because the software does not explicitly reference the symbol.

```
--image_input=$PROJ_DIR$/../../cm33_core1/iar/binary/
cm33_core1.bin,__core1_bin,__core1_image,4
--keep=__core1_bin
```

For detailed information about these options, refer to the IAR document.

6.3.2 Arm Keil MDK/GCC (MCUXpresso IDE)

In Armclang and GCC assembly, use the `.incbin` directive to embed a binary into an image. For example, in the CPU0 configuration, use the following assembly code.

The `core1_image.bin` is included in the section named `.core1_code`, and its start address, end address, and size are defined as `core1_image_start`, `core1_image_end`, and `core1_image_size`.

```
        .section .core1_code, "ax" @progbits @preinit_array
        .global core1_image_start
        .type core1_image_start, %object
```

```
        .align 4
core1_image_start:
        .incbin "core1_image.bin"
        .global core1_image_end
        .type core1_image_end, %object
core1_image_end:
        .global core1_image_size
        .type core1_image_size, %object
        .align 4
core1_image_size:
        .int core1_image_end - core1_image_start
        .end
```

For detailed information about other options, refer to the GNU document.

7 Acronyms

Table 10 lists the acronyms used in this document along with their descriptions.

Table 10. Acronyms

Acronym	Description
DTCM	Data Tightly Coupled Memory
DSP	Digital Signal Processing
FlexIO	Flexible Input/Output
GPIO	General-Purpose Input/Output
GPU	Graphics Processing Unit
MIPI DSI	Mobile Industry Processor Interface Display Serial Interface
MPU	Memory Protection Unit
LCDIF	LCD Interface
LSP	Linker Support Package
M-Bus	Memory Bus
MMU	Memory Management Unit
NPU	Neural Processing Unit
OTF	On-the-Fly
P-Bus	Peripheral Bus
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer-V
ROM	Read-only Memory
RVQ	Receive VirtIO Queue
SPI	Serial Peripheral Interface
SRAM	Static RAM
TVQ	Transmit VirtIO Queue
VirtIO	Virtual Input/Output
XIP	Execute-in-Place

8 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2026 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

9 Revision history

Table 11 summarizes the revisions to this document.

Table 11. Revision history

Document ID	Release date	Description
AN14618 v.3.0	21 January 2026	Added Section 5.3.3 "Building CPU1 project" and Section 5.3.4 "Building CPU0 project"
AN14618 v.2.0	10 November 2025	• Initial public release • Updates: – Several technical and editorial changes – Added Section 5.5 "Debug the i.MX RT700 multi-core application project"
AN14618 v.1.0	23 April 2025	Initial internal release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

1 Introduction 2
2 Memory usage in i.MX RT700 multi-core applications 3
2.1 Memory architecture 3
2.1.1 Compute domain 4
2.1.2 Common domain 4
2.1.3 DSP domain 4
2.1.4 Sense domain 4
2.1.5 Media domain 5
2.2 Cache controllers 5
2.2.1 CPU0 5
2.2.2 CPU1 6
2.2.3 HiFi4 6
2.2.4 HiFi1 7
2.3 Memory map 7
2.4 Memory allocation 8
3 Boot sequence 9
3.1 Boot sequence overview 9
3.2 Clock root 9
3.3 Software implementation 10
4 Inter-core communication 12
4.1 Software implementation 12
4.2 Hardware implementation 13
4.2.1 Shared memory 13
4.2.2 Inter-core interrupt 14
4.2.2.1 Messaging unit (MU) 14
4.2.2.2 Arm to EZH-V and EZH-V to Arm interrupt 15
5 Running the demo 15
5.1 Apply the patch to the MCUXpresso SDK 15
5.2 Hardware and PC setup 15
5.3 Build the i.MX RT700 multi-core application project 16
5.3.1 Build HiFi4 and HiFi1 16
5.3.2 Build EZH-V 18
5.3.3 Building CPU1 project 19
5.3.3.1 IAR Embedded Workbench for Arm 19
5.3.3.2 MCUXpresso for Visual Studio Code 19
5.3.4 Building CPU0 project 19
5.3.4.1 IAR Embedded Workbench for Arm 20
5.3.4.2 MCUXpresso for Visual Studio Code 20
5.4 Result 20
5.5 Debug the i.MX RT700 multi-core application project 22
5.5.1 Debug CPU0/CPU1 22
5.5.2 Debug HiFi4/HiFi1 22
5.5.3 Debug EZH-V 22
6 Linker customization 22
6.1 Configure linker for HiFi4/HiFi1 22
6.2 Configure linker for EZH-V 26
6.3 Configure linker for CPU0/CPU1 26
6.3.1 IAR Embedded Workbench for Arm 26
6.3.2 Arm Keil MDK/GCC (MCUXpresso IDE) 26
7 Acronyms 27
8 Note about the source code in the document 28
9 Revision history 28
Legal information 29