Multi Core Handling Guide in Traveo II

Associated part family

Traveo™ II Family

About this document

Scope and purpose

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

Intended audience

This document is intended for anyone using Traveo II family .

Table of contents

Table of contents

1 Introduction

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

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

In addition, the CYT3 and CYT4 series have cache memory on CPU and some peripherals. A cache memory is a low latency memory, and helps to improve the performance. However, cache memory can cause coherency issues between memories. Therefore, use of 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 SDL. See **Other References** for the SDL.

SDL basically has a configuration part and a driver part. The configuration part mainly configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part. You can configure the configuration part according to your system.

Figure 1 shows block diagram of CPU subsystem for CYT2B series, and **Figure 2** shows block diagram for CYT4B/CYT4D series.

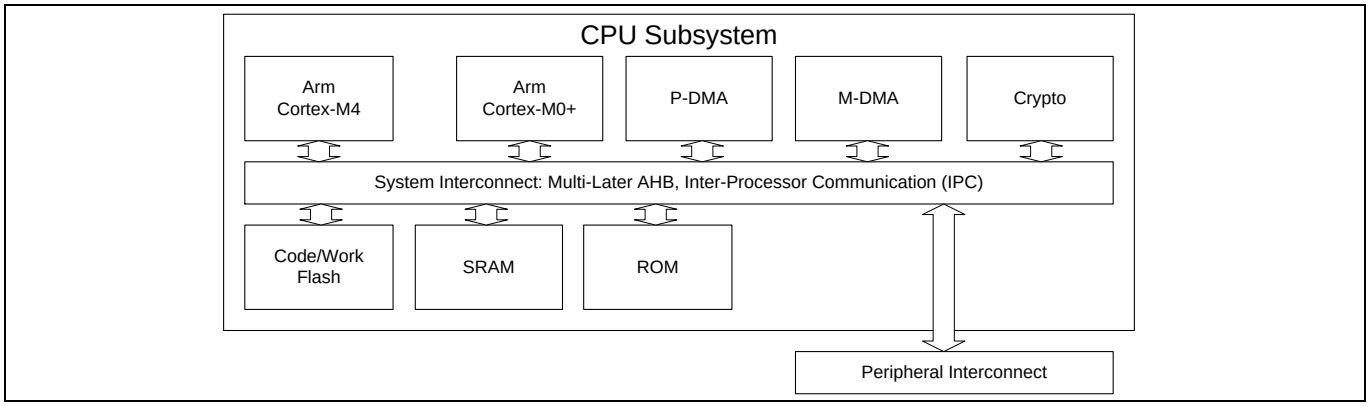

Figure 1 CPU Subsystem for CYT2 Series

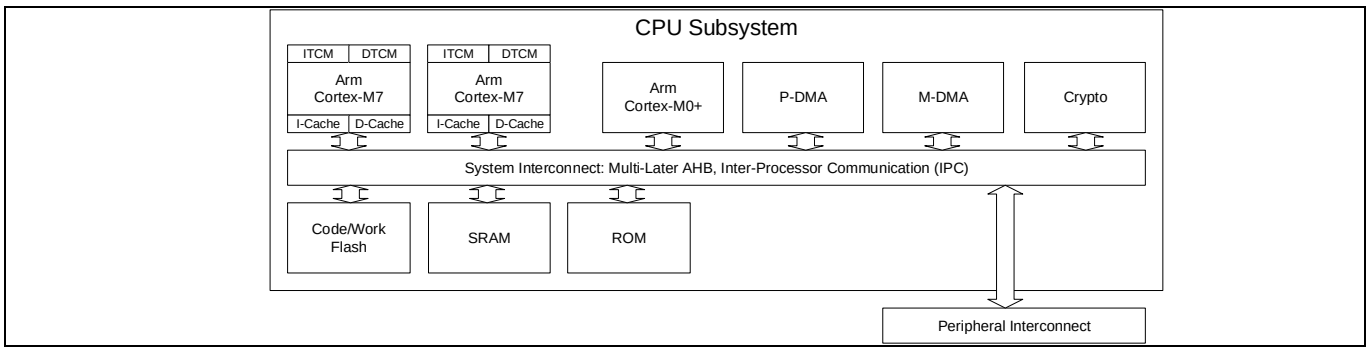

Figure 2 CPU Subsystem for CYT4 Series

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 CPU subsystems of the CYT2, CYT3 and CYT4 series MCUs have bus masters for P-DMA, M-DMA, and Crypto block. See the Arm documentation sets for **CM7**, **CM4**, and **CM0+**, and the Traveo II **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.*

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

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

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

2 Consideration for CPU Start Up

Generally, when 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 BOD.

This is especially important for the CYT4B and CYT4D series, which have two CM7s.

Here is an example of stepping up CPU clock frequency in stages for CYT4B series. **Figure 3** shows CM7 CPUs' clock connection in this example. This example uses CLK_PATH1 with PLL400#0 as the root clock for CLK_HF1, which is the CM7 CPUs' clock.

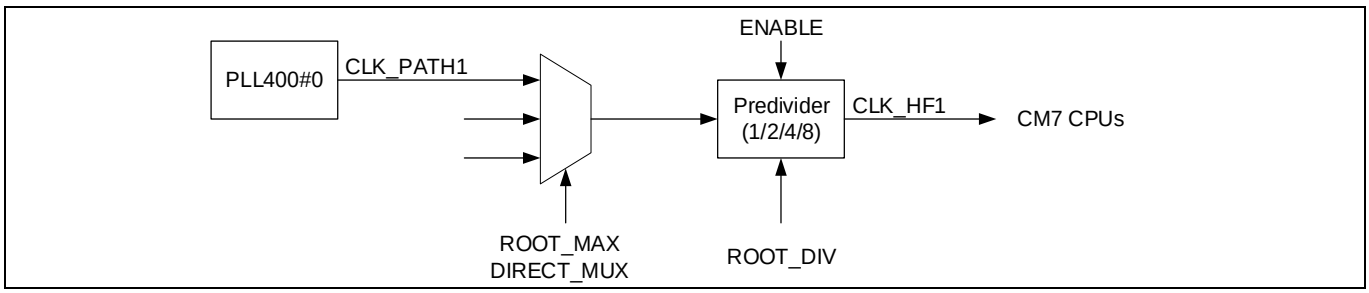

Figure 3 CM7 CPUs Clock Connection

2.1 Example of Step Up CPU Clock Frequency

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

2.1.1 Configuration

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

Table 1 List of 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 List of Functions

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

Consideration for CPU Start Up

The following code shows an example of step up clock frequency:

- ***SRSS***->un***CLK_ROOT_SELECT[x]***.stcField.u4***ROOT_MUX*** is the ROOT_MUX field in 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 of registers.

Code Listing 1 Example of Step up 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);
    }
:
}
```

Set PLL here.

CLK_HF configuration

Step up CPU Clock frequency

Consideration for CPU Start Up

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 CLK_HF clock divider. The `Cy_SysTick_DelayCoreCycle()` provides the required delay between each successive step. See the **Architecture TRM** and **Application Note** for PLL and CLK_HF configuration.

3 Consideration for Resource Access

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

Figure 4 shows resource connection for CYT4BF series. IPC (blue box) access of CM7_0 and CRYPT (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).

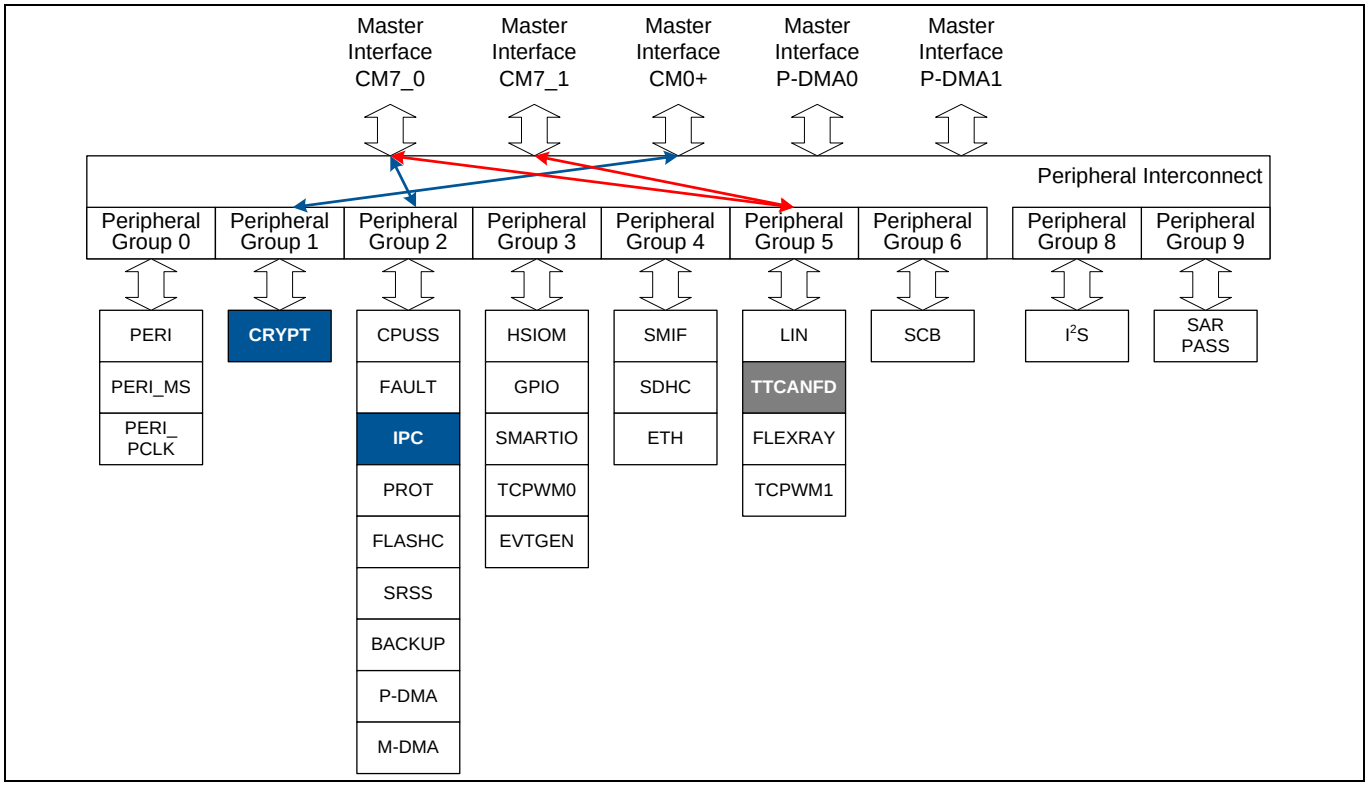

Figure 4 Resource Connection (CYT4BF Series)

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 CM_1 to TTCAN FD management will improve performance.

A similar case occurs for memory access. For example, SRAM0 access of CM7_0 and SRAM1 access of CM7_1 can be performed at 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).

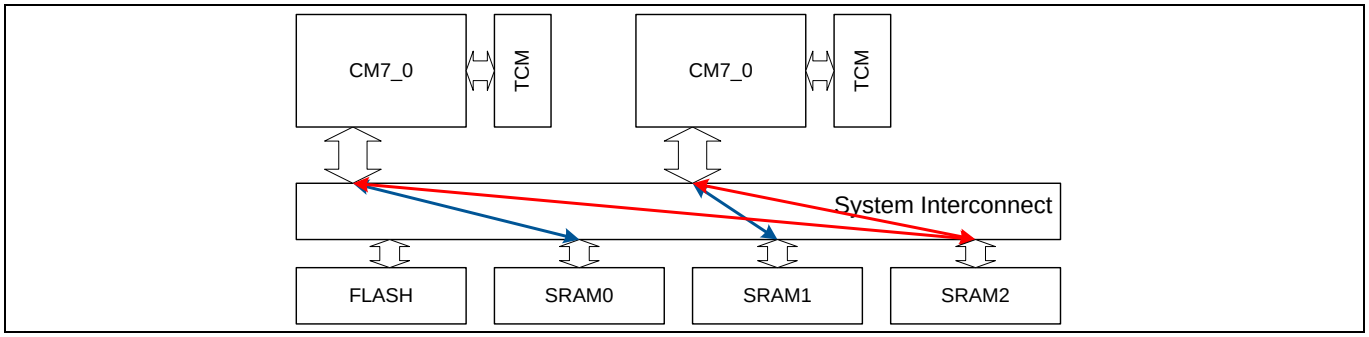

Figure 5 Memory Connection (CYT4BF Series)

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 Communicating between CPUs

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

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

IPC channel structure registers consist of IPC_STRUCTx_ACQUIRE, IPC_STRUCTx_NOTIFY, IPC_STRUCTx_RELEASE, two 32-bit IPC_STRUCTx_DATA0/1, and IPC_STRUCTx_LOCK_STATUS. The ACQUIRE register provides 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 registers can pass a message up to 64 bits.

Note: *Some IPCs are reserved by SROM API, and you cannot use structures of IPC channel and interrupt reserved by the SROM API. See the **Device Datasheet** for more information.*

4.1 CPU Synchronization

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

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

IPC has two solutions for this issue. One solution is to use the IPC_STRUCTx_DATA0/1 registers. Another solution is to use the IPC_STRUCTx_NOTIFY register. The solution using IPC_STRUCTx_DATA0/1 registers 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 Register List of Notify Event

Structure	Register Name	Bit Name	Description
IPCx channel	IPC_STRUCTx_NOTIFY	INTR_NOTTIFY[15:0]	This field allows for the generation of notification events to the IPC interrupt structures. SW always reads a '0' from this field.
	IPC_STRUCTx_RELEASE	INTR_RELEASE[15:0]	This field allows for the generation of release events to the IPC interrupt structures, but only when the lock is acquired. SW always reads a '0' from this field.

Structure	Register Name	Bit Name	Description
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.

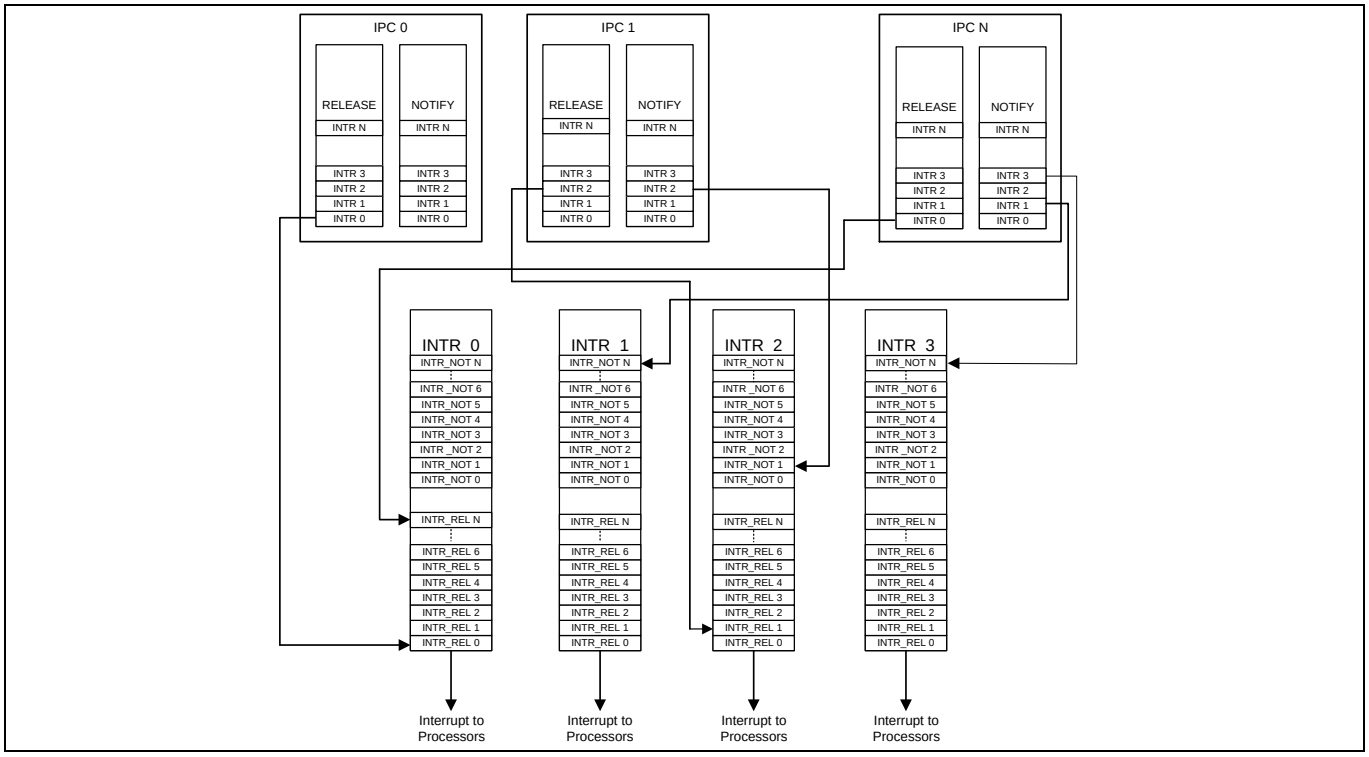

Figure 6 IPC Channel Structures and Interrupt Structures

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.

4.1.1 Implementation Example Operation of Synchronization between CPUs

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

4.1.2 Use case

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

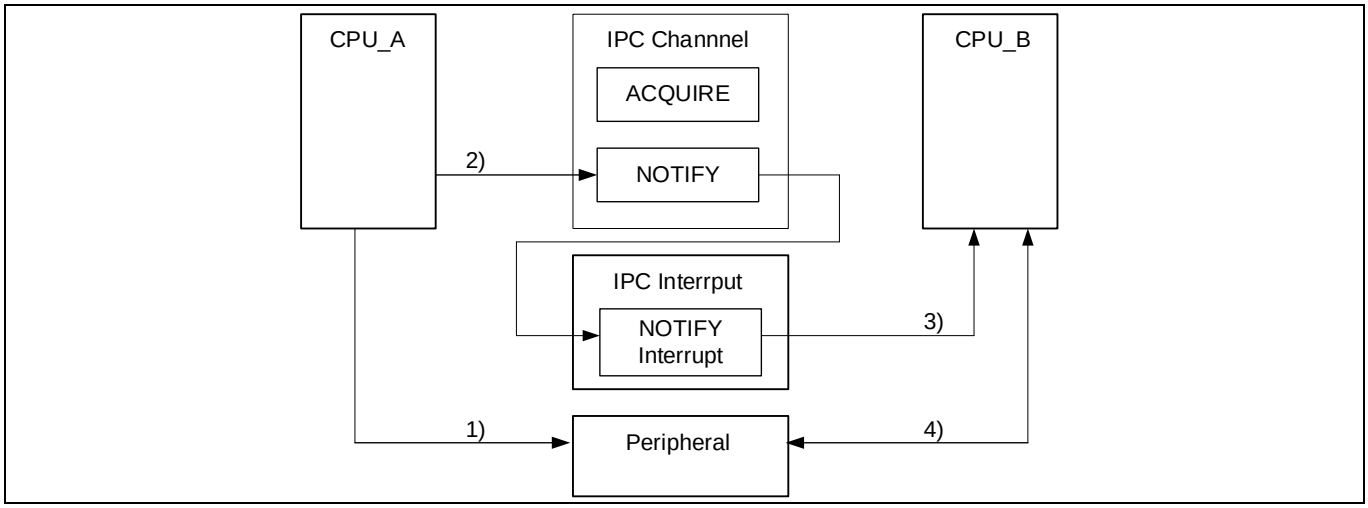

Figure 7 CPU Synchronization using IPC

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

Figure 8 shows the flow.

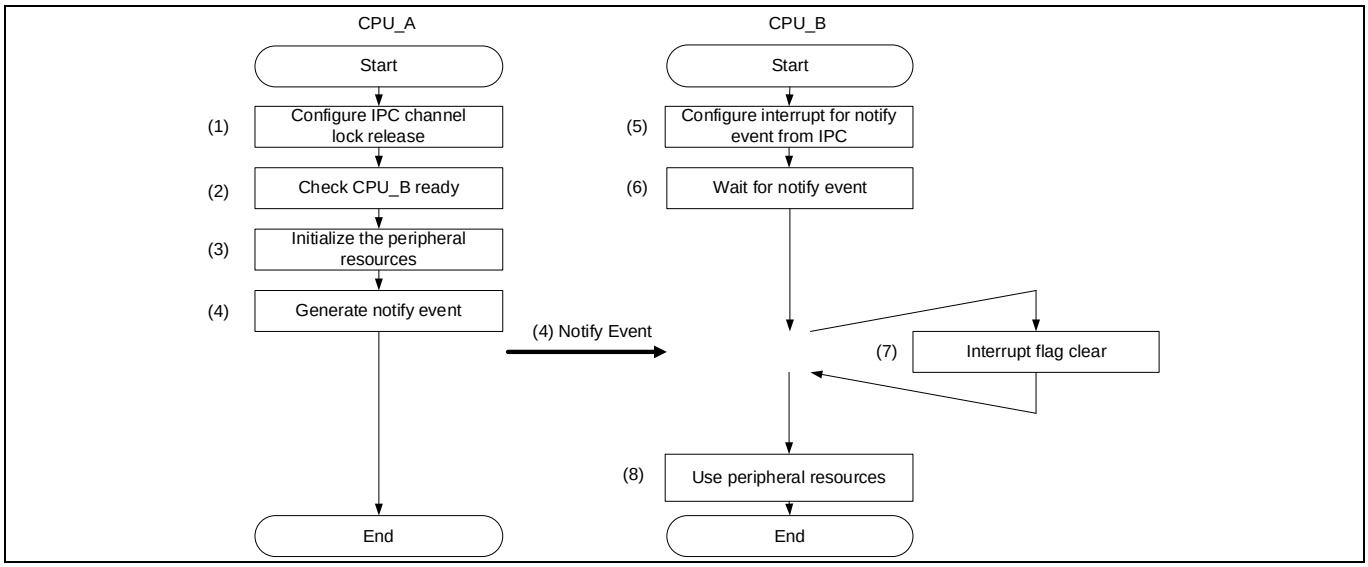

Figure 8 CPU Synchronization Operation

The following is the structure of the sample code.

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

See the **Architecture TRM** and **AN219842 - How to Use Interrupt in Traveo II** for Interrupt configuration details.

4.1.3 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 List of Parameters

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

Table 5 List of Functions

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

Communicating between CPUs

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

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

- 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;
```

Communicating between CPUs

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**. (Get CM0+ configuration status)

(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

4.2 Mutual Exclusion Operation

This section describes how to mutually exclude shared resource access between CPUs using IPC. In a multi-CPU architecture, each CPU 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 memory. CPU_A is supposed to read and update memory data. Then, CPU_B is supposed to read and update the same memory data, but only after CPU A completes the operation. However, if CPU_A reads memory data, but CPU_B updates memory data before CPU_A updates memory data, there will be a mismatch between the actual memory data and the expected memory data, since CPU_B is supposed to update the data written by CPU_A.

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

IPC in Traveo II can easily implement exclusive access using the IPC_STRUCTx_ACQUIRE register. This register has a lock feature of IPC channel structure. A lock of the IPC channel structure is acquired by reading this register.
Table 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 IPC channel structure is provided by the IPC_STRUCTx_LOCK_STATUS register. The acquired state of IPC channel structure is released by writing any value into the IPC_STRUCTx_RELEASE register, and allows for the generation of release events to the IPC interrupt structure.

4.2.1 Implementation Example of Mutual Exclusion

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

4.2.2 Use case

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

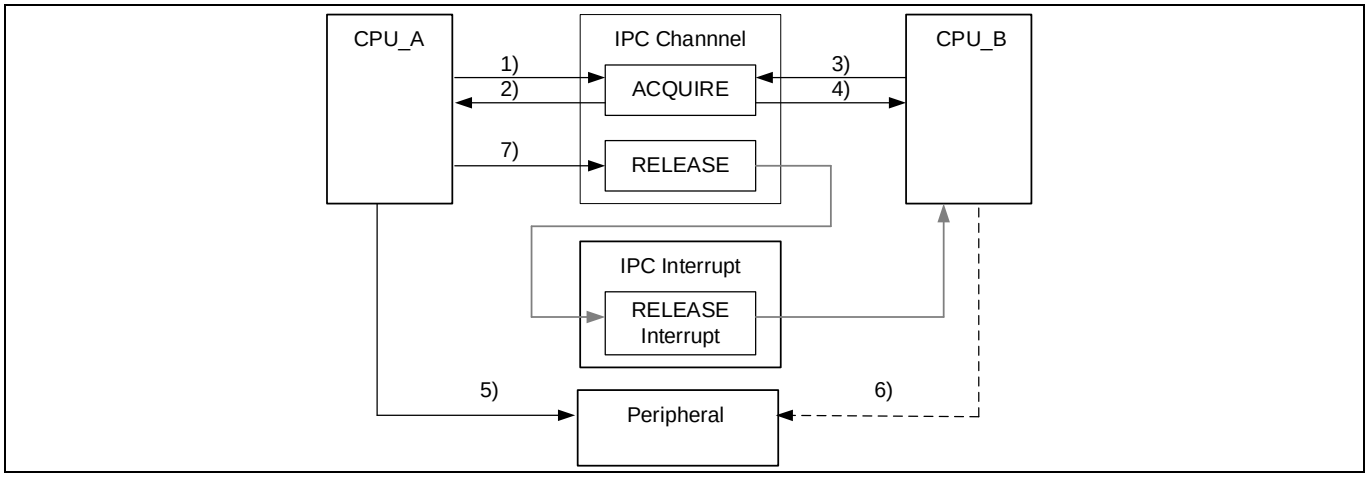

Figure 9 Example of Exclusive Access

The following shows an example of exclusive control implementation:

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

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

Figure 10 shows the example flow for mutual exclusion.

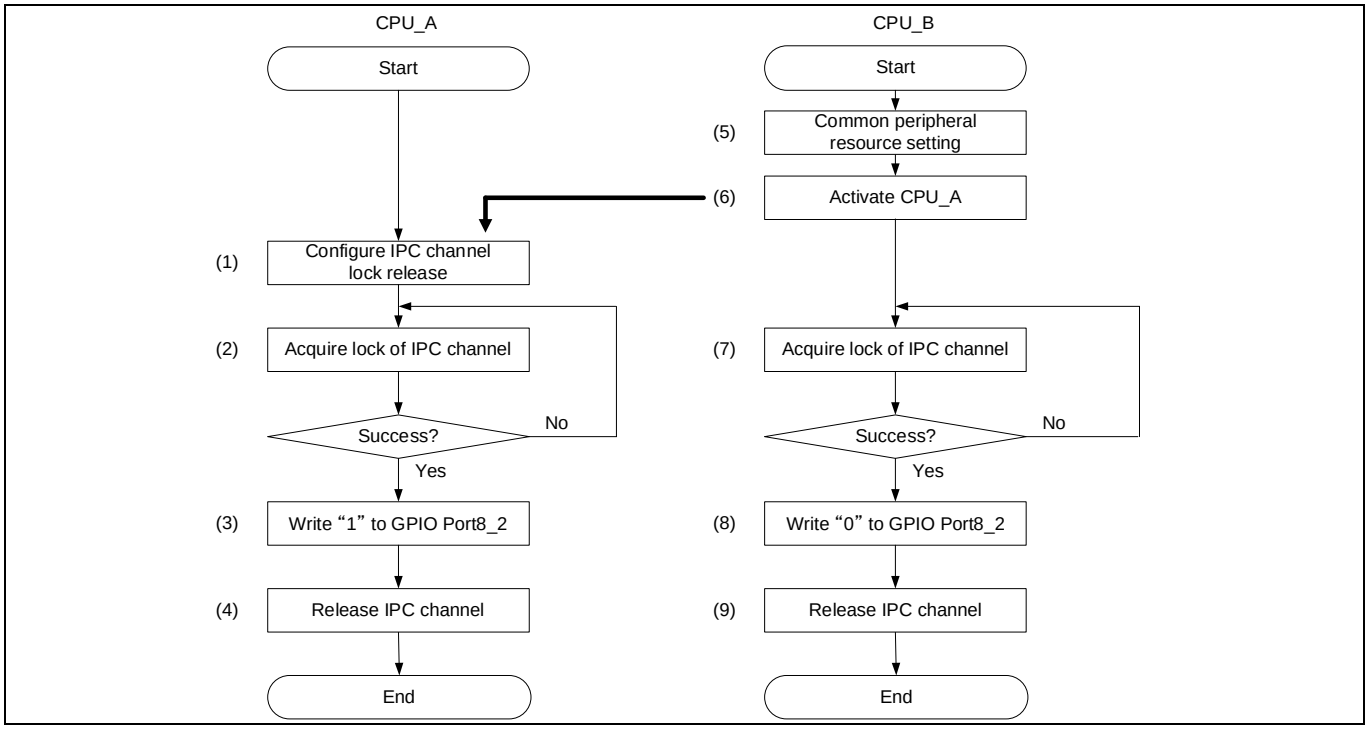

Figure 10 Mutual Exclusion Flow

The following shows the structure of the sample code.

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

See the **Architecture TRM** and **AN220193 - GPIO USAGE SETUP IN TRAVEO II FAMILY** for GPIO configuration details.

4.2.3 Configuration

Table 7 and **Table 8** list the parameters and functions in SDL for mutual exclusion 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 7 List of Parameters

Parameters	Description	Value
IPC_CHANNEL_NUMBER	Define using IPC channel structure number	6ul (IPC6 channel structure)
CY_CB_LED_PORT	Define IO Port number	GPIO_PRT8 (Assign to Port 8)
CY_CB_LED_PIN	Define IO pin	2ul
CY_IPC_NO_NOTIFICATION	Defines a value to indicate that no notification events are needed.	0x00000000ul
user_led_port_pin_cfg.xxxx	IO port configuration See **AN220193 - GPIO USAGE SETUP IN TRAVEO II FAMILY** for GPIO configuration details.	-

Communicating between CPUs

Table 8 List of Functions

Function	Description	Remark
`Cy_IPC_Drv_LockAcquire(base)`	Acquire 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);

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

            break;
        }
    }

    for(;;)
    {
    }
}
```

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.

(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,
```

Define IPC channel number

Configure IO Port.

```
    .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(;;)
    {
    }
}
```

(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**.

4.3 Data Passing

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

4.3.1 Implementation Example of Passing Small Data (Up to 64 Bits)

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

4.3.2 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.

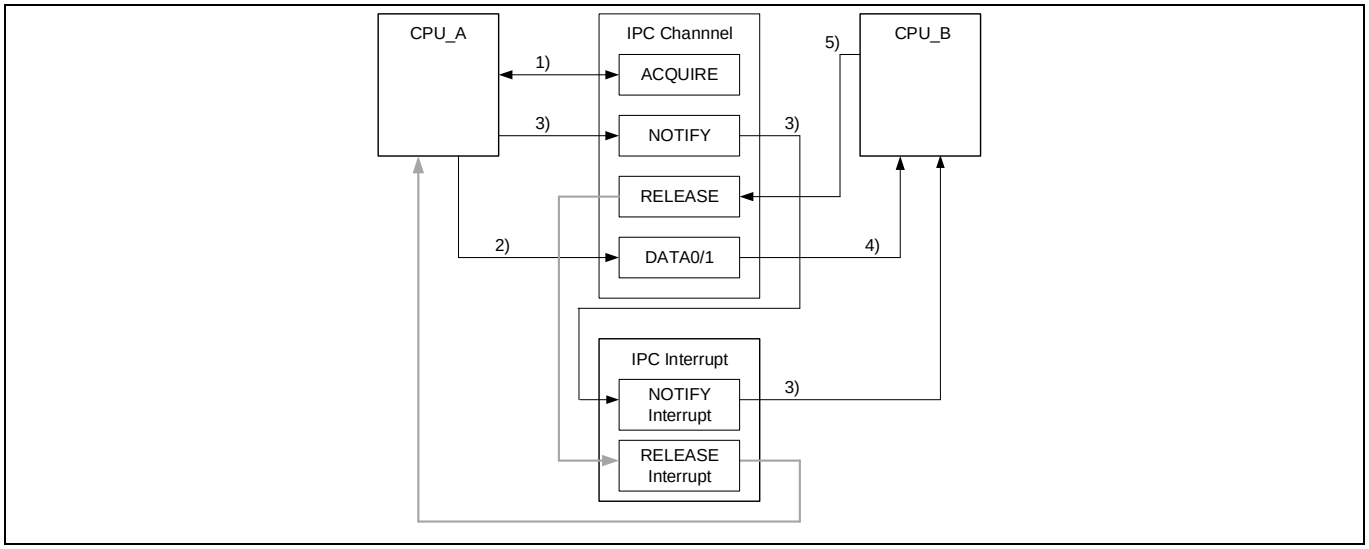

Figure 11 Example of Passing Small Message

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

1. CPU_A reads the IPC_STRUCTx_ACQUIRE register. When CPU_A reads “1” from the IPC_STRUCTx_ACQUIRE register, CPU_A is successful in acquiring the IPC channel structure lock.
2. After the IPC channel structure is locked, CPU_A places message data up to 64 bits in the IPC_STRUCTx_DATA0/1 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).

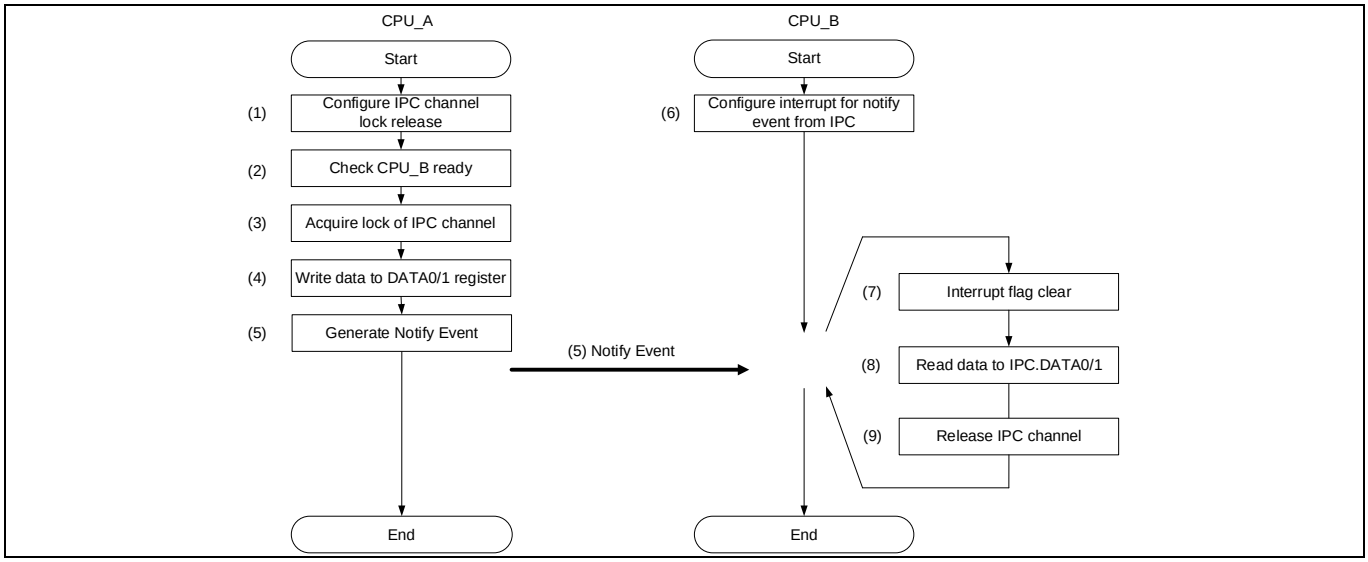

Figure 12 Data Passing (Up to 64 bits) flow

The following shows the structure of the sample code.

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

See the **Architecture TRM** and **AN219842 - How to Use Interrupt in Traveo II** for Interrupt configuration details.

4.3.3 Configuration

Table 9 and **Table 10** list the parameters and functions in SDL for data passing of 64 bits or less 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 9 List of Parameters

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

Communicating between CPUs

Table 10 List of Functions

Functions	Description	Remarks
`Cy_IPC_Drv_SendMsgWord_2(base, notifyEventIntr, message, message2)`	Set DATA0/1 register of IPC channel structure base: Base address of IPC channel to operate notifyEventIntr: Value of notify event setting message: Write data to IPC.DATA0 message2: Write data to IPC.DATA1	It has function of acquire lock and notify event generation.
`Cy_IPC_Drv_WriteDataValue(base, dataValue)`	Write data to DATA0 register base: Base address of IPC channel to operate dataValue: Write data	-
`Cy_IPC_Drv_WriteData1Value(base, dataValue)`	Write data to DATA1 register base: Base address of IPC channel to operate dataValue: Write data	-
`Cy_IPC_Drv_ReadMsgWord_2(base, message, message2)`	Read DATA0/1 register of IPC channel structure base: Base address of IPC channel to operate message: Stored address for IPC.DATA0 message2: Stored address for IPC.DATA0	-
`Cy_IPC_Drv_ReadDataValue(base)`	Read DATA0 register base: Base address of IPC channel to operate	-
`Cy_IPC_Drv_ReadData1Value(base)`	Read DATA1 register base: Base address of 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 value of IPC.INTR_MASK. See **Code Listing 6**.

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

Get value of notify mask. See **Code Listing 7**. (Get CM0+ configuration status)

(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 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 number of IPC interrupt for notify event.

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

4.3.4 Implementation Example of Passing Large Data (More than 64 Bits)

This section describes passing of 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.

4.3.5 Use Case

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

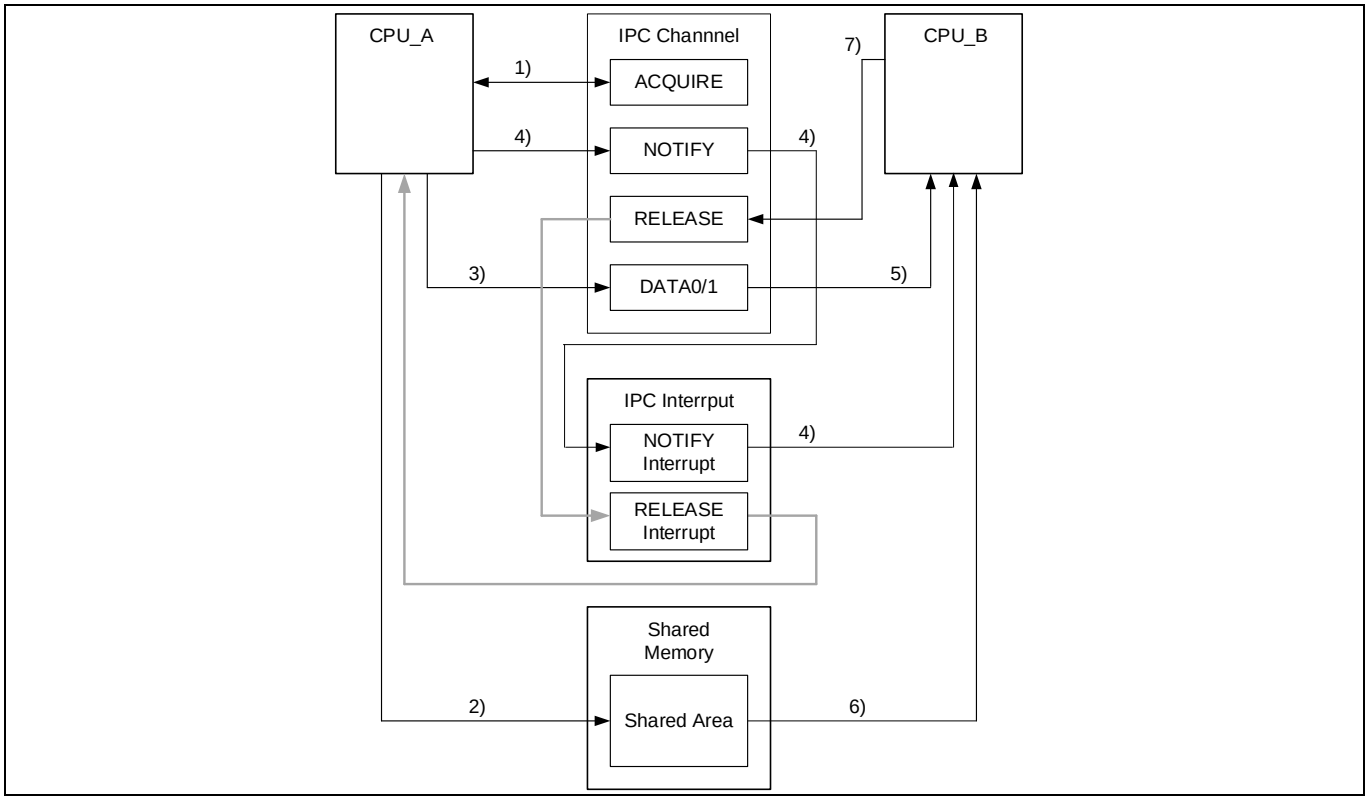

Figure 13 Example of Passing Large Message

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

1. CPU_A reads the IPC_STRUCTx_ACQUIRE register. When CPU_A reads “1” from the IPC_STRUCTx_ACQUIRE register, CPU_A is successful in acquiring the IPC channel structure lock.
2. After the IPC channel structure is locked, CPU_A places message data in the shared memory.
3. Then, CPU_A places 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 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).

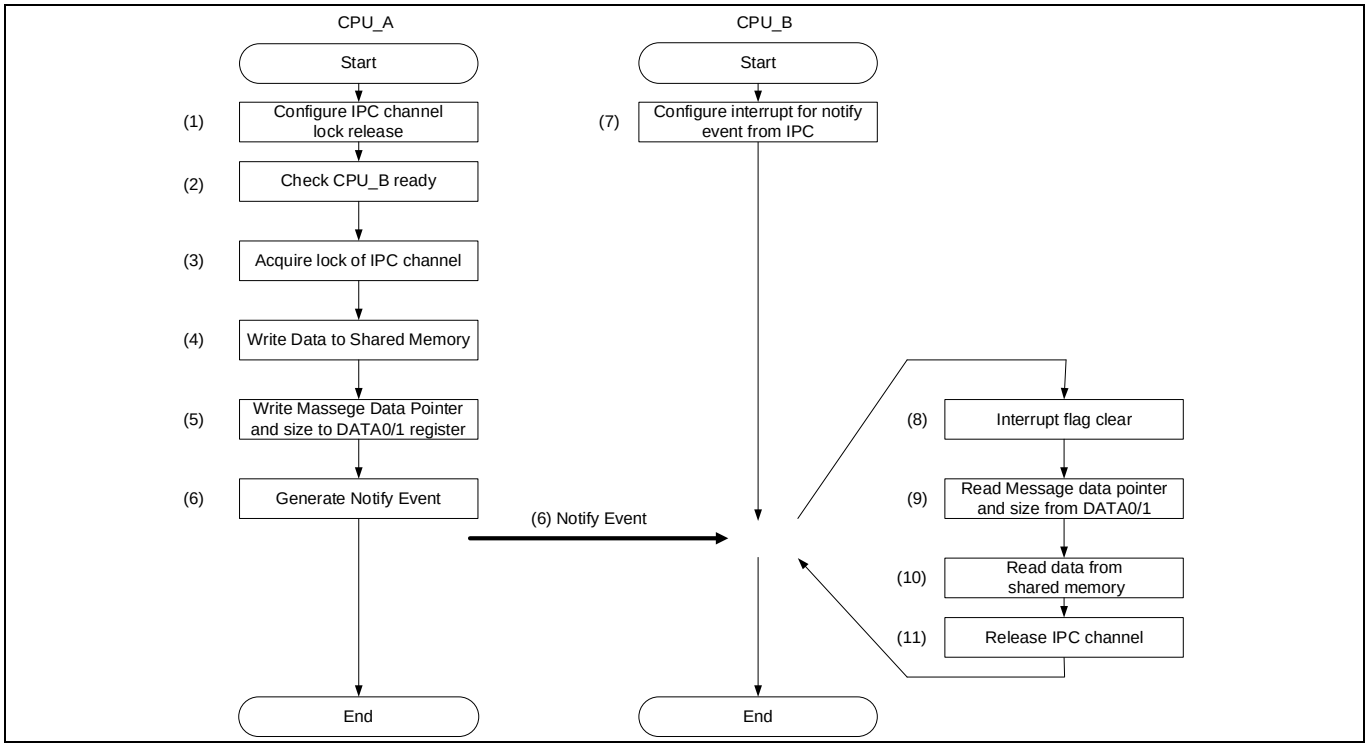

Figure 14 Data Passing (More than 64 bits) flow

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)

4.3.6 Configuration

Table 11 lists the parameters and functions in 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 List of Parameters

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

Communicating between CPUs

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**. (Get CM0+ configuration status)

(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**.

5 Consideration for Cache Coherency Issue

A cache memory helps to improve CPU performance from its high-speed read/write operation. However, the characteristics of cache memory may cause a data mismatch between cache memory and other memories, that is, cache coherency issue. Cache coherency issue should be mainly considered in CYT4B and CYT4D series which have cache memory in CPU. This section provides an overview of cache memory in these series and explains cache coherency issue under different scenarios. In addition, it provides methods to manage or avoid the cache coherency issue. In this section, the shared memory referred to is SRAM unless otherwise specified.

5.1 Cache Coherency

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

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

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

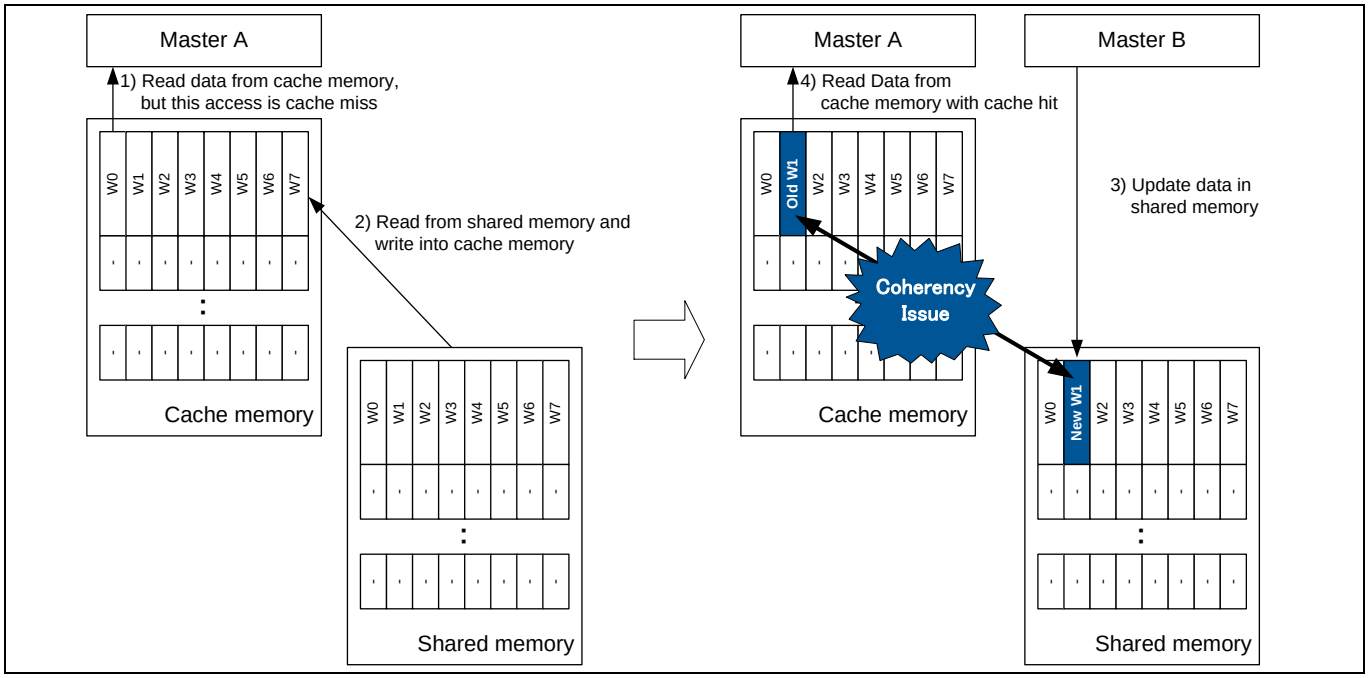

Figure 15 Coherency Issue Example

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

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

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

5.2 Cache Memory Overview

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

5.2.1 Cache Memory Placement

Figure 16 shows the placement of cache memory.

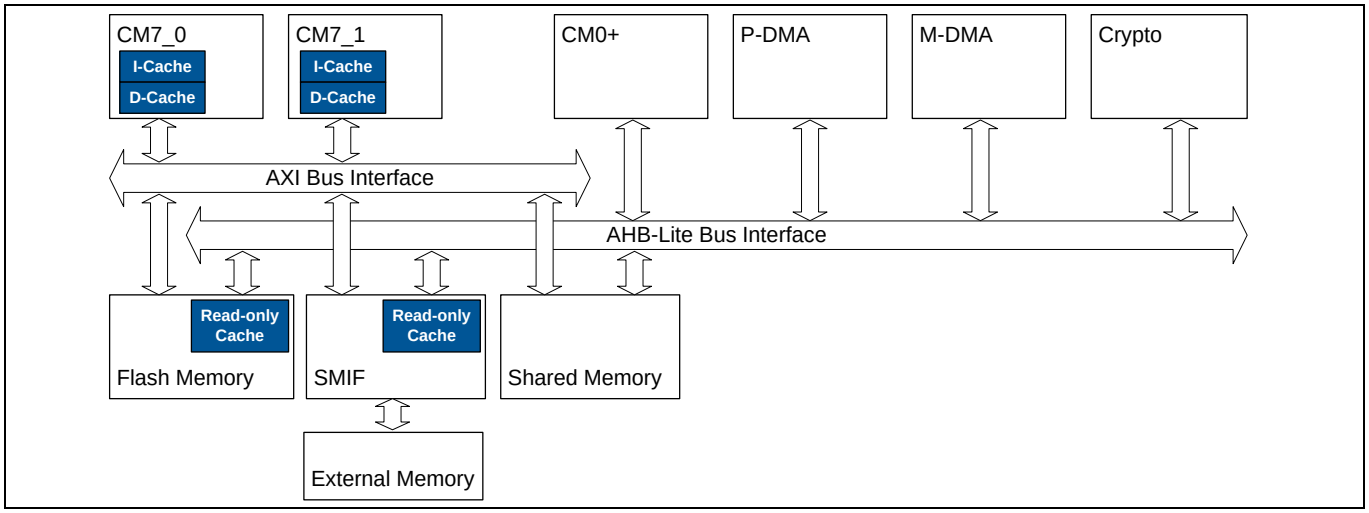

Figure 16 Cache Memory Placement

In these series, CM7 CPUs have I-cache and D-cache, and Flash Memory and Serial memory interfaces (SMIF) have read-only cache memory for AHB-Lite Bus Interface.

5.2.2 I-Cache and D-Cache Operation

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

5.2.2.1 Cache Memory Behavior

When CPU finds data in the cache memory, that is, a cache hit, the data is read from the cache memory or written into the cache memory. **Table 12** lists the behavior of cache memory in CM7. This operation assumes that shared memory is allocated to 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. Cache memory allocates a memory location to a cache line. When a cache line is allocated, the shared memory data is fetched and written to the cache memory. Then, read access to these memory locations will be a cache hit, and data is read from the cache memory.
Write Access	Cache hit	Write-Back	The access is written into the cache memory. The cache line is marked as dirty, and the data in the cache memory is only written to the shared memory when the line is evicted.
		Write-Through	The access is written into the cache memory. The data is also written to the shared RAM, so that the data stored in the cache memory is coherent with the shared memory.
	Cache miss	Write Allocate	Cache memory allocates a memory location to a cache line. When a cache line is allocated, the shared memory data is fetched, and written to the cache memory.
		No Write Allocate	Cache memory does not allocate a memory location to a cache line. The data is written to shared memory.

5.2.2.2 Cache Memory Configuration

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

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

Consideration for Cache Coherency Issue

These configurations are available in MPU Region Attribute and Size Register (MPU_RASR). **Table 13** shows MPU_RASR common combination for cache configuration. The configuration of 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.

5.2.2.3 Cache Maintenance Operation

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

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

To perform these cache maintenances, you can use the Cortex Microcontroller Software Interface Standard (CMSIS). **Table 14** lists 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
`SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Invalidates D-cache by address addr: Address aligned to 32-byte boundary dsaize: 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 dsaize: Size of the memory block in bytes
`SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Cleans and invalidates D-cache by address

Consideration for Cache Coherency Issue

Cache Maintenance APIs	Description
	addr: Address aligned to 32-byte boundary dsaize: 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. Subsequent access is cache miss.

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

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.

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

5.2.3 Cache Memory Operation in Flash Memory

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

Table 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 Flash memory, and 16-Bytes data are refilled from Flash memory to cache memory. Subsequent access result is cache hit.
Write Access		The write access is bypass the cache memory. In Flash memory, the write access without specific sequence is generally causes access error.

In general, flash memory does not rewrite as frequently as RAM. Also, flash memory is most often written under specific conditions according to system requirement. Therefore, 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. Cache memory can be enabled/disabled using register. When cache memory is set to disable and enable again, data in cache memory is invalidated, and read access causes refilling in 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)

5.2.4 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 external memory, and 16 Bytes data are refilled from external memory to cache memory. Subsequent access results in cache hit.
Write Access		The write access bypasses the cache memory. The data is directly written into external memory. A write to an address in the read-only cache invalidates the associated cache subsector.

Consideration for Cache Coherency Issue

SMIF has three interfaces: XIP AXI, XIP AHB-Lite, and MMIO AHB-Lite interface. Out of the three interfaces, only the XIP AHB-Lite interface has cache memory. In addition, this cache memory does not support cache coherency by hardware. Therefore, SMIF has cache coherency issue depending on access between each port. **Table 18** lists the control registers for invalidating and enabling/disabling of 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', the SMIF can be safely disabled or the mode of operation can be safely changed.
SMIF_SLOW_CA_CMD	INV	Cache and prefetch buffer invalidation. Software writes a '1' to clear the cache and prefetch buffer. The cache's LRU structure is also reset to its default state. Note that the software should invalidate the cache and prefetch buffer only when SMIF_STATUS.BUSY is '0'.
SMIF_SLOW_CA_CTL	PREF_EN	Prefetch enable: '0': Disabled '1': Enabled (Default) Prefetching requires the cache to be enabled; ENABLED is '1'.
	ENABLED	Cache enable: '0': Disabled '1': Enabled (Default)

5.3 Cache Coherency Handling

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

5.3.1 Cache Disable

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

5.3.2 Cache Invalidate

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

5.3.3 Cache Clean

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

5.3.4 Cache Configuration Sets to Write-through

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

5.3.5 Use TCM as Shared Memory

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

5.4 Cache Coherency Issue Scenarios

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

5.4.1 Cache Coherency Issue between CM7 CPUs

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

5.4.1.1 Scenario and Solution between CM7 CPUs

CM7 has I-cache and D-cache. Cache coherency issue mainly occurs with D-cache that handles data. **Figure 17** shows 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 common area enables a cache.
- Each CPU cache configuration is Write-back, write, and read allocate.
- Data is sent from CM7_1 to CM7_0. That is, CM7_1 writes the data and CM7_0 reads the data.

Consideration for Cache Coherency Issue

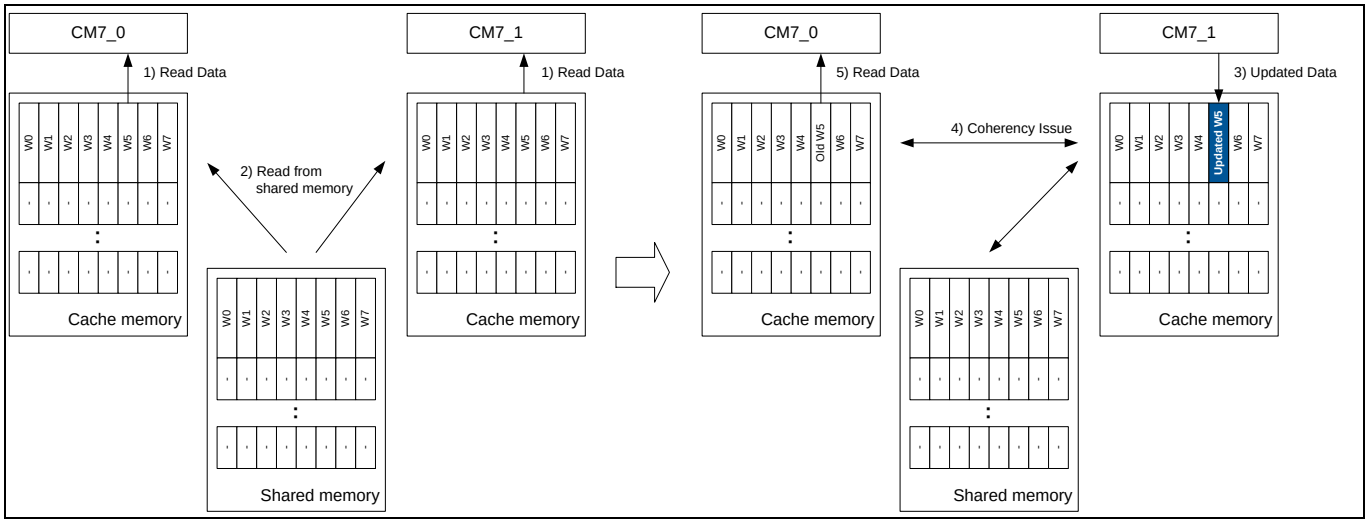

Figure 17 Scenario between CM7 CPUs

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

Here are some solutions for this scenario between CM7 CPUs:

- Solution 1: Disable cache

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

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

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

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

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

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

5.4.2 Cache Coherency Issue between CM7 CPU and Other Masters

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

5.4.2.1 Scenario and Solution for CM7 CPU Read and Other Master Write

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

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

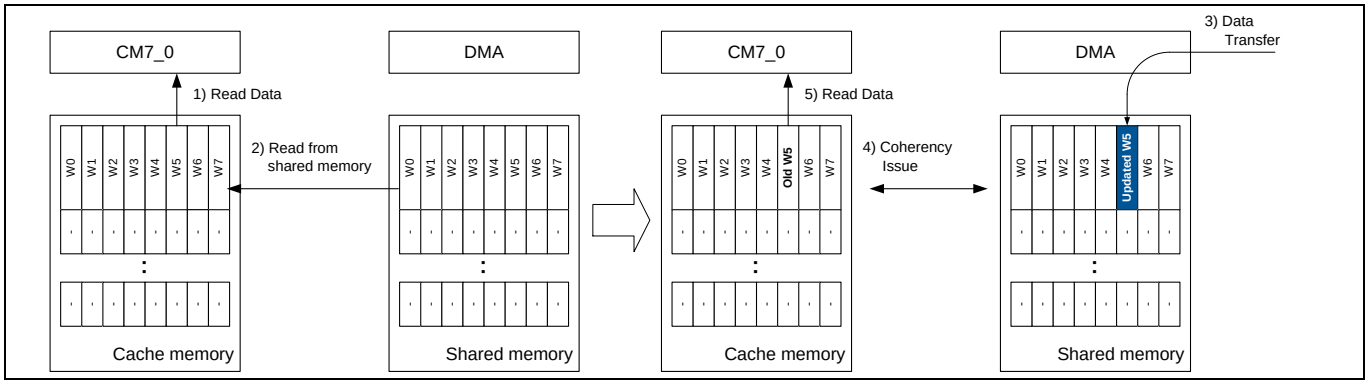

Figure 18 Scenario between CM7 CPU and Other Master (CM7_0 Reads, DMA Writes)

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

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

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

- Solution 1: Disable cache

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

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

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

5.4.2.2 Scenario and Solution for CM7 CPU Write and Other Master Read

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

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

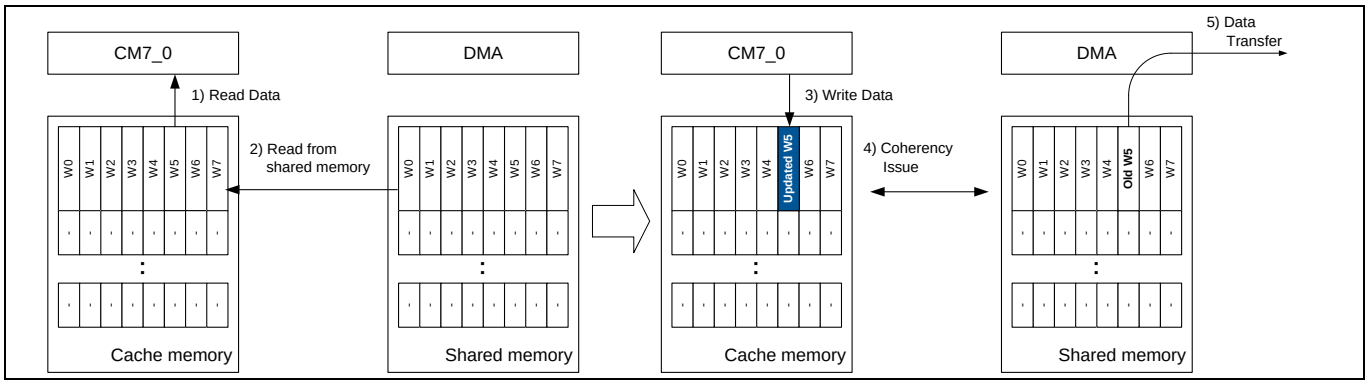

Figure 19 Scenario between CM7 CPU and Other Master (CM7_0 Writes, DMA Reads)

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

Consideration for Cache Coherency Issue

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

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

5.4.3 Cache Coherency Issue for Flash Memory Access

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

5.4.4 Cache Coherency Issue for SMIF Access

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

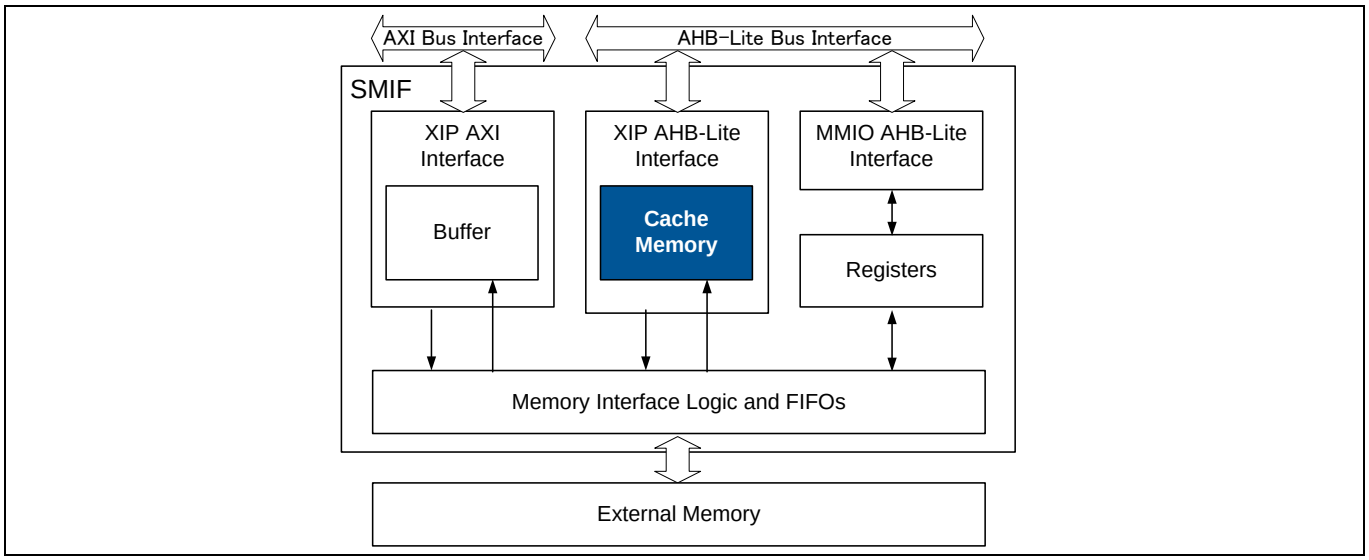

Figure 20 Block Diagram of SMIF Bus Interface

SMIF has three bus interfaces: XIP AXI, XIP AHB-Lite, and MMIO AHI-Lite. The XIP AXI interface is used by CM7 to access external memory in XIP mode. The XIP AHB-Lite interface is used by masters except CM7 to access external memory in XIP mode. The MMIO AHB-Lite interface is used by all master to access external memory in MMIO mode. See the **Architecture TRM** for XIP mode, MMIP mode, and each interface details.

Consideration for Cache Coherency Issue

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

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

5.4.4.1 Scenario and Solution for CM7 Access

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

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

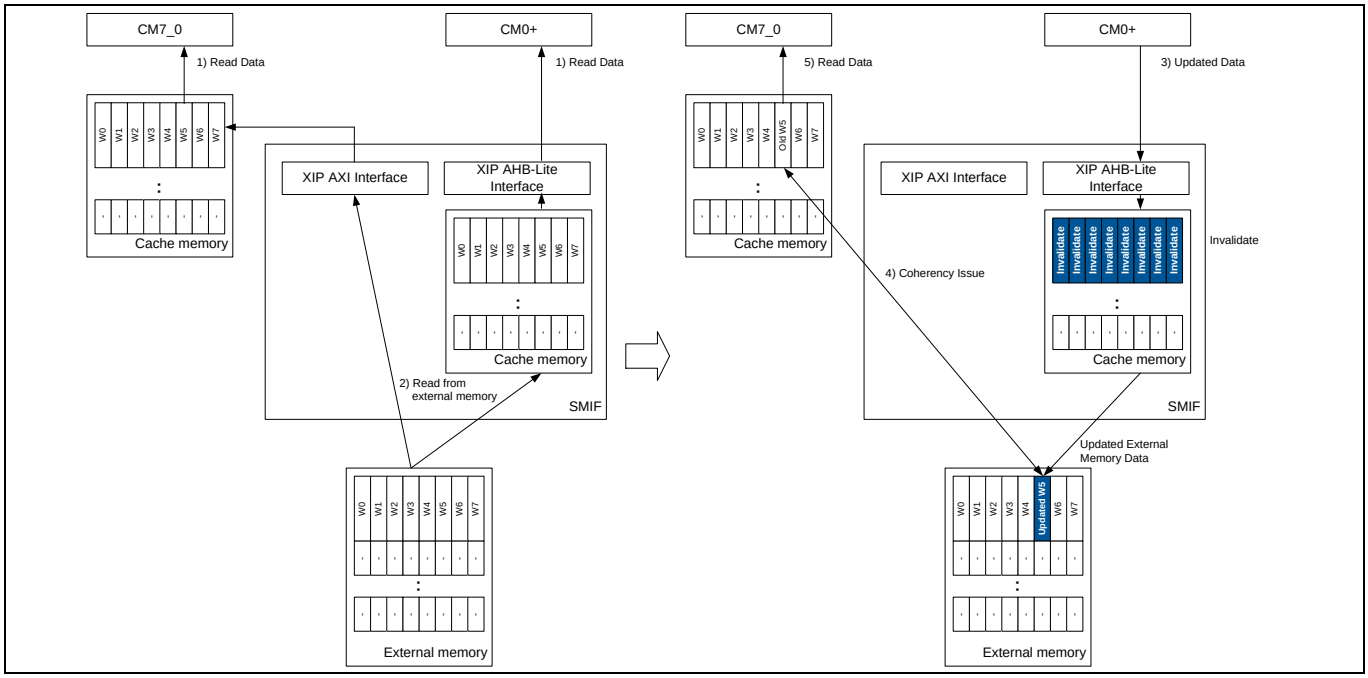

Figure 21 Scenario between CM7 and CM0+ (CM7_0 Reads, CM0+ Writes)

1. CM7_0 and CM0+ try to read data from the cache memory. However, the cache memory does not have data, thus, it is a cache miss.
2. As a result of read access, the cache memories refill the data from the external memory. The cache memories data and the external memory data are same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM0+ updates W5. As a result of write access, W5 in the external memory is updated, and the associated cache subsector is invalidated. Subsequent access to this data results in a cache miss, and cache memory refills the data from the external memory again.

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

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

- Solution 1: Disable cache

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

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

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

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

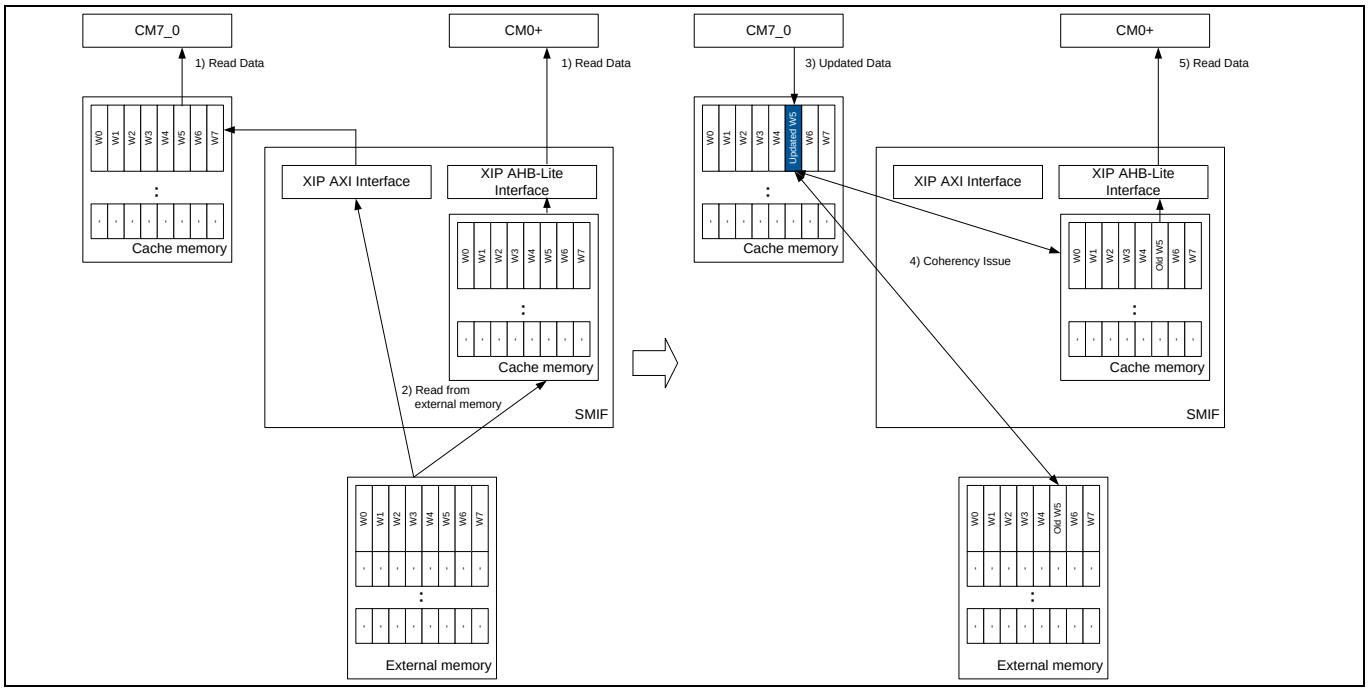

Figure 22 Scenario between CM7 and CM0+ (CM7_0 Writes, CM0+ Reads)

1. CM7_0 and CM0+ try to read data from the cache memory. However, the cache memory does not have data, thus, it is a cache miss.
2. As a result of read access, the cache memories refill the data form the external memory. The cache memories data and the external memory data are same at this point. Therefore, they are coherent. Subsequent access results in a cache hit.
3. CM7_0 updates W5 data in its own cache memory according to cache configuration, but this write access does not update external memory immediately because of Write-back.

4. W5 (Updated W5) in the CM7_0 cache memory is different from W5 (Old W5) in the cache memory in SMIF and external memory. That is, this has cache coherency issue.
5. CM0+ reads old W5 from the cache memory. As a result, CM0+ can cause unintended operation.

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

- Solution 1: Disable cache

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

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

5.4.5 Cache Coherency Issue for Using SROM APIs

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

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

5.4.5.1 Scenario and Solution when Using SROM API (CM0+ API Parameter Read)

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

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

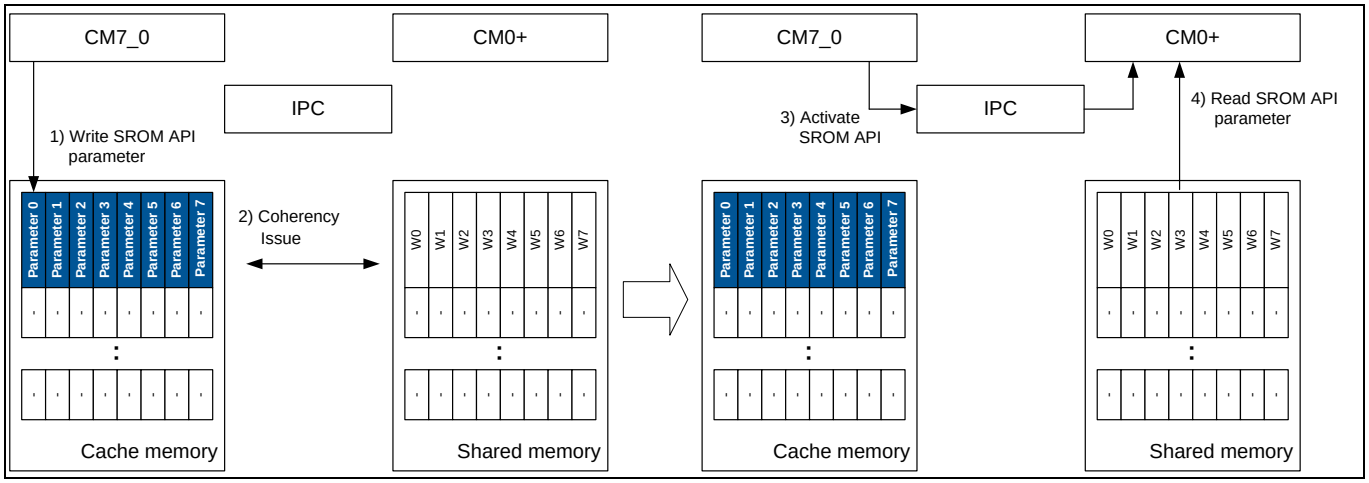

Figure 23 Scenario CM0+ SROM API Parameter Read

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

Here are some solutions for the scenario:

- Solution 1: Disable cache

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

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

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

5.4.5.2 Scenario and Solution when Used SROM API (CM7 Execution Result Read)

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

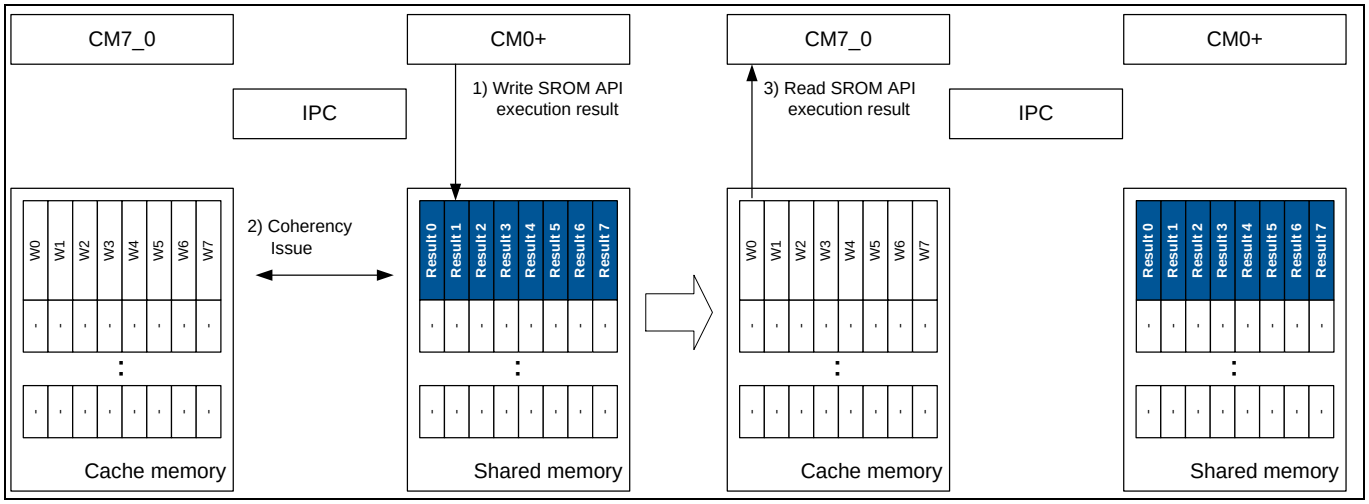

Figure 24 Scenario CM7 SROM API Parameter Read

The preconditions are as follows:

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

Here are some solutions for the scenario:

- Solution 1: Disable cache

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

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

6 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

Related Documents

The following are the Traveo II 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™ II Family
- CYT2B9 Datasheet 32-Bit Arm® Cortex®-M4F Microcontroller Traveo™ II Family
- CYT4BF Datasheet 32-Bit Arm® Cortex®-M7 Microcontroller Traveo™ II Family
- CYT4DN Datasheet 32-Bit Arm® Cortex®-M7 Microcontroller Traveo™ II Family

[2] CYT2B Series

- Traveo™ II Automotive Body Controller Entry Family Architecture Technical Reference Manual (TRM)
- Traveo™ II Automotive Body Controller Entry Registers Technical Reference Manual (TRM) for CYT2B7
- Traveo™ II Automotive Body Controller Entry Registers Technical Reference Manual (TRM) for CYT2B9

[3] CYT4B Series

- Traveo™ II Automotive Body Controller High Family Architecture Technical Reference Manual (TRM)
- Traveo™ II Automotive Body Controller High Registers Technical Reference Manual (TRM)

[4] CYT4D Series

- Traveo™ II Automotive Cluster 2D Family Architecture Technical Reference Manual (TRM)
- Traveo™ II Automotive Cluster 2D Registers Technical Reference Manual (TRM)

[5] Application Note

- AN219842 - How to Use Interrupt in Traveo II
- AN220208 - CLOCK CONFIGURATION SETUP IN TRAVEO II BODY ENTRY FAMILY
- AN224434 - CLOCK CONFIGURATION SETUP IN TRAVEO II FAMILY CYT4B SERIES
- AN226071 - CLOCK CONFIGURATION SETUP IN TRAVEO II FAMILY CYT4D SERIES
- AN229513 - CLOCK CONFIGURATION SETUP IN TRAVEO II FAMILY CYT2C SERIES
- AN220193 - GPIO USAGE SETUP IN TRAVEO II FAMILY

Other References

A Sample Driver Library (SDL) including startup as sample software to access various peripherals is provided. 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
**	12/12/2019	New Application Note.
*A	2021-02-02	Moved to Infineon Template Updated code examples using SDL

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

Edition <2021-02>

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. *A