Multi-core handling guide in XMC7000

About this document

Scope and purpose

This application note provides information for multi-core handling in XMC7000 MCUs. An XMC7000 MCU can have up to three Arm® Cortex®-M CPUs. Multi-CPU architecture helps in improving system performance and efficiency. This document describes how to perform exclusive control, synchronization, and pass data between the different CPUs/cores. In addition, the document provides an overview of the cache coherency issue that occurs between CPUs with cache and other masters and suggests methods to avoid the issue under different scenarios.

Intended audience

This document is intended for anyone using the XMC7000 family.

Associated part family

XMC7000 family XMC7100/XMC7200 series of XMC™ industrial microcontrollers.

Table of contents

Table of contents

1 Introduction

XMC7000 family MCUs include Arm® Cortex®-M CPUs with SRAM, Flash memory, Enhanced Secure Hardware Extension (eSHE), CAN FD, memory, and analog and digital peripheral functions in a single chip.

The CPU subsystem of the XMC7000 family MCUs consists of multiple bus masters, two or three CPUs, two types of DMA controllers (P-DMA (DW), M-DMA (DMAC)), and a cryptography block (Crypto). The CPU subsystem also has an Inter-Processor communication (IPC) block that can be used for exclusive control, synchronization, and data passing between CPUs.

In addition, the XMC7100 and XMC7200 series have cache memory on the CPU and some peripherals. Cache memory is a low latency memory and helps to improve performance. However, the cache memory can cause coherency issues between memories. Therefore, the use of cache memory requires careful handling.

Figure 1 shows a block diagram of the CPU subsystem for the parts of a single CM7 core of the XMC7000 series.

Figure 2 shows a block diagram for the CPU subsystem for the parts of two CM7 cores of the XMC7000 series.

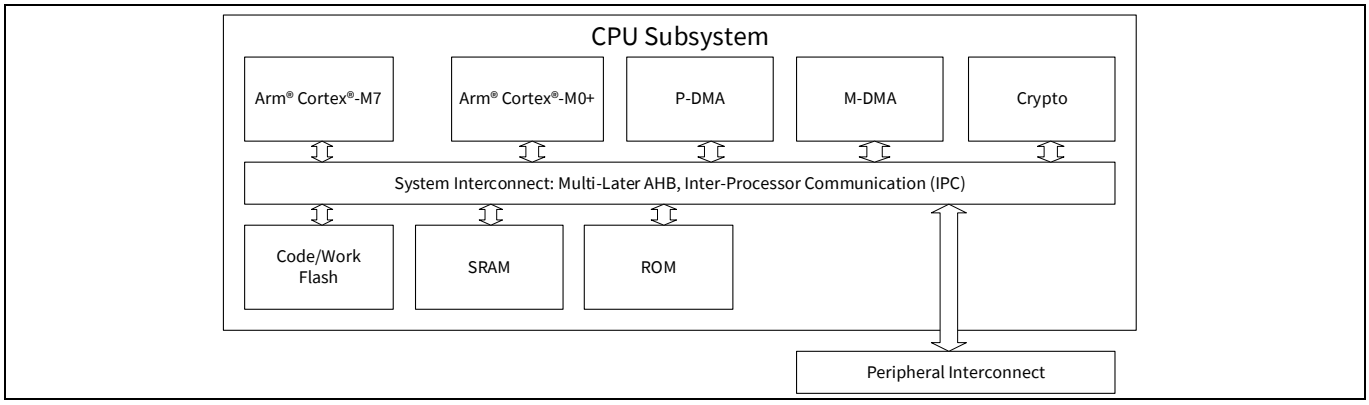

Figure 1 CPU subsystem for part number of single CM7 core of XMC7000 series

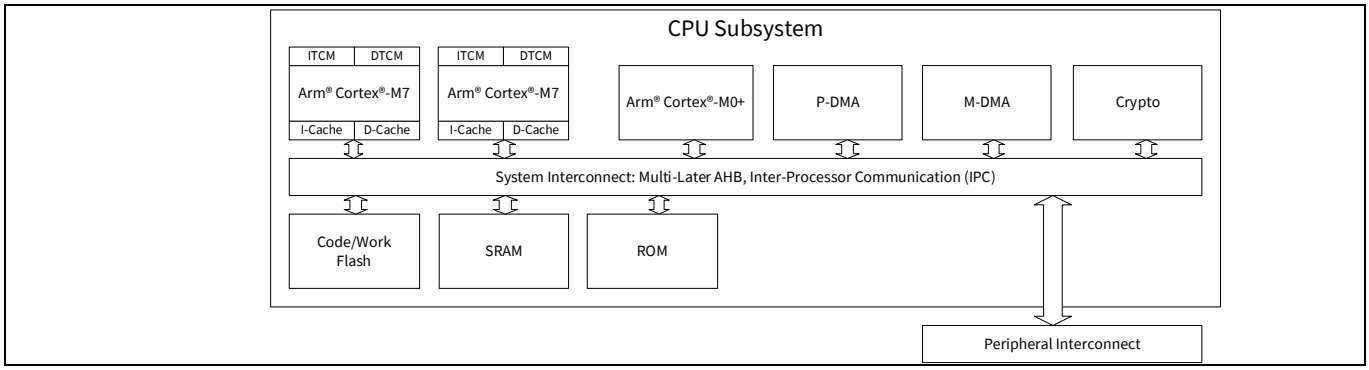

Figure 2 CPU subsystem for part number of two CM7 cores of XMC7000 series

The XMC7000 series have one or two Arm® Cortex®-M7-based CPUs (CM7) and one Cortex®-M0-based CPU (CM0+). CM7 CPUs have Instruction/Data cache (I-cache/D-cache) and Instruction/Data tightly-coupled Memories (ITCM/DTCM). The CPU subsystems of the XMC7000 series MCUs have bus masters for P-DMA (DW), M-DMA (DMAC), and Crypto block. See the Arm® documentation sets for CM7 and CM0+, and the XMC7000 family architecture reference manual for more information.

Introduction

Note: *The contents of the block diagram may vary depending on the device. See the device datasheet for device-specific details.*

All memories and peripherals are shared by all bus masters. Shared resources are accessed through standard Arm® multi-layer bus arbitration. Exclusive accesses are supported by an IPC block.

A multi-CPU architecture presents unique opportunities for system-level design and performance optimization in a single MCU. With multi-CPUs, you can allocate:

- Tasks to CPUs so that multiple tasks may be done at the same time
- Resources to CPUs so that a CPU may be dedicated to managing those resources, therefore, improving efficiency

2 Considerations for CPU start up

Generally, when user application software starts, the CPU uses the PLL to switch to high-speed operation. However, sudden changes in the CPU clock may cause the external or internal supply voltage to drop. If the voltage drops below a defined voltage, the internal brown-out detect (BOD) circuit will trigger a low voltage detection reset.

To avoid a low voltage detection reset, it is recommended to step up the CPU clock in stages to ensure it does not go below the voltage defined by BOD.

This is especially important for the XMC7000 series, which has two CM7 cores.

Here is an example of stepping up CPU clock frequency in stages for the XMC7000 series. Figure 3 shows the CM7 CPUs' clock connection in this example. This example uses `CLK_PATH1` with PLL400#0 as the root clock for `CLK_HF1`, which is the CM7 CPUs' clock.

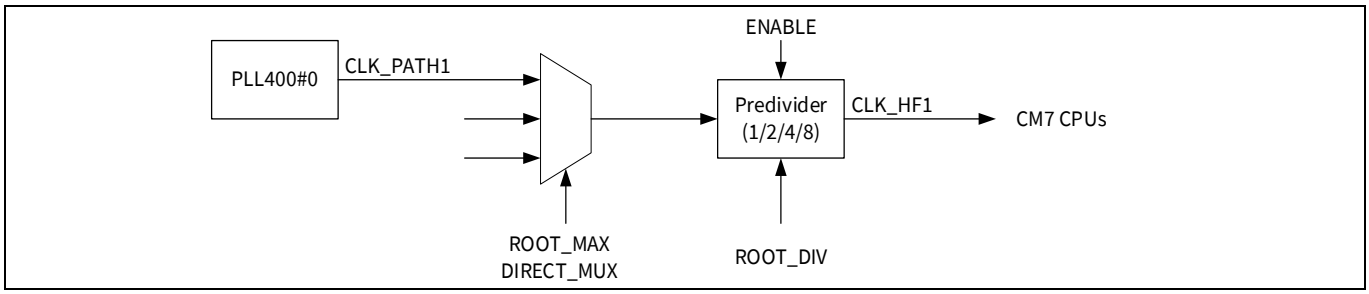

Figure 3 CM7 CPUs clock connection

3 Considerations for resource access

As mentioned in Considerations for CPU start up, all memory and peripherals are accessed through standard Arm® multi-layer bus by all bus masters. Therefore, each master can start accessing the bus at the same time. However, the multiple bus masters can access different memory or peripheral groups at the same time, but cannot access the same memory or peripheral group at the same time.

Figure 4 shows resource connections for the XMC7200 series. IPC (green box) access of CM7_0 and CRYPT (green box) access of CM0+ can be performed at the same time (indicated by green arrows). However, CM7_0 and CM7_1 cannot access TTCAN FD (grey box) at the same time (indicated by orange arrows).

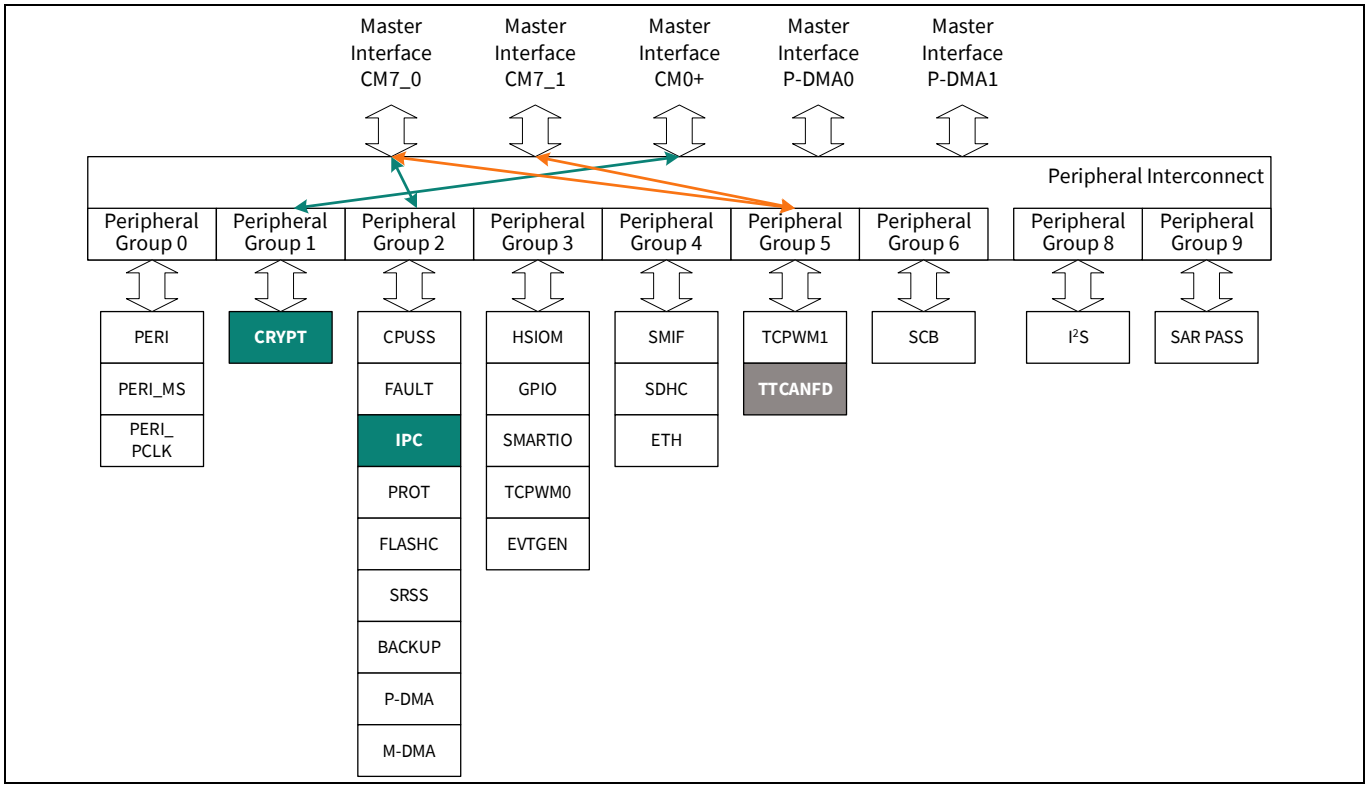

Figure 4 Resource connection (XMC7200 series)

To improve performance, you need to consider CPU resource allocation in system design, so that the CPU may be dedicated to managing those resources. In this case, dedicating either `CM7_0` or `CM_1` to `TTCAN FD` management will improve performance.

A similar case occurs for memory access. For example, SRAM0 access of `CM7_0` and SRAM1 access of `CM7_1` can be performed at the same time (green arrow). `CM7_0` and `CM7_1` cannot access the same SRAM2 at the same time even if the addresses are different within the SRAM (orange arrow).

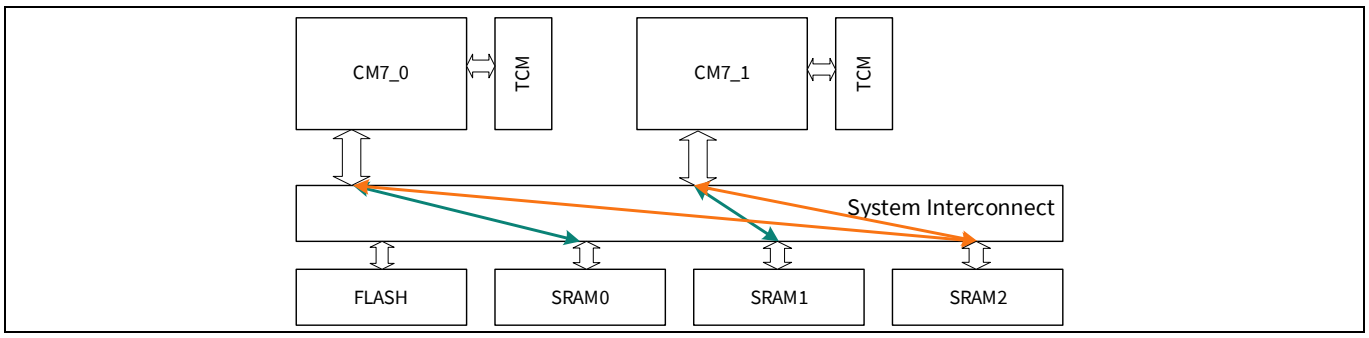

Figure 5 Memory connection (XMC7200 series)

In such cases, you can improve performance by assigning dedicated SRAM for each CPU or using TCM dedicated to each CPU.

Note: The connection of resources and memory may vary depending on the device. See the device datasheet for device-specific details.

4 Communicating between CPUs

Architectures with multiple CPUs often require exclusive control, synchronization, and data passing between CPUs, XMC7000 can use IPC for such control. IPC has support for mutual exclusion (mutex), message passing, and event and release notification.

The IPC hardware contains register structures for the IPC channel and IPC interrupt. IPC channel registers implement lock/release mechanisms and messaging. IPC interrupt structure registers generate interrupts to each CPU for messaging events and lock/release events.

IPC channel structure registers consist of `IPC_STRUCTx_ACQUIRE`, `IPC_STRUCTx_NOTIFY`, `IPC_STRUCTx_RELEASE`, two 32-bit `IPC_STRUCTx_DATA0/1`, and `IPC_STRUCTx_LOCK_STATUS`. The ACQUIRE register provides a lock feature and `IPC_STRUCTx_LOCK_STATUS` indicates lock status. The `IPC_STRUCTx_NOTIFY` register generates notification events, the `IPC_STRUCTx_RELEASE` register releases the IPC channel structure and generates release events. `IPC_STRUCTx_DATA0/1` register can pass a message up to 64 bits.

Note: A few IPCs are reserved by SROM API, and you cannot use structures of IPC channel and interrupt reserved by the SROM API. See the device datasheet for more information.

4.1 CPU synchronization

This section describes how to synchronize CPUs using IPC. In a multi-CPU architecture, the order in which tasks are executed by each CPU needs to be carefully managed.

As an example, consider two CPUs (`CPU_A` and `CPU_B`), where the `CPU_A` initializes resources and then `CPU_B` uses the initialized resources. In this case, however, if `CPU_B` uses the resource before `CPU_A` initializes the resource (wrong order of execution), it causes an unintended operation.

IPC has two solutions for this issue. One solution is to use the `IPC_STRUCTx_DATA0/1` register. Another solution is to use the `IPC_STRUCTx_NOTIFY` register. The solution using `IPC_STRUCTx_DATA0/1` register is easy to implement. CPU_A writes a specific value to the `IPC_STRUCTx_DATA0` register when initialization is complete. `CPU_B` polls the `IPC_STRUCTx_DATA0` register and does not start execution until it reads that specific value from the `IPC_STRUCTx_DATA0` register.

Synchronization using the `IPC_STRUCTx_NOTIFY` register uses a notification event interrupt. Table 1 lists the registers associated with the notification event. `IPC_STRUCTx_NOTIFY` register is used to generate an IPC notify event and `IPC_STRUCTx_RELEASE` to generate an IPC release event.

Table 1 Register list of notify event

Structure	Register name	Bit name	Description
IPCx channel	`IPC_STRUCTx_NOTIFY`	`INTR_NOTTIFY[15:0]`	This field allows for the generation of notification events to the IPC interrupt structures. SW always reads a '0' from this field.

Structure	Register name	Bit name	Description
	IPC_STRUCTx_RELEASE	INTR_RELEASE[15:0]	This field allows for the generation of release events to the IPC interrupt structures, but only when the lock is acquired. SW always reads a '0' from this field.
IPCx interrupt	IPC_INTR_STRUCTx_INTR	NOTIFY[31:16]	These interrupts cause fields to be activated when an IPC notification event is detected. SW writes '1' to these fields to clear the interrupt cause.
		RELEASE[15:0]	These interrupts cause fields to be activated when an IPC release event is detected. SW writes '1' to these fields to clear the interrupt cause.
	IPC_INTR_STRUCTx_INTR_SET	NOTIFY[31:16]	SW writes '1' to this field to set the corresponding field in the INTR register.
		RELEASE[15:0]	SW writes '1' to this field to set the corresponding field in the INTR register.
	IPC_INTR_STRUCTx_INTR_MASK	NOTIFY[31:16]	Mask bit for corresponding field in the INTR register.
		RELEASE[15:0]	Mask bit for corresponding field in the INTR register.
	IPC_INTR_STRUCTx_INTR_MASKED	NOTIFY[31:16]	Logical and of corresponding request and mask bits.
		RELEASE[15:0]	Logical and of corresponding INTR and INTR_MASK fields.

“x” indicates channel number for each IPC structure.

Each bit in the IPC_STRUCTx_NOTIFY and IPC_STRUCTx_RELEASE registers corresponds to the channel number of the IPC interrupt structure, and each bit in the IPC_INTR_STRUCTx_INTR, IPC_INTR_STRUCTx_INTR_SET, IPC_INTR_STRUCTx_INTR_MASK, and IPC_INTR_STRUCTx_INTR_MASKED registers corresponds to the channel number of IPC channel structures. NOTIFY [31:16] corresponds to channel numbers 15 to 0 of the IPC channel structure. See the registers reference manual for more information.

Note: The channel number of IPC channel structure and IPC interrupt structure may vary depending on the device. See the device datasheet for device-specific details.

Figure 6 shows the relation between the IPC channel structures and the IPC interrupt structures. An IPC interrupt structure can be triggered from any of the IPC channel structures and the event generated from an IPC channel structure can trigger any or multiple interrupts in an IPC interrupt structure.

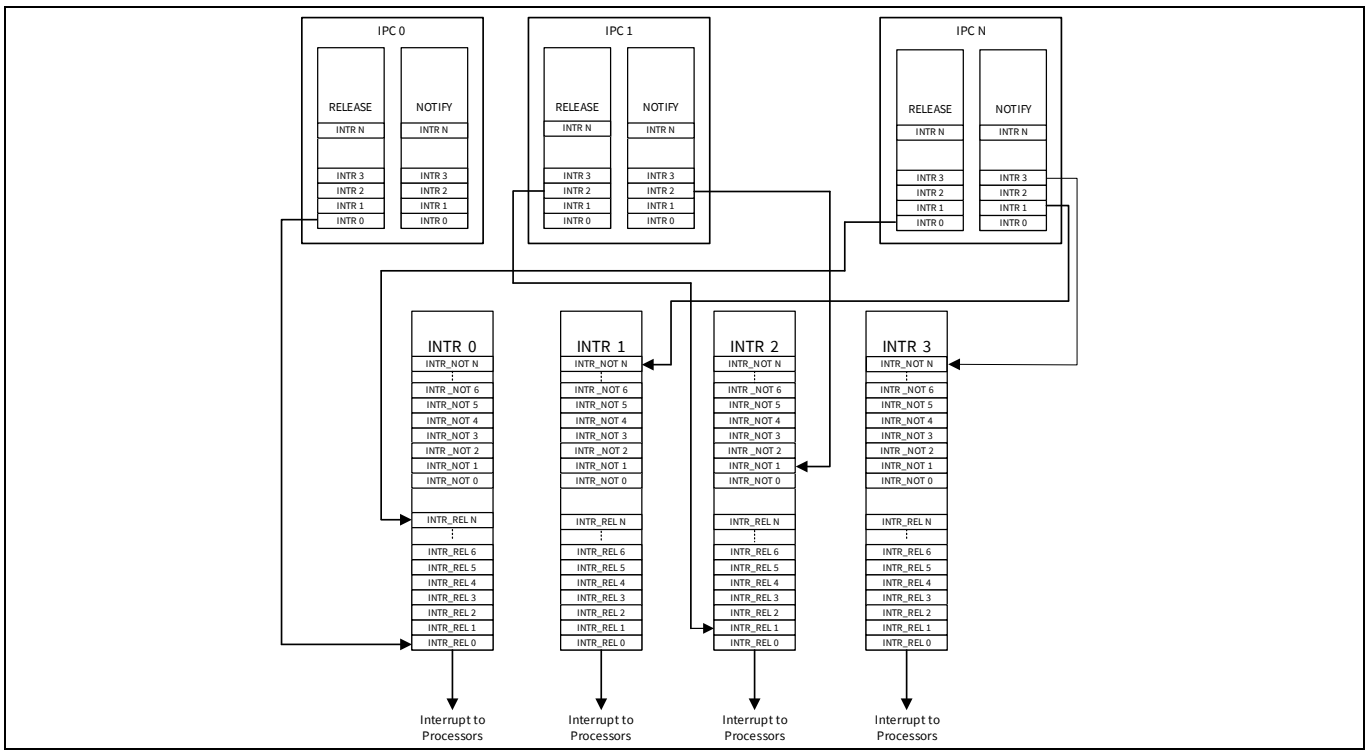

Figure 6 IPC channel structures and interrupt structures

In the example shown in Figure 6, IPC 0 channel structure can trigger the RELEASE event of INTR 0, and IPC 1 channel structure can trigger the NOTIFY and RELEASE event of INTR 2. IPC N channel structure can trigger the NOTIFY event of INTR 1 and INTR 3, and the RELEASE event of INTR 0.

4.1.1 Implementation example operation of synchronization between CPUs

The section describes how to synchronize using the `IPC_STRUCTx_NOTIFY` register. In this use case, when `CPU_A` completes initialization of resources, `CPU_A` notifies interrupt to `CPU_B` using the `IPC_STRUCTx_NOTIFY` register. `CPU_B` waits to execute until it receives the notify interrupt.

4.1.2 Use case

Figure 7 shows an implementation example of CPU synchronization using IPC.

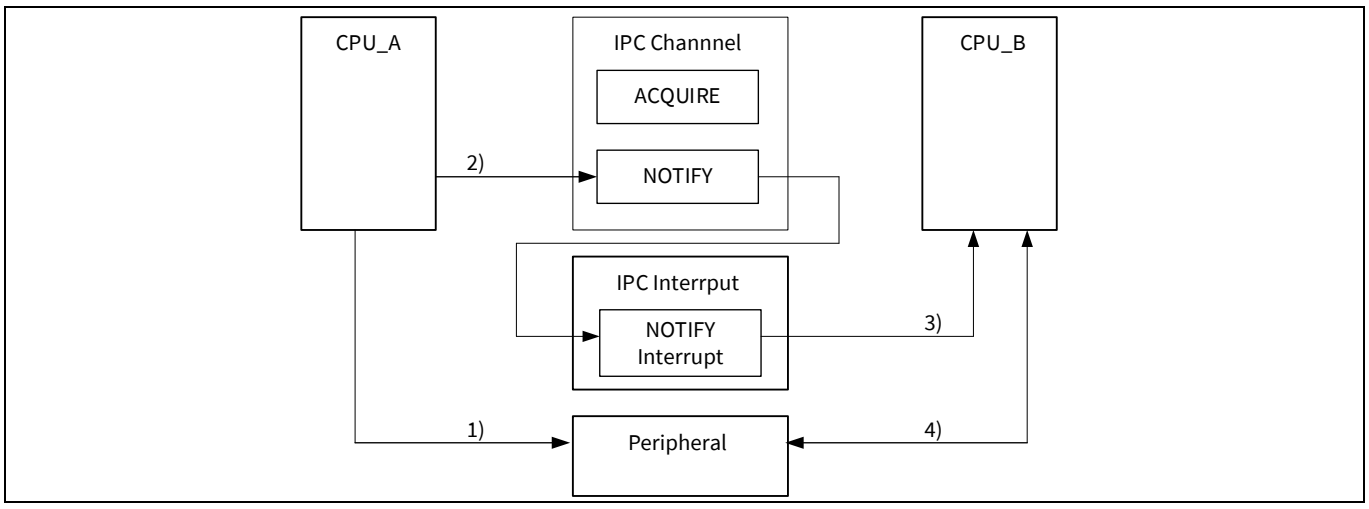

Figure 7 CPU synchronization using IPC

1. `CPU_A` initializes the peripheral.
2. After completing peripheral initialization, `CPU_A` generates a notify interrupt to `CPU_B`.
3. Then, a notify interrupt occurs in `CPU_B`.
4. `CPU_B` can start running the operation using the peripheral (initialized by `CPU_A`) after returning from the interrupt routine.

Figure 8 shows the flow of CPU synchronization.

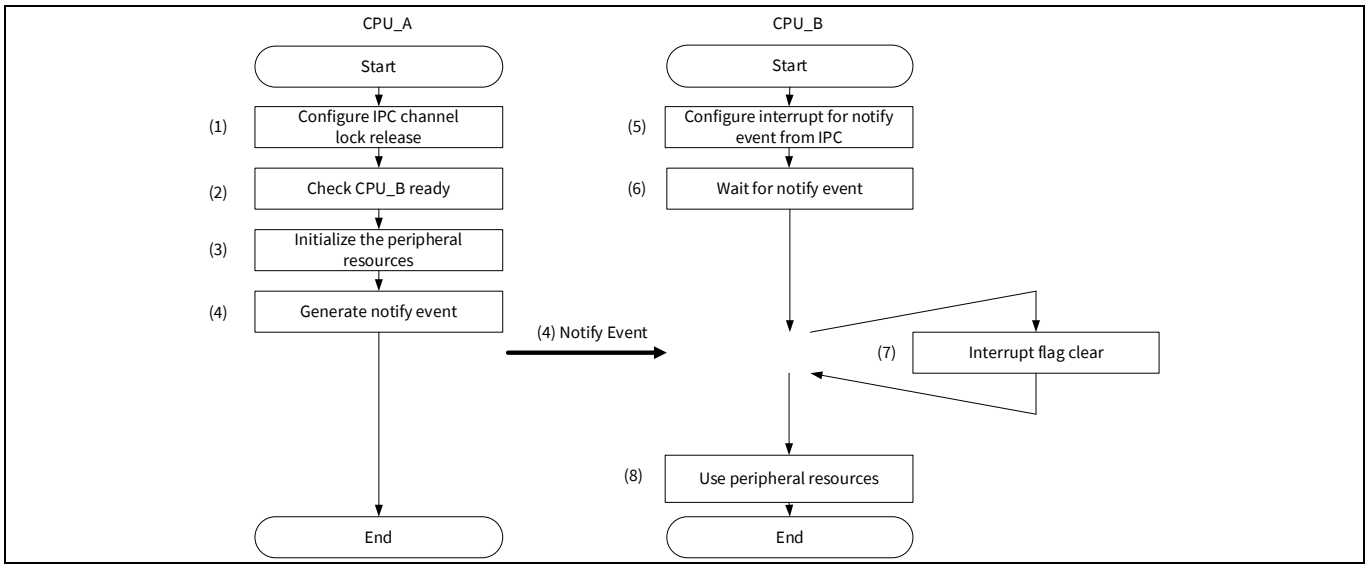

Figure 8 CPU synchronization operation

The following is the structure of the sample code.

- IPC channel structure: 6
- IPC interrupt structure: 5

See the architecture reference manual and AN234226 - XMC7000 MCU: Usage of Interrupts for interrupt configuration details.

4.1.3 Configuration

Table 2 and Table 3 list the parameters and functions in MTB CAT1 Peripheral Driver Library for CPU synchronization using IPC. This is an example of the XMC7000 series. Here, it assumes that `CPU_A` is CM7 and `CPU_B` is CM0+.

Table 2 List of parameters

Parameters	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Defines using IPC interrupt structure number for notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Defines using IPC channel structure number	6ul (IPC6 channel structure)
`CY_IPC_NO_NOTIFICATION`	Defines a value to indicate that no notification events are needed.	0x00000000ul
waitFlag	Indicates if peripheral initialization is complete	0: Completed 1: Not complete (Default)

Table 3 List of functions

Functions	Description	Remarks
`Cy_IPC_Drv_GetIpcBaseAddress(ipcIndex)`	Gets base address of IPC channel structure **ipcIndex**: IPC channel structure number	-
`Cy_IPC_Drv_LockRelease(base, releaseEventIntr)`	Releases a Lock of IPC channel **base**: Base address of IPC channel to operate **releaseEventIntr**: Specifies the release events	-
`Cy_IPC_Drv_GetIntrBaseAddr(ipcIntrIndex)`	Gets base address of IPC interrupt structure **ipcIndex**: IPC interrupt structure number	-
`Cy_IPC_Drv_GetInterruptMask(base)`	Gets value of INTR_MASK register **base**: Base address of IPC interrupt structure to operate	-
`Cy_IPC_Drv_ExtractAcquireMask(intMask)`	Gets value of NOTIFY field in INTR_MASK **intMask**: Value of INTR_MASK	-

Communicating between CPUs

Functions	Description	Remarks
`Cy_IPC_Drv_AcquireNotify(base, notifyEventIntr)`	Sets notify event to NOTIFY register **base**: Base address of IPC channel structure **notifyEventIntr**: Value of notify event setting	-
`Cy_IPC_Drv_IsLockAcquired(base)`	Checks if the lock is acquired **base**: Base address of IPC channel structure to operate	-
`Cy_IPC_Drv_ReleaseNotify(base, notifyEventIntr)`	Sets release event to RELEASE register. **base**: Base address of IPC channel structure **notifyEventIntr**: Value of release event setting	-
`Cy_IPC_Drv_SetInterruptMask(base, ipcReleaseMask, ipcNotifyMask)`	Sets interrupt to INTR_MASK register **base**: Base address of IPC interrupt structure number **ipcReleaseMask**: Value of release event setting **ipcNotifyMask**: Value of notify event setting	-
`Cy_IPC_Drv_GetInterruptStatusMasked(base)`	Gets value of INTR_MASKD register **base**: Base address of IPC interrupt structure to operate	-
`Cy_IPC_Drv_ClearInterrupt(base, ipcReleaseMask, ipcNotifyMask)`	Clears interrupt flag **base**: Base address of IPC interrupt structure to operate **ipcReleaseMask**: Clears data for release event **ipcNotifyMask**: Clears data for notify event	-

4.2 Mutual exclusion operation

This section describes how to mutually exclude shared resource access between CPUs using IPC. In a multi-CPU architecture, each CPU can share memory and peripherals, such as data exchange or external serial communication.

As an example, consider the situation where two CPUs (CPU_A and CPU_B) share memory. CPU_A is supposed to read and update memory data. Then, CPU_B is supposed to read and update the same memory data, but only after CPU_A completes the operation. However, if CPU_A reads memory data, but CPU_B updates memory data before CPU_A updates memory data, there will be a mismatch between the actual memory data and the expected memory data because CPU_B is supposed to update the data written by CPU_A.

To avoid this issue, CPU_B should not be allowed to access the memory while CPU_A is reading and updating data. That is reads and updates by each CPU need atomic operations.

IPC in XMC7000 can easily implement exclusive access using the IPC_STRUCTx_ACQUIRE register. This register has a lock feature of IPC channel structure. A lock of the IPC channel structure is acquired by reading this register.
Table 4 shows the result of the ACQUIRE register read operation.

Table 4 IPC_STRUCTx_ACQUIRE register operation

Result of Read access	IPC channel structure status
0	IPC channel structure lock failed.
1	IPC channel structure lock successful.

If the register is already in an acquired state, another master cannot acquire it. The acquired state of the IPC channel structure is provided by the IPC_STRUCTx_LOCK_STATUS register. The acquired state of the IPC channel structure is released by writing any value into the IPC_STRUCTx_RELEASE register and allows for the generation of release events to the IPC interrupt structure.

4.2.1 Implementation example of mutual exclusion

This section describes an example of mutual exclusion access. This use case assumes that CPU_A and CPU_B access common peripheral resources. Each CPU write access must be atomic access. An IPC channel structure is associated with a common peripheral resource, and when accessing a common peripheral resource, each CPU must acquire a lock on the associated IPC channel structure. Therefore, a CPU that cannot acquire the IPC channel structure lock is not allowed to access common peripheral.

4.2.2 Use case

Figure 9 shows an implementation example of common peripheral exclusive access using IPC.

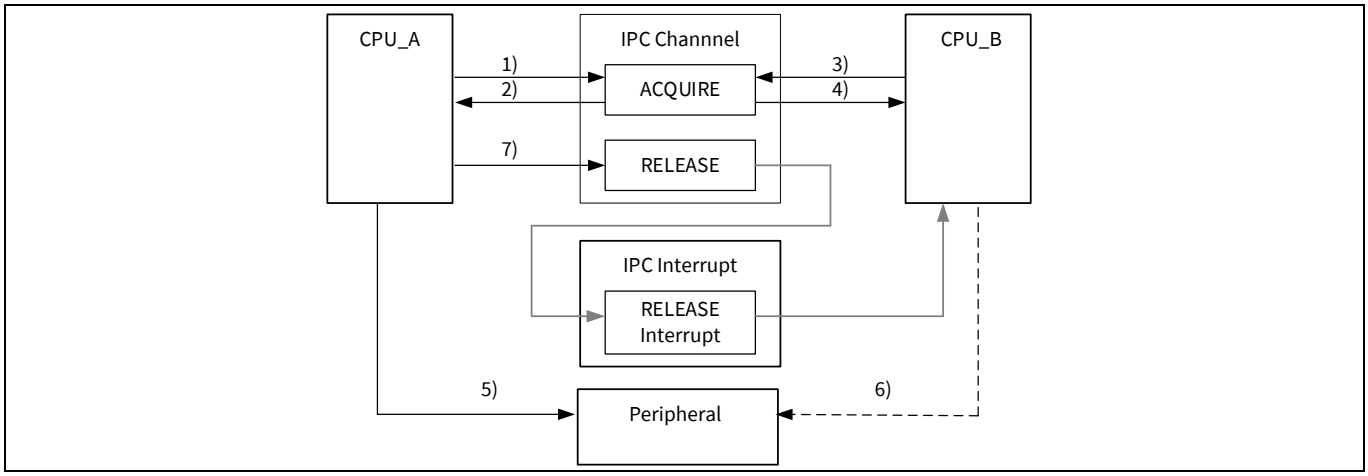

Figure 9 Example of exclusive access

The following shows an example of exclusive control implementation:

1. CPU_A reads the IPC_STRUCTx_ACQUIRE register before CPU_A accesses the common peripheral.
2. When CPU_A reads “1” from the IPC_STRUCTx_ACQUIRE register, CPU_A is successful in acquiring the IPC channel structure lock.
3. CPU_B reads the IPC_STRUCTx_ACQUIRE register for accessing common peripheral after CPU_A has acquired the IPC channel structure lock.
4. CPU_B reads "0" from the IPC_STRUCTx_ACQUIRE register. This indicates that CPU_B cannot acquire the IPC channel structure lock.
5. CPU_A reads and writes to common peripheral.
6. CPU_B which could not acquire the IPC channel structure lock is not allowed to access the common peripheral.
7. CPU_A releases the IPC channel structure lock by writing to the IPC_STRUCTx_RELEASE register when writing to a common peripheral is complete. If IPC interrupt structure is set to generate release interrupt by IPC_STRUCTx_RELEASE register write, IPC interrupt structure notifies the release interrupt to CPU_B.

Note: IPC has no hardware to restrict resource access. Therefore, software must have strict rules not to access shared memory if it cannot acquire the lock.

Figure 10 shows the example flow for mutual exclusion.

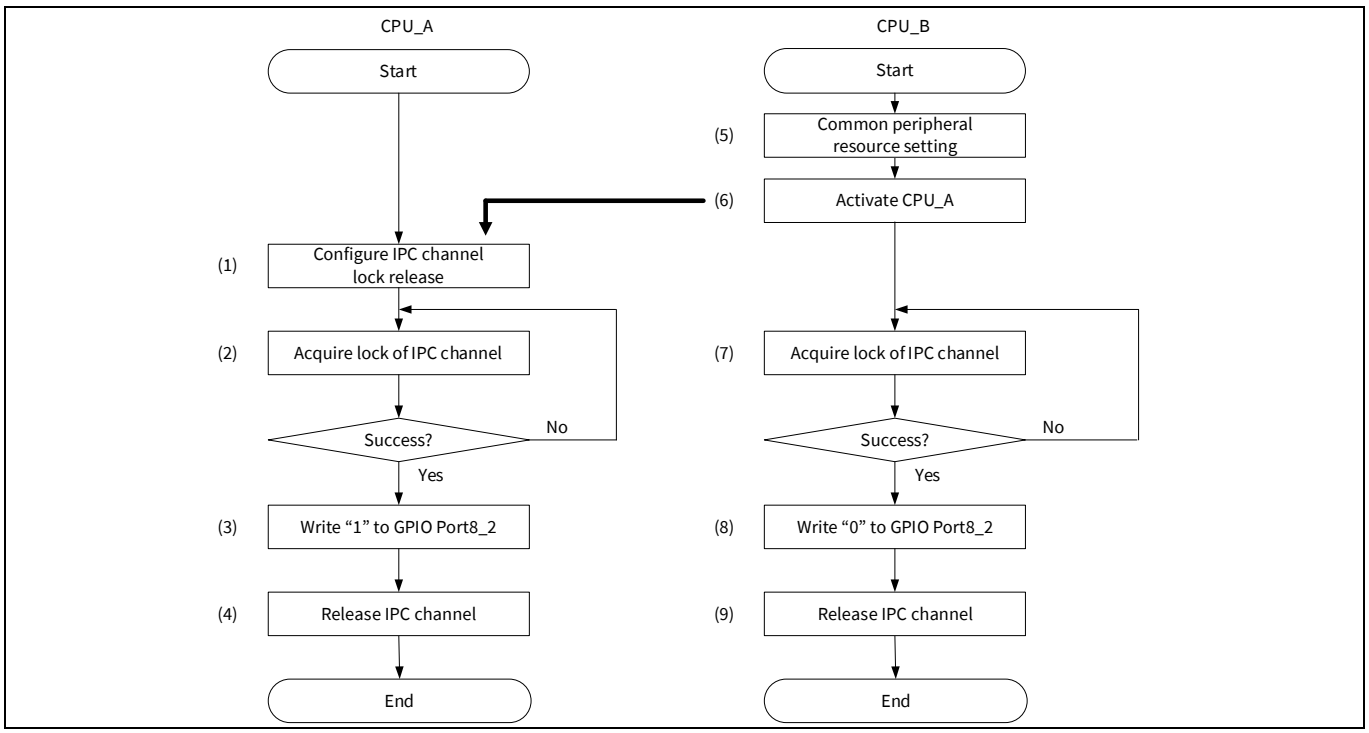

Figure 10 Mutual exclusion flow

The following shows the structure of the sample code.

- IPC channel structure: 6
- Common Peripheral: Port8 (2pin)

See the architecture reference manual and AN234118 - GPIO usage setup in XMC7000 family for GPIO configuration details.

4.2.3 Configuration

Table 5 and Table 6 list the parameters and functions in MTB CAT1 Peripheral Driver Library for mutual exclusion using IPC. This is example in XMC7000 series. In this case, it is assumed that `CPU_A` is CM7 and `CPU_B` is CM0+.

Table 5 List of parameters

Parameters	Description	Value
`IPC_CHANNEL_NUMBER`	Define using IPC channel structure number	6ul (IPC6 channel structure)
`CY_IPC_NO_NOTIFICATION`	Defines a value to indicate that no notification events are needed.	0x00000000ul

Table 6 List of functions

Function	Description	Remark
`Cy_IPC_Drv_LockAcquire(base)`	Acquire IPC channel lock base: Base address of IPC channel to operate	-

4.3 Data passing

This section describes how to pass data between CPUs using IPC. In a multi-CPU architecture, each CPU may pass a message to the other CPUs. In this case, IPC can be used.

4.3.1 Implementation example of passing small data (up to 64 bits)

This section describes passing data of 64 bits or less. If the message data is 64 bits or less, `IPC_STRUCTx_DATA0/1` can be used for data passing. `IPC_STRUCTx_DATA0/1` has two 32-bit registers. A message of up to 64 bits can be written to these registers to be sent to other CPUs.

4.3.1.1 Use case

Figure 11 shows an implementation example of small message communication using IPC. In this example, `CPU_A` passes the message to `CPU_B`.

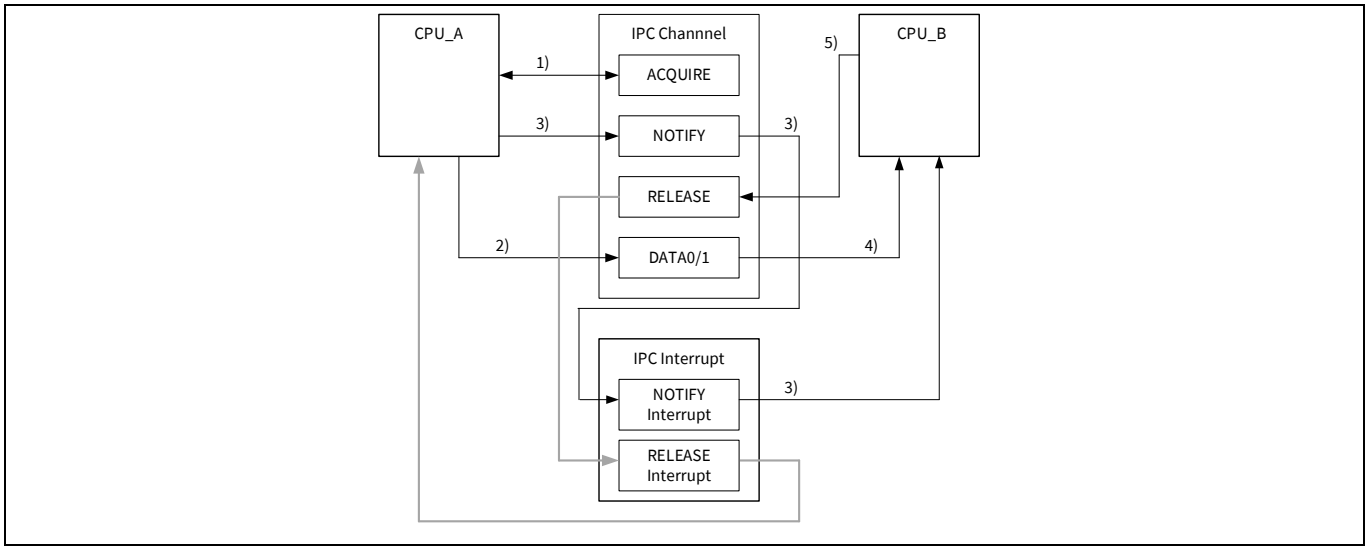

Figure 11 Example of passing small message

The following shows an example of passing data up to 64 bits:

1. `CPU_A` reads the `IPC_STRUCTx_ACQUIRE` register. When `CPU_A` reads “1” from the `IPC_STRUCTx_ACQUIRE` register, `CPU_A` is successful in acquiring the IPC channel structure lock.
2. After the IPC channel structure is locked, `CPU_A` places message data up to 64 bits in the `IPC_STRUCTx_DATA0/1` register.
3. Now that the message is placed in the IPC channel, `CPU_A` generates a notify event to `CPU_B` by setting the corresponding bit in the `IPC_STRUCTx_NOTIFY` register.

4. When CPU_B accepts the notify interrupt, CPU_B can read the IPC_INTR_STRUCTx_INTR_MASKED register to know which IPC channel triggered the notify event. Based on this, CPU_B identifies the channel to read and reads from the IPC_STRUCTx_DATA0/1 register.
5. After receiving the message, CPU_B releases the IPC channel structure so that other processors/processes can use it. It also optionally generates a release event to CPU_A. This will generate a release event interrupt to CPU_A when the corresponding bit of IPC_INTR_STRUCTx_INTR_MASK is not masked.

Note: IPC has no hardware to restrict resource access. Therefore, CPU_B software must have strict rules not to access IPC_STRUCTx_DATA0/1 if it does not receive notify interrupt.

Figure 12 shows the example flow for data passing (up to 64 bits).

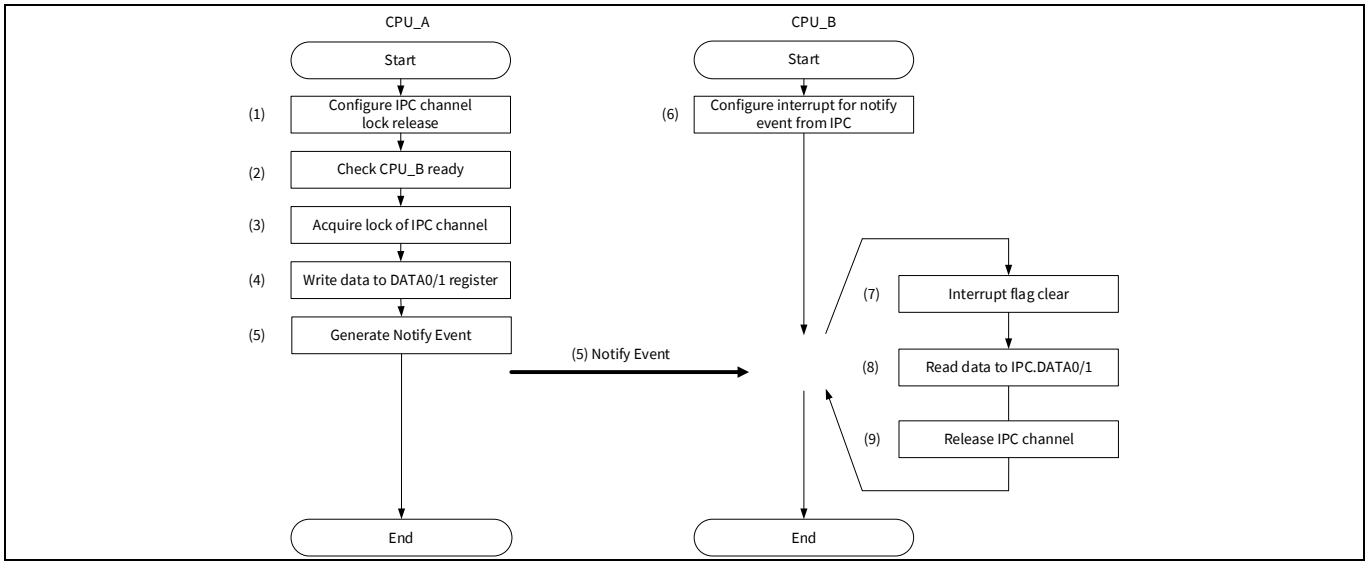

Figure 12 Data passing (up to 64 bits) flow

The following shows the structure of the sample code.

- IPC channel structure: 6
- IPC interrupt structure: 5

See the architecture reference manual and AN234226 - XMC7000 MCU: Usage of Interrupts for interrupt configuration details.

4.3.1.2 Configuration

Table 7 and Table 8 list the parameters and functions in the MTB CAT1 Peripheral Driver Library for data passing of 64 bits or less using IPC. This is an example in the XMC7000 series. In this case, it is assumed that `CPU_A` is CM7 and `CPU_B` is CM0+.

Table 7 List of parameters

Parameters	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Define using IPC interrupt structure number for notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Define using IPC channel structure number	6ul (IPC6 channel structure)
`IPC_DATA`	Define a passing data 0	0x5A5A5A5Aul
`IPC_DATA2`	Define a passing data 1	0x12345678ul
`CY_IPC_NO_NOTIFICATION`	Defines a value to indicate that no notification events are needed	0x00000000ul

Table 8 List of functions

Functions	Description	Remarks
`Cy_IPC_Drv_SendMsgWord(base, notifyEventIntr, message, message2)`	Set DATA0/1 register of IPC channel structure. **base**: Base address of IPC channel to operate. **notifyEventIntr**: Value of notify event setting. **message**: Write data to IPC.DATA0. **message2**: Write data to IPC.DATA1.	It has function of acquire lock and notify event generation.
`Cy_IPC_Drv_ReadMsgWord(base, message, message2)`	Read DATA0/1 register of IPC channel structure. **base**: Base address of IPC channel to operate. **message**: Stored address for IPC.DATA0. **message2**: Stored address for IPC.DATA0.	-

4.3.2 Implementation example of passing large data (more than 64 bits)

This section describes the passing of a large message. Larger messages can be sent as pointers. CPU_A can allocate a larger message structure in the shared memory and use the 32-bit IPC_STRUCTx_DATA0/1 register to pass the pointer and size on which the message is placed to CPU_B.

4.3.2.1 Use case

Figure 13 shows an implementation example of large message communication using IPC.

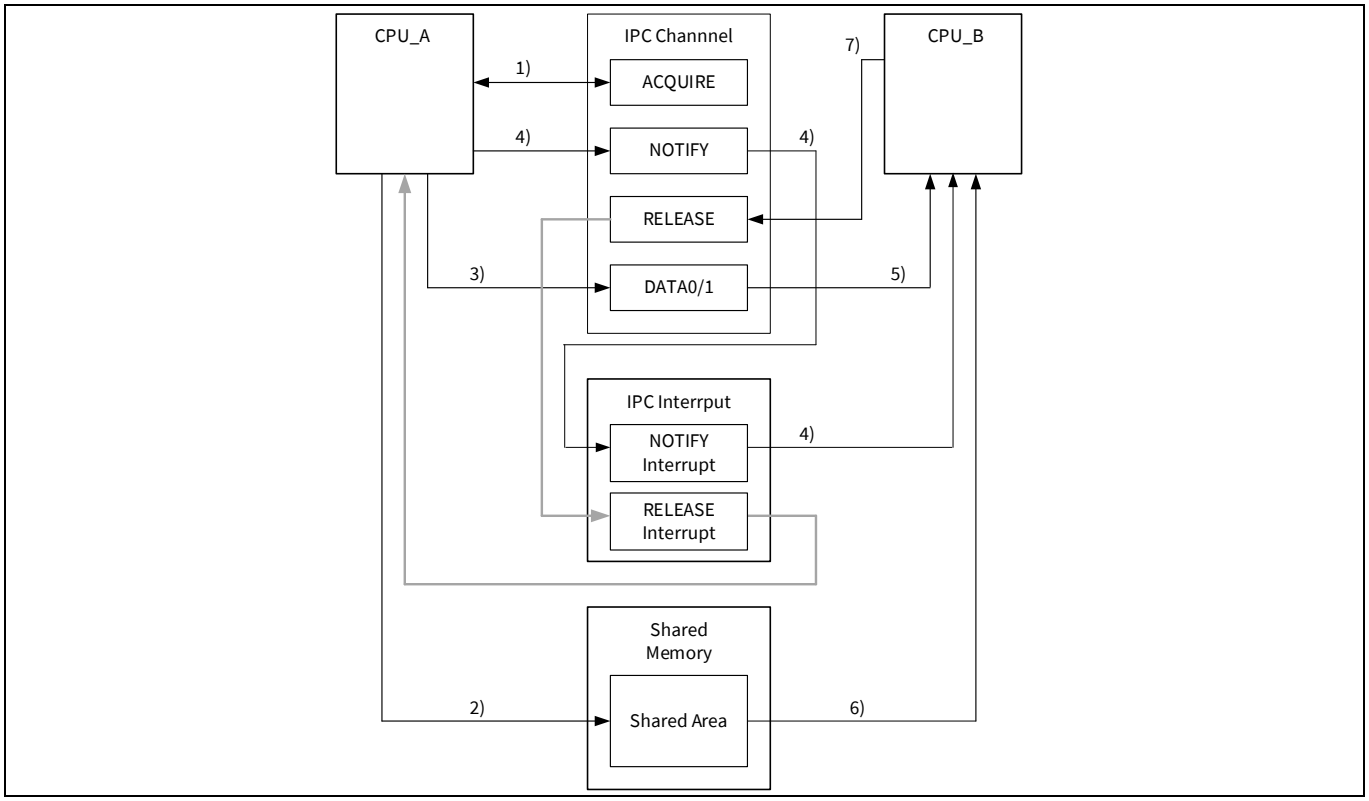

Figure 13 Example of passing large message

The following shows an example of passing data more than 64 bits:

1. CPU_A reads the IPC_STRUCTx_ACQUIRE register. When CPU_A reads “1” from the IPC_STRUCTx_ACQUIRE register, CPU_A is successful in acquiring the IPC channel structure lock.
2. After the IPC channel structure is locked, CPU_A places message data in the shared memory.
3. Then, CPU_A places the message data pointer and size of the shared memory in the IPC_STRUCTx_DATA0/1 register.
4. Now that the message and pointer are placed, CPU_A generates a notify event to CPU_B by setting the corresponding bit in the IPC_STRUCTx_NOTIFY register.
5. When CPU_B accepts the notify interrupt, CPU_B can read the IPC_INTR_STRUCTx_INTR_MASKED register to know which IPC channel had triggered the notify event. Based on this, CPU_B identifies the channel to read and reads pointer and size from IPC_STRUCTx_DATA0/1 register.
6. CPU_B reads message data of the specified size from the address indicated by the pointer.

7. After receiving the message, CPU_B releases the IPC channel structure so that other processors/processes can use it. It also optionally generates a release event to CPU_A. This will generate a release event interrupt to the CPU_A when the corresponding bit of IPC_INTR_STRUCTx_INTR_MASK is not masked.

Note: IPC has no hardware to restrict resource access. Therefore, CPU_A and CPU_B software must have strict rules not to access IPC_STRUCTx_DATA0/1 and message data in shared memory if it does not receive notify interrupt.

Figure 14 shows the example flow for data passing (more than 64 bits).

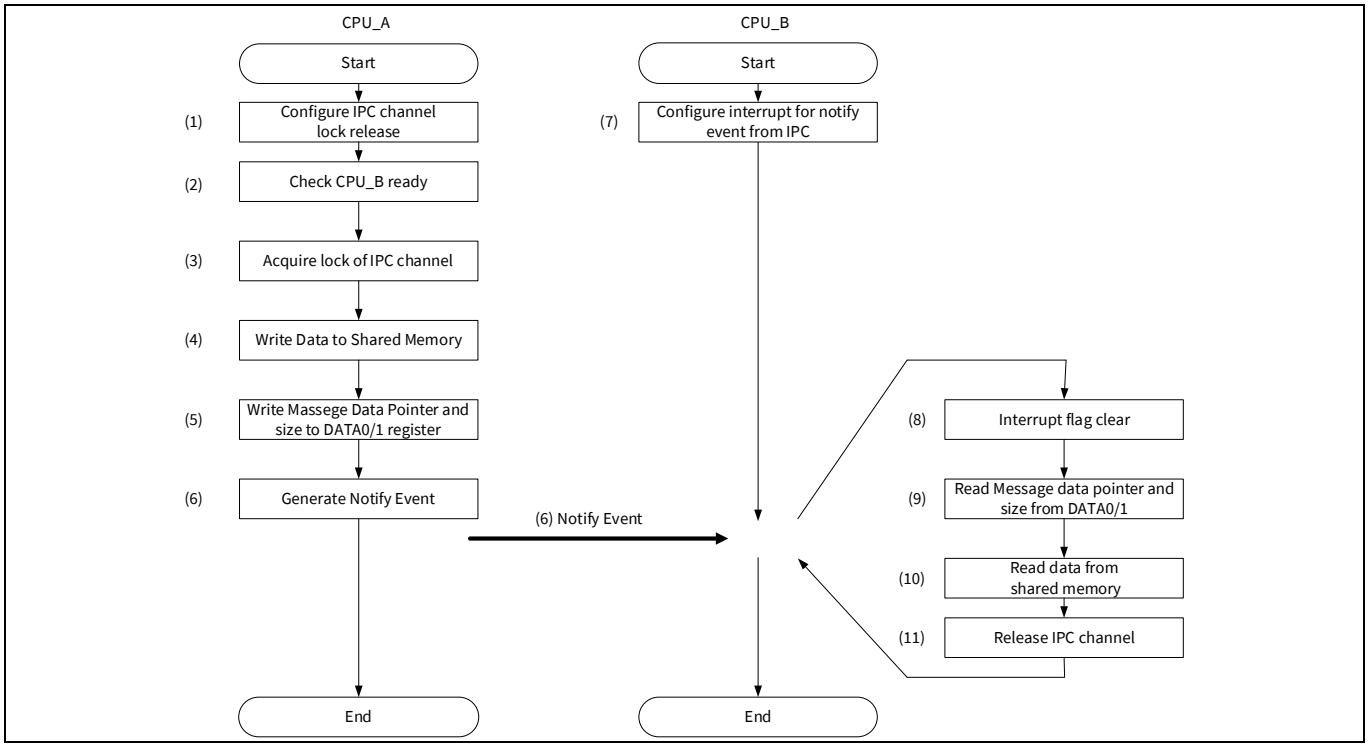

Figure 14 Data passing (more than 64 bits) flow

The following shows the structure of the sample code.

- IPC channel structure: 6
- IPC interrupt structure: 5
- Shared memory: SRAM
- Data size: 4 words (16 bytes)

4.3.2.2 Configuration

Table 9 lists the parameters and functions in MTB CAT1 Peripheral Driver Library for data passing of more than 64 bits using IPC. This is an example in the XMC7000 series. In this case, it is assumed that `CPU_A` is CM7 and `CPU_B` is CM0+.

Table 9 List of parameters

Parameters	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Define using IPC interrupt structure number for notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Define using IPC channel structure number	6ul (IPC6 channel structure)
`DATA_SIZE`	Define Passing data size	4ul (4 word)
`sharedData[]`	Shared memory area on SRAM	-
`ipc_data[]`	Passing data	0x12345678ul, 0x87654321ul, 0x12345678ul, 0x87654321ul
`CY_IPC_NO_NOTIFICATION`	Defines a value to indicate that no notification events are needed.	0x00000000ul

5 Considerations for cache coherency issue

A cache memory helps to improve CPU performance from its high-speed read/write operation. However, the characteristics of cache memory can cause a data mismatch between cache memory and other memories, that is, cache coherency issue. Cache coherency issue should be mainly considered in the XMC7000 series which has cache memory in CPU. This section provides an overview of cache memory in these series and explains the cache coherency issue under different scenarios. In addition, it provides methods to manage or avoid the cache coherency issue. In this section, the shared memory referred to is SRAM unless otherwise specified.

5.1 Cache coherency

Coherency is a consistency of the common area used by multiple bus masters. When the common area is the same view for multiple bus masters, this area is coherent.

CPU can read or update only the cache memory depending on the cache memory configuration. If the CPU reads data from the cache memory after another master updated the shared memory that is allocated to cache memory, the view of CPU (cache memory) and the other masters (shared memory) will be different. Thus, this area is not coherent.

In this case, the CPU and other masters may operate using different data, causing an unintended operation. It is a cache coherency issue. Figure 15 shows a general example of coherency issue occurrence. As a precondition, shared memory is allocated to the cache memory.

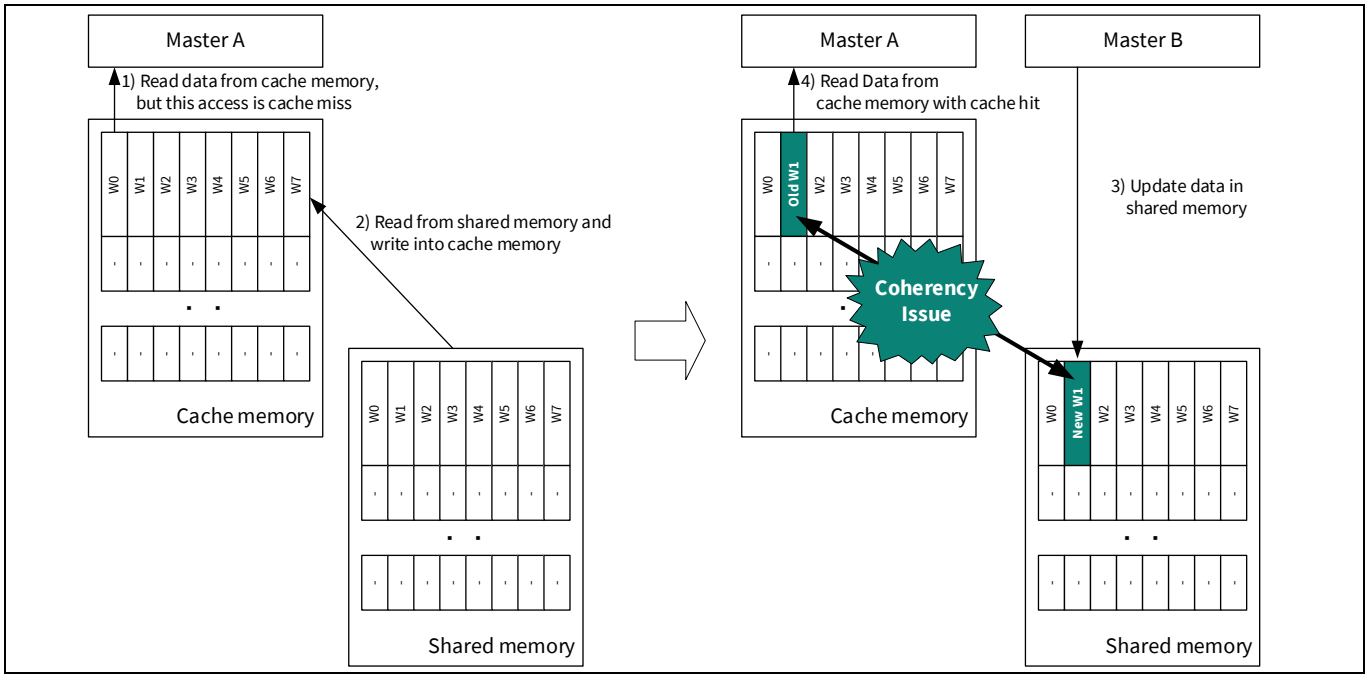

Figure 15 Coherency issue example

1. The cache memory does not have data before the start of the operation.
2. Master A tries to read data from the cache memory. However, the cache memory does not have data. Therefore, this access causes a cache miss.
3. As a result of a cache miss, the cache memory reads data from the shared memory. The cache memory data and the shared memory data are the same at this point. Therefore, they are coherent. Subsequent accesses to this address are cache hit.

4. Master B updates data (New W1) in shared memory. As a result, the cache memory data and the shared memory data are different. Therefore, they are not coherent.
5. Master A reads data from the cache memory. The cache memory has data (old W1), therefore, the cache hit. As a result of the cache hit, master A reads old W1 from cache memory. Master A starts to operate using different data. A coherency issue occurs.

Cache management is important for a system with the cache memory and multiple masters.

5.2 Cache memory overview

This section describes the location and behavior of cache memory implemented in this series.

5.2.1 Cache memory placement

Figure 16 shows the placement of cache memory.

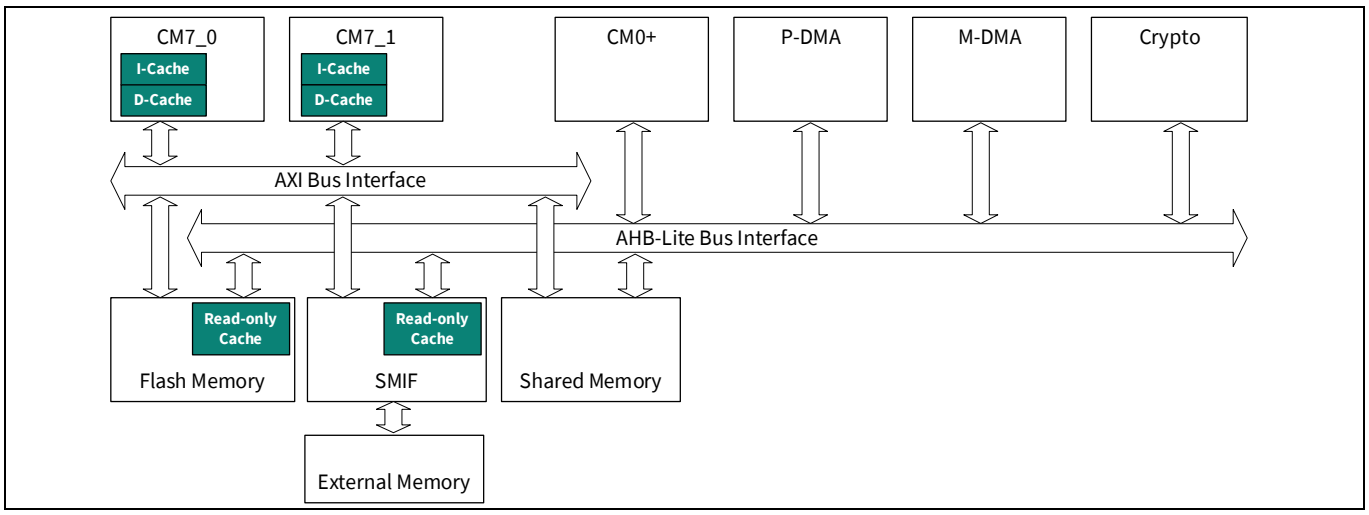

Figure 16 Cache memory placement

In these series, CM7 CPUs have I-cache and D-cache, and flash memory and serial memory interfaces (SMIF) have read-only cache memory for AHB-Lite Bus interface.

5.2.2 I-cache and D-cache operation

The I-cache and D-cache are implemented as part of the CM7. These cache memories are valid for the access that is for an AXI bus interface. When the access to cacheable memory on the AXI bus interface and the cache is enabled, this access attempts a lookup in the cache memory.

5.2.2.1 Cache memory behavior

When the CPU finds data in the cache memory, that is, a cache hit, the data is read from the cache memory or written into the cache memory. Table 10 lists the behavior of cache memory in CM7. This operation assumes that shared memory is allocated to a cache memory.

Table 10 Behavior of CM7 cache memories

Operation			Description
Read Access	Cache hit		Data is read from the cache memory
	Cache miss		All cacheable area is Read-Allocate. Cache memory allocates a memory location to a cache line. When a cache line is allocated, the shared memory data is fetched and written to the cache memory. Then, read access to these memory locations will be a cache hit, and data is read from the cache memory.
Write Access	Cache hit	Write-Back	The access is written into the cache memory. The cache line is marked as dirty, and the data in the cache memory is only written to the shared memory when the line is evicted.
		Write-Through	The access is written into the cache memory. The data is also written to the shared RAM so that the data stored in the cache memory is coherent with the shared memory.
	Cache miss	Write Allocate	Cache memory allocates a memory location to a cache line. When a cache line is allocated, the shared memory data is fetched, and written to the cache memory.
		No Write Allocate	Cache memory does not allocate a memory location to a cache line. The data is written to shared memory.

5.2.2.2 Cache memory configuration

Following configurations are supported for cache memory in CM7. Cache memories in CM7 can be configured using a CM7 specific register.

- Non-cache
  - Cache memory does not work. Always read and write on the shared memory.
  - This configuration does not require consideration of cache coherency issues.
- Write-back, write, and read allocate
  - The cache hit of read access reads from the cache memory.
  - The cache hit of write access updates only the cache memory.
  - The cache miss of read and write access copies data from the shared memory to the cache memory.
  - This configuration must require full consideration of coherency issue.
- Write-back, no write allocate
  - The cache hit of read access reads from the cache memory.
  - The cache miss of read access copies data from the shared memory to the cache memory.
  - The cache hit of write access updates only the cache memory.
  - The cache miss of write access does not copy data from the shared memory to the cache memory.
  - This configuration must require full consideration of coherency issue.
- Write-through, no write allocate
  - The cache hit of read access reads from the cache memory.
  - The cache miss of read access copies data from the shared memory to the cache memory.
  - The cache hit or miss of write access performs on the shared memory.
  - This configuration solves cache coherency issue partially.

These configurations are available in the MPU Region Attribute and Size Register (MPU_RASR). Table 11 shows MPU_RASR common combination for cache configuration. The configuration of cache memory is defined by TEX, C, and B in MPU_RASR.

Table 11 TEX, C, and B encoding

TEX	C	B	Memory type	Description
000b	0b	0b	Strongly-ordered	Non-cacheable
	0b	1b	Device	Non-cacheable
	1b	0b	Normal	Write-through, no write allocate
	1b	1b		Write-back, no write allocate
001b	0b	0b		Non-cacheable
	1b	1b		Write-back; write, and read allocate

See the Arm® documentation sets of CM7 for the complete details related to TEX, C, and B encoding.

5.2.2.3 Cache maintenance operation

I-cache and D-cache support the following operations for cache maintenance:

- **Enable and Disable**: Cache ON/OFF. A CPU access is direct to the shared memory when the cache is OFF.
- **Invalidate**: Clear the valid bit of the cache line. Data in the cache memory is invalidated. Subsequent access is cache miss; data is fetched from shared memory and written to the cache memory.
- **Clean**: Write the updated data in cache memory back to the shared memory. The data of the shared memory match cache memory.

To perform these cache maintenances, you can use the Cortex® Microcontroller Software Interface Standard (CMSIS). Table 12 lists cache maintenance APIs supported by CMSIS.

Table 12 Cache maintenance APIs

Cache maintenance APIs	Description
`SCB_EnableICache (void)`	Invalidates and then enables I-cache
`SCB_DisableICache (void)`	Disables I-cache and invalidates its contents
`SCB_InvalidateICache (void)`	Invalidates I-cache
`SCB_EnableDCache (void)`	Invalidates and then enables D-cache
`SCB_DisableDCache (void)`	Disables D-cache and then cleans and invalidates its contents
`SCB_InvalidateDCache (void)`	Invalidates D-cache
`SCB_CleanDCache (void)`	Cleans D-cache
`SCB_CleanInvalidateDCache (void)`	Cleans and invalidates D-cache
`SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Invalidates D-cache by address **addr**: Address aligned to 32-byte boundary **dsize**: Size of the memory block in bytes
`SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Cleans D-cache by address **addr**: Address aligned to 32-byte boundary **dsize**: Size of the memory block in bytes

Considerations for cache coherency issue

Cache maintenance APIs	Description
`SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Cleans and invalidates D-cache by address **addr**: Address aligned to 32-byte boundary **dsize**: Size of the memory block in bytes

See the Arm® documentation sets of CM7 for more details.

Code Listing 1 to Code Listing 3 show examples of using some cache maintenance APIs.

Code Listing 1 Example of using the cache maintenance API (1)

```
Void Startup_Init(void)
{
    SCB_EnableICache();
    SCB_EnableDCache();
}
```

Code Listing 2 Example of using the cache maintenance API (2)

```
void SystemInit (void)
{
    // Ensure cache coherency (e.g., in case ROM-to-RAM copy of code
sections happened during startup)
    SCB_CleanInvalidateDCache();
    SCB_InvalidateICache();

}
```

Code Listing 3 Example of using the cache maintenance API (3)

```
#define BUFFER_SIZE                256ul

static uint8_t srcBuffer[BUFFER_SIZE]  __ALIGNED(32); // Align to 32-byte
boundary to simplify cache maintenance
static uint8_t dstBuffer[BUFFER_SIZE]  __ALIGNED(32); // Align to 32-byte
boundary to simplify cache maintenance

int main(void)
{
    /* Set up internal routing, pins, and clock-to-peripheral connections
*/
```

Code Listing 3 Example of using the cache maintenance API (3)

```
    init_cycfg_all();
    // Preset source buffer with test pattern and clear destination
    for(uint32_t i = 0; i < BUFFER_SIZE; i++)
    {
        srcBuffer[i] = (uint8_t) i;
        dstBuffer[i] = 0;
    }
    // Ensure buffer data is cleaned out to SRAM (so that it can be
accessed by DMA later on)
    SCB_CleanDCache_by_Addr((uint32_t *) srcBuffer, sizeof(srcBuffer));
    SCB_CleanDCache_by_Addr((uint32_t *) dstBuffer, sizeof(dstBuffer));


    // Initialize DMA
    // Ensure descriptor data is cleaned out to SRAM (so that it can be
accessed by DMA later on)
    SCB_CleanDCache_by_Addr((uint32_t *) &descriptor3D,
sizeof(descriptor3D));


    // Trigger DMA transfer by SW


    // Destination buffer has been modified by DMA, so the corresponding
area needs to be invalidated before accessing it by CPU
    SCB_InvalidateDCache_by_Addr((uint32_t *) dstBuffer,
sizeof(dstBuffer));


    // Check for expected data
    for(uint32_t i = 0; i < BUFFER_SIZE; i++)
    {
    }
    for(;;)
    {
   }
}
```

5.2.3 Cache memory operation in flash memory

Table 13 shows the behavior of flash memory cache memory. This cache memory is read-cache. Therefore, write access data is directly written into associated memories.

Table 13 Behavior of cache memory in flash memory

Operation		Description
Read Access	Cache hit	Data is read from the cache memory
	Cache miss	Access occurs to Flash memory, and 16-Bytes data are refilled from Flash memory to cache memory. Subsequent access result is cache hit.
Write Access		The write access is bypass the cache memory. In Flash memory, the write access without specific sequence is generally causes access error.

In general, flash memory does not rewrite as frequently as RAM. Also, flash memory is most often written under specific conditions according to system requirements. Therefore, the cache memory can avoid the coherency issues by clearing the cache memory after rewriting the flash memory. Table 14 lists the control registers to invalidate and enable/disable the cache memory. Cache memory can be enabled/disabled using the register. When cache memory is set to disable and enable again, data in the cache memory is invalidated and read access causes refilling in the cache memory. See the registers reference manual for more details.

Table 14 Flash memory cache invalidate and enable control register

Register name	Bit field	Description
FLASHC_FLASH_CMD	INV	Invalidation of all caches and buffers: Software writes a "1" to clear the caches. Hardware sets this field to "0" when the operation is completed.
FLASHC_CM0_CA_CTL	CA_EN	Cache enable: 0: Disabled 1: Enabled (Default)

5.2.4 Cache memory operation in SMIF

Table 15 lists the behavior of SMIF cache memories. This cache memory is a read-cache. Therefore, write access data is directly written into associated memories.

Table 15 Behavior of cache memory in SMIF

Operation		Description
Read Access	Cache hit	Data is read from the cache memory
	Cache miss	Access occurs to external memory and 16 bytes data are refilled from external memory to cache memory. Subsequent access results in a cache hit.
Write Access		The write access bypasses the cache memory. The data is directly written into external memory. A write to an address in the read-only cache invalidates the associated cache subsector.

SMIF has three interfaces: XIP AXI, XIP AHB-Lite, and MMIO AHB-Lite interface. Out of the three interfaces, only the XIP AHB-Lite interface has cache memory. In addition, this cache memory does not support cache coherency by hardware. Therefore, SMIF has a cache coherency issue depending on access between each port. Table 16 lists the control registers for invalidating and enabling/disabling cache memory. See the registers reference manual for more details.

Table 16 SMIF cache invalidate and enable control register

Register name	Bit field	Description
SMIF_STATUS	BUSY	SMIF status: '0': Not busy '1': Busy When BUSY is '0', the SMIF can be safely disabled or the mode of operation can be safely changed.
SMIF_SLOW_CA_CMD	INV	Cache and prefetch buffer invalidation. Software writes a '1' to clear the cache and prefetch buffer. The cache's LRU structure is also reset to its default state. Note that the software should invalidate the cache and prefetch buffer only when SMIF_STATUS.BUSY is '0'.
SMIF_SLOW_CA_CTL	PREF_EN	Prefetch enable: '0': Disabled '1': Enabled (Default) Prefetching requires the cache to be enabled; ENABLED is '1'.
	ENABLED	Cache enable: '0': Disabled '1': Enabled (Default)

5.3 Cache coherency handling

Cache coherency issues are caused when a cache memory and shared memory cannot keep their consistency. This section describes how to manage or avoid cache memory and shared memory coherency issues.

5.3.1 Cache disable

Each CPU is configured to be ‘cache disable’. A read/write access of each CPU is performed for the shared memory without cache memory. No actions are required for the cache memory coherency issue.

5.3.2 Cache invalidate

The ‘cache invalidate’ is used to update the cache memory when the shared memory has been updated by the other master. When cache invalidate is performed, the valid bit in the cache memory is cleared and the data in the cache memory is invalid. Subsequent read accesses result in a cache miss. As a result, the cache memory reads the shared memory data. The cache memory and shared memory can keep their coherency. This handling can use cache maintenance API such as `SCB_InvalidateDCache_by_Addr`.

5.3.3 Cache clean

The cache clean is used to update the shared memory when the cache memory has been updated by the CPU. The updated data in cache memory write back to shared memory by this handling. The cache memory and shared memory can keep their coherency. This handling can use cache maintenance API such as `SCB_CleanDCache_by_Addr`.

5.3.4 Cache configuration sets to Write-through

In Write-through configuration, the CPU writes to shared memory directly, not cache memory. This configuration keeps the coherency between cache memory and shared memory for only write access. This configuration solves the cache coherency issue partially.

5.3.5 Use TCM as shared memory

Each CM7 CPU has ITCM/DTCM. These memories can be accessed by each master through the AHB bus interface. As mentioned above, I-cache and D-cache memories are valid access for an AXI bus interface. Thus, ITCM and DTCM can access without cache memory. Therefore, ITCM/DTCM can be used as shared memory without consideration for cache coherency issues, except when CM7 accesses the TCM area of another CM7. Note that CM7 uses the AXI bus interface when accessing another CM7 TCM. All bus masters can access ITCM and DTCM using dedicated address space. No actions are required for the cache memory coherency issue. See the device datasheet for TCM address mapping.

5.4 Cache coherency issue scenarios

This section describes cache coherency issue under different scenarios and provides solutions.

5.4.1 Cache coherency issue between CM7 CPUs

This section describes the scenario of cache coherency issue between CPUs. The coherency issue between each CPU cache memory is complex. The coherency must be considered between the cache memory of each CPU and shared memory.

5.4.1.1 Scenario and solution between CM7 CPUs

CM7 has I-cache and D-cache. Cache coherency issue mainly occurs with D-cache that handles data. Figure 17 shows the cache coherency issue scenario in this case. The preconditions are as follows:

- Each CPU uses a part of the shared memory as a common area, and the common area enables a cache.
- Each CPU cache configuration is Write-back, write, and read allocate.
- Data is sent from `CM7_1` to `CM7_0`. That is, `CM7_1` writes the data and `CM7_0` reads the data.

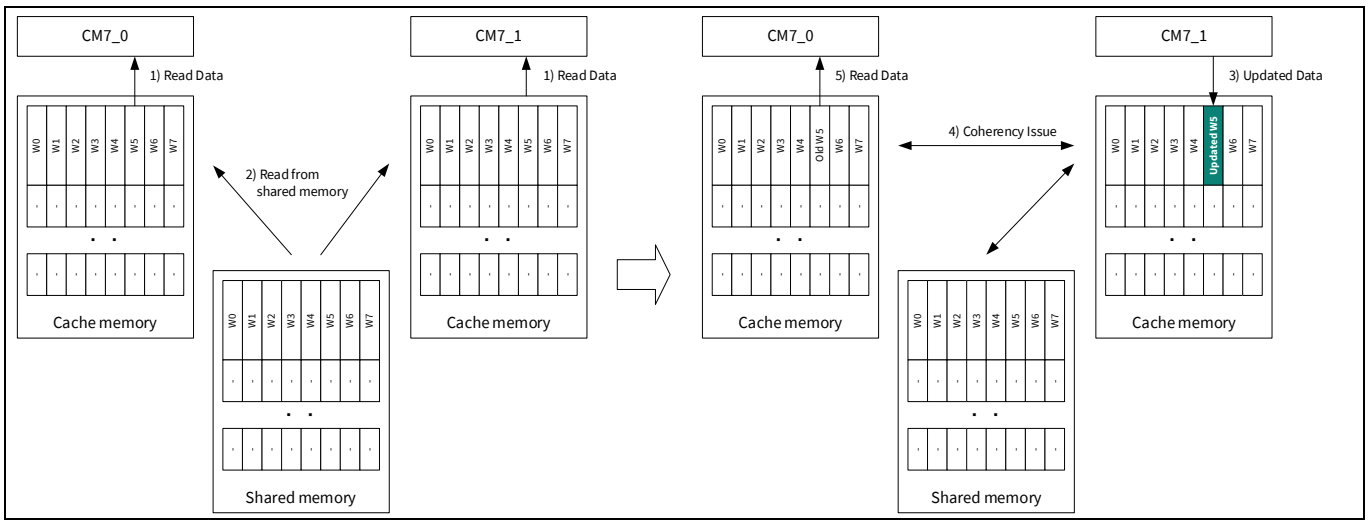

Figure 17 Scenario between CM7 CPUs

1. Each CPU tries to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.
2. As a result of read access, the cache memory refills the data from the shared memory. The cache memory data and the shared memory data are the same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM7_1 updates W5 data in its own cache memory according to cache configuration, but this write access does not update shared memory immediately because of Write-back.
4. W5 (Updated W5) in the CM7_1 cache memory is different from W5 (Old W5) of CM7_0 cache memory and shared memory. That is, this has a cache coherency issue.
5. CM7_0 reads W5 (Old W5) data from its own cache memory. As a result, CM7_0 can cause unintended operations.

Here are some solutions for this scenario between CM7 CPUs:

- Solution 1: Disable cache

  Both CM7 CPUs configure cache disable to the common area. Cache memory does not operate, and each CPU reads/writes to the shared memory directly. Both CPUs have no cache coherency issue. Therefore, there is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs

  CM7_1 performs cache clean after write access to the cache memory. Cache clean writes data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean.

  CM7_0 performs cache invalidate before read access from the cache memory. Cache invalidate invalidates data in the cache memory, and subsequent read access refills the cache memory data with shared memory data. The cache memory and the shared memory are coherent after read access with cache invalidate is performed.
- Solution 3: Change cache configuration

  CM7_1 cache memory is configured to Write-Through. CM7_1 writes data to the cache memory and the shared memory. The write access of CM7_1 has no coherency issue between the cache memory and the shared memory. However, the read access of CM7_0 still has a coherency issue. Therefore, CM7_0 requires a read access with cache invalidate handling.

- Solution 4: Use TCM
  In this case, handling is different depending on the CPU using TCM.
  - Case of Using CM7_1 TCM:
    CM7_1 is not required for handling cache coherency issue regardless of cache configuration. CM7_1 always writes to TCM.
    However, the read access of CM7_0 still has a coherency issue. Therefore, CM7_0 requires a read access with cache invalidate handling.
  - Case of Using CM7_0 TCM:
    The write access of CM7_1 has a coherency issue. Therefore, CM7_1 needs to perform cache clean after write access to cache memory or configure cache memory to Write-Through.
    CM7_0 is not required for handling cache coherency issue regardless of cache configuration. CM7_0 always reads from TCM directly without having to go through cache memory.

These solutions are for CM7_1 write and CM7_0 read. Both CPUs need to be considered for cache coherency issues when read/write access by both CPUs.

5.4.2 Cache coherency issue between CM7 CPU and other masters

This section describes the scenario of cache coherency issue between CM7 CPU and other masters. Other masters except CM7 have no cache memory for shared memory (SRAM). Therefore, these masters operate the shared memory directly.

5.4.2.1 Scenario and solution for CM7 CPU read and other master write

In this scenario, DMA transfers data from the peripheral to the shared memory, and CM7_0 reads the data. That is, DMA writes the data and CM7_0 reads the data. Figure 18 shows the cache coherency issue scenario in this case. The preconditions are as follows:

- CPU and DMA use a part of the shared memory as a common area, and the common area enables a cache.
- CPU cache configuration is Write-back, write, and read allocate.

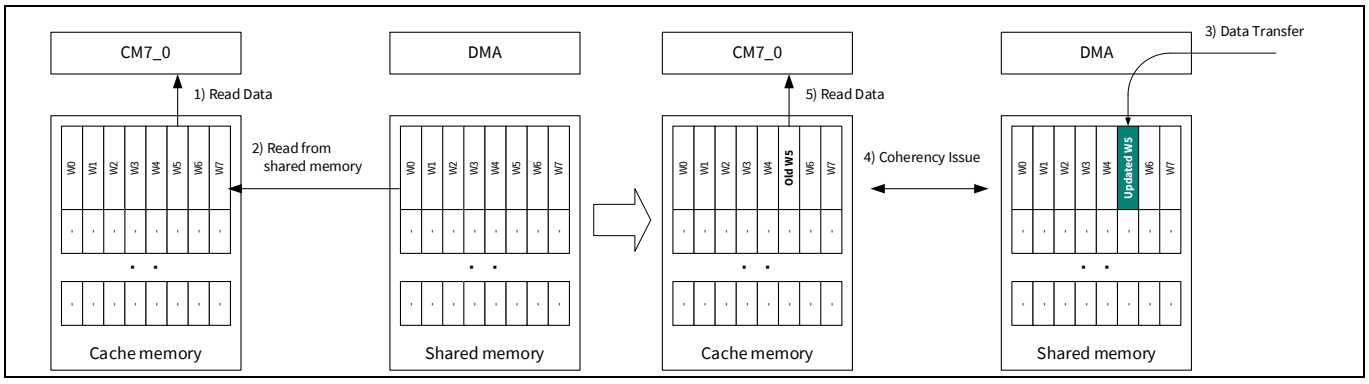

Figure 18 Scenario between CM7 CPU and other master (CM7_0 Reads, DMA Writes)

1. CM7_0 tries to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.
2. As a result of read access, the cache memory refills the data from the shared memory. The cache memory data and the shared memory data are the same at this point. Therefore, they are coherent. Subsequent access result is a cache hit.
3. The DMA writes data to the shared memory by data transfer.

4. W5 (Updated W5) in the shared memory is different from W5 (Old W5) in CM7_0 cache memory. That is, this has a cache coherency issue.
5. CM7_0 reads Old W5 from the cache memory. As a result, CM7_0 can cause unintended operations.

Here are some solutions for the scenario where CM7 CPU reads and other master writes:

- Solution 1: Disable cache

  CM7_0 configures cache disable to the common area. Cache memory does not operate, and CM7_0 reads from the shared memory directly. CM7_0 has no cache coherency issue. Therefore, there is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache invalidate before read access from the cache memory. The cache memory and the shared memory are coherent after read access with cache invalidate is performed.
- Solution 3: Use TCM

  In the case of using CM7_0 TCM, CM7_0 has no cache coherency issue. CM7_0 is not required for handling cache coherency issue regardless of cache configuration. CM7_0 always reads from TCM directly without having to go through the cache memory.

5.4.2.2 Scenario and solution for CM7 CPU write and other master read

In this scenario, CM7_0 writes data, DMA transfers the data from the shared memory to the peripheral. That is, DMA reads the data and CM7_0 writes the data. Figure 19 shows the cache coherency issue scenario in this case. The preconditions are as follows:

- CM7_0 and DMA use a part of the shared memory as a common area, and the common area enables a cache.
- CM7_0 cache configuration is Write-back, write, and read allocate.

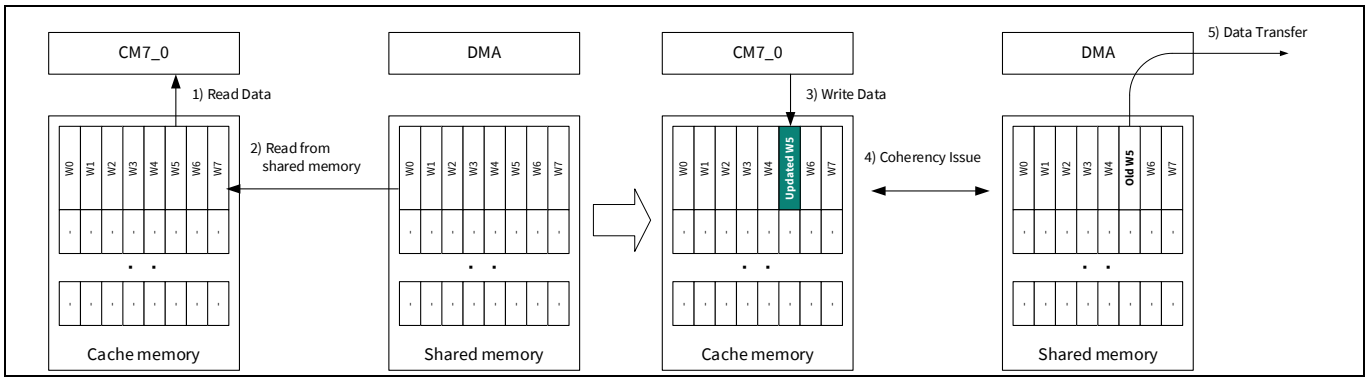

Figure 19 Scenario between CM7 CPU and other master (CM7_0 Writes, DMA Reads)

1. CM7_0 tries to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.
2. As a result of read access, the cache memory refills the data from the shared memory. The cache memory data and the shared memory data are the same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM7_0 updates W5 data in its own cache memory according to cache configuration, but this write access does not update the shared memory immediately because of Write-back.
4. W5 (Updated W5) in the CM7_0 cache memory is different from W5 (Old W5) in the shared memory. That is, this has a cache coherency issue.

5. DMA reads and transfers old W5 in the shared memory. As a result, DMA transfer can cause unintended operations.

Here are some solutions for the scenario where CM7 CPU writes and other master reads:

- Solution 1: Disable cache
  CM7 CPU configures cache disable to the common area. Cache memory does not operate, and the CPU writes to the shared memory directly. CPU has no cache coherency issue. Therefore, there is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs
  `CM7_0` performs cache clean after write access to the cache memory. Cache clean writes data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean.
- Solution 3: Using TCM
  In the case of using `CM7_0` TCM, `CM7_0` has no cache coherency issue. `CM7_0` is not required for handling cache coherency issue regardless of cache configuration. `CM7_0` always writes to TCM directly without through cache memory.

5.4.3 Cache coherency issue for flash memory access

Flash memory has read-only cache memory for the AHB-Lite Bus interface. It helps to improve the read performance of the flash memory from the CM0+ CPU. As mentioned above, the flash memory does not rewrite as frequently as RAM. In the XMC7000 family, flash memory programming is performed using the SROM API. The SROM API invalidates the cache memory in the flash memory after programming. Subsequent read access, the cache memory refills data from the flash memory. There is no need to manage the cache coherency issue.

5.4.4 Cache coherency issue for SMIF access

SMIF has cache memory for the AHB-Lite Bus interface. It helps to improve the read performance of external memories from a master with AHB-Lite interface. Figure 20 shows a block diagram overview of the SMIF bus interface.

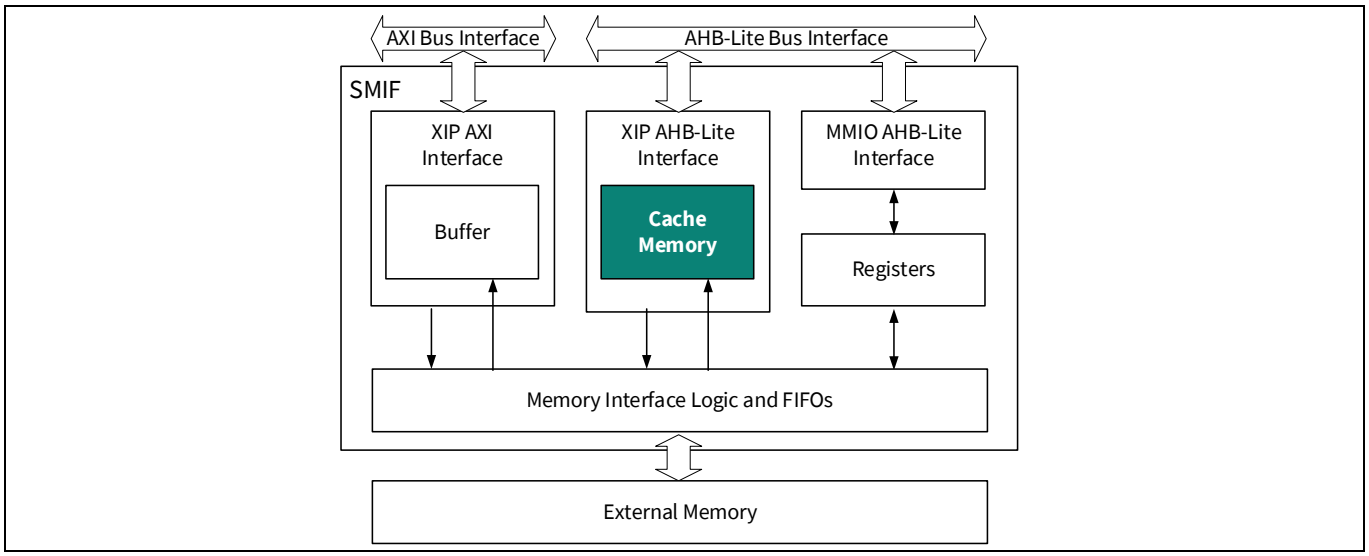

Figure 20 Block diagram of SMIF bus interface

SMIF has three bus interfaces: XIP AXI, XIP AHB-Lite, and MMIO AHI-Lite. The XIP AXI interface is used by CM7 to access external memory in XIP mode. The XIP AHB-Lite interface is used by masters except CM7 to access external memory in XIP mode. The MMIO AHB-Lite interface is used by all masters to access external memory in MMIO mode. See the architecture reference manual for XIP mode, MMIO mode, and each interface detail.

Out of the three interfaces, only the XIP AHB-Lite interface has cache memory with read-only. The cache memory refills the data from the external memory by a read access via the XIP AHB-Lite interface.

This cache memory does not have hardware control of cache consistency by access between interfaces. That is, the cache memory is not affected by writing to the external memory via the XIP AXI interface and MMIO AHB-Lite interface. Therefore, a write access from XIP AXI and MMIO interfaces may cause cache coherency issues. In addition, CM7 with cache memory has a cache coherency issue for write access from XIP AHB-Lite and MMIO interfaces.

5.4.4.1 Scenario and solution for CM7 access

In this scenario, CM7_0 accesses external memory via the XIP AXI interface. Also, CM0+ accesses external memory via the XIP AHB-Lite interface. Two scenarios need to be considered in this case. One scenario where CM0+ writes data to the external memory and CM7_0 reads data from the external memory. Another scenario where CM7_0 writes data to the external memory and CM0 + reads data from the external memory. Figure 21 shows cache coherency issue when CM0+ writes and CM7_0 reads. The preconditions are as follows:

- CM7_0 and CM0+ use a part of the external memory as a common area.
- CM7 cache memory of common area is enabled for CM7_0 XIP mode access, and CM7_0 cache configuration is Write-back, write, and read allocate.
- SMIF cache memory of the common area is enabled for CM0+ XIP mode access.

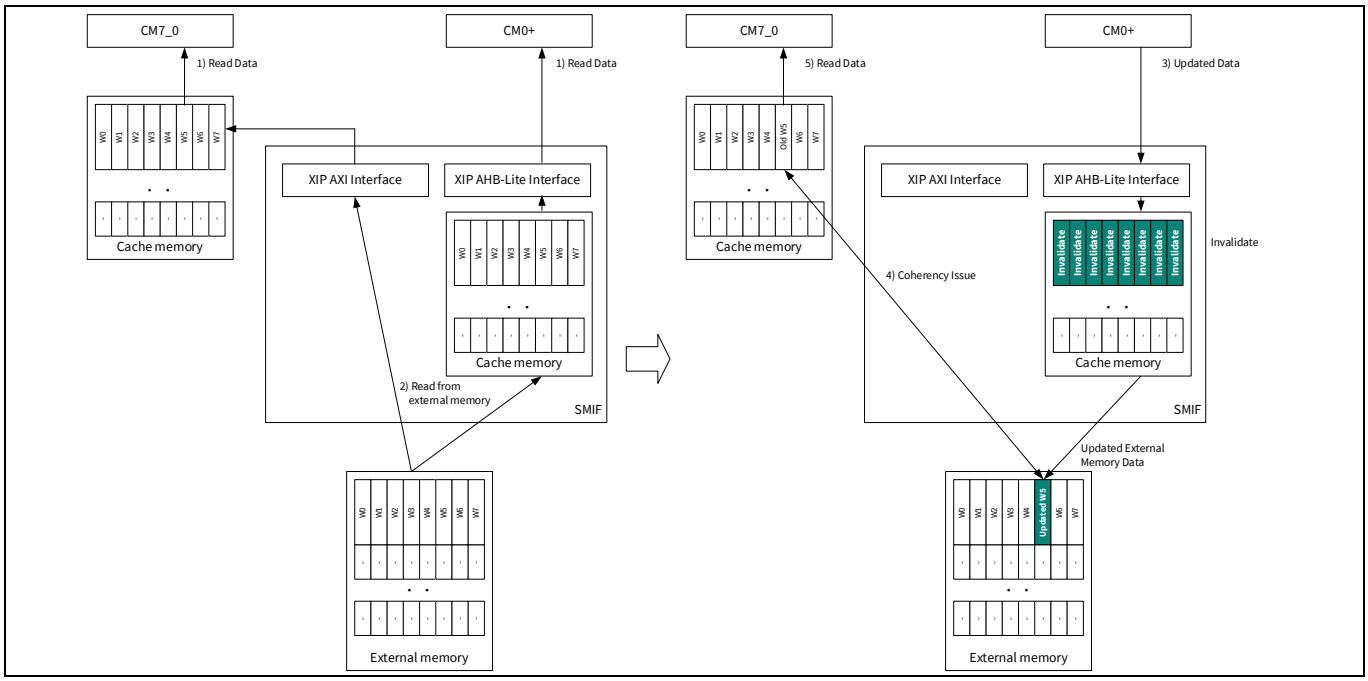

Figure 21 Scenario between CM7 and CM0+ (CM7_0 Reads, CM0+ Writes)

1. CM7_0 and CM0+ try to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.

2. As a result of read access, the cache memories refill the data from the external memory. The cache memories data and the external memory data are the same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM0+ updates W5. As a result of write access, W5 in the external memory is updated, and the associated cache subsector is invalidated. Subsequent access to this data results in a cache miss, and cache memory refills the data from the external memory again.
4. W5 (Old W5) in the CM7_0 cache memory is different from W5 (Updated W5) in the external memory. That is, this has a cache coherency issue.
5. CM7_0 reads old W5 from the cache memory. As a result, CM7_0 can cause unintended operations.

Here are some solutions for the scenario where CM7_0 reads and CM0+ writes:

- Solution 1: Disable cache

  CM7_0 configures cache disable to the common area. Cache memory does not operate, and CM7_0 reads from the external memory directly. CM7_0 has no cache coherency issue. Therefore, handling is not required for the cache coherency issue.
- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache invalidate before read access from cache memory. The cache memory and the shared memory are coherent after performing read access with cache invalidate.

Figure 22 shows the cache coherency issue scenario in CM0+ reads and CM7_0 writes. The preconditions are as follows:

- CM7_0 and CM0+ use a part of the external memory as the common area.
- CM7 cache memory of the common area is enabled for CM7_0 XIP mode access, and CM7_0 cache configuration is Write-back, write, and read allocate.
- SMIF cache memory of the common area is enabled for CM0+ XIP mode access.

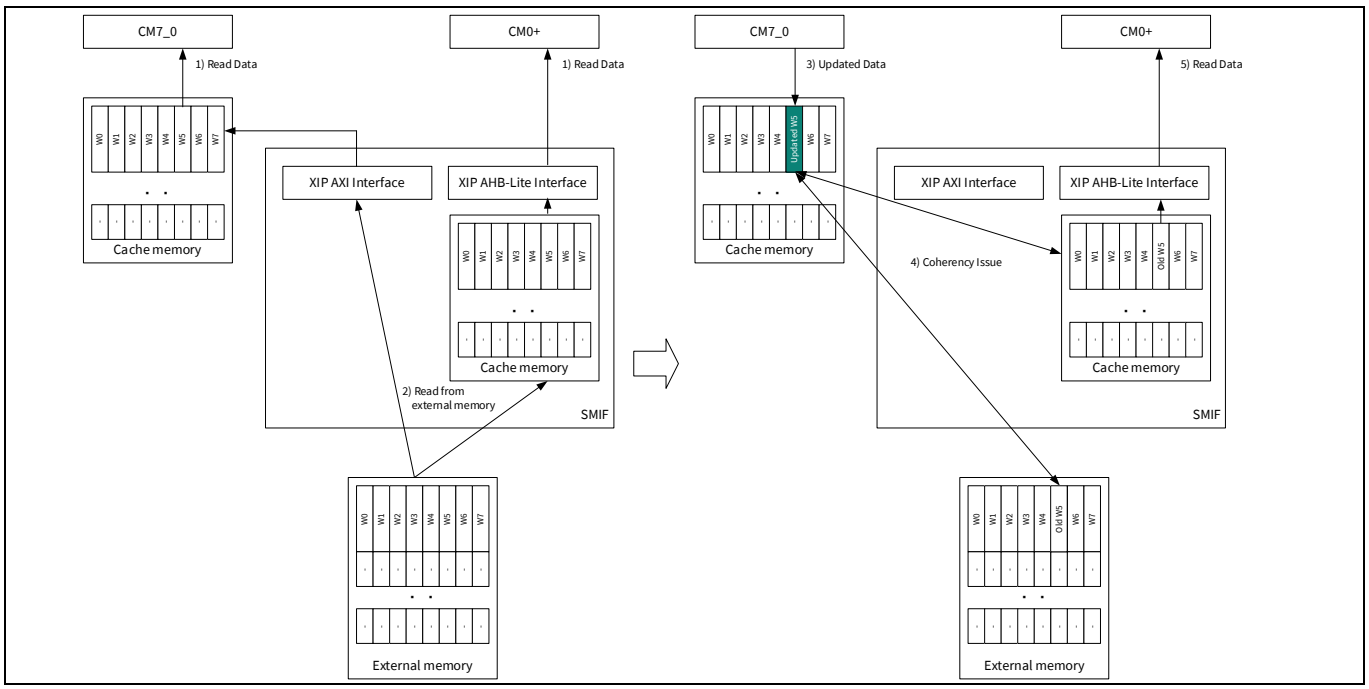

Figure 22 Scenario between CM7 and CM0+ (CM7_0 Writes, CM0+ Reads)

1. CM7_0 and CM0+ try to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.
2. As a result of read access, the cache memories refill the data from the external memory. The cache memories data and the external memory data are the same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM7_0 updates W5 data in its own cache memory according to cache configuration, but this write access does not update external memory immediately because of Write-back.
4. W5 (Updated W5) in the CM7_0 cache memory is different from W5 (Old W5) in the cache memory in SMIF and external memory. That is, this has a cache coherency issue.
5. CM0+ reads old W5 from the cache memory. As a result, CM0+ can cause unintended operations.

Here are some solutions for the scenario where CM7_0 writes and CM0+ reads

- Solution 1: Disable cache

  CM7_0 and CM0+ configure cache disable to the common area. Cache memory does not operate, and both CPUs write to the external memory directly. Both CPUs have no cache coherency issue. There is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache clean after write access to the cache memory. Cache clean writes data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean. SMIF cache memory needs to be invalidated with CM7_0 write access. Therefore, the application software needs to monitor write access from the XIP AXI interface and MMIO AHB-Lite interface.

5.4.5 Cache coherency issue for using SROM APIs

This section describes the scenario of cache coherency issue when using SROM APIs. This scenario is very similar to the cache coherency scenario between the CM7 CPUs and other masters described in the Cache coherency issue between CM7 CPU and other masters.

SROM APIs perform various supervisory tasks via CM0+ such as flash programming and changing system configuration. SROM APIs use IPC, and in many cases, use shared memory to pass parameters and execution results.

5.4.5.1 Scenario and solution when using SROM API (CM0+ API parameter Read)

In this scenario, CM7 uses the SROM API to read specific memory data. The CM7 writes the SROM API parameters to the shared memory, and CM0+ reads it and executes the SROM API. Then, CM0+ writes the execution result and memory data to the shared memory, and CM7 CPU reads the data. That is, in this scenario, CM7 writes, CM0+ reads, and CM7 reads, CM0+ writes occur. Two cache coherency issues occur when writing and reading CM7. Figure 23 shows the cache coherency issue scenario in the CM0+ API parameter read. The preconditions are as follows:

- CM7 and CM0+ use a part of the shared memory as a common area, and the common area enables a cache.
- CM7 cache configuration is Write-back, write, and read allocate.

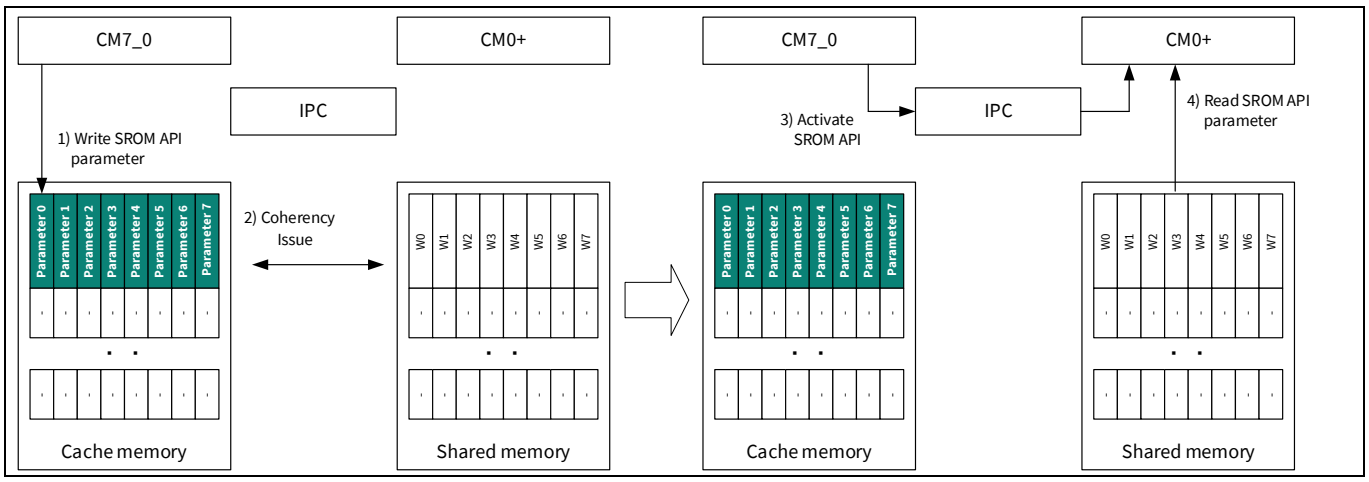

Figure 23 Scenario CM0+ SROM API parameter Read

1. CM7_0 writes SROM API parameters in its own cache memory according to cache configuration, but this write access does not update the shared memory immediately because of Write-back.
2. SROM API parameters in the CM7_0 cache memory are different from the shared memory. That is, this has a cache coherency issue.
3. CM7_0 notifies SROM API activation to CM0+ via IPC.
4. CM0+ reads SROM API parameters from the shared memory when notified by IPC. However, CM0+ reads non-updated SROM API parameters. As a result, CM0+ cannot perform correctly.

Here are some solutions for the scenario:

- Solution 1: Disable cache

  CM7 CPU configures cache disable to the common area. Cache memory does not operate, and the CPU writes to the shared memory directly. CPU has no cache coherency issue. There is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache clean after write access to the cache memory. Cache clean writes data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean.

  After that, CM7_0 notifies SROM API activation to CM0+ via IPC.

5.4.5.2 Scenario and solution when used SROM API (CM7 execution result read)

Figure 24 shows the cache coherency issue scenario in CM7 SROM API execution result read.

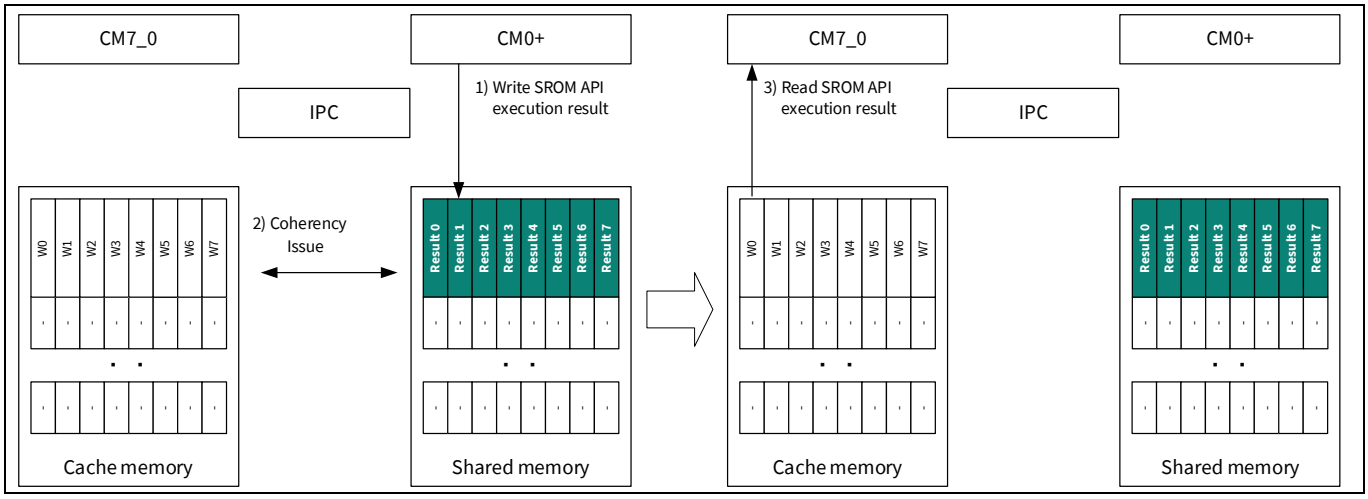

Figure 24 Scenario CM7 SROM API parameter Read

The preconditions are as follows:

1. After executing the SROM API, CM0+ writes the execution result to the shared memory.
2. The execution result in the shared memory is different from `CM7_0` cache memory. That is, this has a cache coherency issue.
3. `CM7_0` reads the execution result from the cache memory. However, `CM7_0` reads a non-updated execution result. As a result, `CM7_0` cannot perform correctly.

Here are some solutions for the scenario:

- Solution 1: Disable cache
  CM7 CPU configures cache disable to the common area. Cache memory does not operate, and the CPU writes to the shared memory directly. CPU has no cache coherency issue. There is no need to manage the cache coherency issue.
- Solution 2: Use cache maintenance APIs
  `CM7_0` performs cache invalidate before read access from the cache memory. The cache memory and the shared memory are coherent after performing read access with cache invalidate.

5.5 Additional cache issue scenarios

This section describes additional cache issues for different scenarios and provides solutions.

5.5.1 Cache issue for protection attribute switching

5.5.1.1 Scenario and solution for protection attribute switching

The Protection Context (PC) and Secure attributes, which are the access protection attributes of SMPU and PPU, are added outside the CPU. Therefore, access to cache memory does not detect protection violations of these access attributes. Figure 25 shows the cache issue scenario in this case. The preconditions are as follows:

- CPU uses a part of the shared memory, and this area enables a cache.

Considerations for cache coherency issue

- Shared memory with cache enabled contains an area accessible only on PC = 4 and another area accessible only on PC = 5.
- CPU cache configuration is Write-back, write, and read allocate.

See architecture reference manual for PC and Secure attributes.

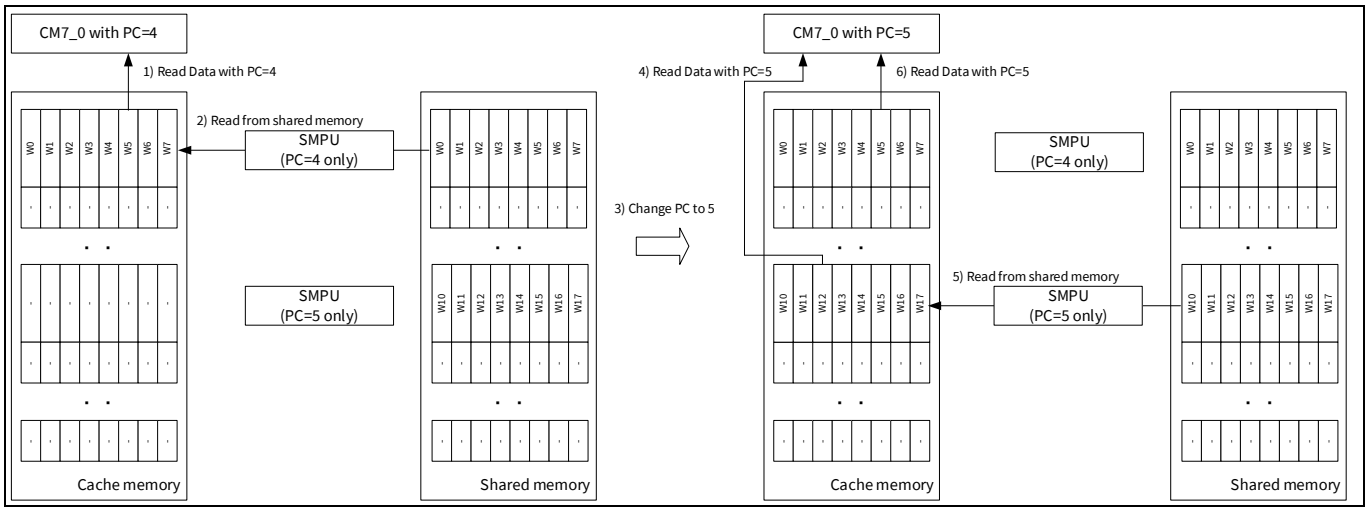

Figure 25 Scenario switching PC

1. CM7_0 is operating on PC=4 and tries to read data from the cache memory. However, the cache memory does not have data, therefore, it is a cache miss.
2. As a result of read access, the cache memory refills data from the shared memory. The cache memory data and the shared memory data are the same at this point. Subsequent access results in a cache hit.
3. CM7_0 changes protection context from PC=4 to PC=5.
4. CM7_0 tries to read data from the cache memory. However, the cache memory does not have data, thus, it is a cache miss.
5. As a result of read access, the cache memory refills data from the shared memory. The cache memory data and the shared memory data are the same at this point. Subsequent access results in a cache hit.
6. Here, the data that is allowed by PC=4 in the cache memory can be accessed by PC=5, because this access does not go through the SMPU.

Here are some solutions for the scenario:

- Solution 1: Disable cache

  CM7 CPU configures cache disable in the common area. Cache memory does not operate, and the CPU reads and writes to the shared memory directly. CPU has no caching issue.
- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache clean and invalidates before switching PC. Cache clean writes data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean. Cache invalidate invalidates data in the cache memory, and subsequent read access refills the cache memory data with shared memory data via SMPU. Therefore, while accessing the area of PC = 4 with PC=5, an SMPU protection violation will occur.

References

[1] Device datasheets:

- XMC7100 series 32-bit Arm® Cortex®-M7 microcontroller datasheet
- XMC7200 series 32-bit Arm® Cortex®-M7 microcontroller datasheet

[2] Device reference manuals:

- XMC7000 MCU family architecture technical reference manual
- 002-33817: XMC7100 MCU registers reference manual
- 002-33812: XMC7200 MCU registers reference manual

[3] Application notes

- AN234226 - XMC7000 MCU: Usage of interrupts
- AN234118 - GPIO usage setup in XMC7000 family

Contact Technical support to obtain XMC7000 family references documents.

Glossary

AHB
advanced high-performance bus

AXI
Advanced eXtensible Interface

BOD
brown-out detection

CAN FD
Controller Area Network with Flexible Data Rate. See the CAN FD controller chapter of the XMC7000 family architecture reference manual for details

CPU
central processing unit

D-cache
Data cache memory

DTCM
data tightly-coupled memory

eSHE
Enhanced Secure Hardware Extension

I-cache
Instruction cache memory

IPC
inter-processor communication

ITCM
instruction tightly-coupled memory

Glossary

LRU
Least Recently Used. An algorithm that determines the allocation of data handled by cache memory to resources.

M-DMA
Memory DMA. See the Direct Memory Access chapter of the XMC7000 family architecture reference manual for details.

P-DMA
Peripheral DMA. See the Direct Memory Access chapter of the XMC7000 family architecture reference manual for details.

PLL
phase-locked loop

SMIF
Serial Memory Interface

SROM API
SROM Application Programming Interface. It performs various supervisory tasks such as flash programming and changing system configuration. See the Nonvolatile Memory Programming chapter of the XMC7000 family architecture reference manual for details.

XIP
eXecute-In-Place

Revision history

Document revision	Date	Description of changes
**	2021-11-23	Initial release.
*A	2022-05-06	Updated Table 12.
*B	2023-09-25	Updated Introduction Removed Other references Updated References

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-09-25

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-34254 Rev. *B

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.