Multi core handling in TRAVEO™ T2G family

Associated part family

TRAVEO™ T2G family

About this document

Scope and purpose

This application note provides useful information for multi-core (multi-CPU) handling in TRAVEO™ T2G MCUs. A TRAVEO™ T2G 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. In addition, the document provides an overview of cache coherency issues that occur between CPUs with cache and other masters, and suggests methods to avoid the issue in different scenarios.

Intended audience

This document is intended for anyone using TRAVEO™ T2G family.

Table of contents

Table of contents

1 Introduction

TRAVEO™ T2G family MCUs include Arm® Cortex®-M CPUs with SRAM, and flash memories, hardware-based Cryptography (eSHE/HSM support), communication peripherals (CAN FD and LIN), digital peripherals (Timer, Counter, and Pulse Width Modulator (PWM)), and analog peripherals (SAR ADC) in a single chip.

The CPU subsystem (CPUSS) of the TRAVEO™ T2G MCU consists of multiple bus masters, two or three CPUs, two types of DMA controllers (P-DMA and M-DMA), and a hardware cryptography block (Crypto). The CPUSS also has an Inter-Processor communication (IPC) block that can be used for exclusive control, synchronization, and data passing between CPUs.

The CYT2, CYT3, and CYT4 series have cache memory on some peripherals. In addition, the CYT3 and CYT4 series have cache memory on the CPU. A cache memory is a low-latency memory, and helps to improve the performance. However, the cache memory can cause coherency issues between memories. Therefore, use of the cache memory requires careful handling.

In addition, this application note describes example code with the Sample Driver Library (SDL). The code snippets in this application note are part of the SDL. See **Other references** for the SDL.

The SDL consists of a configuration, and a driver associated with the peripheral driver. The configuration helps to set the chosen parameter values for the desired operation, whereas the peripheral driver configures each register based on the parameter values in the configuration. You can adapt the configuration according to your system.

Figure 1 shows block diagram of CPUSS for the CYT2B series, and **Figure 2** shows block diagram for the CYT4B series.

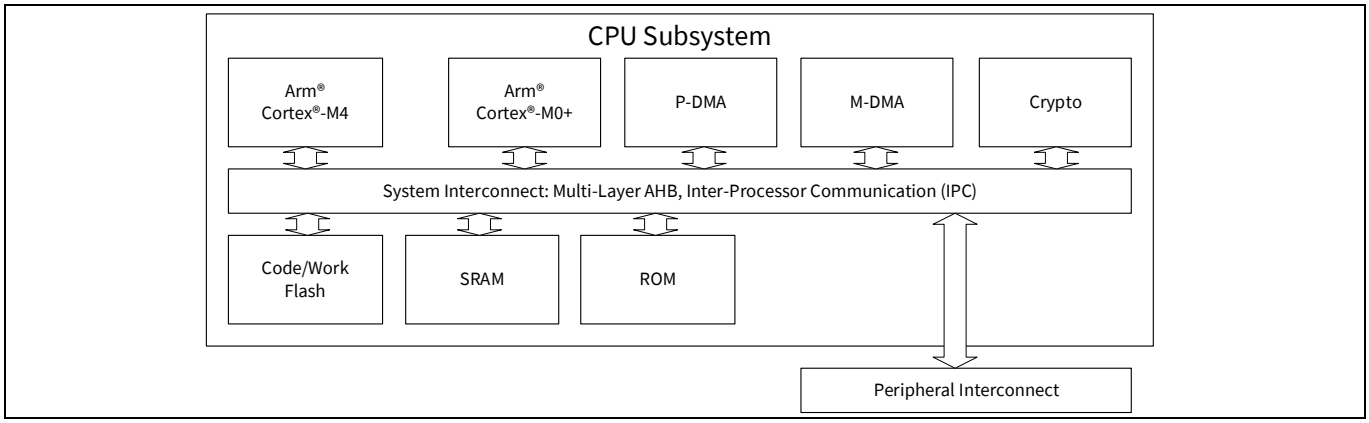

Figure 1 CPUSS for CYT2B series

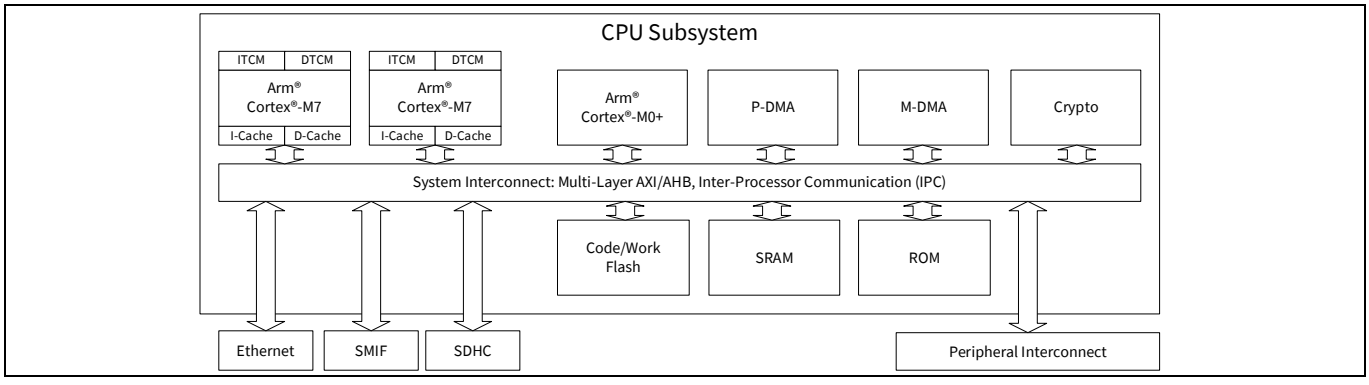

Figure 2 CPUSS for CYT4B series

Introduction

The CYT2 series MCUs have an Arm® Cortex®-M4F-based CPU (CM4) and a Cortex®-M0+-based CPU (CM0+). The CYT4 series have two Arm® Cortex®-M7-based CPUs (CM7) and one CM0+. The CYT3 series has one Arm® Cortex®-M7-based CPU (CM7) and one CM0+. CM7 CPUs have Instruction/Data cache (I-cache/D-cache) and Instruction/Data Tightly-Coupled Memories (ITCM/DTCM). The CPUSS of the CYT2, CYT3, and CYT4 series MCUs have bus masters for P-DMA, M-DMA, and Crypto.

In addition, CYT4B and CYT3B series MCUs have SDHC and Ethernet as the bus muster. CYT4D series MCU has Ethernet, JPEG decoder, VIDEO subsystem, and AXI M-DMA as the bus master. CYT3D series MCU has Ethernet and VIDEO subsystem as the bus master.

See the Arm® documentation sets for **CM7**, **CM4**, and **CM0+**, and the TRAVEO™ T2G **architecture technical reference Manual (TRM)** for more information.

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

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

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

2 Considerations for CPU startup

Generally, when a user application software is started, the CPU uses the PLL to switch to high-speed operation. However, sudden changes in 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 CPU clock in stages to make sure it does not go below the voltage defined by the BOD. See the specific **device datasheet** for BOD detection voltage. This is especially important for the CYT4 series, which have two CM7s.

Here is an example of stepping up the CPU clock frequency in stages for the CYT4B 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.

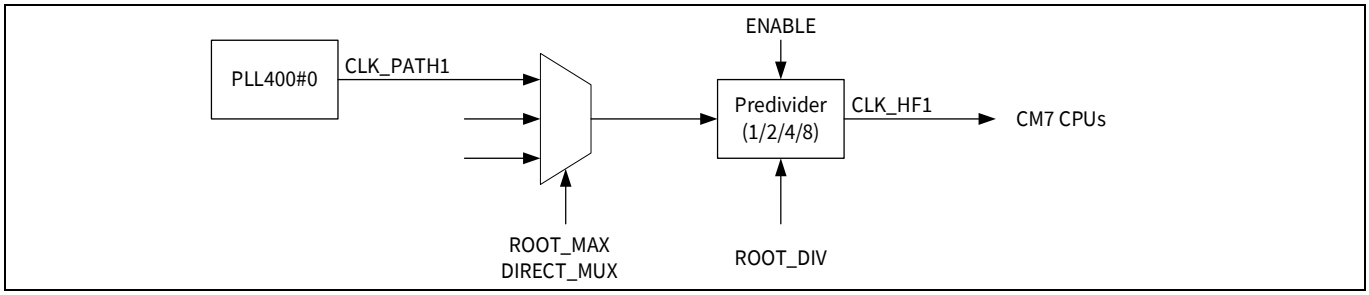

Figure 3 CM7 CPU's clock connection

2.1 Example of step-up CPU clock frequency

This section describes how to step up the CPU clock frequency in stages. This is an example for the CYT4B series.

2.1.1 Configuration

Table 1 and **Table 2** list the parameters and functions in the SDL for stepping up the CPU clock frequency.

Table 1 Clock configuration parameters

Parameters	Description	Value
`WAIT_CYCLE_WHILE_DISTRIBUTING_CLOCK`	Rising time adjustment	50 ul
`SRSS_NUM_HFROOT`	Number of HF clocks	8 ul
`clkHfSetting.targetDiv`	Set target CLK_HF division	-
`clkHfSetting.source`	Selection of target CLK_HF root.	-

Table 2 Clock configuration functions

Functions	Description	Remarks
`Cy_SysTick_DelayCoreCycle(wait)`	Delay input core cycle	-

Considerations for CPU startup

The following code shows an example of stepping up the clock frequency:

- ***SRSS***->un***CLK_ROOT_SELECT[x]***.stcField.u4***ROOT_MUX*** is the ROOT_MUX field in the SRSS_CLK_ROOT_SELECT[x] register mentioned in the **registers TRM**. Other registers are also described in the same manner.“***x***” signifies the register suffix number.

See *cyip_srss_v3.h* under *hdr/rev_x/ip* for more information on the union and structure representation.

Code Listing 1 Example of stepping up the CPU clock frequency

```
#define WAIT_CYCLE_WHILE_DISTRIBUTING_CLOCK (50ul)
#define SRSS_NUM_HFROOT                 8ul

typedef enum
{
    CY_SYSCLK_HFCLK_NO_DIVIDE   = 0u,    /**< don't divide hf_clk. */
    CY_SYSCLK_HFCLK_DIVIDE_BY_2 = 1u,    /**< divide hf_clk by 2 */
    CY_SYSCLK_HFCLK_DIVIDE_BY_4 = 2u,    /**< divide hf_clk by 4 */
    CY_SYSCLK_HFCLK_DIVIDE_BY_8 = 3u     /**< divide hf_clk by 8 */
} cy_en_hf_clk_dividers_t;

typedef enum
{
    CY_SYSCLK_HFCLK_IN_CLKPATH0  = 0u,    /**< hf_clk input is Clock Path 0 */
    CY_SYSCLK_HFCLK_IN_CLKPATH1  = 1u,    /**< hf_clk input is Clock Path 1 */
    CY_SYSCLK_HFCLK_IN_CLKPATH2  = 2u,    /**< hf_clk input is Clock Path 2 */
    CY_SYSCLK_HFCLK_IN_CLKPATH3  = 3u,    /**< hf_clk input is Clock Path 3 */
    CY_SYSCLK_HFCLK_IN_CLKPATH4  = 4u,    /**< hf_clk input is Clock Path 4 */
    CY_SYSCLK_HFCLK_IN_CLKPATH5  = 5u,    /**< hf_clk input is Clock Path 5 */
:
    CY_SYSCLK_HFCLK_IN_CLKIMO    = 0xFFu, /**< hf_clk input is directly connected to IMO */
} cy_en_hf_clk_sources_t;

void SystemInit (void)
{
    /***     PLL setting and enabling        ***/
:
    /***  Setting for each clk_hf        ***/
    {
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH3}, // setting for clk_hf0
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH1}, // setting for clk_hf1
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH4}, // setting for clk_hf2
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH1}, // setting for clk_hf3
        { .targetDiv = CY_SYSCLK_HFCLK_DIVIDE_BY_4, .source = CY_SYSCLK_HFCLK_IN_CLKPATH3}, // setting for clk_hf4
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH2}, // setting for clk_hf5
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH3}, // setting for clk_hf6
        { .targetDiv = CY_SYSCLK_HFCLK_NO_DIVIDE,   .source = CY_SYSCLK_HFCLK_IN_CLKPATH5}, // setting for clk_hf7
    };

    for(int8_t i_clkHfNo = 0ul; i_clkHfNo < SRSS_NUM_HFROOT; i_clkHfNo++)
    {
:
        SRSS->unCLK_ROOT_SELECT[i_clkHfNo].stcField.u4ROOT_MUX   = clkHfSetting[i_clkHfNo].source;
        SRSS->unCLK_ROOT_SELECT[i_clkHfNo].stcField.u2ROOT_DIV   = CY_SYSCLK_HFCLK_DIVIDE_BY_8;
        SRSS->unCLK_ROOT_SELECT[i_clkHfNo].stcField.u1DIRECT_MUX = 1u; /* Select ROOT_MUX */
        SRSS->unCLK_ROOT_SELECT[i_clkHfNo].stcField.u1ENABLE     = 1u; /* 1 = enable */

        /* Gradually decrease the current root clock divider until the target divider is reached */
        for(int8_t i_divRegValue = 2; i_divRegValue >= clkHfSetting[i_clkHfNo].targetDiv; i_divRegValue--)
        {
            Cy_SysTick_DelayCoreCycle(WAIT_CYCLE_WHILE_DISTRIBUTING_CLOCK);
            SRSS->unCLK_ROOT_SELECT[i_clkHfNo].stcField.u2ROOT_DIV = i_divRegValue;
        }
        Cy_SysTick_DelayCoreCycle(WAIT_CYCLE_WHILE_DISTRIBUTING_CLOCK);
    }
:
}
```


In this example, the CPU clock frequency is stepped up using the CLK_HF clock divider in steps from 8 division to target division using the CLK_HF clock divider. The call to the `Cy_SysTick_DelayCoreCycle()` API provides the required delay between each successive step. See the “Clocking System” chapter in the **architecture TRM** and **application note** for PLL and CLK_HF configuration.

3 Considerations for resource access

As mentioned above, all memory and peripherals are accessed through standard Arm® multi-layer bus architecture 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 same time.

Figure 4 shows resource connection for CYT4BF series. IPC (blue box) access of CM7_0 and Crypto (blue box) access of CM0+ can be performed at same time (indicated by blue arrows). However, CM7_0 and CM7_1 cannot access TTCAN FD (grey box) at the same time (indicated by red arrows).

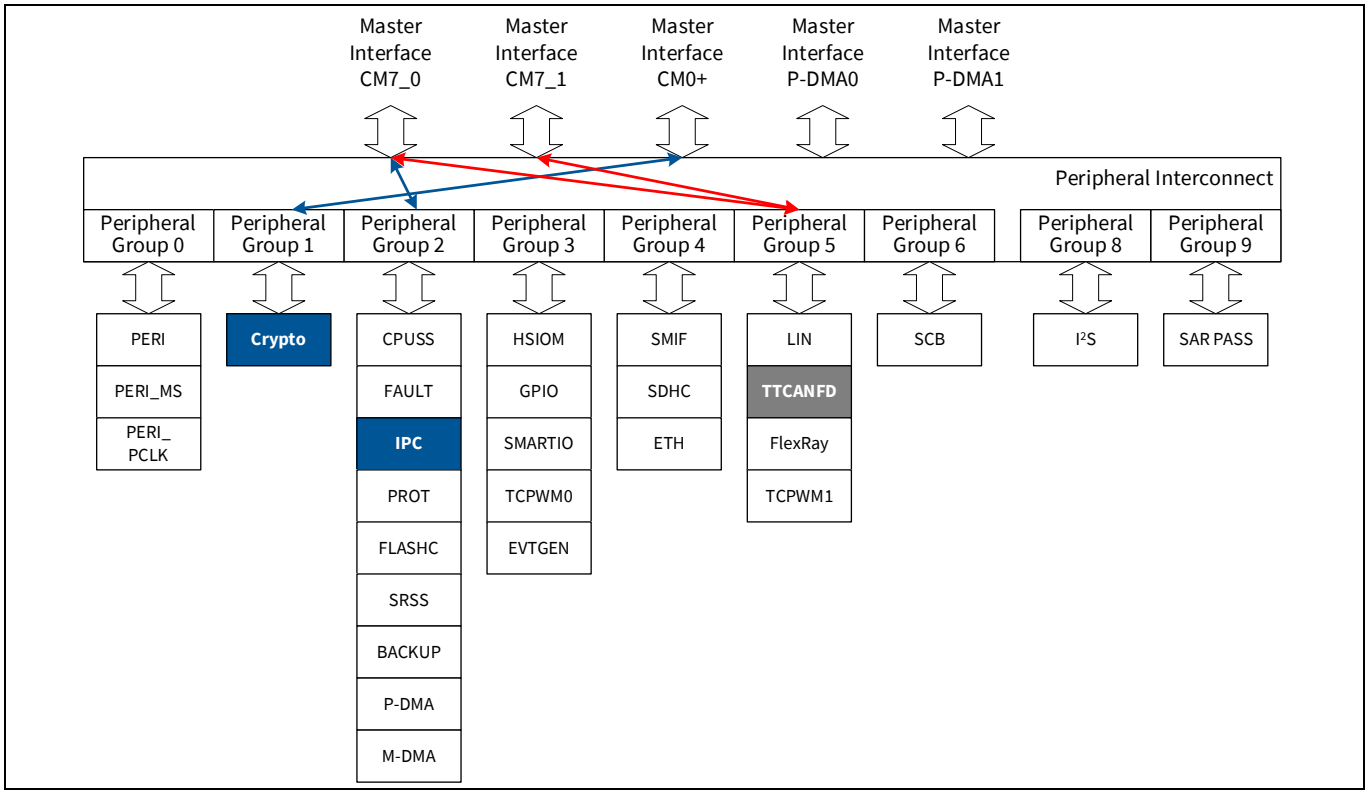

Figure 4 Resource connection (CYT4BF series)

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 CM7_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 same time (blue arrow). CM7_0 and CM7_1 cannot access same SRAM at the same time even if the addresses are different within the SRAM (red arrow).

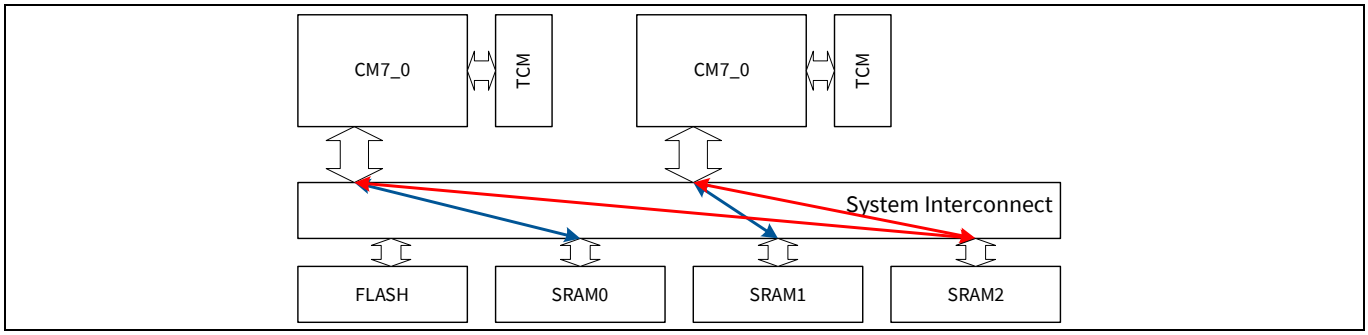

Figure 5 Memory connection (CYT4BF series)

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 Consideration for write buffer in CPU and bus infrastructure

TRAVEO™ T2G has write buffers in CPU and bus infrastructure. A write buffer has increases the execution performance, because subsequent instructions can be executed without waiting for the write complition. However, this means that the application cannot assume the impact of a register write in a timely way. Therefore, it might require extra steps to ensure the correctness of the program operation.

For the write buffer in the CPU, you can use memory barrier instructions to support the sequencing between memory accesses, or between memory accesses and other operations.

For the write buffer in bus infrastructure, if the MPU region is set to the device type for the concerned register, the CPU will continue the execution and not wait for the register write. The writing to register is ensured, but the timing may differ. For example, the timing depends on the following conditions:

- Write buffer in the CPU
- Clock ratio between the CPU and the register clock
- Buffers in the bus infrastructure
- Other masters accessing and occupying the bus system (arbitration)

A readback of a register is necessary when you want to ensure that the changes in the registers are effective before proceeding with the code execution. A readback operation reads the data from the concerned register after writing register the write buffer data.

For example, when leaving an interrupt handler, clear the interrupt flag in the peripheral register and execute a readback to ensure that the flag is cleared in the peripheral register before clearing the flag in the NVIC. As a result, it can prevent an immediate resetting of the flag in the NVIC.

See the Arm® Cortex®-M Programming Guide to Memory Barrier Instructions in **Related documents[5]** for more details.

5 Communicating between CPUs

Architectures with multiple CPUs often require exclusive control, synchronization, and data passing between CPUs, TRAVEO™ T2G can use IPC for such control. IPC has support for mutual exclusion (mutex), message passing, and event/release notification.

The IPC hardware contains register structures for 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.

The IPC channel structure ACQUIRE register provides lock feature and IPC_STRUCTx_LOCK_STATUS indicates lock status. The IPC_STRUCTx_NOTIFY register generates notification event, the IPC_STRUCTx_RELEASE register releases IPC channel structure and generates release event. IPC_STRUCTx_DATA0/1 register can pass a message up to 64 bits.

TRAVEO™ T2G devices have a supervisory ROM that contains the boot ROM code and SROM APIs. The SROM APIs are designed to be used with Arm® Cortex®-M0+ (CM0+). The specific IPC channel structures and interrupt structures are used to activate the APIs. See the “Nonvolatile memory programming” chapter in the **architecture TRM** for more information.

Note: LDREX/STREX and other exclusive access instructions are not supported by the TRAVEO™ T2G devices. Therefore, implementation inter-process communication in system of multiple CPUs and shared memory needs to use IPC.

5.1 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 completed. 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 IPC_STRUCTx_NOTIFY register uses a notification event interrupt. **Table 3** lists the registers associated with the notification event. IPC_STRUCTx_NOTIFY register is used to generate an IPC notify event and IPC_STRUCTx_RELEASE is used to generate an IPC release event.

Table 3 Notify event registers

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

Structure	Register name	Bit name	Description
			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 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 channel number of IPC channel structures. NOTIFY [31:16] corresponds to channel numbers 15 to 0 of IPC channel structure. See the **registers TRM** 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.

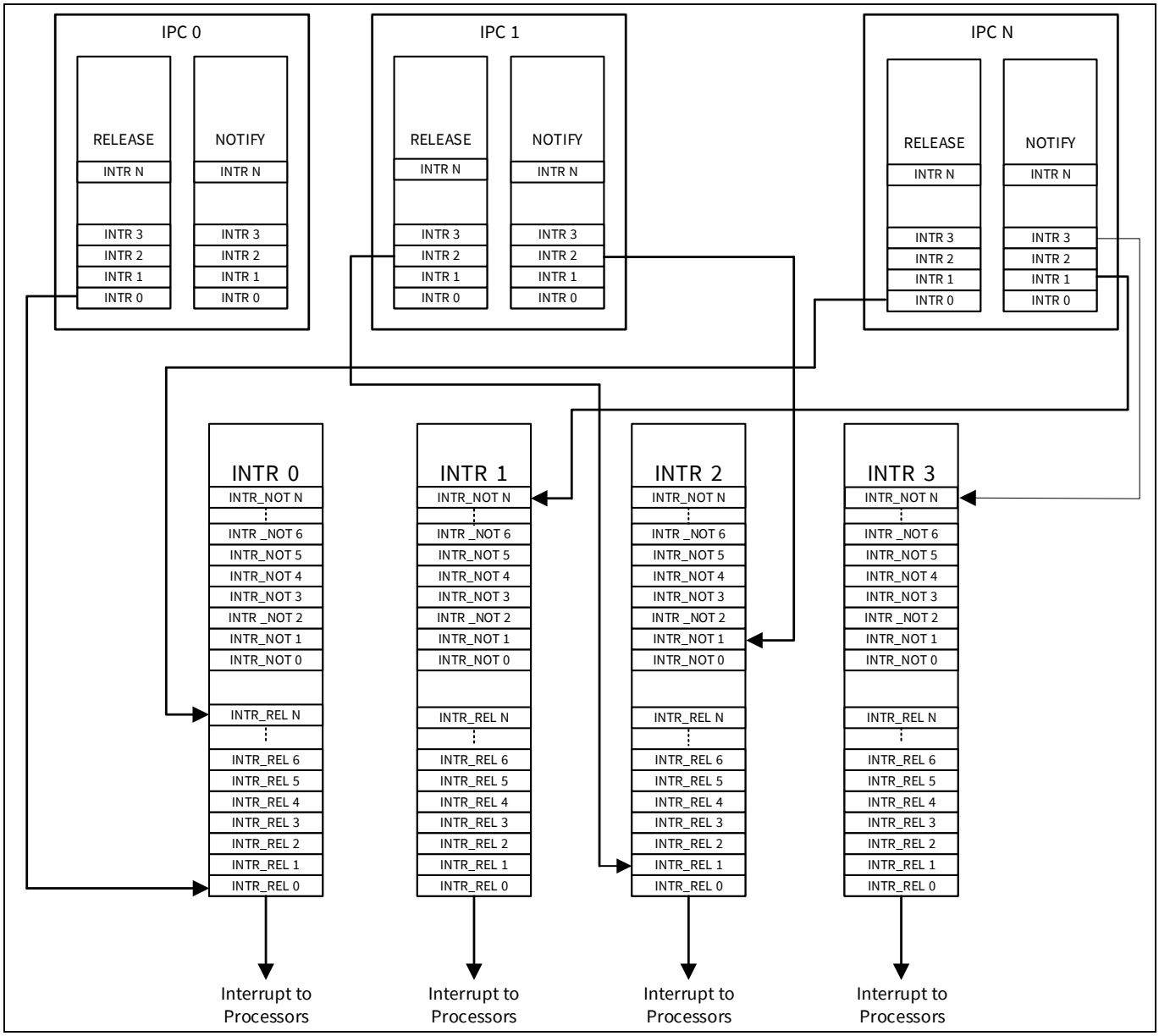

Figure 6 IPC channel structures and interrupt structures

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.

5.1.1 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 in the following example.

5.1.2 Use case

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

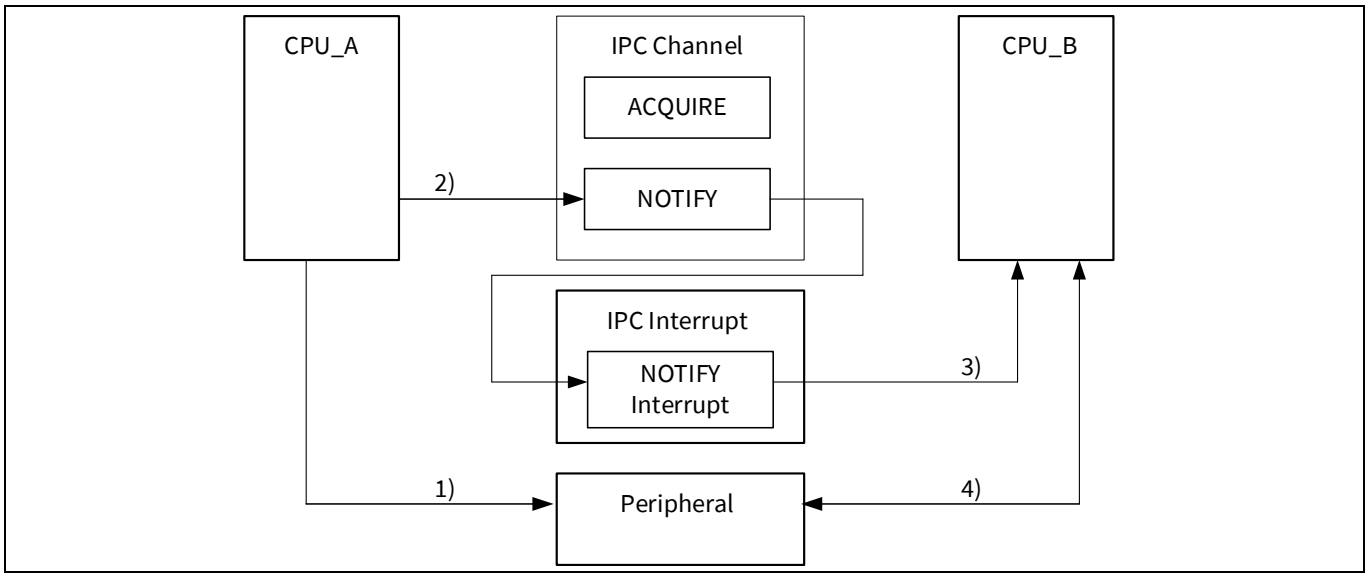

Figure 7 CPU synchronization using IPC

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.

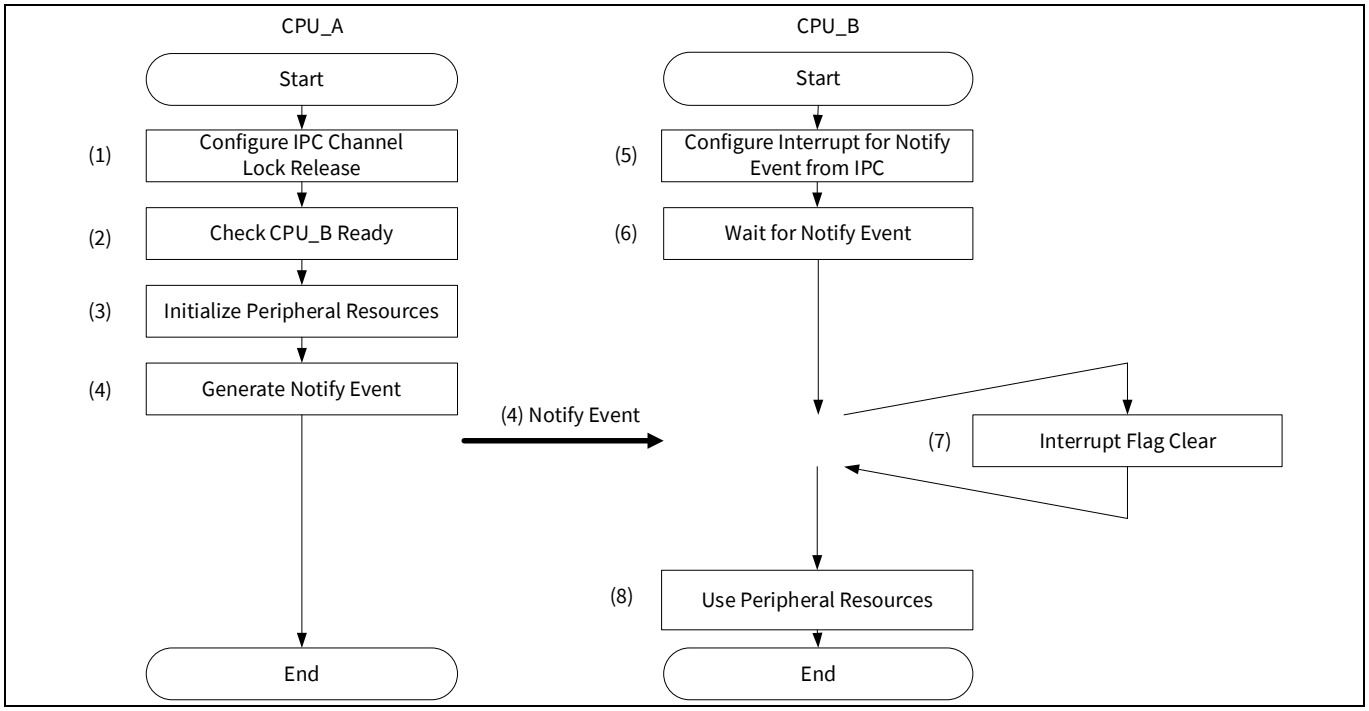

Figure 8 CPU synchronization operation

Communicating between CPUs

The following is the structure of the sample code.

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

See the “Interrupts” chapter in the **architecture TRM** and **AN219842 - How to use interrupt in TRAVEO™ T2G** for interrupt configuration details.

5.1.3 Configuration

Table 4 and **Table 5** list the parameters and functions in SDL for CPU synchronization using IPC. This is example for CYT2B series. Here, it assumes that CPU_A is CM4 and CPU_B is CM0+.

Table 4 Configuration parameters for CPU synchronization

Parameter	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Defines using IPC interrupt structure number for the notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Defines using the 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 whether peripheral initialization is complete	0: Completed 1: Not complete (Default)

Table 5 Configuration functions for CPU synchronization

Function	Description	Remarks
`Cy_IPC_Drv_GetIpcBaseAddress(ipcIndex)`	Gets the base address of the IPC channel structure. ipcIndex: IPC channel structure number	-
`Cy_IPC_Drv_LockRelease(base, releaseEventIntr)`	Releases the IPC channel lock. base: Base address of the IPC channel to operate releaseEventIntr: Specifies the release event	-
`Cy_IPC_Drv_GetIntrBaseAddr(ipcIntrIndex)`	Gets the base address of the IPC interrupt structure. ipcIndex: IPC interrupt structure number	-
`Cy_IPC_Drv_GetInterruptMask(base)`	Gets the value of INTR_MASK register. base: Base address of the IPC interrupt structure to operate	-

Communicating between CPUs

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

The following code shows an example of CPU synchronization using IPC.

Communicating between CPUs

- Base signifies the pointer to the IPC register base address.
- ***base***->unNOTIFY.u32Register is the IPC_STRUCTx_NOTIFY register mentioned in the **registers TRM**. Other registers are also described in the same manner. "***x***" signifies the register suffix number.
- To improve the register setting performance, the SDL writes a complete 32-bit data to the register. Each bit field is generated and written to the register as the final 32-bit data.

```
un_IPC_INTR_STRUCT_INTR_t reg = { 0 };
reg.stcField.u16NOTIFY  = ipcNotifyMask;
reg.stcField.u16RELEASE = ipcReleaseMask;
base->unINTR.u32Register = reg.u32Register;
```

See *cyip_ipc.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

Code Listing 2 Example of CPU synchronization for CM4

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */

#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)

#define _FLD2VAL(field, value)    (((uint32_t)(value) & field ## _Msk) >> field ## _Pos)

int main(void)
{
:
    /* At first force release the lock state. */
    volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
    (void)Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);


    /* Wait until the CM0+ IPC server is started */
    /* Note:
     * After the CM0+ IPC server is started, the corresponding number of the INTR_MASK is set.
     * So in this case CM4 can recognize whether the server has started or not by the INTR_MASK status.
     */
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMask;
    uint32_t notifyMask;
    do

    {
        intrMask = Cy_IPC_Drv_GetInterruptMask(ipcIntrStrBase);
        notifyMask = Cy_IPC_Drv_ExtractAcquireMask(intrMask);
    } while((notifyMask & (1ul << IPC_CHANNEL_NUMBER)) == 0);


    /* Initialize peripherals */

    /* Generate a notify interrupt */
    Cy_IPC_Drv_AcquireNotify(ipcBase, (1ul << IPC_NOTIFY_INT_NUMBER));

    for(;;)
    {
    }
}
```

Define number of IPC interrupt for notify event.

Define IPC channel number

Define Lock release data without release event

Get base address of IPC channel structure. See **Code Listing 3**.

IPC channel initialization (Release a lock). See **Code Listing 4**.

Get base address of IPC interrupt structure. See **Code Listing 5**.

Get value of IPC.INTR_MASK. See **Code Listing 6**.

Get value of notify mask. See **Code Listing 7**.

(2) Check if CM0+ ready.

(3) Initialize peripherals.

Generate notify interrupt. See **Code Listing 8**.

Code Listing 3 Cy_IPC_Drv_GetIpcBaseAddress () function

```
__STATIC_INLINE volatile stc_IPC_STRUCT_t* Cy_IPC_Drv_GetIpcBaseAddress (uint32_t ipcIndex)
{
    CY_ASSERT((uint32_t)CPUSS_IPC_IPC_NR > ipcIndex);
    return ( (volatile stc_IPC_STRUCT_t*) ( &IPC->STRUCT[ipcIndex] ) );
}
```

Code Listing 4 Cy_IPC_Drv_LockRelease() function

```
cy_en_ipcdrv_status_t  Cy_IPC_Drv_LockRelease (volatile stc_IPC_STRUCT_t* base, uint32_t releaseEventIntr)
{
    cy_en_ipcdrv_status_t retStatus;

   /* Check to make sure the IPC is Acquired */
   if( Cy_IPC_Drv_IsLockAcquired(base) )
   {
        /* The IPC was acquired, release the IPC channel */
        Cy_IPC_Drv_ReleaseNotify(base, releaseEventIntr);

        retStatus = CY_IPC_DRV_SUCCESS;
    }
    else   /* The IPC channel was already released (not acquired) */
    {
        retStatus = CY_IPC_DRV_ERROR;
    }

    return(retStatus);
}
```

Check if the lock is acquired. See Code Listing 9.

Release the IPC channel. See **Code Listing 10**.

Code Listing 5 Cy_IPC_Drv_GetIntrBaseAddr() function

```
__STATIC_INLINE volatile stc_IPC_INTR_STRUCT_t* Cy_IPC_Drv_GetIntrBaseAddr (uint32_t ipcIntrIndex)
{
    CY_ASSERT((uint32_t)CPUSS_IPC_IPC_IRQ_NR > ipcIntrIndex);
    return ( (volatile stc_IPC_INTR_STRUCT_t*) ( &IPC->INTR_STRUCT[ipcIntrIndex] ) );
}
```

Get base address of IPC interrupt structure.

Code Listing 6 Cy_IPC_Drv_GetInterruptMask() function

```
__STATIC_INLINE uint32_t Cy_IPC_Drv_GetInterruptMask(volatile stc_IPC_INTR_STRUCT_t const * base)
{
    return (base->unINTR_MASK.u32Register);
}
```

Get value of the INTR_MASK.

Code Listing 7 Cy_IPC_Drv_ExtractAcquireMask() function

```
__STATIC_INLINE uint32_t Cy_IPC_Drv_ExtractAcquireMask (uint32_t intMask)
{
    return _FLD2VAL(IPC_INTR_STRUCT_INTR_MASK_NOTIFY, intMask);
}
```

Get value of INTR_MASK.NOTIFY value.

Code Listing 8 Cy_IPC_Drv_AcquireNotify() function

```
__STATIC_INLINE void  Cy_IPC_Drv_AcquireNotify (volatile stc_IPC_STRUCT_t* base, uint32_t notifyEventIntr)
{
    un_IPC_STRUCT_NOTIFY_t reg = { 0 };
    reg.stcField.u16INTR_NOTIFY = notifyEventIntr;
    base->unNOTIFY.u32Register = reg.u32Register;
}
```

(4) Generate Notify event.

Code Listing 9 Cy_IPC_Drv_IsLockAcquired() function

```
__STATIC_INLINE bool Cy_IPC_Drv_IsLockAcquired (volatile stc_IPC_STRUCT_t const * base)
{
    return ( 0u != base->unLOCK_STATUS.stcField.u1ACQUIRED );
}
```

Read value of IPC channel structure acquired

Code Listing 10 Cy_IPC_Drv_ReleaseNotify() function

```
__STATIC_INLINE void  Cy_IPC_Drv_ReleaseNotify (volatile stc_IPC_STRUCT_t* base, uint32_t notifyEventIntr)
{
    un_IPC_STRUCT_RELEASE_t reg = { 0 };
    reg.stcField.u16INTR_RELEASE = notifyEventIntr;
    base->unRELEASE.u32Register = reg.u32Register;
}
```

(1) Write to RELEASE register for releasing the IPC channel structure.

Code Listing 11 Example of CPU synchronization for CM0+

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)

cy_stc_sysint_irq_t irq_cfg =
{
    .sysIntSrc  = (cy_en_intr_t)(cpuss_interrupts_ipc_0_IRQn + IPC_NOTIFY_INT_NUMBER),
    .intIdx     = CPUIntIdx2_IRQn,
    .isEnabled  = true,
};

uint8_t waitFlag = 1ul;

int main(void)
{
:
    /* Enable application core CM4.
    *  CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout is changed.
    */
    Cy_SysEnableApplCore(CY_CORTEX_M4_APPL_ADDR);

    /* Enable IPC interrupt mask */
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t releaseMask = CY_IPC_NO_NOTIFICATION;
    uint32_t notifyMask = (1ul << IPC_CHANNEL_NUMBER);
    Cy_IPC_Drv_SetInterruptMask(ipcIntrStrBase, releaseMask, notifyMask);

    /* Interrupt setting */
    Cy_SysInt_InitIRQ(&irq_cfg);
    Cy_SysInt_SetSystemIrqVector(irq_cfg.sysIntSrc, IpcIntHandler);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(CPUIntIdx2_IRQn, 0ul);
    NVIC_EnableIRQ(CPUIntIdx2_IRQn);

    /* Wait until IPC interrupt is generated */
    while(waitFlag == 1ul);

    for(;;)
    {
:
    }
}
```

Define number of IPC interrupt for release event.

Define IPC channel

Configure IPC interrupt.

Activate CM4

Get base address of IPC interrupt structure. See **Code Listing 5**.

(5)-1 Enable IPC notify event. See **Code Listing 13**.

(5)-2 Configure interrupt for IPC notify event

(6) Wait for interrupt

(8) CM0+ can use peripheral.

Code Listing 12 Notify interrupt handler for CM0+

```
void IpcIntHandler(void)
{
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMasked    = Cy_IPC_Drv_GetInterruptStatusMasked(ipcIntrStrBase);
    uint32_t releaseMasked = CY_IPC_NO_NOTIFICATION; /* Do not care */
    uint32_t notifyMasked  = Cy_IPC_Drv_ExtractAcquireMask(intrMasked);



    /* Check if the interrupt is caused by the notifier channel */
    if (notifyMasked & (1ul << IPC_CHANNEL_NUMBER))
    {
        /* Clear IPC interrupt */
        Cy_IPC_Drv_ClearInterrupt(ipcIntrStrBase, releaseMasked, notifyMasked);

        /* Clear wait flag */
        waitFlag = 0ul;
    }
}
```

Get base address of IPC interrupt structure. See **Code Listing 5**.

Get value of INTR_MASKD. See **Code Listing 14**.

Get value of notify mask. See **Code Listing 7**.

Check if interrupts are valid.

Clear Interrupt flag. See **Code Listing 15**.

Code Listing 13 Cy_IPC_Drv_SetInterruptMask() function

```
__STATIC_INLINE void  Cy_IPC_Drv_SetInterruptMask (volatile stc_IPC_INTR_STRUCT_t* base,
                                                   uint32_t ipcReleaseMask, uint32_t ipcNotifyMask)
{
    un_IPC_INTR_STRUCT_INTR_MASK_t reg = { 0 };
    reg.stcField.u16NOTIFY  = ipcNotifyMask;
    reg.stcField.u16RELEASE = ipcReleaseMask;
    base->unINTR_MASK.u32Register = reg.u32Register;
}
```

Set notify and release interrupt mask.

Code Listing 14 Cy_IPC_Drv_GetInterruptStatusMasked() function

```
__STATIC_INLINE uint32_t Cy_IPC_Drv_GetInterruptStatusMasked (volatile stc_IPC_INTR_STRUCT_t const * base)
{
    return (base->unINTR_MASKED.u32Register);
}
```

Get value of INTR_MASKD.

Code Listing 15 Cy_IPC_Drv_ClearInterrupt() function

```
__STATIC_INLINE void  Cy_IPC_Drv_ClearInterrupt(volatile stc_IPC_INTR_STRUCT_t* base, uint32_t ipcReleaseMask,
uint32_t ipcNotifyMask)
{
    un_IPC_INTR_STRUCT_INTR_t reg = { 0 };
    reg.stcField.u16NOTIFY  = ipcNotifyMask;
    reg.stcField.u16RELEASE = ipcReleaseMask;
    base->unINTR.u32Register = reg.u32Register;
    (void)base->unINTR.u32Register;  /* Read the register to flush the cache */
}
```

(7) Clear interrupt flag

Read back

5.2 Mutual exclusion operation

This section describes how to mutually exclude shared resource access between CPUs using IPC. In a multi-CPU architecture, each CPU may 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 the memory. CPU_A is supposed to read and update the 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 the memory data before CPU_A updates the 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 the data.

IPC in TRAVEO™ T2G MCUs can easily implement exclusive access using the IPC_STRUCTx_ACQUIRE register. This register has a lock feature for the IPC channel structure. A lock of the IPC channel structure is acquired by reading this register. **Table 6** shows the result of the ACQUIRE register read operation.

Table 6 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; this allows for the generation of release events to the IPC interrupt structure.

5.2.1 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 a common peripheral resource. An IPC channel structure is associated with a common peripheral resource; when accessing the common peripheral resource, each CPU must acquire a lock on the associated IPC channel structure. Therefore, the CPU that cannot acquire the IPC channel structure lock is not allowed to access the common peripheral.

5.2.2 Use case

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

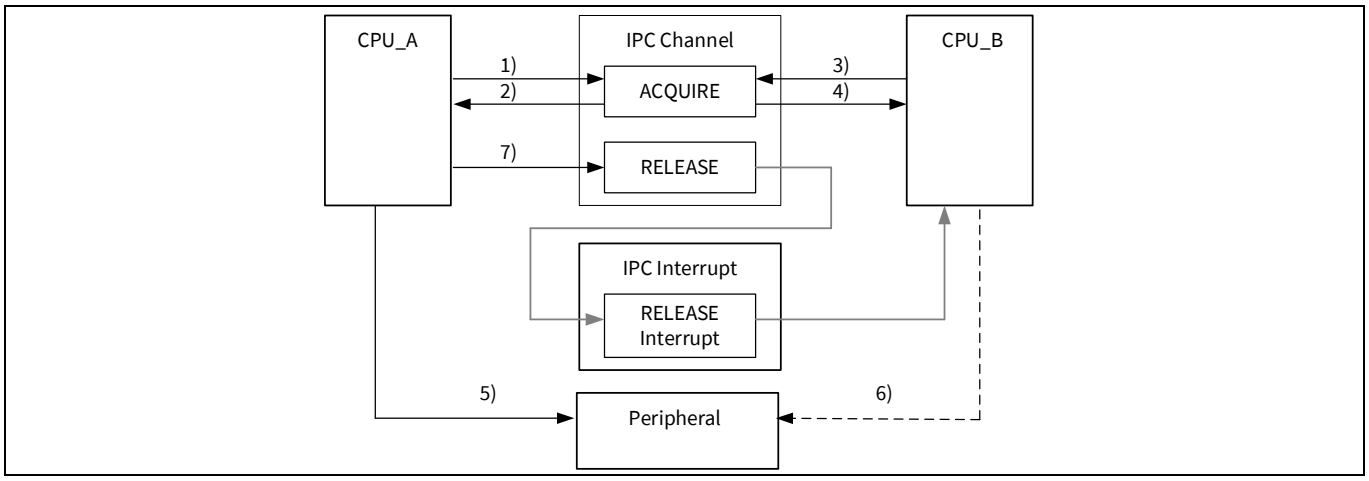

Figure 9 Example of exclusive access

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.

Communicating between CPUs

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 the 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 the 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 the write to the common peripheral is complete. If the IPC interrupt structure is set to generate a release interrupt by the IPC_STRUCTx_RELEASE register write, the 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.

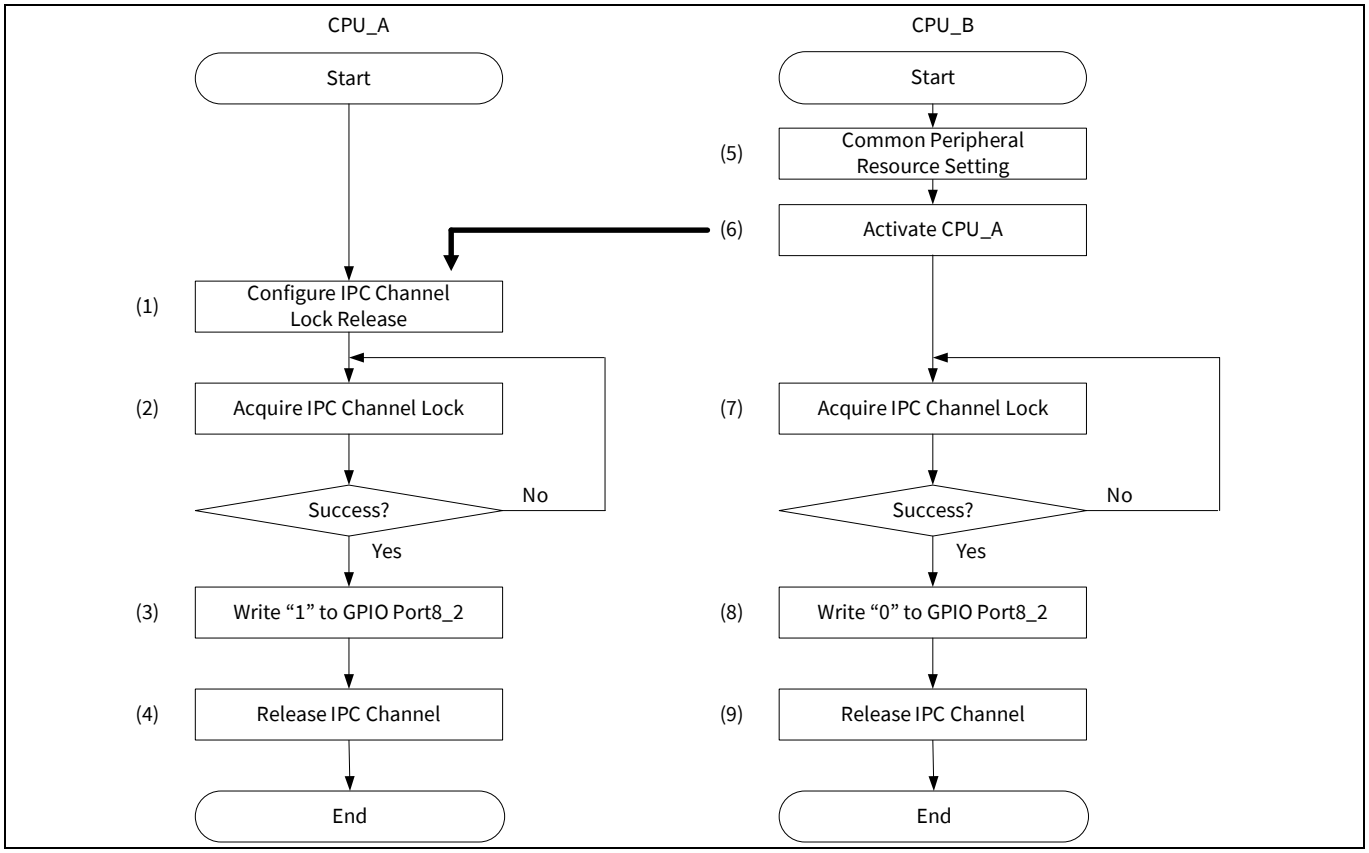

Figure 10 Mutual exclusion flow

The following shows the structure of the sample code.

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

Communicating between CPUs

See the “I/O System” chapter in the **architecture TRM** and **AN220193 - GPIO usage setup in TRAVEO™ T2G family** for GPIO configuration details.

5.2.3 Configuration

Table 7 and **Table 8** list the parameters and functions in the SDL for mutual exclusion using IPC. This example uses CYT2B series. In this case, it is assumed that CPU_A is CM4 and CPU_B is CM0+.

Table 7 Configuration parameters for mutual exclusion

Parameters	Description	Value
`IPC_CHANNEL_NUMBER`	Define using the IPC channel structure number	6ul (IPC6 channel structure)
`CY_CB_LED_PORT`	Define the I/O port number	GPIO_PRT8 (Assign to port 8)
`CY_CB_LED_PIN`	Define the I/O pin	2ul
`CY_IPC_NO_NOTIFICATION`	Defines a value to indicate that no notification events are needed	0x00000000ul
`user_led_port_pin_cfg.xxxx`	I/O port configuration See **AN220193 - GPIO usage setup in TRAVEO™ T2G family** for GPIO configuration details.	-

Table 8 Configuration functions for mutual exclusion

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

Code Listing 16 shows an example of mutual exclusion using IPC.

Code Listing 16 Example of mutual exclusion for CM4

```
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

#define CY_CB_LED_PORT                 GPIO_PRT8
#define CY_CB_LED_PIN                  2ul

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)

int main(void)
{
:
    /* At first force release the lock state. */
    volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
    (void)Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);



    /* Lock IPC channnel */
    for(;;)
    {
        if(CY_IPC_DRV_SUCCESS == Cy_IPC_Drv_LockAcquire(ipcBase))
        {
            /* Set IO port to 1 */
            Cy_GPIO_Write(CY_CB_LED_PORT, CY_CB_LED_PIN, 1ul);
```

Define IPC channel number

Define port and pin Number

Get base address of IPC channel structure. See **Code Listing 3**.

(1) IPC channel initialization (Release a lock). See **Code Listing 4**.

Acquire lock of IPC channel. See **Code Listing 17**.

(3) Set IO Port to 1.

Code Listing 16 Example of mutual exclusion for CM4

```
            /* Release the lock state */
            Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);

            break;
        }
    }

    for(;;)
    {
    }
}
```

(4) Release IPC channel without release event. See **Code Listing 4**.

Code Listing 17 Cy_IPC_Drv_LockAcquire() function

```
cy_en_ipcdrv_status_t Cy_IPC_Drv_LockAcquire (volatile stc_IPC_STRUCT_t const * base)
{
    cy_en_ipcdrv_status_t retStatus;

    if( 0ul != base->unACQUIRE.stcField.u1SUCCESS )
    {
        retStatus = CY_IPC_DRV_SUCCESS;
    }
    else
    {
        retStatus = CY_IPC_DRV_ERROR;
    }
    return(retStatus);
}
```

(2) Acquire lock of IPC channel

Code Listing 18 Example of mutual exclusion for CM0+

```
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

static cy_stc_gpio_pin_config_t user_led_port_pin_cfg =
{
    .outVal    = 0ul,
    .driveMode = CY_GPIO_DM_STRONG_IN_OFF,
    .hsiom     = CY_CB_LED_PIN_MUX,
    .intEdge   = 0ul,
    .intMask   = 0ul,
    .vtrip     = 0ul,
    .slewRate  = 0ul,
    .driveSel  = 0ul,
};

int main(void)
{
:
    /* Initialize the port pin for LED */
    Cy_GPIO_Pin_Init(CY_CB_LED_PORT, CY_CB_LED_PIN, &user_led_port_pin_cfg);

    /* Enable application core CM4.
    *  CY_CORTEX_M4_APPL_ADDR must be updated if CM4 memory layout is changed
    */
    Cy_SysEnableApplCore(CY_CORTEX_M4_APPL_ADDR);

    volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
    for(;;)
    {
        if(CY_IPC_DRV_SUCCESS == Cy_IPC_Drv_LockAcquire(ipcBase))
        {
            /* Set IO port to 0 */
            Cy_GPIO_Write(CY_CB_LED_PORT, CY_CB_LED_PIN, 0ul);


            /* Release the lock state */
            Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);

            break;
        }
    }

    for(;;)
    {
    }
}
```

Configure IO Port.

Define IPC channel number

(5) Initialize IO Port.

(6) Activate CM4

Get base address of IPC channel structure. See **Code Listing 3**.

(7) Acquire lock of IPC channel. See **Code Listing 17**.

(8) Set IO Port to 0.

(9) Release IPC channel without release event. See **Code Listing 4**.

5.3 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.

5.3.1 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.

5.3.2 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.

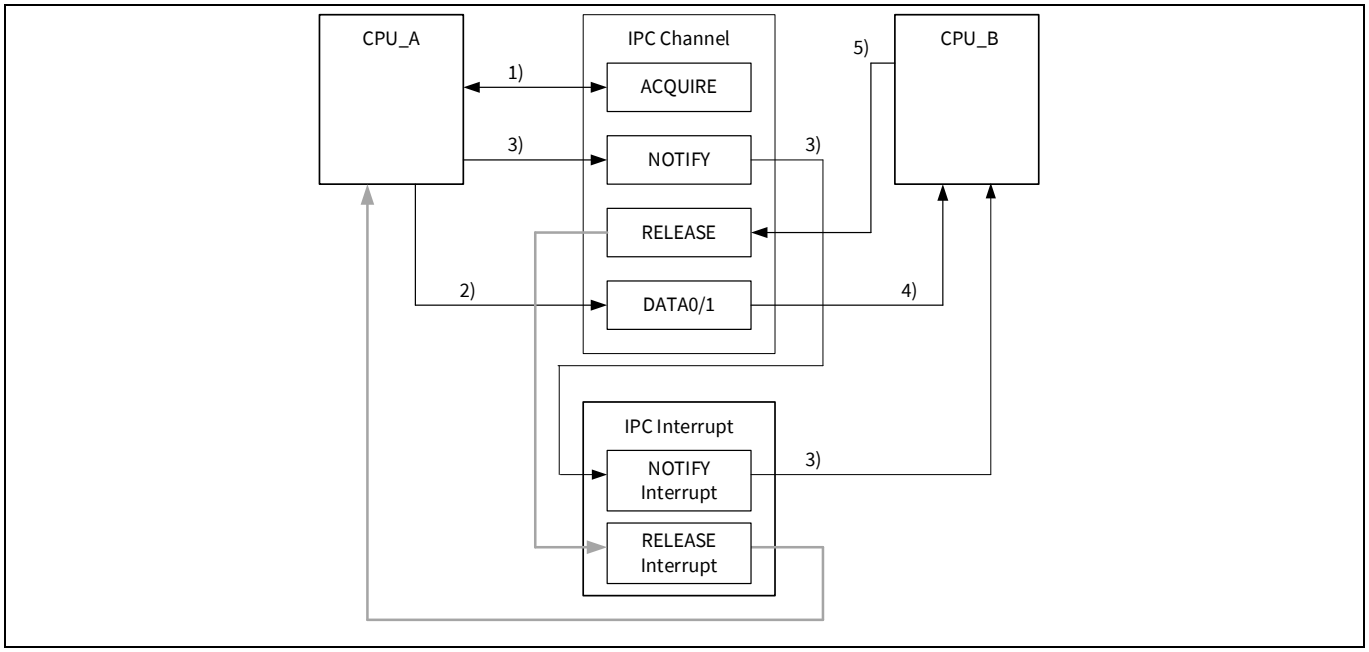

Figure 11 Example of passing a small message

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 the message data up to 64 bits in the IPC_STRUCTx_DATA0/1 registers.
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 registers.
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 was 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 bit).

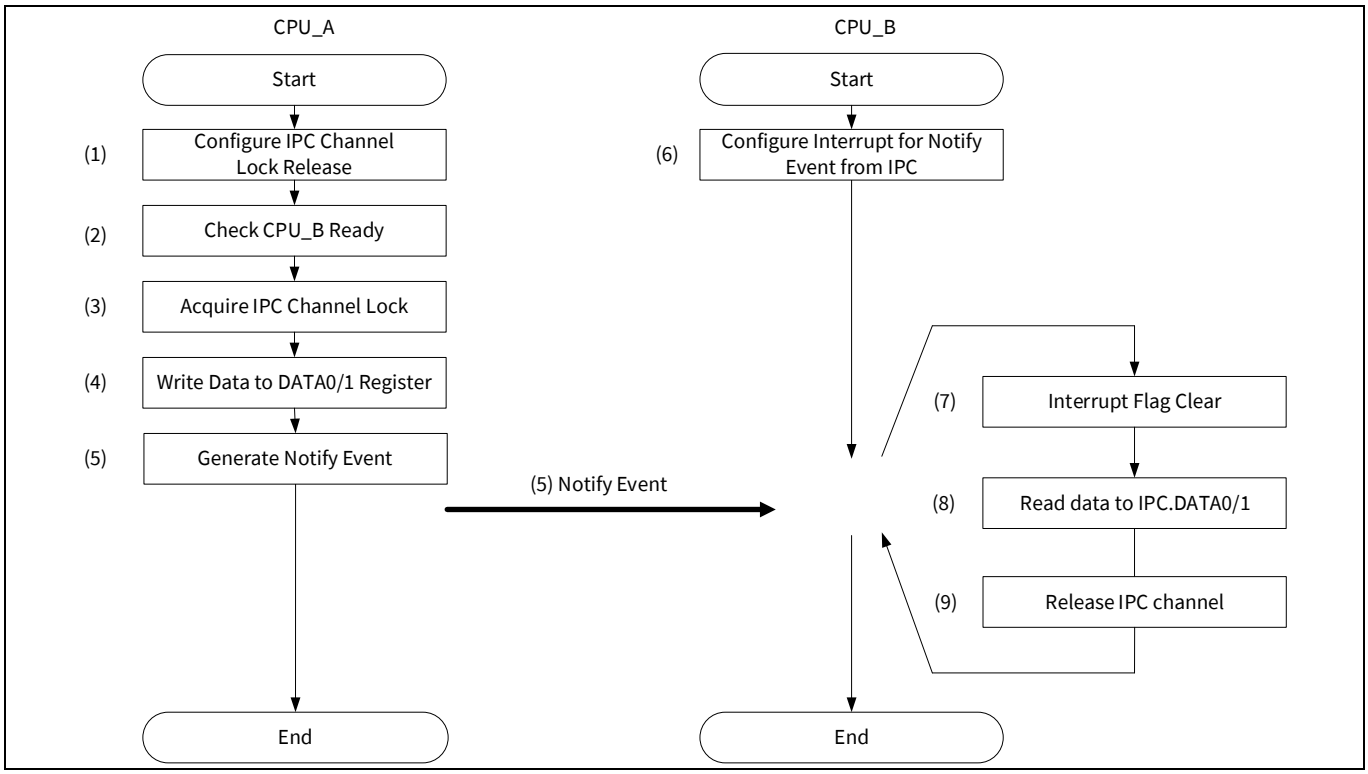

Figure 12 Data passing (up to 64 bits) flow

The following shows the structure of the sample code.

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

See the “Interrupts” chapter in the **architecture TRM** and **AN219842 - How to use interrupt in TRAVEO™ T2G** for interrupt configuration details.

5.3.3 Configuration

Table 9 and **Table 10** list the parameters and functions in the SDL for data passing of 64 bits or less using IPC. This example uses CYT2B series. In this case, it is assumed that CPU_A is CM4 and CPU_B is CM0+.

Table 9 Configuration parameters

Parameters	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Define using the IPC interrupt structure number for the notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Define using the IPC channel structure number	6ul (IPC6 channel structure)
`IPC_DATA`	Define passing data 0	0x5A5A5A5Aul
`IPC_DATA2`	Define passing data 1	0x12345678ul

Communicating between CPUs

Parameters	Description	Value
CY_IPC_NO_NOTIFICATION	Defines a value to indicate that no notification events are needed	0x00000000ul

Table 10 Configuration functions

Functions	Description	Remarks
Cy_IPC_Drv_SendMsgWord_2(base, notifyEventIntr, message, message2)	Set the DATA0/1 register of the IPC channel structure. base: Base address of the IPC channel to operate notifyEventIntr: Value of then 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_WriteDataValue(base, dataValue)	Write data to the DATA0 register. base: Base address of the IPC channel to operate dataValue: Write data	-
Cy_IPC_Drv_WriteData1Value(base, dataValue)	Write data to the DATA1 register. base: Base address of the IPC channel to operate dataValue: Write data	-
Cy_IPC_Drv_ReadMsgWord_2(base, message, message2)	Read the DATA0/1 register of the IPC channel structure. base: Base address of the IPC channel to operate message: Stored address for IPC.DATA0 message2: Stored address for IPC.DATA0	-
Cy_IPC_Drv_ReadDataValue(base)	Read the DATA0 register. base: Base address of the IPC channel to operate	-
Cy_IPC_Drv_ReadData1Value(base)	Read the DATA1 register. base: Base address of the IPC channel to operate	-

Communicating between CPUs

Code Listing 19 shows an example of data passing of 64 bits or less using IPC.

Code Listing 19 Example of data passing (up to 64 bits) for CM4

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */
#define IPC_DATA                0x5A5A5A5Aul
#define IPC_DATA2               0x12345678ul

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)


int main(void)
{
:
    /* At first force release the lock state. */
    volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPCs_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
    (void)Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);


    /* Wait until the CM0+ IPC server is started */
    /* Note:
     * After the CM0+ IPC server is started, the corresponding number of the INTR_MASK is set.
     * So in this case CM4 can recognize whether the server has started or not by the INTR_MASK status.
     */

    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMask;
    uint32_t notifyMask;
    do
    {
        intrMask = Cy_IPC_Drv_GetInterruptMask(ipcIntrStrBase);
        notifyMask = Cy_IPC_Drv_ExtractAcquireMask(intrMask);
    } while((notifyMask & (1ul << IPC_CHANNEL_NUMBER)) == 0);

    /* Send the message to the M0+ through IPC */
    Cy_IPC_Drv_SendMsgWord_2(ipcBase, (1ul << IPC_NOTIFY_INT_NUMBER), (uint32_t)IPC_DATA, (uint32_t)IPC_DATA2);

    for(;;)
    {
    }
}
```

Define number of IPC interrupt for notify event.

Define IPC channel number

Define send message data

Define Lock release data without release event

Get base address of IPC channel structure. See **Code Listing 3**.

(1) IPC channel initialization (Release a lock). See **Code Listing 4**.

Get base address of IPC interrupt structure. See **Code Listing 5**.

Get value of IPC.INTR_MASK. See **Code Listing 6**.

Get value of notify mask. See **Code Listing 7**.

(2) Check if CM0+ ready.

Write Data to DATA0/1 register. See **Code Listing 22**.

Code Listing 20 Cy_IPC_Drv_SendMsgWord_2() function

```
cy_en_ipcdrv_status_t  Cy_IPC_Drv_SendMsgWord_2 (volatile stc_IPC_STRUCT_t* base, uint32_t notifyEventIntr, uint32_t
message, uint32_t message2)
{
    cy_en_ipcdrv_status_t retStatus;

    if( CY_IPC_DRV_SUCCESS == Cy_IPC_Drv_LockAcquire(base) )
    {
        /* If the channel was acquired, send the message. */
        Cy_IPC_Drv_WriteDataValue(base, message);
        Cy_IPC_Drv_WriteData1Value(base, message2);

        Cy_IPC_Drv_AcquireNotify(base, notifyEventIntr);

        retStatus = CY_IPC_DRV_SUCCESS;
    }
    else
    {
        /* Channel was already acquired, return Error */
        retStatus = CY_IPC_DRV_ERROR;
    }
    return(retStatus);
}
```

(3) Acquire lock of IPC channel. See **Code Listing 17**.

Set passing data. See **Code Listing 23**.

(5) Generate notify interrupt. See **Code Listing 8**.

Code Listing 21 Cy_IPC_Drv_WriteDataValue() and Cy_IPC_Drv_WriteData1Value() functions

```
__STATIC_INLINE void     Cy_IPC_Drv_WriteDataValue (volatile stc_IPC_STRUCT_t* base, uint32_t dataValue)
{
      base->unDATA0.u32Register = dataValue;
}

__STATIC_INLINE void     Cy_IPC_Drv_WriteData1Value (volatile stc_IPC_STRUCT_t* base, uint32_t dataValue)
{
      base->unDATA1.u32Register = dataValue;
}
```

(4) Set IPC.DATA0 register

(4) Set IPC.DATA1 register

Code Listing 22 Example of data passing (up to 64 bits) for CM0+

Define number of IPC interrupt for notify event.

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

cy_stc_sysint_irq_t irq_cfg =
{
    .sysIntSrc  = (cy_en_intr_t)(cpuss_interrupts_ipc_0_IRQn + IPC_NOTIFY_INT_NUMBER),
    .intIdx     = CPUIntIdx2_IRQn,
    .isEnabled  = true,
};

int main(void)
{
:
    /* Enable IPC interrupt mask */
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t releaseMask = CY_IPC_NO_NOTIFICATION;
    uint32_t notifyMask = (1ul << IPC_CHANNEL_NUMBER);
    Cy_IPC_Drv_SetInterruptMask(ipcIntrStrBase, releaseMask, notifyMask);

    /* Interrupt setting */
    Cy_SysInt_InitIRQ(&irq_cfg);
    Cy_SysInt_SetSystemIrqVector(irq_cfg.sysIntSrc, IpcNotifyInt_ISR);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(CPUIntIdx2_IRQn, 0ul);
    NVIC_EnableIRQ(CPUIntIdx2_IRQn);

    for(;;)
    {
    }
}
```

Define IPC channel

Configure notify interrupt

(6)-1 Enable IPC release event.

(6)-2 Configure interrupt for IPC notify event

Code Listing 23 Notify interrupt handler

```
void IpcNotifyInt_ISR(void)
{
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMasked    = Cy_IPC_Drv_GetInterruptStatusMasked(ipcIntrStrBase);
    uint32_t releaseMasked = CY_IPC_NO_NOTIFICATION; /* Do not care */
    uint32_t notifyMasked  = Cy_IPC_Drv_ExtractAcquireMask(intrMasked);

    /* Check if the interrupt is caused by the notifier channel */
    if (notifyMasked & (1ul << IPC_CHANNEL_NUMBER))
    {
        /* Clear the interrupt */
        Cy_IPC_Drv_ClearInterrupt(ipcIntrStrBase, releaseMasked, notifyMasked);

        /* Read DATA */
        uint32_t Data;
        uint32_t Data2;
        volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
        Cy_IPC_Drv_ReadMsgWord_2(ipcBase, &Data, &Data2);

        /* Finally relase the lock */
        Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);
    }
}
```

Get value of notify mask. See **Code Listing 12**.

Check if interrupts are valid.

(7) Clear Interrupt flag. See **Code Listing 15**.

Read passing data. See **Code Listing 26**.

(9) Release the IPC channel. See **Code Listing 4**.

Code Listing 24 Cy_IPC_Drv_ReadMsgWord_2() function

```
cy_en_ipcdrv_status_t  Cy_IPC_Drv_ReadMsgWord_2 (volatile stc_IPC_STRUCT_t const * base, uint32_t * message, uint32_t*
message2)
{
    cy_en_ipcdrv_status_t retStatus;

    if ( Cy_IPC_Drv_IsLockAcquired(base) )
    {
        /* The channel is locked; message is valid. */
        *message  = Cy_IPC_Drv_ReadDataValue(base);
        *message2 = Cy_IPC_Drv_ReadData1Value(base);

        retStatus = CY_IPC_DRV_SUCCESS;
    }
    else
    {
        /* The channel is not locked so channel is invalid. */
        retStatus = CY_IPC_DRV_ERROR;
    }
    return(retStatus);
}
```

Check if the lock is acquired. See **Code Listing 9**.

Read passing data. See **Code Listing 27**.

Code Listing 25 Cy_IPC_Drv_ReadDataValue() and Cy_IPC_Drv_ReadData1Value() function

```
__STATIC_INLINE uint32_t Cy_IPC_Drv_ReadDataValue (volatile stc_IPC_STRUCT_t const * base)
{
    return (base->unDATA0.u32Register);
}

__STATIC_INLINE uint32_t Cy_IPC_Drv_ReadData1Value (volatile stc_IPC_STRUCT_t const * base)
{
    return (base->unDATA1.u32Register);
}
```

(8) Read IPC.DATA0 register

(8) Read IPC.DATA1 register

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

This section describes 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 registers to pass the pointer and size on which the message is placed to CPU_B.

*Note: This section describes an implementation example using the shared memory. This case can cause cache coherency issue. See **Consideration for cache coherency issue** for more details.*

5.3.5 Use case

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

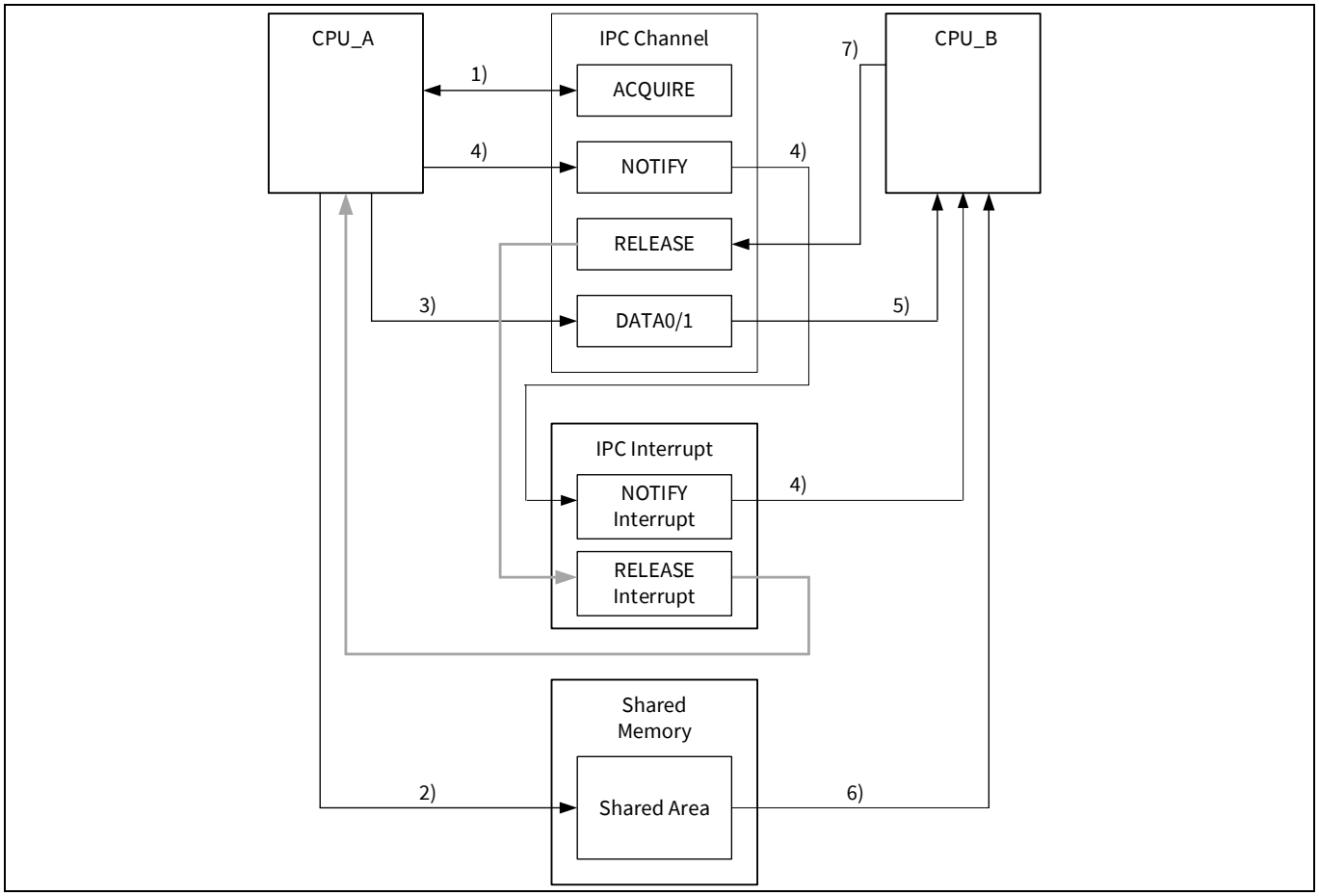

Figure 13 Example of passing a large message

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 registers.
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 registers.
6. CPU_B reads the 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 was 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).

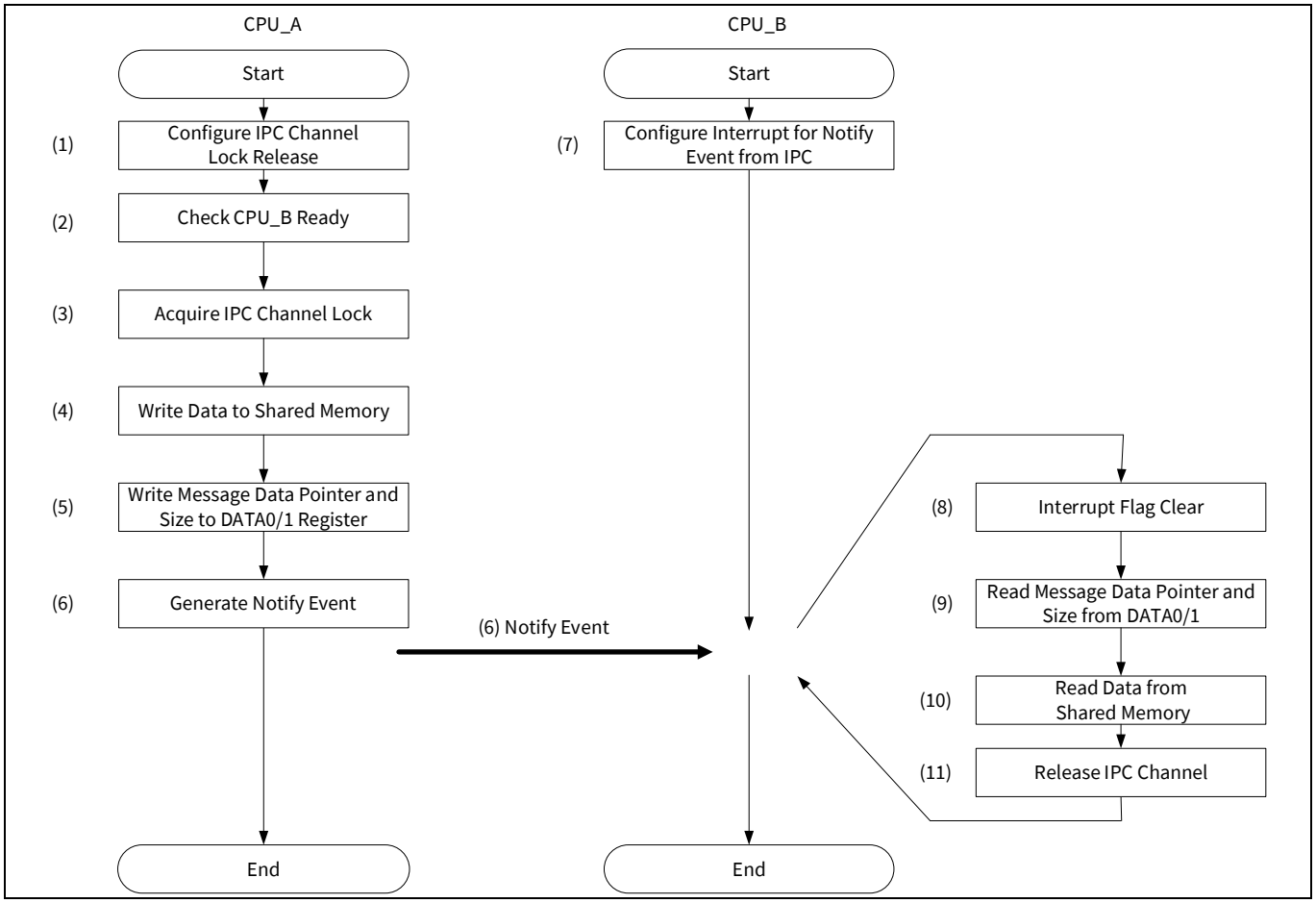

Figure 14 Data passing (more than 64 bits) flow

The following shows the structure of the sample code.

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

5.3.6 Configuration

Table 11 lists the parameters and functions in the SDL for data passing of more than 64 bits using IPC. This is example in CYT2B series. In this case, it is assumed that CPU_A is CM4 and CPU_B is CM0+.

Table 11 Configuration parameters

Parameters	Description	Value
`IPC_NOTIFY_INT_NUMBER`	Define using the IPC interrupt structure number for notify event	5ul (IPC5 interrupt structure)
`IPC_CHANNEL_NUMBER`	Define using the IPC channel structure number	6ul (IPC6 channel structure)
`DATA_SIZE`	Define the passing data size	4ul (4 word)
`sharedData[]`	Shared memory area on the SRAM	-

Communicating between CPUs

Parameters	Description	Value
ipc_data[]	Passing data	0x12345678ul, 0x87654321ul, 0x12345678ul, 0x87654321ul
CY_IPC_NO_NOTIFICATION	Defines a value to indicate that no notification events are needed	0x00000000ul

Code Listing 26 shows an example of data passing of more than 64 bits using IPC.

Code Listing 26 Example of data passing (more than 64 bits) for CM4

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */
#define DATA_SIZE              4ul /* Word */

/* Use shared memory(.bss_share) defined in the "linker_directives.ld" file */
#pragma ghs section bss=".bss_share"
uint32_t sharedData[DATA_SIZE];
#pragma ghs section bss=default

uint32_t ipc_data[DATA_SIZE] = {0x12345678ul, 0x87654321ul, 0x12345678ul, 0x87654321ul};

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)

int main(void)
{
:
    /* At first force release the lock state. */
    volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
    (void)Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);


    /* Wait until the CM0+ IPC server is started */
    /* Note:
     * After the CM0+ IPC server is started, the corresponding number of the INTR_MASK is set.
     * So in this case CM4 can recognize whether the server has started or not by the INTR_MASK status.
     */


    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMask;
    uint32_t notifyMask;
    do
    {
        intrMask = Cy_IPC_Drv_GetInterruptMask(ipcIntrStrBase);
        notifyMask = Cy_IPC_Drv_ExtractAcquireMask(intrMask);
    } while((notifyMask & (1ul << IPC_CHANNEL_NUMBER)) == 0);


    if(CY_IPC_DRV_SUCCESS == Cy_IPC_Drv_LockAcquire(ipcBase))
    {
        /* Write to Shared momory */
        for(uint32_t i = 0ul; i < DATA_SIZE; i++ )
        {
            sharedData[i] = ipc_data[i];
        }

        /* Send message */
        Cy_IPC_Drv_WriteDataValue(ipcBase, (uint32_t)&sharedData[0]);
        Cy_IPC_Drv_WriteDataIValue(ipcBase, (uint32_t)DATA_SIZE);
        Cy_IPC_Drv_AcquireNotify(ipcBase, (1ul << IPC_NOTIFY_INT_NUMBER));
    }

    for(;;)
    {
    }
}
```

Define number of IPC interrupt for notify event.

Define IPC channel

Define passing data size

Shared area reserved on SRAM.

Define passing data

Get base address of IPC channel structure. See **Code Listing 3**.

(1) IPC channel initialization (Release a lock). See **Code Listing 4**.

Get base address of IPC interrupt structure. See **Code Listing 5**.

Get value of IPC.INTR_MASK. See **Code Listing 6**.

Get value of notify mask. See **Code Listing 7**.

(2) Check if CM0+ ready.

(3) Acquire lock of IPC channel. See **Code Listing 17**.

(4) Write a message to the shared area on SRAM.

(5) Set passing data pointer and size. See **Code Listing 23**.

(6) Generate notify interrupt. See **Code Listing 8**.

Code Listing 27 Example of passing data (more than 64 bits) for CM0+

```
#define IPC_NOTIFY_INT_NUMBER  5ul /* Notify interrupt number */
#define IPC_CHANNEL_NUMBER     6ul /* IPC number which is used in this example */

#define CY_IPC_NO_NOTIFICATION         (uint32_t)(0x00000000ul)

cy_stc_sysint_irq_t irq_cfg =
{
    .sysIntSrc  = (cy_en_intr_t)(cpuss_interrupts_ipc_0_IRQn + IPC_NOTIFY_INT_NUMBER),
    .intIdx     = CPUIntIdx2_IRQn,
    .isEnabled  = true,
};

uint32_t receivedData[64] = {0ul};

int main(void)
{
:
    /* Enable IPC interrupt mask */
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t releaseMask = CY_IPC_NO_NOTIFICATION;
    uint32_t notifyMask = (1ul << IPC_CHANNEL_NUMBER);
    Cy_IPC_Drv_SetInterruptMask(ipcIntrStrBase, releaseMask, notifyMask);

    /* Interrupt setting */
    Cy_SysInt_InitIRQ(&irq_cfg);
    Cy_SysInt_SetSystemIrqVector(irq_cfg.sysIntSrc, IpcNotifyInt_ISR);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(CPUIntIdx2_IRQn, 0ul);
    NVIC_EnableIRQ(CPUIntIdx2_IRQn);

    for(;;)
    {
    }
}
```

Define IPC interrupt number for notify event

Define IPC channel

Configure release interrupt

Configure Receive area

(7)-1 Enable IPC release event.

(7)-2 Configure interrupt for IPC notify event

Code Listing 28 Notify interrupt handler

```
void IpcNotifyInt_ISR(void)
{
    volatile stc_IPC_INTR_STRUCT_t* ipcIntrStrBase = Cy_IPC_Drv_GetIntrBaseAddr(IPC_NOTIFY_INT_NUMBER);
    uint32_t intrMasked     = Cy_IPC_Drv_GetInterruptStatusMasked(ipcIntrStrBase);
    uint32_t releaseMasked = CY_IPC_NO_NOTIFICATION; /* Do not care */
    uint32_t notifyMasked  = Cy_IPC_Drv_ExtractAcquireMask(intrMasked);

    /* Check if the interrupt is caused by the notifier channel */
    if (notifyMasked & (1uL << IPC_CHANNEL_NUMBER))
    {
        /* Clear the interrupt */
        Cy_IPC_Drv_ClearInterrupt(ipcIntrStrBase, releaseMasked, notifyMasked);

        /* Read DATA */
        uint32_t Address;
        uint32_t Size;
        volatile stc_IPC_STRUCT_t* ipcBase = Cy_IPC_Drv_GetIpcBaseAddress(IPC_CHANNEL_NUMBER);
        Cy_IPC_Drv_ReadMsgWord_2(ipcBase, &Address, &Size);

        for(uint32_t i = 0ul; i < Size; i++ )
        {
            receivedData[i] = *(uint32_t*)(Address + (i*4ul));
        }

        /* Finally relase the lock */
        Cy_IPC_Drv_LockRelease(ipcBase, CY_IPC_NO_NOTIFICATION);
    }
}
```

Get value of notify mask. See **Code Listing 12**.

Check if interrupts are valid.

(8) Clear Interrupt flag. See **Code Listing 15**.

(9) Read passing data. See **Code Listing 26**.

(10) Read passing data from shared memory.

(11) Release the IPC channel. See **Code Listing 4**.

6 Consideration for cache coherency issue

A cache memory helps to improve the CPU performance from its high-speed read/write operation. However, the characteristics of the cache memory may cause a data mismatch between the cache memory and other memories, that is, cache coherency issue. Cache coherency issues should be considered mainly in CYT4B and CYT4D series which have a cache memory in the CPU.

This section provides an overview of the cache memory in these series MCUs and explains the cache coherency issue under different scenarios. In addition, it provides methods to manage or avoid the cache coherency issue. The shared memory shown in this section includes read and writable memory, including the external memory.

6.1 Cache coherency

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

The 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 the cache memory, the view of the 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 a different data, causing an unintended operation (cache coherency issue). **Figure 15** shows a general example of how a coherency issue occurs. As a precondition, the shared memory is allocated to the cache memory.

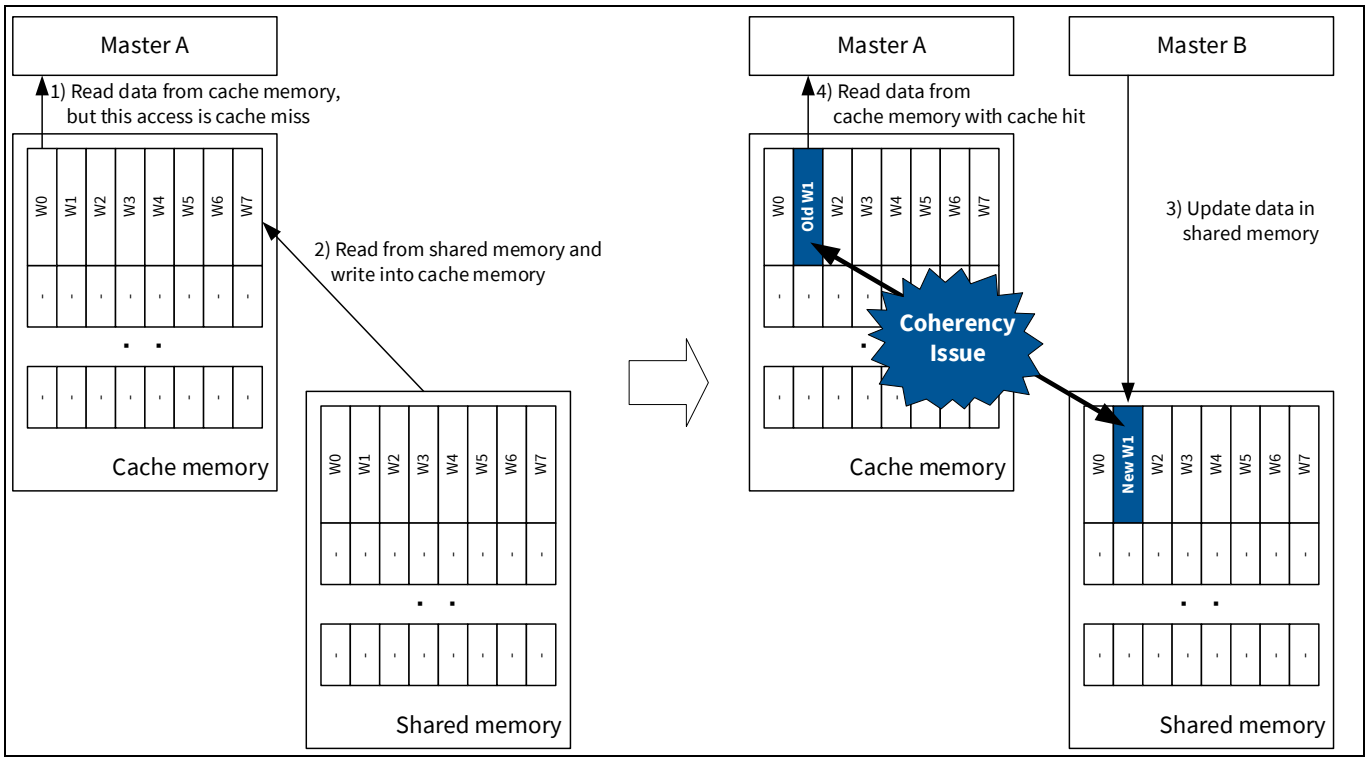

Figure 15 Coherency issue example

The cache memory does not have data before the start of the operation.

1. Master A tries to read the data from the cache memory. However, the cache memory does not have the data. Therefore, this access causes a “cache miss”.

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

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

6.2 Cache memory overview

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

6.2.1 Cache memory placement

Figure 16 shows the placement of the cache memory.

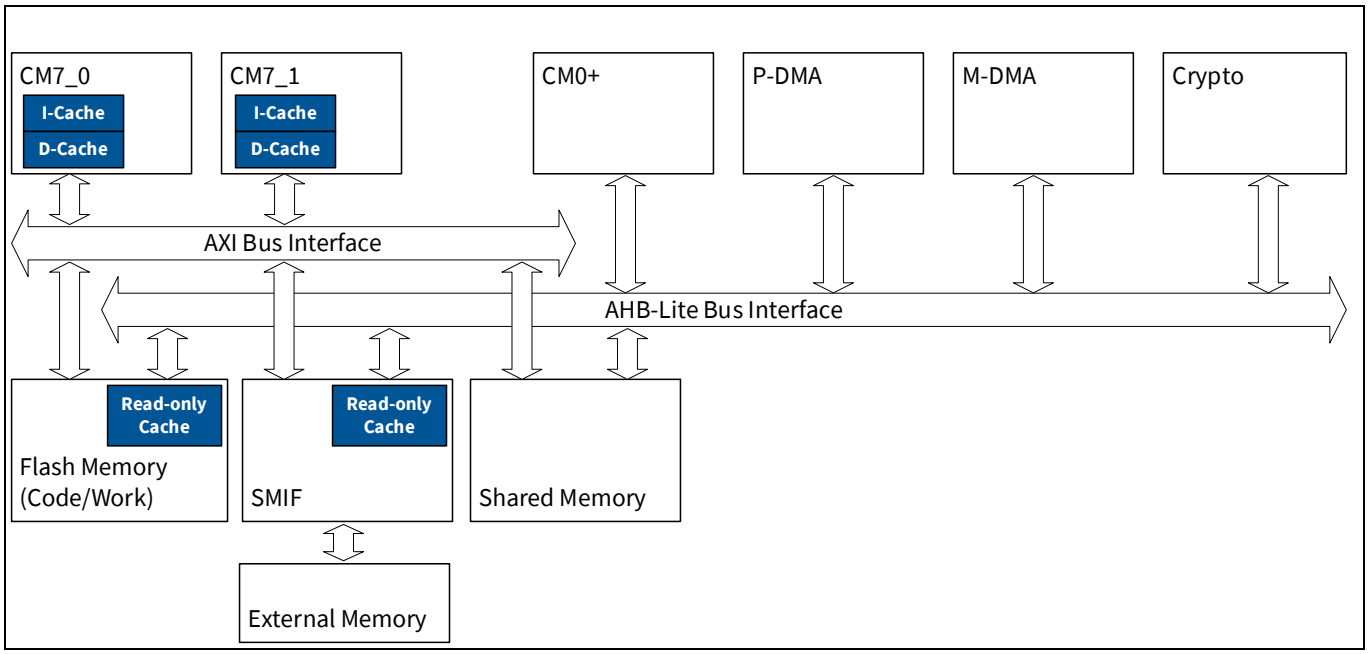

Figure 16 Cache memory placement

In these series MCUs, CM7 CPUs have I-cache and D-cache, and flash memory and Serial Memory Interfaces (SMIF) have read-only cache memory for the AHB-Lite bus interface.

6.2.2 I-cache and D-cache operation

The I-cache and D-cache are implemented as part of CM7. These cache memories are valid for the access that is for an AXI bus interface. When access to the cacheable memory on the AXI bus interface and cache is enabled, this access attempts a lookup in the cache memory. These caches are independent for each CPU. There is no synchronization (snoop) between the caches of each CPU.

6.2.2.1 Cache memory behavior

When the CPU finds data in the cache memory (cache hit), the data is read from the cache memory or written into the cache memory. **Table 12** lists the behavior of the cache memory in CM7. This operation assumes that the shared memory is allocated to the cache memory.

Table 12 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. The 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, a read access to these memory locations will be a cache hit, and data is read from the cache memory.
Write access	Cache hit	Writeback	The data is written into the cache memory. The cache line is marked as “dirty”, and the data in the cache memory is written to the shared memory only when the line is evicted.
		Write-through	The data is written into the cache memory. The data is also written to the shared memory, so that the data stored in the cache memory is coherent with the shared memory.
	Cache miss	Write allocate	The 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	The cache memory does not allocate a memory location to a cache line. The data is written to the shared memory.

6.2.2.2 Cache memory configuration

The following configurations are supported for cache memory in CM7. Cache memories in CM7 can be configured using the CM7-specific register.

- Non-cache:
  - Cache memory does not work. Always read and write on the shared memory.
  - This configuration is not affected by the cache coherency issue.
- Write-back, write and read allocate
  - Cache hit of the read access reads from the cache memory.
  - Cache hit of the write access updates only the cache memory.
  - Cache miss of the read and write access copies the data from the shared memory to the cache memory.
  - This configuration must handle the coherency issue.
- Writeback, no write allocate
  - Cache hit of the read access reads from the cache memory.
  - Cache miss of the read access copies the data from the shared memory to the cache memory.
  - Cache hit of the write access updates only the cache memory.
  - Cache miss of the write access does not copy the data from the shared memory to the cache memory.
  - This configuration must handle the coherency issue.

Consideration for cache coherency issue

- Writethrough, no write allocate
  - Cache hit of the read access reads from the cache memory.
  - Cache miss of the read access copies the data from the shared memory to the cache memory.
  - Cache hit or miss of the write access performs on the shared memory.
  - This configuration solves the cache coherency issue partially.

These configurations are available in the MPU Region Attribute and Size Register (MPU_RASR). **Table 13** shows the MPU_RASR common combination for cache configuration. The configuration of the cache memory is defined by TEX, C, B in MPU_RASR.

Table 13 TEX, C, 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, B encoding.

6.2.2.3 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 made directly to the shared memory when cache is OFF.
- Invalidate: Clear the valid bit of the cache line. Data in the cache memory is invalidated. Subsequent access is a cache miss; data is fetched from the shared memory and written to the cache memory.
- Clean: Write the updated data in the cache memory back to the shared memory. The shared memory data matches the cache memory.

To perform these cache maintenance operations, you can use the Cortex® Microcontroller Software Interface Standard (CMSIS). **Table 14** lists the cache maintenance APIs supported by CMSIS.

Table 14 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

Consideration for cache coherency issue

Cache maintenance APIs	Description
`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
`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 29 to **Code Listing 31** show examples of using some cache maintenance APIs.

Code Listing 29 Example of using the cache maintenance API (1)

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

Invalidates and then enables I-cache

Invalidates and then enables D-cache

Code Listing 30 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();

:
}
```

Cleans and invalidates D-cache. The data of the shared memory match cache memory.

Invalidates I-cache. Subsequent access is cache miss.

Code Listing 31 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)
{
    SystemInit();
:
    // 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
:
```

Define buffer size

See **Code Listing 30**.

Initialize buffer. It executes in cache memory.

Clean D-cache for buffer area. The data of the shared memory match cache memory.

Code Listing 31 Example of using the cache maintenance API (3)

```
    // 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(;;)
    {
:   }
}
```

Clean D-cache for descriptor area

DMA transfer from srcBuffer to dstBuffer.

Invalidate D-cache for dstBuffer area. Subsequent access is cache miss.

Read from dstBuffer. Data is fetched from shared memory

6.2.3 Cache memory operation in flash memory (Code and Work)

Table 15 shows the behavior of the flash memory cache memory. This cache memory is read-cache. Therefore, write access data is directly written into associated memories.]

Table 15 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 the flash memory; 16-byte data is refilled from the flash memory to the cache memory. Subsequent access result is cache hit.
Write access		The write access bypasses the cache memory. In the flash memory, the write access without a specific sequence generally causes an access error.

In general, the flash memory does not rewrite as frequently as the RAM. Also, the 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 16** lists the control registers to invalidate and enable/disable the cache memory. The cache memory can be enabled/disabled using a register. When the cache memory is set to disable and enable again, data in the cache memory is invalidated; a read access causes refilling the cache memory. See the **registers TRM** for more details.

Table 16 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)

6.2.4 Cache memory operation in SMIF

Table 17 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 17 Behavior of cache memory in SMIF

Operation		Description
Read access	Cache hit	Data is read from the cache memory.
	Cache miss	Access occurs to the external memory, and 16-byte data is refilled from the external memory to the cache memory. A subsequent access results in a cache hit.
Write access		The write access bypasses the cache memory. The data is directly written into the 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 these, only the XIP AHB-Lite interface has cache memory. In addition, this cache memory does not support cache coherency by hardware. Therefore, SMIF has the cache coherency issue depending on the access between each port. Table 18 lists the control registers for invalidating and enabling/disabling of the cache memory. See the registers TRM for more details.

Table 18 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', 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 least recently used (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)

6.3 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 a cache memory and shared memory coherency issues.

6.3.1 Cache disable

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

6.3.2 Cache invalidate

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 invalidated. 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 APIs such as `SCB_InvalidateDCache_by_Addr`.

6.3.3 Cache clean

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

6.3.4 Cache configuration sets to write-through

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

6.3.5 Use TCM as shared memory

Each CM7 CPU has instruction/data tightly coupled memory (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 targets for an AXI bus interface. Thus, ITCM and DTCM can access without the cache memory. Therefore, ITCM/DTCM can be used as shared memory without being affected by 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. Therefore, the cache coherency issue must be considered when CM7 accesses another CM7 TCM memory.

All bus masters can access ITCM and DTCM using a dedicated address space. No actions are required for the cache memory coherency issue. See the **device datasheet** for TCM address mapping.

6.4 Cache coherency issue scenarios

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

6.4.1 Cache coherency issue between CM7 CPUs

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

6.4.1.1 Scenario and solution between CM7 CPUs

CM7 has I-cache and D-cache. The cache coherency issue occurs mainly with the 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 CPUs enable the cache memory to access the common area.
- 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.

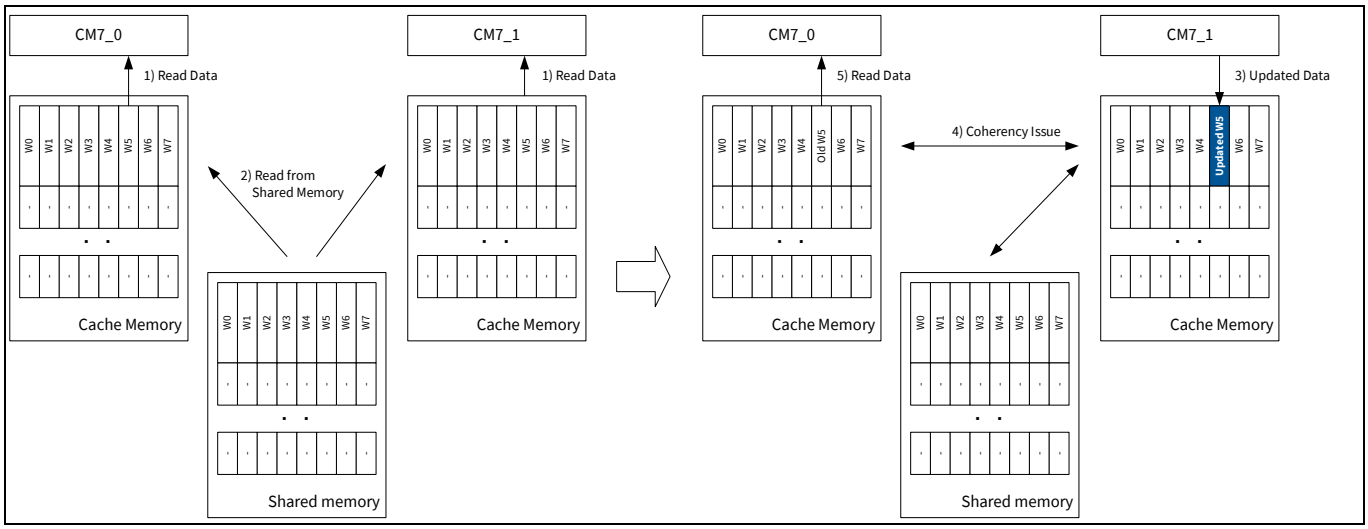

Figure 17 Scenario between CM7 CPUs

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

Here are some solutions for this scenario between CM7 CPUs:

- Solution 1: Disable the cache

  Both CM7 CPUs configure cache disable to the common area. The 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 a write access to the cache memory. Cache clean writes the data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean.

Consideration for cache coherency issue

CM7_0 performs cache invalidate before the read access from the cache memory. Cache invalidate invalidates the data in the cache memory; a subsequent read access refills the cache memory data with the shared memory data. The cache memory and the shared memory are coherent after the read access with cache invalidate performed.

- Solution 3: Change the cache configuration

  CM7_1 cache memory is configured as write-through. CM7_1 writes the 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 the 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 the TCM.

  - Case of using CM7_1 TCM:

    CM7_1 is not required for handling of the cache coherency issue regardless of the cache configuration. CM7_1 always writes to the TCM.

    However, the read access of CM7_0 still has the 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 the coherency issue. Therefore, CM7_1 needs to perform cache clean after the write access to the cache memory, or configure the cache memory as write-through.

    CM7_0 is not required for handling of the cache coherency issue regardless of the cache configuration. CM7_0 always reads from the TCM directly without having to go through the cache memory.

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

6.4.2 Cache coherency issue between CM7 CPU and other masters

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

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

In this scenario, the DMA transfers the data from the peripheral to the shared memory, and CM7_0 reads the data. That is, the 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:

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

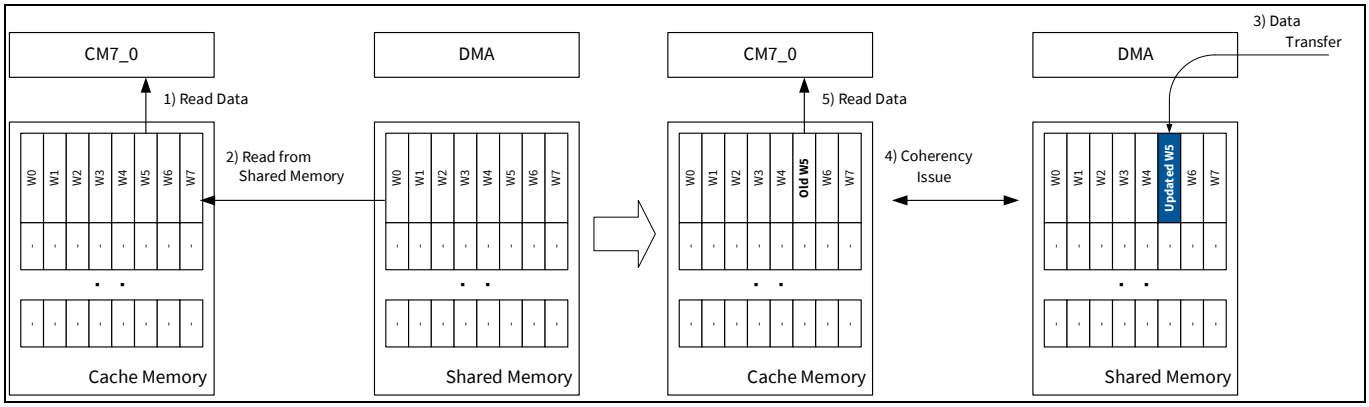

Figure 18 Scenario between CM7 CPU and other master (CM7_0 reads, DMA writes)

1. CM7_0 tries to read the data from the cache memory. However, the cache memory does not have the data, thus, it is a cache miss.
2. As a result of the 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. A subsequent access result is cache hit.
3. The DMA writes the data to the shared memory through data transfer.
4. W5 (Updated W5) in the shared memory is different from W5 (Old W5) in the CM7_0 cache memory. That is, this has the cache coherency issue.
5. CM7_0 reads old W5 from the cache memory. As a result, CM7_0 can cause an unintended operation.

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

- Solution 1: Disable the cache

  CM7_0 configures cache disable to the common area. The 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 a read access from the cache memory. The cache memory and the shared memory are coherent after the read access with cache invalidate performed.

- Solution 3: Use the TCM

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

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

In this scenario, CM7_0 writes the data, the DMA transfers the data from the shared memory to the peripheral. That is, the 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:

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

Consideration for cache coherency issue

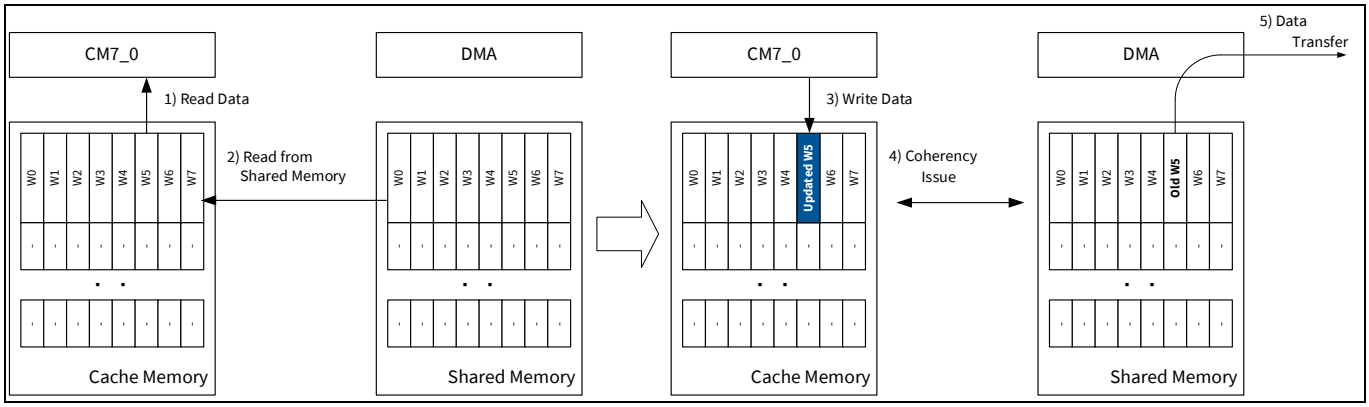

Figure 19 Scenario between CM7 CPU and other master (CM7_0 writes, DMA reads)

1. CM7_0 tries to read the data from the cache memory. However, the cache memory does not have data, thus, it is a cache miss.
2. As a result of the 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. A subsequent access results in a cache hit.
3. CM7_0 updates the W5 data in its own cache memory according to the 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 the cache coherency issue.
5. The DMA reads and transfers old W5 in the shared memory. As a result, a DMA transfer can cause an unintended operation.

Note: *DMA has a data prefech function. Source data transfers are initiated as soon as the channel is enabled if the current descriptor pointer is not "0" and there is space available in the channel's data FIFO. After the input trigger is activated, the DMA initiates destination data transfers with the data that is already in the channel's data FIFO. Therefore, the data prefech function can cause coherency issues. See the "Direct Memory Access" chapter in the* ***architecture TRM*** *for details.*

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

- Solution 1: Disable the cache

  CM7 CPU configures cache disable to the common area. The cache memory does not operate, and the CPU writes to the shared memory directly. The 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 the write access to the cache memory. Cache clean writes the 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: Use the TCM

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

6.4.3 Cache coherency issue for flash memory access

The flash memory has read-only cache memory for 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 the RAM. In TRAVEO™ T2G MCUs, flash memory programming is performed using the SROM API. The SROM API invalidates the cache memory in the flash memory after programming. During a subsequent read access, the cache memory refills the data from the flash memory. There is no need to manage the cache coherency issue.

6.4.4 Cache coherency issue for SMIF access

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

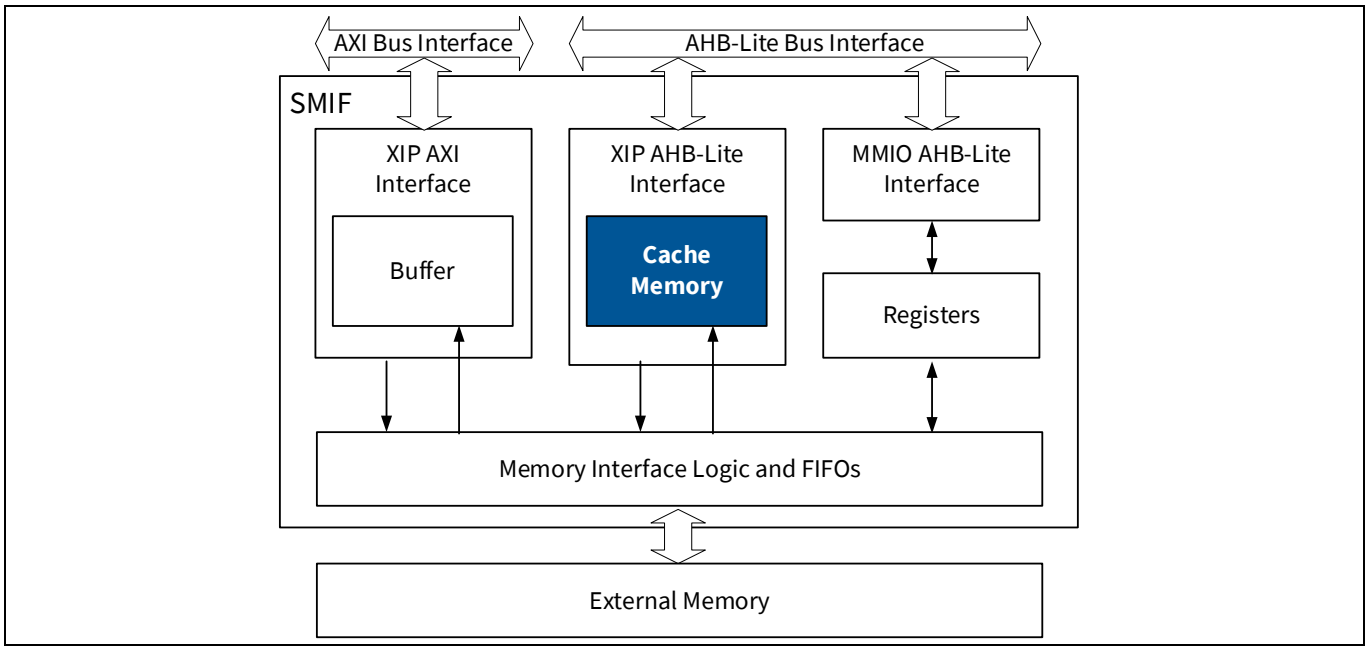

Figure 20 Block diagram of SMIF bus interface

SMIF has three bus interfaces: XIP AXI, XIP AHB-Lite, and MMIO AHI-Lite. The XIP AXI interface is used by CM7 to access the external memory in XIP mode. The XIP AHB-Lite interface is used by masters except CM7 to access the external memory in XIP mode. The MMIO AHB-Lite interface is used by all master to access the external memory in MMIO mode. See the “Serial Memory Interface” chapter in the **architecture TRM** for XIP mode, MMIO mode, and interface details.

Out of the three interfaces, only XIP AHB-Lite interface has cache memory with read-only access. 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 the cache coherency issue for write access from XIP AHB-Lite and MMIO interfaces.

6.4.4.1 Scenario and solution for CM7 access (CM0+ writes and CM7_0 reads)

In this scenario, CM7_0 accesses the external memory via the XIP AXI interface. Also, CM0+ accesses the external memory via the XIP AHB-Lite interface. Two scenarios need to be considered in this case. One scenario where CM0+ writes the data to the external memory and CM7_0 reads data from the external memory. Another scenario where CM7_0 writes the 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 the common area is enabled for CM7_0 XIP mode access, and CM7_0 cache configuration is write-back, write, and read allocate.
- The SMIF cache memory of the common area is enabled for CM0+ XIP mode access.

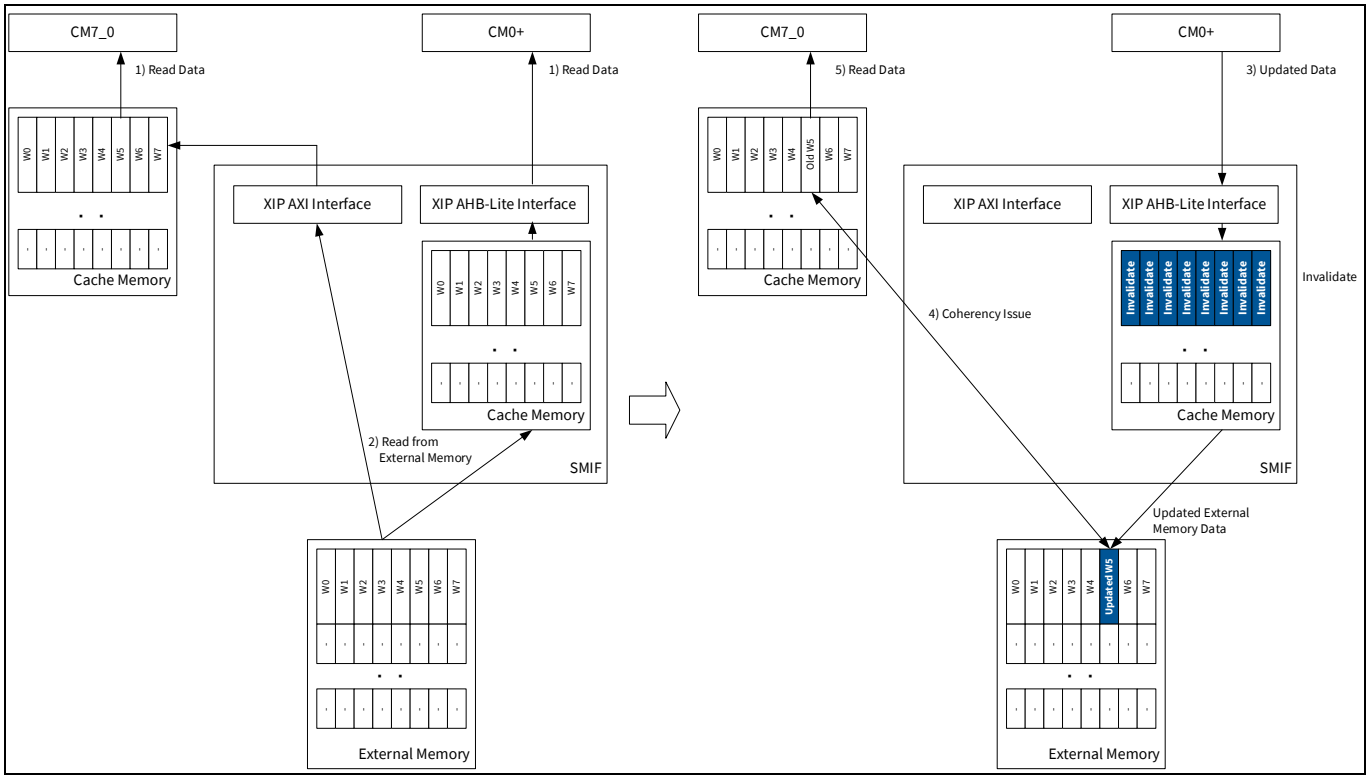

Figure 21 Scenario between CM7 and CM0+ (CM7_0 reads, CM0+ writes)

1. CM7_0 and CM0+ try to read the data from the cache memory. However, the cache memory does not have data; thus, it is a cache miss.
2. As a result of the read access, the cache memories refill the data from the external memory. The data in the cache memories and the external memory are the same at this point. Therefore, they are coherent. A subsequent access results in a cache hit.
3. CM0+ updates W5. As a result of the 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 the 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 the cache coherency issue.
5. CM7_0 reads old W5 from the cache memory. As a result, CM7_0 can cause an unintended operation.

Consideration for cache coherency issue

Note: *In this scenario, the cache memory is configured as write-back and write allocate. Therefore, when a write access occurs before reading, the cache coherency problem can occur. See **Scenario and solution for CM7 access (CM0+ reads and CM7_0 writes)** for more details.*

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

- Solution 1: Disable the 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, the handling is not required to the cache coherency issue.

- Solution 2: Use cache maintenance APIs

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

6.4.4.2 Scenario and solution for CM7 access (CM0+ reads and CM7_0 writes)

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.

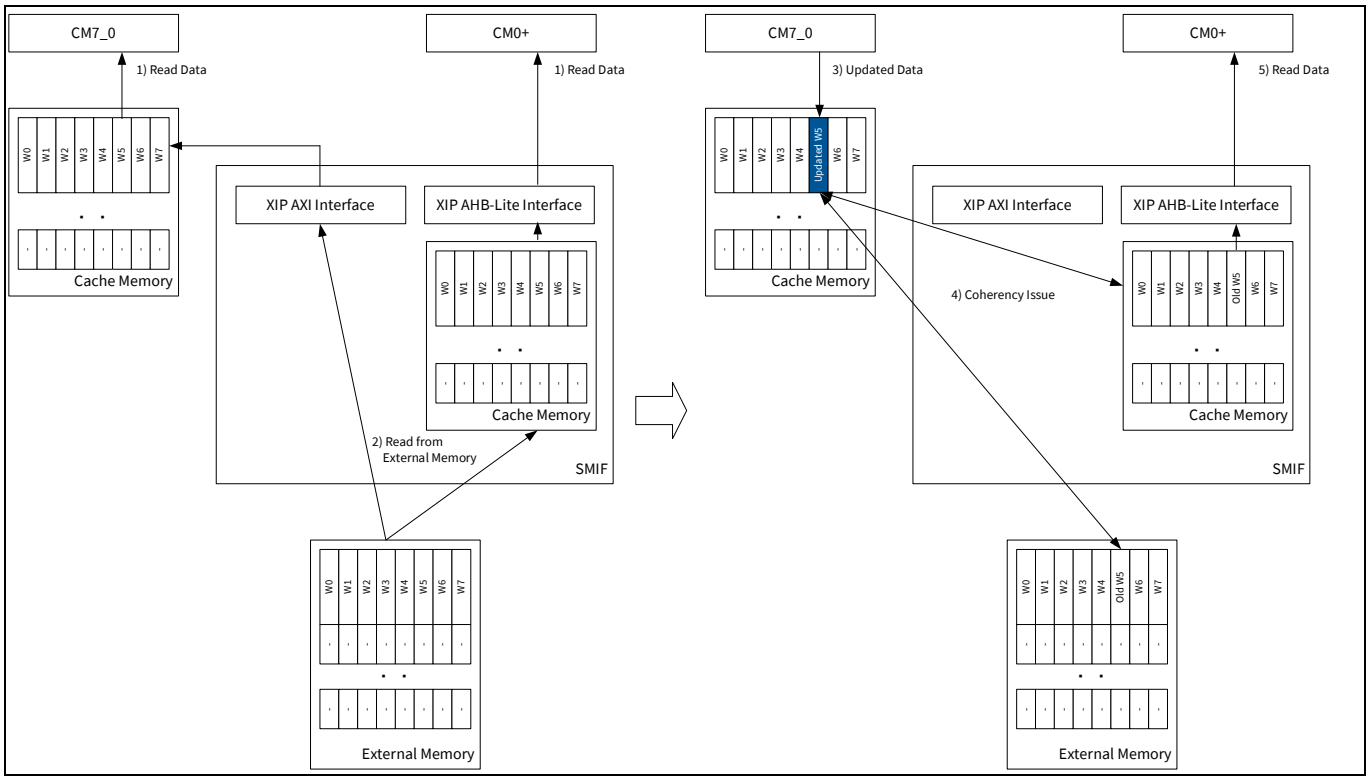

Figure 22 Scenario between CM7 and CM0+ (CM7_0 writes, CM0+ reads)

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

Consideration for cache coherency issue

2. As a result of the read access, cache memories refill the data form the external memory. The data in cache memories and the external memory are the same at this point. Therefore, they are coherent. A subsequent access results in a cache hit.
3. CM7_0 updates the W5 data in its own cache memory according to the cache configuration, but this write access does not update the 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 the cache coherency issue.
5. CM0+ reads old W5 from the cache memory. As a result, CM0+ can cause an unintended operation.

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

- Solution 1: Disable the cache

  CM7_0 and CM0+ configure cache disable to the common area. The 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 a write access to the cache memory. Cache clean writes the data from the cache memory back to the shared memory. The cache memory and the shared memory are coherent after performing cache clean. When CM7_0 performs a write access, the SMIF cache memory must be invalidated before reading from another master. Therefore, the application software must monitor the write access from the XIP AXI interface and MMIO AHB-Lite interface. See **CPU synchronization** for synchronization between CPUs.

- Solution 3: Change the cache configuration

  CM7_0 cache memory is configured as write-through cache. CM7_0 writes the data to the cache memory and shared memory. The write access of CM7_0 has no coherency issue between the cache memory and shared memory.

6.4.5 Cache coherency issue for using SROM APIs

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

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

6.4.5.1 Scenario and solution when using SROM API (CM0+ API parameter read)

In this scenario, CM7 uses the SROM API to read a specific memory data. 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 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 of CM7. **Figure 23** shows the cache coherency issue scenario in a CM0+ API parameter read. The preconditions are as follows:

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

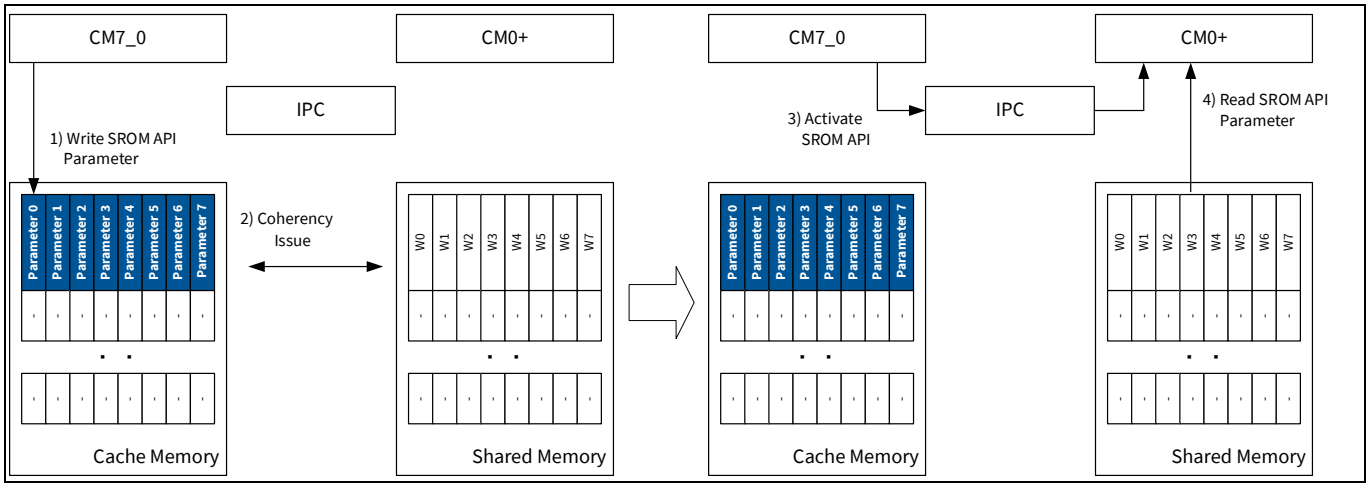

Figure 23 Scenario CM0+ SROM API parameter read

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

  CM7 CPU configures cache disable to the common area. The cache memory does not operate, and the CPU writes to the shared memory directly. The 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 a write access to the cache memory. Cache clean writes the 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.

6.4.5.2 Scenario and solution when using SROM API (CM7 execution result read)

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

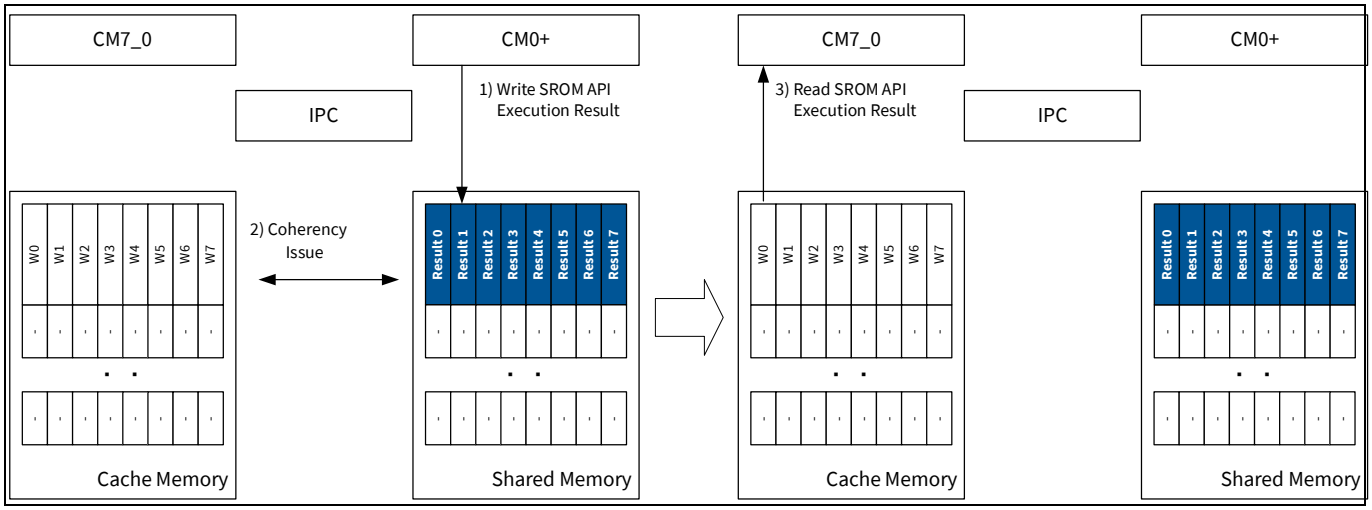

Figure 24 Scenario CM7 SROM API parameter read

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 the CM7_0 cache memory. That is, this has the cache coherency issue.
3. CM7_0 reads the execution result from the cache memory. However, CM7_0 reads the non-updated execution result. As a result, CM7_0 cannot perform correctly.

Here are some solutions for the scenario:

- Solution 1: Disable the cache

  CM7 CPU configures cache disable to the common area. The cache memory does not operate; the CPU writes to the shared memory directly. The 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 a read access from the cache memory. The cache memory and the shared memory are coherent after performing read access with cache invalidate.

6.5 Additional cache issue scenarios

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

6.5.1 Cache issue for protection attribute switching

6.5.1.1 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 the 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:

- The CPU uses a part of the shared memory, and CM7 enables the cache memory to access the shared memory.
- The shared memory with the cache enabled contains an area accessible only on PC = 4 and another area accessible only on PC = 5.
- The CPU cache configuration is write-back, write, and read allocate.

See the “Protection Unit” chapter in the **architecture TRM** for PC and “secure” attributes.

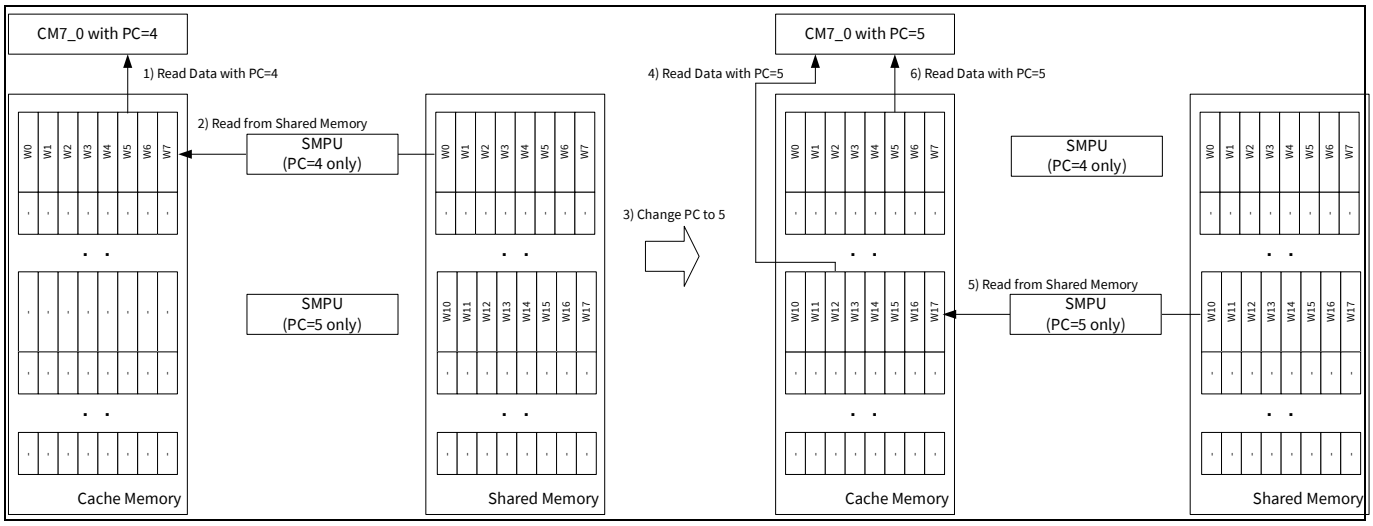

Figure 25 Scenario of switching the PC

1. CM7_0 is operating on PC=4, and tries to read the data from the cache memory. However, the cache memory does not have data; thus, it is a cache miss.
2. As a result of the read access, the cache memory refills the data from the shared memory. The data in the cache memory and the shared memory are the same at this point. A subsequent access results in a cache hit.
3. CM7_0 changes the protection context from PC=4 to PC=5.
4. CM7_0 tries to read the data from the cache memory. However, the cache memory does not have data; thus, it is a cache miss.
5. As a result of the read access, the cache memory refills the data from the shared memory. The data in the cache memory and the shared memory are the same at this point. A 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.

Consideration for cache coherency issue

Here are some solutions for the scenario:

- Solution 1: Disable the cache

  CM7 configures cache disable in the common area. The cache memory does not operate; the CPU reads and writes to the shared memory directly. The CPU has no cache issue.

- Solution 2: Use cache maintenance APIs

  CM7_0 performs cache clean and invalidate before switching the PC. Cache clean writes the 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 the data in the cache memory; a subsequent read access refills the cache memory data with the shared memory data via the SMPU. Therefore, while accessing an area of PC = 4 with PC=5, an SMPU protection violation will occur.

7 Glossary

Terms	Description
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 **architecture TRM** 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
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 **architecture TRM** for details.
P-DMA	Peripheral DMA. See the ‘Direct Memory Access” chapter of the **architecture TRM** 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 **architecture TRM** for details.
XIP	eXecute-In-Place

8 Related documents

The following are the TRAVEO™ T2G family series datasheets and technical reference manuals. Contact **Technical Support** to obtain these documents.

[1] Device datasheet

- **CYT2B7 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family**
- **CYT2B9 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family**
- **CYT4BF datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family**
- CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-24601)
- **CYT3BB/4BB datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family**
- CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-27763)

[2] Body controller entry family

- **TRAVEO™ T2G automotive body controller entry family architecture technical reference manual (TRM)**
- **TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B7**
- **TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B9**

[3] Body controller high family

- **TRAVEO™ T2G automotive body controller high family architecture technical reference manual (TRM)**
- **TRAVEO™ T2G Automotive body controller high registers technical reference manual (TRM) for CYT4BF**
- **TRAVEO™ T2G automotive body controller high registers technical reference manual (TRM) for CYT3BB/4BB**

[4] Cluster 2D family

- TRAVEO™ T2G automotive cluster 2D family architecture technical reference manual (TRM) (Doc No. 002-25800)
- TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT4DN (Doc No. 002-25923)
- TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT3DL (Doc No. 002-29584)

[5] Application note

- **AN219842 - How to use interrupt in TRAVEO™ T2G**
- **AN220208 - Clock configuration setup in TRAVEO™ T2G body entry family**
- **AN224434 - Clock configuration setup in TRAVEO™ T2G family CYT4B series**
- **AN226071 - Clock configuration setup in TRAVEO™ T2G family CYT4D series**
- **AN220193 - GPIO usage setup in TRAVEO™ T2G family**
- **Arm® Cortex®-M Programming Guide to Memory Barrier Instructions**

9 Other references

A Sample Driver Library (SDL) including startup as sample software to access various peripherals is provided. The SDL also serves as a reference, to customers, for drivers that are not covered by the official AUTOSAR products. The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. Contact **Technical Support** to obtain the SDL.

Revision history

Document version	Date of release	Description of changes
**	2019-12-12	New Application Note.
*A	2021-02-02	Moved to Infineon Template Updated code examples using SDL
*B	2021-04-19	Added Chapter 5.5. Added target parts number (CYT3 series).
*C	2021-06-18	Updated TRAVEO™ trademarks.
*D	2021-12-07	Updated **Figure 1**, **Figure 2** and **Figure 16**. Changed description and added note in section **1** and **0**. Changed description: Section **5.2**, **5.2.1**, **6**, **6.4.1.1**, **6.4.2.1**, and **6.4.5.1**. Added note in section **5.3.4**, **6.4.2.2** and **6.4.4.1**. Added description in section **6.2.2** and **6.3.5**. Added section **6.4.4.2**. Fixed typo.
*E	2022-03-08	Added section **4**.

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

Edition 2022-03-08

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to: www.cypress.com/support

Document reference

002-24432 Rev. *E

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.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

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.