XMC7000 MCU: Usage of Interrupts

About this document

Scope and purpose

This application note explains how to setup and use interrupts for the XMC7000 MCU family. This document serves as a guide for developing interrupt-based projects. It also describes the interrupt structure and vector selection and guides coding interrupt handlers.

Intended audience

This document is intended for anyone who uses the XMC7000 MCU to use the interrupt functions.

Associated part family

XMC7000 MCU family of XMC™ industrial microcontrollers.

Table of contents

1 Introduction

This application note describes interrupts in XMC7000 MCU family series. The series includes Arm® Cortex®-M CPUs with enhanced secure hardware extension (eSHE), CAN FD, memory, and analog and digital peripheral functions in a single chip.

Both the XMC7100 and XMC7200 series have two Arm® Cortex®-M7 CPUs and an Arm® Cortex®-M0+ CPU.

Interrupts are theimportant part of any embedded application. They free the CPU from continuously polling for a specific event; notifies the CPU only when that event occurs. An interrupt occurs in the current program flow being stopped and the interrupt service routine (ISR) being executed by the CPU.

In the embedded applications, interrupts are frequently used to communicate the status of the on-chip peripherals to the CPU. They are also used for notifying when errors are detected from the on-chip peripherals.

In this document, an interrupt from a peripheral is called a system interrupt. The series supports up to 1023 system interrupts. For the list of system interrupts supported by the device variants, see the device datasheet.

The fault report structures capture faults that occur while the software is running. The XMC7000 platform uses a centralized fault report structure, which allows for system-wide, consistent handling of faults and simplifies software development.

Fault report structures have a signaling interface to notify the system of the captured fault.

To understand the functionality described and terminology used in this application note, see the “Interrupts” and “Fault Subsystem” chapters of the architecture technical reference manual (TRM).

2 Interrupt overview

2.1 Interrupt structure

Figure 1 shows a simplified block diagram of the interrupt architecture in the XMC7200 MCU.

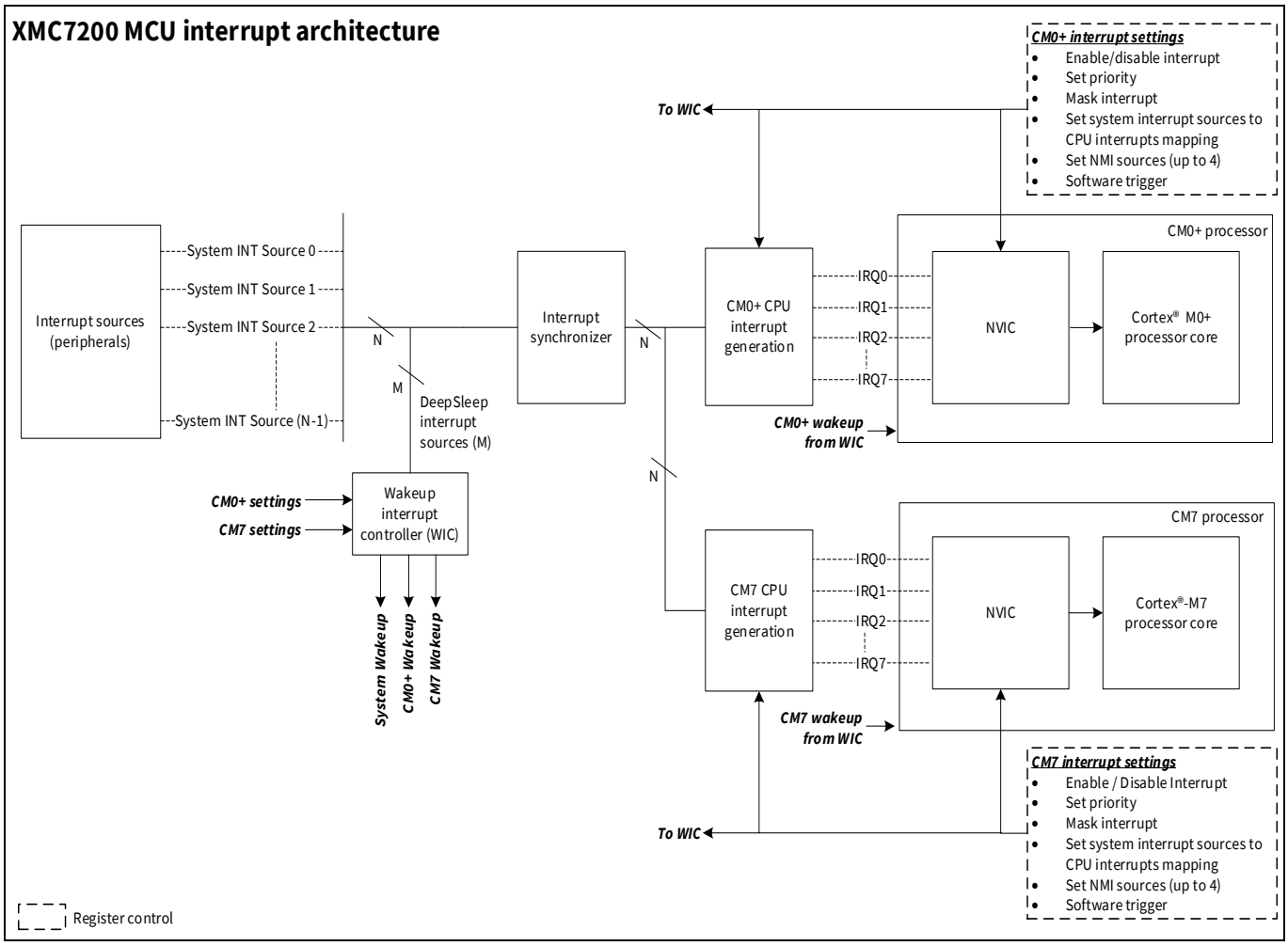

Figure 1 XMC7200 MCU interrupt architecture

See the architecture TRM for other series interrupt architecture.

The system interrupts of the series are processed by the NVIC of the individual cores.

In the XMC7000 MCU interrupt architecture, each CPU uses eight CPU interrupts IRQ[7:0], and any of the 'N' system interrupts are mapped to any of the IRQ[7:0] of each CPU. All the system interrupts are mapped onto any CPU interrupt simultaneously, and the interrupt is mapped independently for both CPUs.

The multiple systems interrupts are mapped on to the same CPU interrupt. Therefore, an active CPU interrupt indicates one or multiple active system interrupts.

For each interrupt, there are eight levels of configurable priority levels for CM7 and four levels for CM0+.

In addition to the NVIC, XMC7000 MCU supports a wakeup interrupt controller (WIC) and interrupt synchronizer block.

Interrupt overview

The WIC provides detection of Deep Sleep interrupts in the Deep Sleep CPU power mode. When an interrupt signal of one or more wakeup sources is detected, the WIC generates a wakeup signal and causes the CPU to enter Active mode. See the device datasheet for the interrupt sources available for Deep Sleep wakeup.

The Interrupt synchronizer block synchronizes the interrupts to the CPU clock frequency.

Up to four system interrupts can be mapped to the NMI for each CPU.

2.2 Operation overview

Figure 2 shows CPU operation in an interrupt sequence.

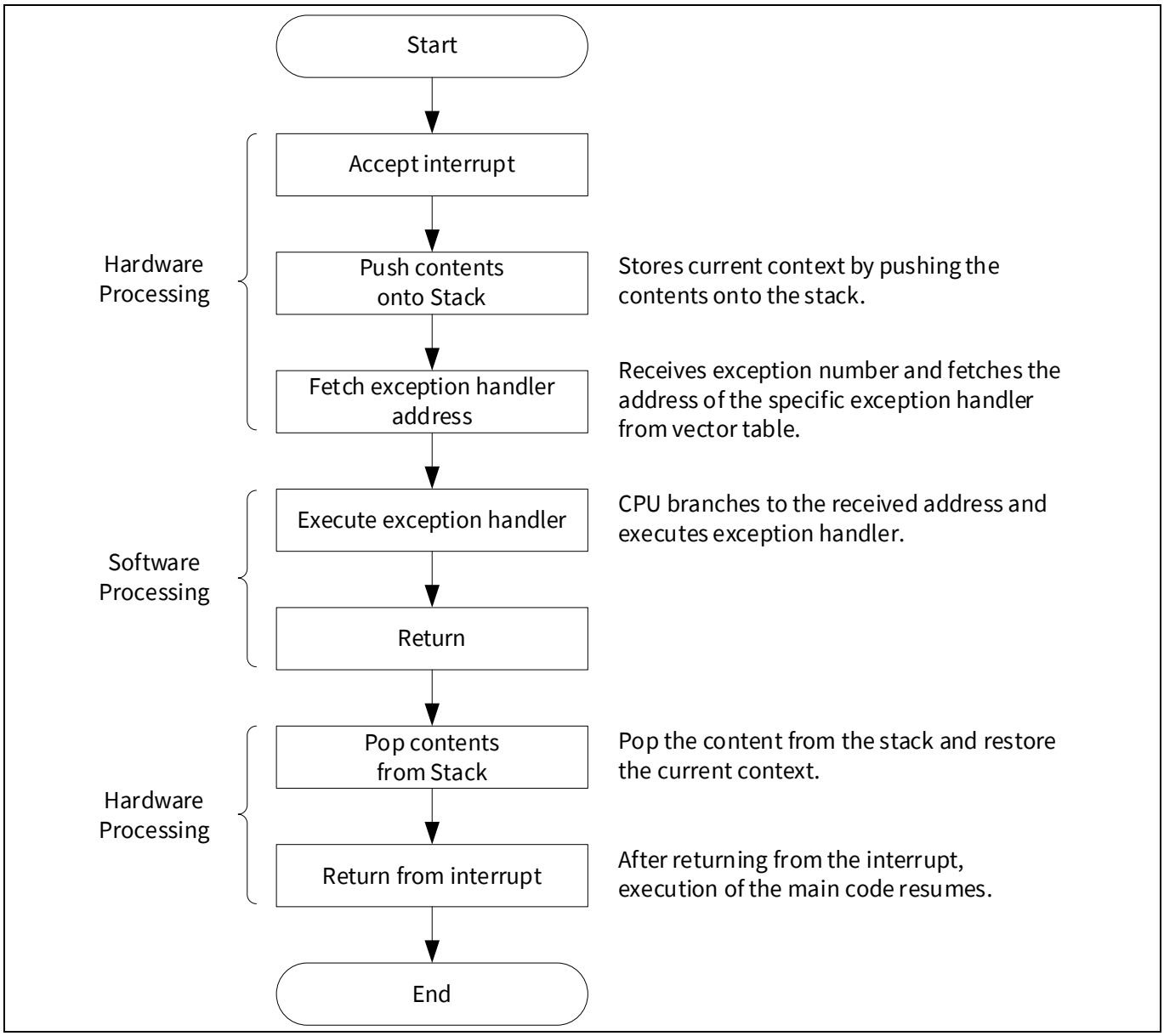

Figure 2 Interrupt sequence

When the NVIC receives an interrupt request while another interrupt is being serviced or receives multiple interrupt requests at the same time, it evaluates the priority of all the interrupts and sends the exception number of the highest-priority interrupt to the CPU. Therefore, a higher priority interrupt blocks the the processing of lower-priority interrupt.

Interrupt overview

Figure 3 shows the interrupt operation by interrupt priority.

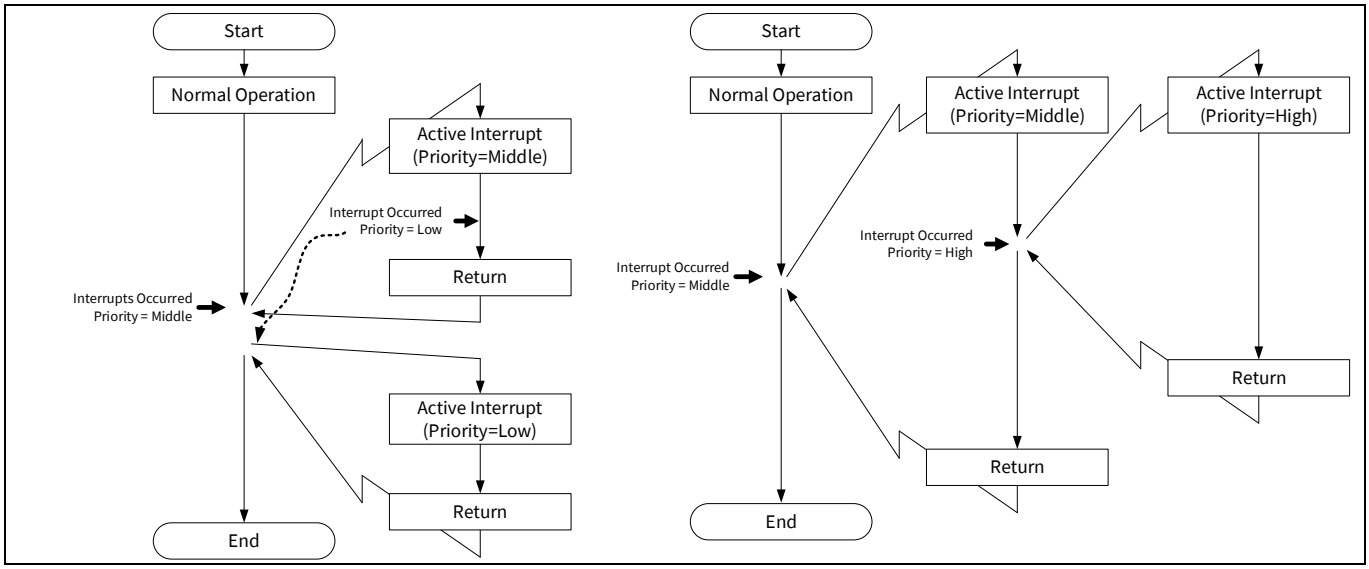

Figure 3 Operation when interrupt occurs during interrupt processing

For more details, see the Arm® documentation sets for CM7 and CM0+.

2.3 Initial setting

This section describes how to initialize interrupts using the peripheral driver library (PDL). The code snippets in this application note are part of PDL.

PDL code has the following parts:

- **The configuration part**: 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 4 shows the initializing interrupts.

Interrupt overview

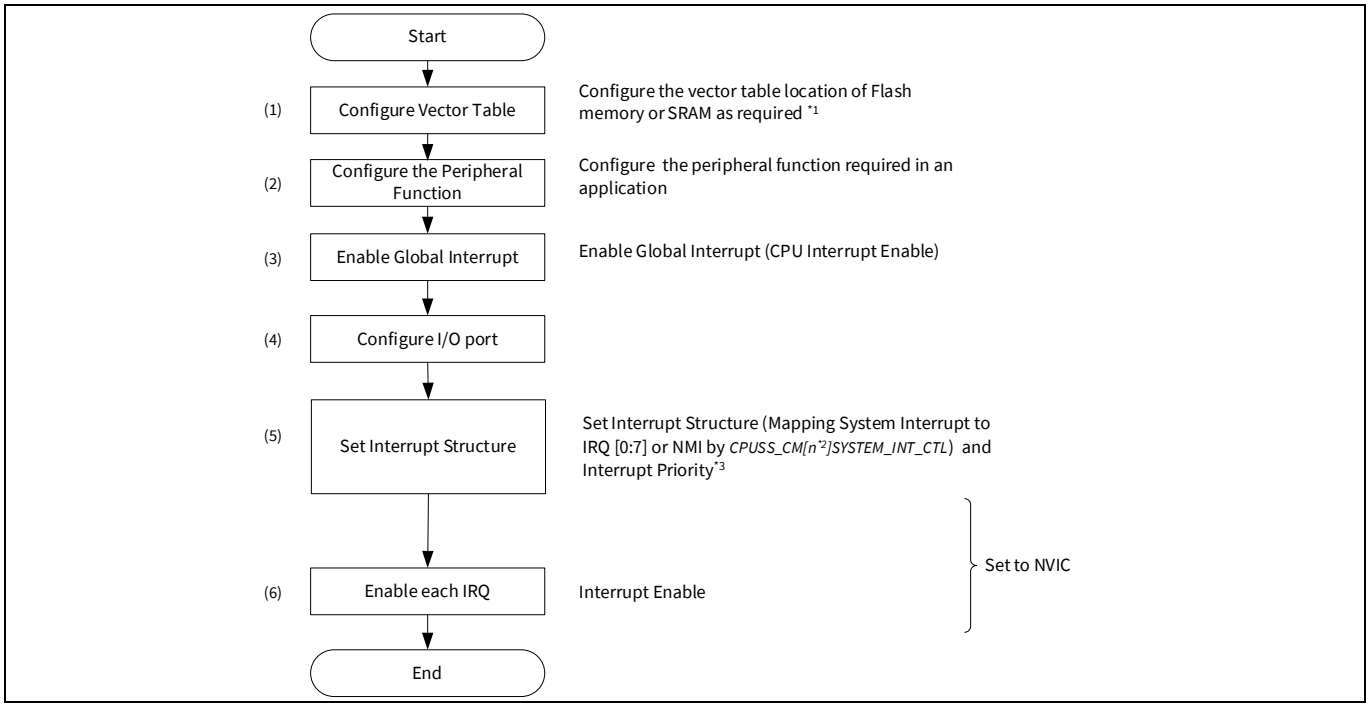

Figure 4 Interrupt initial setting flow

*Note: (*1) Vector table offset register (VTOR) is used in the vector table setting. Lower 7 bits in VTOR are reserved. Therefore, the vector table needs to be aligned on a 128-byte boundary.*

*Note: (*2) "n" in Figure 4 indicates 0, 7. For example, `CPUSS_CM0_SYSTEM_INT_CTL` shows system interrupt control register for CM0. The XMC7000 series MCUs have `CPUSS_CM7_0_SYSTEM_INT_CTL`, `CPUSS_CM7_1_SYSTEM_INT_CTL`, and `CPUSS_CM0_SYSTEM_INT_CTL` registers. See the registers TRM for more information.*

*Note: (*3) Interrupt priority can be set with the 8-bit `PRI_N` field in the `NVIC_IPR` register. The following equation can express the correspondence between interrupt numbers and register bit fields:*

$$IRQ\ number = NVIC\_IPR\ register\ number * 4 + PRI\_N\ bit\ field\ number$$

(for example, IRQ5 = NVIC_IPR1.PRI_N1)

Each CPU has a VTOR. It can be used for relocating the vector table from the flash to the SRAM, thus allowing the change of interrupt handlers dynamically. When VTOR is set to the SRAM area, the vector table must be placed in the SRAM.

Note: The software must ensure that the default hard fault vector entry is replaced with the specific user-handler. The default SROM handler is designed to be used only during boot.

2.3.1 Use case

This section explains an example of the configure interrupt structure using the following use cases:

- System Interrupt source: GPIO Port Interrupt#21 (IDX: 43)
- Mapped to CPU Interrupt: IRQ3
- CPU Interrupt Priority: 2

In the Interrupt structure configuration, GPIO Port 21_4 is connected to the button switch. An interrupt is generated by pressing the switch. The IDX numbers used in this section denote the system interrupt index numbers of the XMC7200 MCU series. See the device datasheet for the actual index number to be used. See the "I/O System" chapter in architecture TRM and application note for GPIO setting.

2.3.2 Configuration

Table 1 and Table 2 list the parameters and functions of the configuration part in PDL for interrupt initialization.

Table 1 Interrupt initialization parameters

Parameters	Description	Value
`irq_cfg.intrSrc`	Bit 0-15 of intrSrc is used to store the system interrupt value and bit 16-31 are used to store the CPU IRQ value. System interrupt index number [0: 1022]. CPU interrupt number [0: 7] NvicMux0_IRQn is assigned to IRQ 0, NvicMux1_IRQn is assigned to IRQ 1, NvicMux2_IRQn is assigned to IRQ 2, NvicMux3_IRQn is assigned to IRQ 3, NvicMux4_IRQn is assigned to IRQ 4, NvicMux5_IRQn is assigned to IRQ 5, NvicMux6_IRQn is assigned to IRQ 6, NvicMux7_IRQn is assigned to IRQ 7.	System interrupt number: ioss_interrupts_gpio_21_IRQn It is assigned to Port 21 (Index number = 43) CPU interrupt number: IRQ3
`irq_cfg.intrPriority`	Set interrupt priority	intrPriority: 2ul

Table 2 Interrupt initialization functions

Function	Description	Value
`Cy_SysInt_Init(config, userIsr)`	Configure the interrupt structure. config: Interrupt structure parameters address userIsr: Address of the ISR	& config &userIsr, See Code Listing 6.
`NVIC_ClearPendingIRQ(intSrc)`	Clear pending flag: intSrc: CPU interrupt number	NvicMux3_IRQn
`NVIC_EnableIRQ(intSrc)`	Set interrupt enable intSrc: CPU interrupt number	NvicMux3_IRQn

Code Listing 1 and Code Listing 2 show an example program of the interrupt configuration part.

The following description will help you understand the register notation of the driver part of PDL:

- `CPUSS_CM7_0_SYSTEM_INT_CTL[x]` Register is the `CPUSS_CM7_0_SYSTEM_INT_CTLx` register mentioned in the registers TRM. Other registers are also described similarly. "***x***" signifies the system interrupt index number.
- Performance improvement measures:

  For CM7/CM0+ core, the configuration structure (`cy_stc_sysint_t`) must specify the system interrupt source, CPU IRQ, and the CPU IRQ priority. The system interrupt source is mapped to bit 0–15 of the `intrSrc` parameter; CPU IRQ is mapped to bit 16–31 of the `intrSrc` parameter.

  To improve the performance of the register setting, the PDL writes a complete 32-bit data to the register with macro define -- `_VAL2FLD`.

Interrupt overview

```
IRQn_Type IRQn = (IRQn_Type)(config->intrSrc >> 16); // Fetch bit 16-31 to get CPU
IRQ value

cy_en_intr_t devIntrSrc = (cy_en_intr_t)(config->intrSrc & 0xFFFFUL); // Fetch bit
0-15 to get system interrupt value

CPUSS_CM7_0_SYSTEM_INT_CTL[devIntrSrc] =
_VAL2FLD(CPUSS_CM7_0_SYSTEM_INT_CTL_CPU_INT_IDX, IRQn)|
CPUSS_CM7_0_SYSTEM_INT_CTL_CPU_INT_VALID_Msk;


CPUSS_CM7_1_SYSTEM_INT_CTL[devIntrSrc] =
_VAL2FLD(CPUSS_CM7_1_SYSTEM_INT_CTL_CPU_INT_IDX, IRQn)|
CPUSS_CM7_1_SYSTEM_INT_CTL_CPU_INT_VALID_Msk;
```

See *cy_sysint.c* under *mtb-pdl-cat1/drivers/source* for more information on the registers' union and structure representation.

Code Listing 1 Example of vector table configuration

```
static void CopyVectorTable(void)
{
    const uint8_t    u8NrOfVectors = (uint8_t) ((uint32_t) &__Vectors_Size / 4);
    uint32_t * const pu32RamTable = (uint32_t *) __ramVectors;
    uint32_t * const pu32RomTable = (uint32_t *) (&__vector_table);



    for(uint8_t u8Index = 0; u8Index < u8NrOfVectors; u8Index++)
    {
        pu32RamTable[u8Index] = pu32RomTable[u8Index];
    }


    SCB->VTOR = (uint32_t) pu32RamTable;
}
```

This code is called during start up.

Interrupt overview

Code Listing 2 Example of interrupt configuration

```
#define CYBSP_USER_BTN1_PORT GPIO_PRT21
#define CYBSP_USER_BTN1_PIN 4U

const cy_stc_gpio_pin_config_t P21_4_Pin_Init =
 {
.outVal = 1u,                    /* Pin output state */
.driveMode = CY_GPIO_DM_PULLUP,  /* Drive mode */
.hsiom = HSIOM_SEL_GPIO,         /* HSIOM selection */
.intEdge = CY_GPIO_INTR_FALLING, /* Interrupt Edge type */
.intMask = CY_GPIO_INTR_EN_MASK, /* Interrupt enable mask */
.vtrip = CY_GPIO_VTRIP_CMOS,     /* Input buffer voltage trip type */
.slewRate = CY_GPIO_SLEW_FAST,   /* Output buffer slew rate */
.driveSel = CY_GPIO_DRIVE_FULL,  /* Drive strength */
.vregEn = 0u,                    /* SIO pair output buffer mode */
.ibufMode = 0u,                  /* SIO pair input buffer mode */
.vtripSel = 0u,                  /* SIO pair input buffer trip point */
.vrefSel = 0u,                   /* SIO pair reference voltage for input buffer trip
  point */
.vohSel = 0u                     /* SIO pair regulated voltage output level */
};

/* This structure initializes the Port21 interrupt for the NVIC */
cy_stc_sysint_t intrCfg =
{
.intrSrc = ((NvicMux3_IRQn << 16) | ioss_interrupts_gpio_21_IRQn), /* Interrupt
  source is GPIO port 21 interrupt */
.intrPriority = GPIO_INTERRUPT_PRIORITY   /* Interrupt priority is 7 */
};

static void gpio_interrupt_handler_PDL()
{
    gpio_intr_flag = true;

    /* Clear pin interrupt logic. Required to detect next interrupt */
    Cy_GPIO_ClearInterrupt(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN);
}
int main(void)
{
     /* Initialize the device and board peripherals */
     result = cybsp_init();
```

Interrupt overview

Code Listing 2 Example of interrupt configuration

```
    /* Enable global interrupts */
    __enable_irq();

    /* Board initialize failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
}

/* Initialize the user button */
Cy_GPIO_Pin_FastInit(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN, CY_GPIO_DM_PULLUP, 1UL,
  HSIOM_SEL_GPIO);

    /* Pin Interrupts */
    /* Configure GPIO pin to generate interrupts */
    Cy_GPIO_SetInterruptEdge(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN,
  CY_GPIO_INTR_RISING);
Cy_GPIO_SetInterruptMask(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN,
  CY_GPIO_INTR_EN_MASK);

    /* Configure CPU GPIO interrupt vector */
    Cy_SysInt_Init(&intrCfg, gpio_interrupt_handler_PDL);
    NVIC_ClearPendingIRQ((IRQn_Type)intrCfg.intrSrc);
   NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);

   for(;;)
   {
   }
}
```

(*1) Programming hint: The CM7/CM0+ interrupt enable and NVIC operation instructions are provided by Cortex® Microcontroller Software Interface Standard (CMSIS) with intrinsic functions.

Code Listing 3 Cy_SysInt_Init() function

```
cy_en_sysint_status_t Cy_SysInt_Init(const cy_stc_sysint_t* config, cy_israddress
  userIsr)
{
    cy_en_sysint_status_t status = CY_SYSINT_SUCCESS;

    if(NULL != config)
    {
        CY_ASSERT_L3(CY_SYSINT_IS_PRIORITY_VALID(config->intrPriority));
```

Code Listing 3 Cy_SysInt_Init() function

```
        #if defined (CY_IP_M4CPUSS)
            #if (CY_CPU_CORTEX_M0P)
                if (config->intrSrc > SysTick_IRQn)
                {
                    Cy_SysInt_SetInterruptSource(config->intrSrc, config-
>cm0pSrc);
                }
                else
                {
                    status = CY_SYSINT_BAD_PARAM;
                }
            #endif

                NVIC_SetPriority(config->intrSrc, config->intrPriority);

                /* Set the new vector only if it was moved to __ramVectors */
                if (SCB->VTOR == (uint32_t)&__ramVectors)
                {
                    (void)Cy_SysInt_SetVector(config->intrSrc, userIsr);
                 }
        #endif


        #if defined (CY_IP_M7CPUSS)
            CY_MISRA_DEVIATE_LINE('MISRA C-2012 Rule 10.8','Intentional typecast
to IRQn_Type enum.');
            IRQn_Type IRQn = (IRQn_Type)(config->intrSrc >> 16) ; /* Fetch bit 16-
31 to get CPU IRQ value */
            CY_MISRA_DEVIATE_LINE('MISRA C-2012 Rule 10.8','Intentional typecast
to cy_en_intr_t enum.');
            cy_en_intr_t devIntrSrc = (cy_en_intr_t)(config->intrSrc & 0xFFFFUL);
/* Fetch bit 0-15 to get system interrupt value */
            CY_ASSERT_L3(CY_SYSINT_IS_PRIORITY_VALID(config->intrPriority));
        #if (CY_CPU_CORTEX_M0P)
            if (IRQn > NvicMux1_IRQn && IRQn < Internal0_IRQn) /* CPU IRQ0 and
IRQ1 for CM0+ are used by ROM and not meant for user. */
        #else
            if (IRQn > SysTick_IRQn && IRQn < Internal0_IRQn)
        #endif
            {
                Cy_SysInt_SetInterruptSource(IRQn, devIntrSrc);
            }
            else
```

Code Listing 3 Cy_SysInt_Init() function

```
            {
                status = CY_SYSINT_BAD_PARAM;
            }

            NVIC_SetPriority(IRQn, config->intrPriority);

            (void)Cy_SysInt_SetSystemIrqVector(devIntrSrc, userIsr);
        #endif
    }
    else
    {
        status = CY_SYSINT_BAD_PARAM;
    }

    return(status);
}
```

Code Listing 4 Cy_SysInt_SetSystemIrqVector() function

```
void  Cy_SysInt_SetSystemIrqVector(cy_en_intr_t sysIntSrc, cy_systemIntr_Handler
  userIsr)
{
    if (Cy_SysInt_SystemIrqUserTableRamPointer != NULL)
    {
        Cy_SysInt_SystemIrqUserTableRamPointer[sysIntSrc] = userIsr;
    }
}
```

2.4 Interrupt handling

In this MCU series, multiple system interrupts are mapped to the same CPU interrupt. Therefore, they share a CPU interrupt handler. To identify the system interrupt that occurred, each CPU interrupt has a `CPUSS_CMn_INTx_STATUS` register. This register has the following fields:

- `CPUSS_ V2_CMn_INTx_STATUS_SYSTEM_INT_VALID`: Specifies if any system interrupt is active for the CPU interrupt
- `CPUSS_V2_CMn_INTx_STATUS_SYSTEM_INT_IDX [9:0]`: Specifies the index (a number in the range [0, 1022]) of the lowest active system interrupt mapped to the corresponding CPU interrupt.

The CPU interrupt handler uses the `SYSTEM_INT_IDX` field to index a system interrupt lookup table and jumps to the system interrupt handler.

Note that in this series, these interrupts are level interrupts, so clear the peripheral interrupt in the interrupt handler (step 5 in Figure 5). Ensure toread back the register for the completion of register write access and the pending registers are clearing when returning from the interrupt. Figure 5 shows the interrupt handling example in the XMC7200 series.

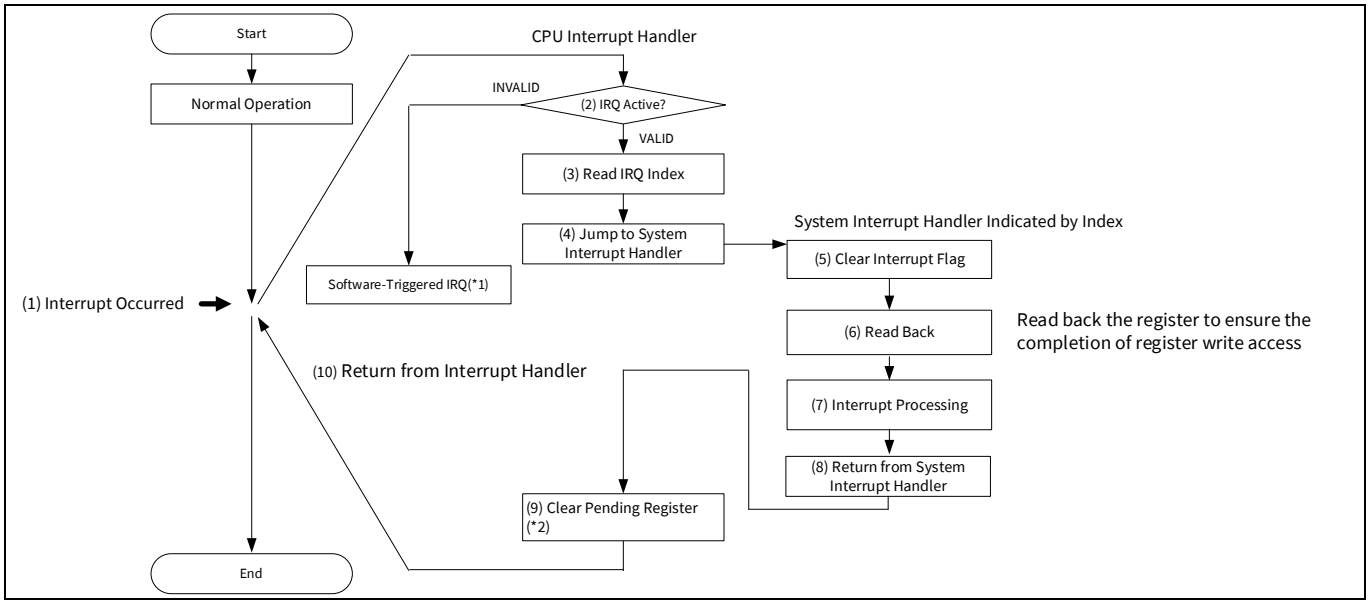

Figure 5 Interrupt handling

Interrupt overview

Code Listing 5 and Code Listing 6 show an example program of the interrupt handler.

Code Listing 5 CPU interrupt handler

```
void CM7_CpuIntr_Handler(uint8_t intrNum)
{
    uint32_t system_int_idx;
    cy_israddress handler;

    if(CY_IS_CM7_CORE_0 && (_FLD2VAL(CPUSS_CM7_0_INT_STATUS_SYSTEM_INT_VALID,
CPUSS_CM7_0_INT_STATUS[intrNum])))
    {
        system_int_idx = _FLD2VAL(CPUSS_CM7_0_INT_STATUS_SYSTEM_INT_IDX,
CPUSS_CM7_0_INT_STATUS[intrNum]);
        handler = Cy_SystemIrqUserTable[system_int_idx];
        if(handler != NULL)
        handler(); // jump to system interrupt handler
    }
    else if(CY_IS_CM7_CORE_1 &&
(_FLD2VAL(CPUSS_CM7_1_INT_STATUS_SYSTEM_INT_VALID,
CPUSS_CM7_1_INT_STATUS[intrNum])))
    {
        system_int_idx = _FLD2VAL(CPUSS_CM7_1_INT_STATUS_SYSTEM_INT_IDX,
CPUSS_CM7_1_INT_STATUS[intrNum]);
        handler = Cy_SystemIrqUserTable[system_int_idx];
        if(handler != NULL)
        handler(); // jump to system interrupt handler
    }
    else
    {
        // Triggered by software or because of software cleared a peripheral
interrupt flag but did not clear the pending flag at NVIC
    }
    NVIC_ClearPendingIRQ((IRQn_Type)intrNum);
}
```

Code Listing 6 gpio_interrupt_handler_PDL() handler

```
static void gpio_interrupt_handler_PDL()
{
   gpio_intr_flag = true;

   /* Clear pin interrupt logic. Required to detect next interrupt */
   Cy_GPIO_ClearInterrupt(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN);
}
```

Interrupt overview

Code Listing 7 Cy_GPIO_ClearInterrupt() function

```
void Cy_GPIO_ClearInterrupt(volatile stc_GPIO_PRT_t* base, uint32_t pinNum)
{
    uint32_t prtIntr;
#if (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE)
    cy_en_pra_pin_prot_type_t pinType;
#endif /* (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE) */

    CY_ASSERT_L2(CY_GPIO_IS_FILTER_PIN_VALID(pinNum));

#if (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE)
    pinType = CY_PRA_GET_PIN_PROT_TYPE(base, pinNum);
#if defined(CY_DEVICE_PSOC6ABLE2)
    if (pinType == CY_PRA_PIN_SECURE_UNCONSTRAINED)
    {
        (void)CY_PRA_REG32_GET(CY_PRA_GET_PORT_REG_INDEX(base,
CY_PRA_SUB_INDEX_PORT_INTR));
    }
    else if (pinType == CY_PRA_PIN_SECURE_NONE)
    {
      /* Any INTR MMIO registers AHB clearing must be preceded with an AHB read
 access */
        (void)GPIO_PRT_INTR(base);
    }
    else
    {
      /* secure pin */
    }
#else
   /* Any INTR MMIO registers AHB clearing must be preceded with an AHB read
 access */
   (void)GPIO_PRT_INTR(base);
#endif /* defined(CY_DEVICE_PSOC6ABLE2) */
#else
   /* Any INTR MMIO registers AHB clearing must be preceded with an AHB read
 access */
   (void)GPIO_PRT_INTR(base);
#endif /* (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE) */

   prtIntr = CY_GPIO_INTR_STATUS_MASK << pinNum;

#if (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE)
    if (pinType != CY_PRA_PIN_SECURE)
    {
        if (pinType == CY_PRA_PIN_SECURE_UNCONSTRAINED)
        {
            CY_PRA_REG32_SET(CY_PRA_GET_PORT_REG_INDEX(base,
CY_PRA_SUB_INDEX_PORT_INTR), prtIntr);
        }
        else
        {
            GPIO_PRT_INTR(base) = prtIntr;
        }
    }
    else
    {
```

Interrupt overview

Code Listing 7 Cy_GPIO_ClearInterrupt() function

```
        /* Secure PIN can't be modified using register policy */
    }
#else
    GPIO_PRT_INTR(base) = prtIntr;
#endif /* (CY_CPU_CORTEX_M4) && defined(CY_DEVICE_SECURE) */

   /* This read ensures that the initial write has been flushed out to the
hardware */
    (void)GPIO_PRT_INTR(base);
}
```

Interrupt overview

*Note: (*1) The `SYSTEM_INT_VALID` bit is not set when a software interrupt occurs using the software-triggered interrupt register (STIR). Software interrupt (using CPU register)software interrupts.*

*Note: (*2) The flag on the pending register (pending flag) is cleared when the CPU interrupt is accepted. Therefore, it is not necessary to clear it in the CPU interrupt handler. However, note that when handling multiple interrupts with one interrupt handler, invalid interrupts can be prevented by clearing the pending flag when returning from an interrupt.*

Figure 6 shows two (Interrupt 1 and Interrupt 2) factors are processed in one system interrupt handler.

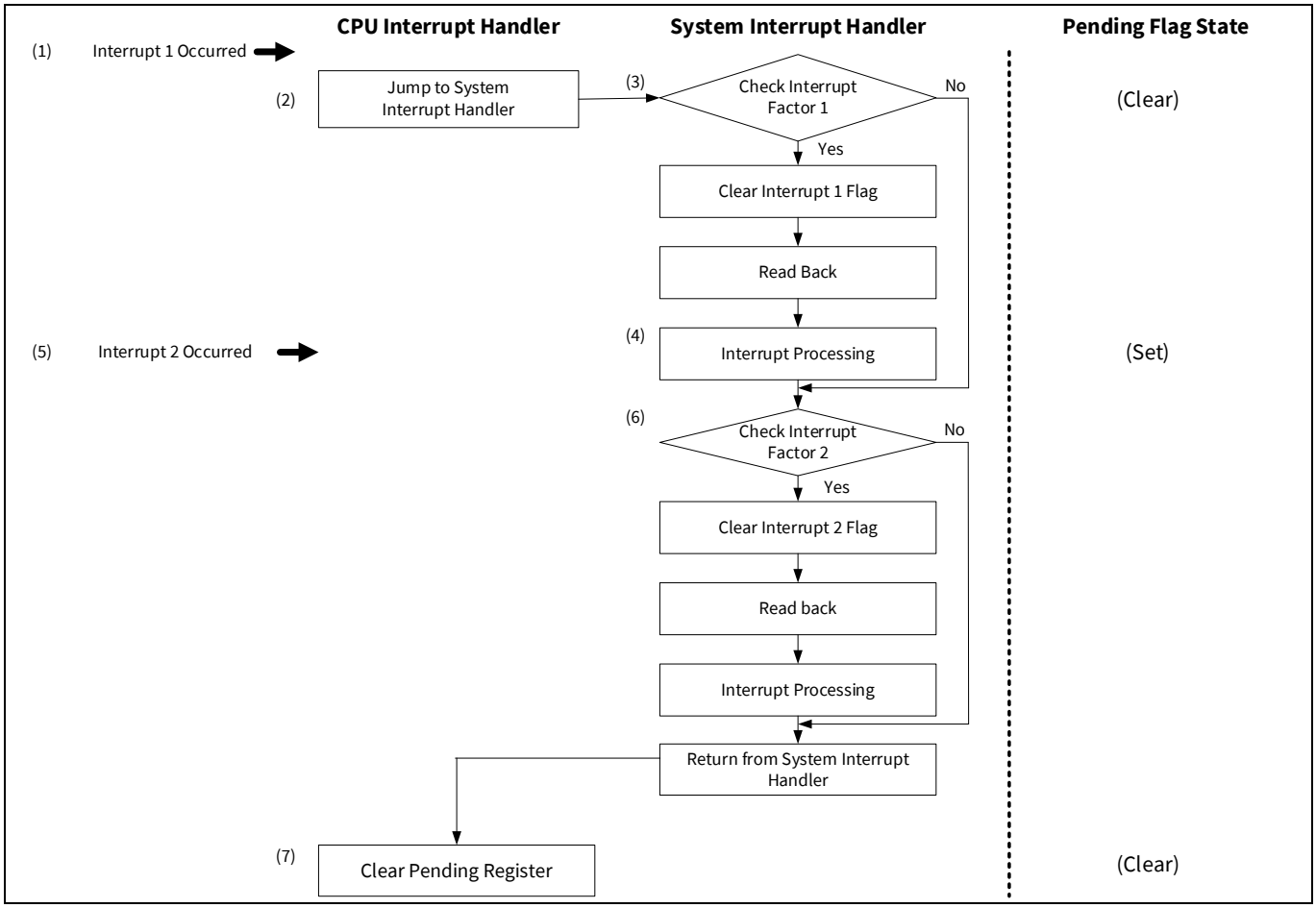

Figure 6 Example of pending flag clear

1. Interrupt 1 occurs.
2. The CPU interrupt handler checks the interrupt factor and jumps to the corresponding system interrupt handler. The pending flag has been cleared because the interrupt was accepted.
3. Checks the interrupt factor. If interrupt 1, the flag of interrupt 1 is cleared.
4. Processes interrupt 1.
5. If the Interrupt 2 occurs during the processing of interrupt 1, the pending flag is set again.
6. Checks interrupt 2 when the processing of interrupt 1 is completed. In this example, executes the interrupt flag clear and the processing of interrupt 2.
7. Clears the pending flag and returns from the interrupt handler. If the pending flag is not cleared, an interrupt will be generated again after returning from the CPU interrupt handler.

2.5 Peripheral interrupt structure

The interrupt structure of most peripheral functions consists of INTR, INTR_SET, INTR_MASK, and INTR_MASKED registers.

- INTR: Indicates that an interrupt factor has occurred. The software can clear these by writing '1' to these bits.
- INTR_SET: Sets an interrupt. The software can write '1' to these bits to set the corresponding INTR bit.
- INTR_MASK: Masks an interrupt. The software can selectively enable interrupts by writing '1' to the corresponding bit.
- INTR_MASKED: Indicates that a masked interrupt factor has occurred. It reflects a bitwise AND between the interrupt request (INTR) and mask (INTR_MASK) registers.

A software interrupt can be generated by using the INTR_SET register. See Software interrupt (using peripheral) for more details.

2.6 Example usage of interrupt structure

An example of using the interrupt structure is explained according to the following usage assumptions.

Note: The IDX numbers in this section denote system interrupt index numbers. IDX numbers are the IDX number of the XMC7200 series. See the device datasheets for the actual index number to use.

2.6.1 Peripheral interrupt handling

This section explains general interrupts from an external pin (GPIO Port#17, 21) and shows its operation, initial setting, and interrupt handling. CM7 processes each interrupt. The following shows the use case.

2.6.1.1 Use case

- **Interrupt setting 1**
  - System interrupt source: GPIO Port Interrupt#17 (IDX: 39) rising-edge detection
  - Mapped to CPU interrupt: *NvicMux4_IRQn*
  - CPU interrupt priority: 2
- **Interrupt setting 2**
  - System interrupt source: GPIO Port Interrupt#21 (IDX: 43) rising-edge detection
  - Mapped to CPU interrupt: *NvicMux5_IRQn*
  - CPU interrupt priority: 3

Figure 7 shows interrupt operation for this configuration.

Interrupt overview

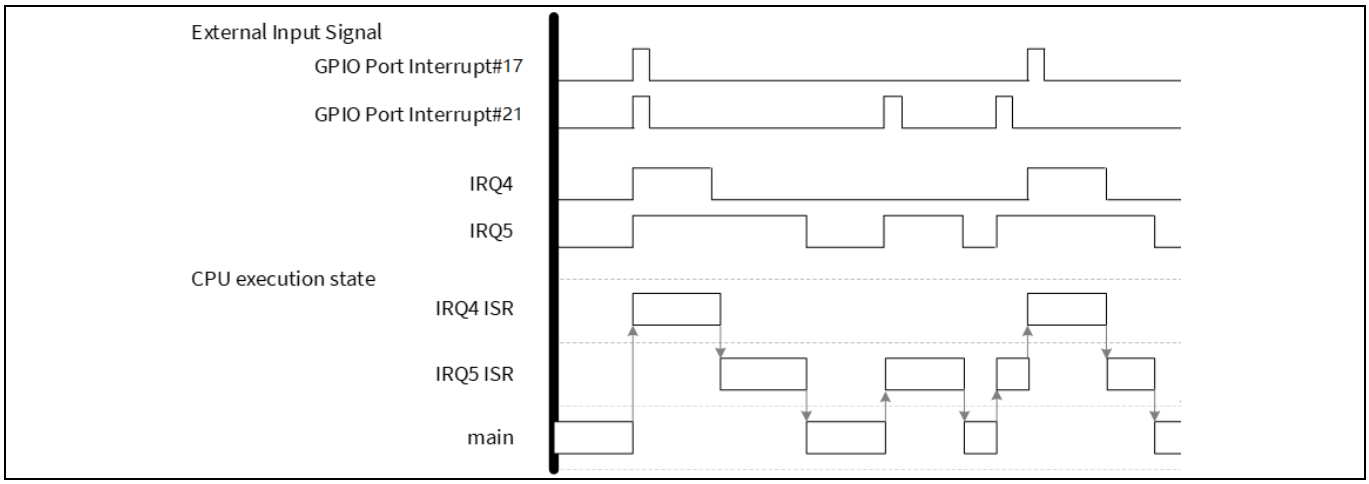

Figure 7 Example interrupt operation usage

If GPIO Port Interrupt#17 and GPIO Port Interrupt#21 interrupts occur at the same time, then the GPIO Port Interrupt#17 t takes precedence because it is mapped to IRQ4, which has a higher priority than IRQ5, which GPIO Port Interrupt#1 is mapped.

If GPIO Port Interrupt#17 interrupts occur during GPIO Port Interrupt#21 processing. The GPIO Port Interrupt#0 takes precedence because it is mapped to IRQ4, which has a higher priority than IRQ5 to which the GPIO Port Interrupt#21 interrupt is mapped.

2.6.1.2 Configuration

- **Initial setting procedure**

  For thethe initial setting procedure, see the Initial setting. After each peripheral resource (GPIO Port Interrupt#17,21) is set up, each resource interrupt is routed to the CPU interrupt. sSet the level and permission of each CPU interrupt to the NVIC. After setting up the interrupt connection, start each resource. See the “I/O System” chapter of the architecture TRM for the GPIO setting.

- **Interrupt handler operation**

  The CPU jumps to the interrupt handler specified in the vector table when an interrupt occurs. The CPU interrupt handler identifies the system interrupt that triggered the CPU interrupt and jumped to the corresponding ISR. After clearing the system interrupt flag in the CPU interrupt handler and executing the ISR, remove the relevant CPU Pending register bit and return to the main routine. See Interrupt handling for the interrupt handler configuration.

Figure 8 and Figure 9 show the software flow when single, and multiple interrupts occur.

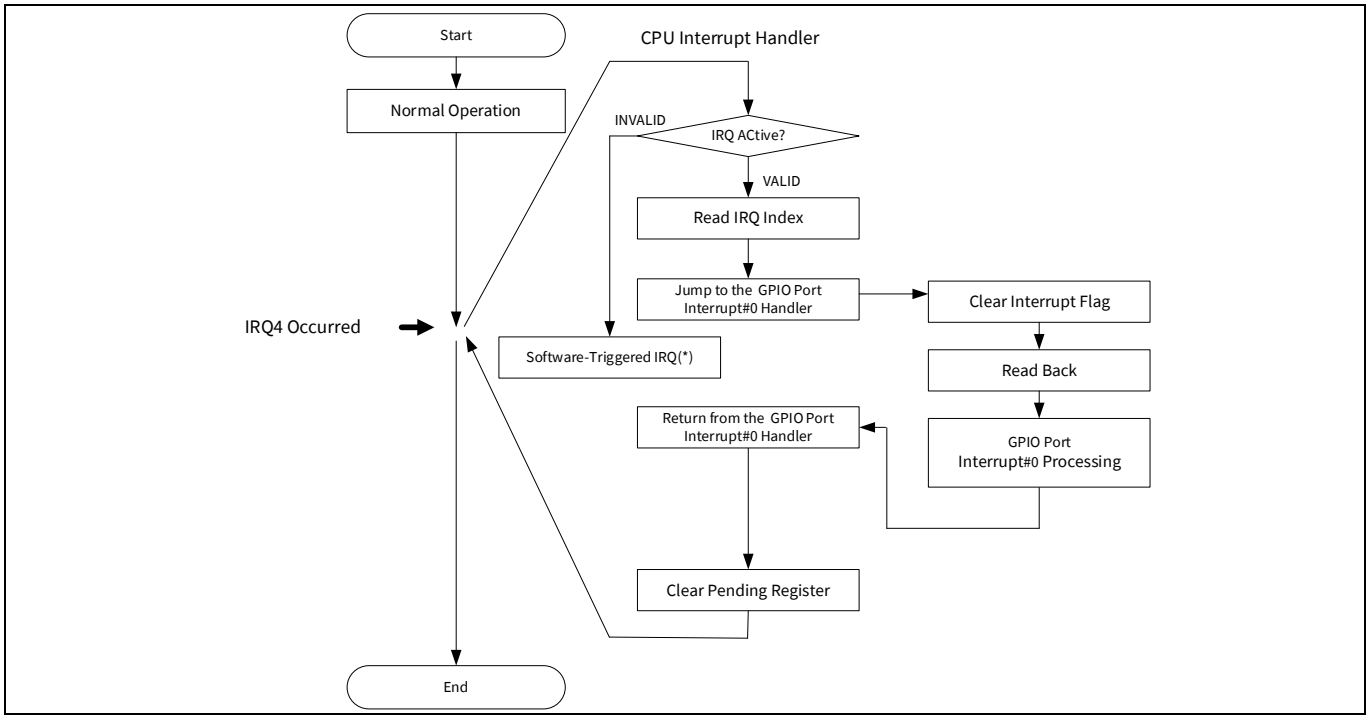

Figure 8 Interrupt operation flow

Note: (*) The `SYSTEM_INT_VALID` bit is not set when a software interrupt occurs using the software triggered interrupt register (STIR). See Software interrupt (using CPU register) for software interrupts.

If a high-priority CPU interrupt occurs in the current ISR, suspend the current ISR processing and execute the ISR with the high priority.

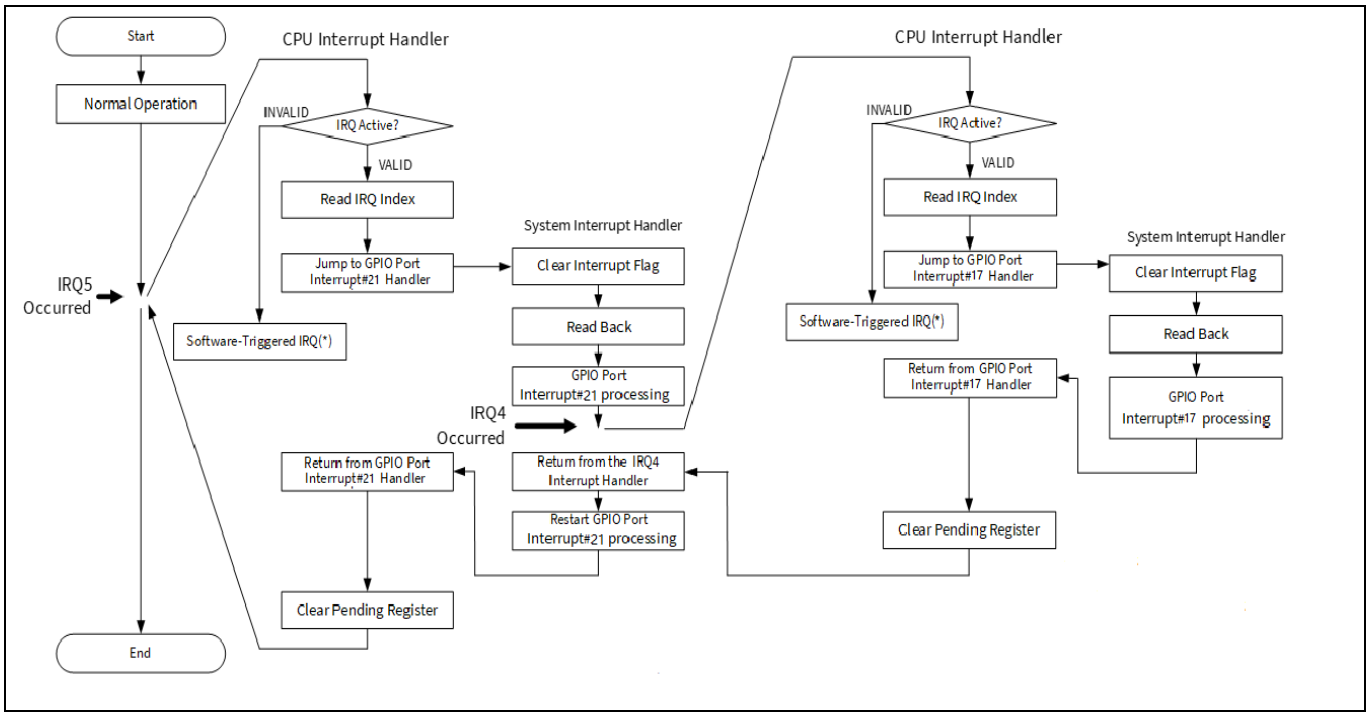

Figure 9 Nested interrupt operation flow

Interrupt overview

Note: (*) The SYSTEM_INT_VALID bit is not set when a software interrupt occurs using the software-triggered interrupt register (STIR). See Software interrupt (using CPU register) for software interrupts.

2.6.2 NMI generation of system interrupts

In this series of MCUs, all available system interrupts (up to four) can be mapped to the NMI of each CPU. This section explains the NMI's timing protection configuration and shows its operation, initial setting, and interrupt handling. CM7 processes each interrupt.

2.6.2.1 Use case

In this configuration, monitor the processing time of the periodic interrupt routine called by TCPWM0 Group#0 Counter#1.

TCPWM1 Group#0 Counter#2 is used to monitor the processing time of the interrupt routine. TCPWM0 Group#0 Counter#0 is a counter that counts upwards and generates an interrupt when the count value is more than the compare value. It starts counting when the periodic interrupt starts and stops when interrupt processing is completed. If interrupt processing is not completed within a specific time, that is, if the counting is not stopped, an NMI is notified to the CPU by the TCPWM0 Group#0 Counter#1 overflow interrupt.

- TCPWM0 Group#0 Counter#1
  - System interrupt source: TCPWM0 Group#0 Counter#1(IDX: 520) TC interrupt
  - Mapped to CPU NMI
  - Monitor NvicMux5_IRQn ISR operation time
  - Compare value: 100 (Monitoring cycle)
- TCPWM1 Group#0 Counter#2
  - System interrupt source: TCPWM1 Group#0 Counter#2(IDX: 437) TC interrupt
  - Mapped to CPU interrupt: NvicMux5_IRQn
  - CPU interrupt priority: 5
  - Compare value: 500 (interrupt period for system operation)

Figure 10 shows interrupt operation for this configuration.

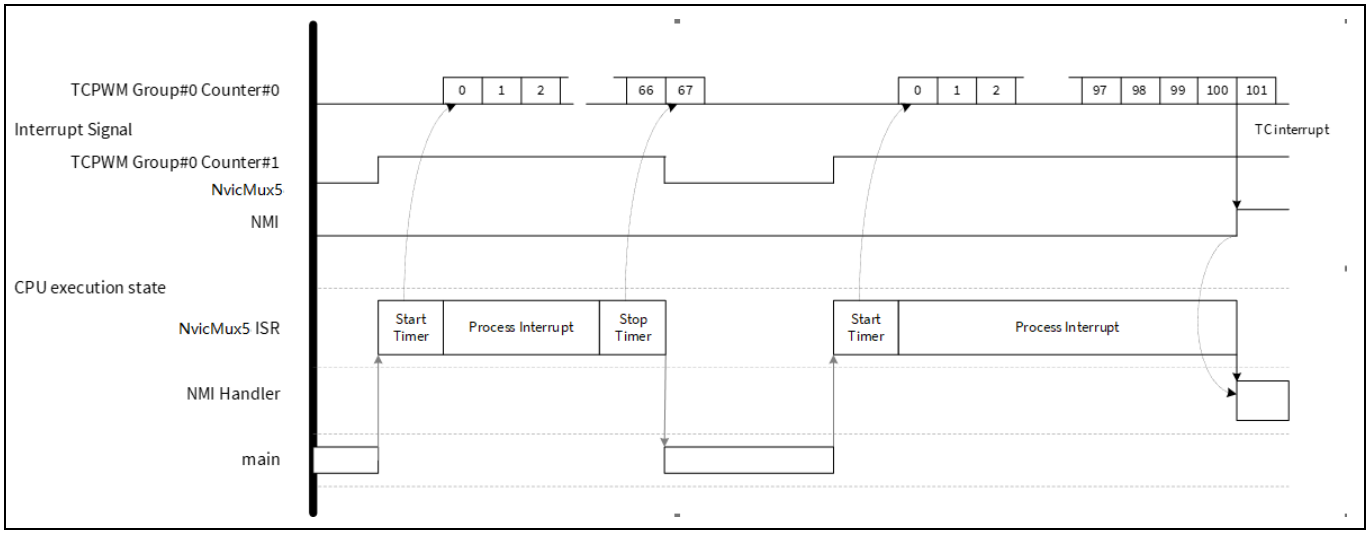

Figure 10 Timing protection operation example

2.6.2.2 Configuration

- Initial setting procedure:

  For the initial setting procedure, see the Initial setting. After each peripheral resource (TCPWM0 Group#0 Counter#1 and TCPWM1 Group#0 Counter#2) is set up, each resource interrupt is routed to the CPU interrupt. Set the level and permission of each CPU interrupt to the NVIC. After setting up the interrupt connection, start each resource.

Figure 11 shows the initializing interrupts for NMI generation.

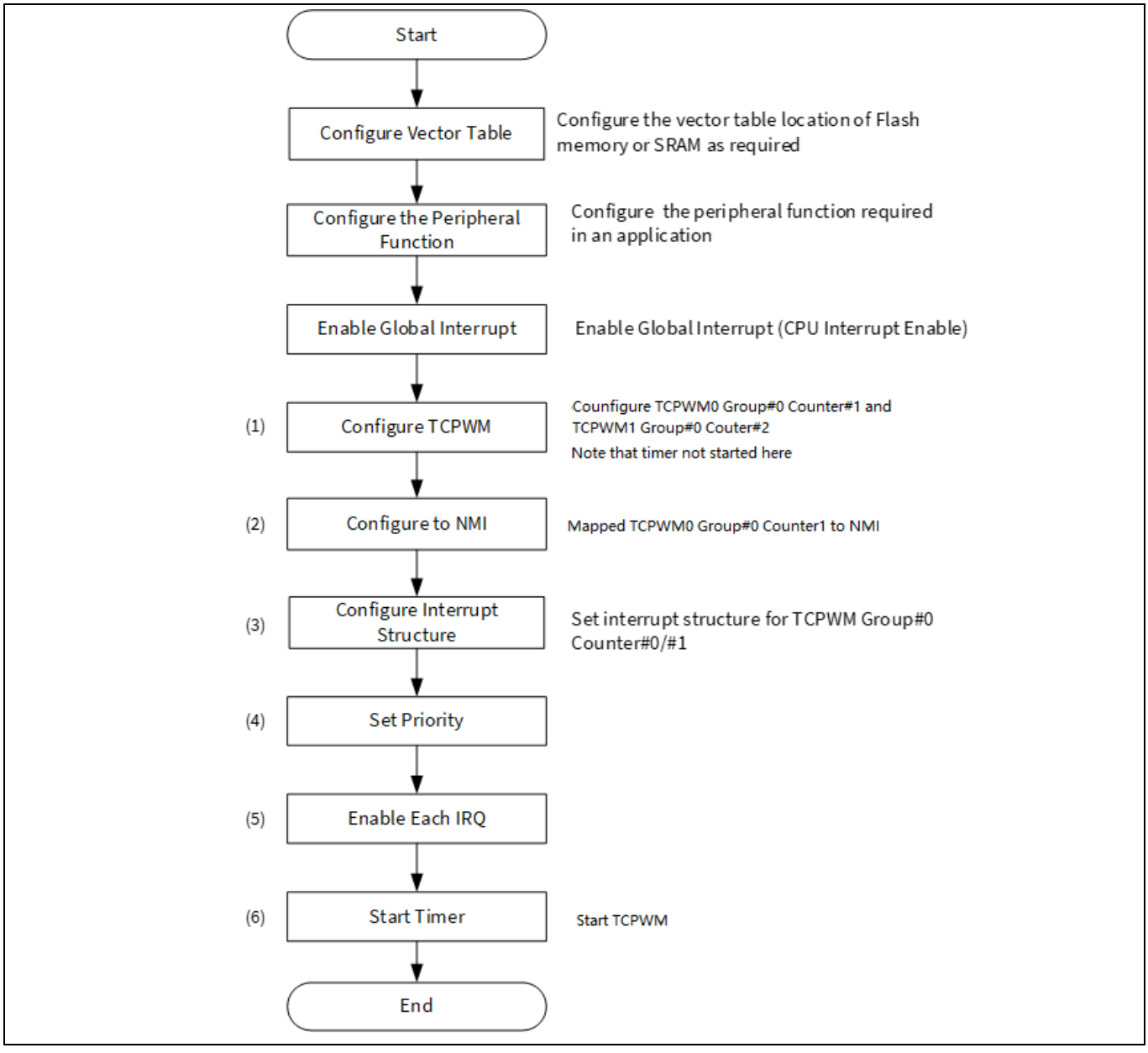

Figure 11 Interrupt structure initial setting procedure for NMI

Table 3 and Table 4 list the parameters and functions of the configuration part in PDL for interrupt initialization.

Interrupt overview

Table 3 Interrupt initialization parameters

Parameters	Description	Value
`irq_cfg_0.intrSrc`	Bits 0–15 of `intrSrc` are used to store the system interrupt value and bits 16–31 are used to store the CPU IRQ value. System interrupt index number [0: 1022]. CPU interrupt number [0: 7] `NvicMux0_IRQn` is assigned to IRQ 0, NvicMux1_IRQn is assigned to IRQ 1, NvicMux2_IRQn is assigned to IRQ 2, NvicMux3_IRQn is assigned to IRQ 3, NvicMux4_IRQn is assigned to IRQ 4, NvicMux5_IRQn is assigned to IRQ 5, NvicMux6_IRQn is assigned to IRQ 6, NvicMux7_IRQn is assigned to IRQ 7.	System interrupt number: `tcpwm_0_interrupts_1_IRQn`. It is assigned to TCPWM0 Group#0 Counter#1 (Index number = 520). CPU interrupt number: `NvicMux4_IRQn`
`irq_cfg_0. intrPriority`	Set interrupt Priority	intrPriority: 6ul

`irq_cfg_1. intrSrc`	Bits 0–15 of `intrSrc` are used to store the system interrupt value; bits 16–31 are used to store the CPU IRQ value.
	
	System interrupt index number [0: 1022].
	
	CPU interrupt number [0: 7]
	NvicMux0_IRQn is assigned to IRQ 0,
	NvicMux1_IRQn is assigned to IRQ 1,
	NvicMux2_IRQn is assigned to IRQ 2,
	NvicMux3_IRQn is assigned to IRQ 3,
	NvicMux4_IRQn is assigned to IRQ 4,
	NvicMux5_IRQn is assigned to IRQ 5,
	NvicMux6_IRQn is assigned to IRQ 6,
	NvicMux7_IRQn is assigned to IRQ 7.
		System interrupt number:
		`tcpwm_1_interrupts_2_IRQn`
		It is assigned to TCPWM1 Group#0 Counter#2 (Index number =437)
		
		CPU interrupt number:
		`NvicMux5_IRQn`

`irq_cfg_1. intrPriority`	Set interrupt priority	intrPriority: 5ul

Interrupt overview

Table 4 Interrupt initialization functions

Function	Description	Value
`Cy_SysInt_SetNmiSource(nmiNum, intrSrc)`	Set the NMI register. nmiNum: CM7 NMI control register number `CPUSS_CM7_0_NMI_CTLx` register (x = 0 to 3) intrSrc: System interrupt index number	nmiNum: 0 Use `CPUSS_CM7_0_NMI_CTL0` register intrSrc: `tcpwm_0_interrupts_1_IRQn`
`Cy_SysInt_Init(config, userIsr)`	Configure the interrupt structure. config: Interrupt structure parameters address userIsr: Address of the ISR	&config 0/1 userIsr: NMI_Handler / Counter_Handler See Code Listing 11 and Code Listing 13.
`NVIC_EnableIRQ(IRQn)`	Set interrupt enable. IRQn: CPU interrupt number	Bit 16-31 of irq_cfg_0/1. IRQn

Interrupt overview

Code Listing 8 and Code Listing 9 show an example program of the interrupt configuration part. See the “Timer, Counter, and PWM” chapter of the architecture TRM and application note for TCPWM setting details.

Code Listing 8 Example of interrupt configuration 1

```
#define TCPWM_COUNTER1_HW TCPWM0
#define TCPWM_COUNTER1_NUM 1UL
#define TCPWM_COUNTER1_IRQ tcpwm_0_interrupts_1_IRQn
#define TCPWM_COUNTER1_INPUT_DISABLED 0x7U

const cy_stc_tcpwm_counter_config_t TCPWM_COUNTER1_config =
{
        .period = 60000,
        .clockPrescaler = CY_TCPWM_COUNTER_PRESCALER_DIVBY_1,
        .runMode = CY_TCPWM_COUNTER_CONTINUOUS,

        .countDirection = CY_TCPWM_COUNTER_COUNT_UP,

        .compareOrCapture = CY_TCPWM_COUNTER_MODE_COMPARE,

        .compare0 = 50000,

        .compare1 = 16384,

        .enableCompareSwap = false,

        .interruptSources = (CY_TCPWM_INT_ON_TC & 0U) | (CY_TCPWM_INT_ON_CC0 ) |
(CY_TCPWM_INT_ON_CC1 & 0U),

        .captureInputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,

        .captureInput = CY_TCPWM_INPUT_0,

        .reloadInputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,

        .reloadInput = CY_TCPWM_INPUT_0,

        .startInputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,

        .startInput = CY_TCPWM_INPUT_0,

        .stopInputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,

        .stopInput = CY_TCPWM_INPUT_0,
        .countInputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,
        .countInput = CY_TCPWM_INPUT_1,
        .capture1InputMode = TCPWM_COUNTER1_INPUT_DISABLED & 0x3U,
        .capture1Input = CY_TCPWM_INPUT_0,
        .compare2 = 30000,
        .compare3 = 16384,
        .enableCompare1Swap = false,
        .trigger0Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,

        .trigger1Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,

};

#define TCPWM_COUNTER2_HW TCPWM1
```

Code Listing 8 Example of interrupt configuration 1

```
#define TCPWM_COUNTER2_NUM 2UL
#define TCPWM_COUNTER2_IRQ tcpwm_1_interrupts_2_IRQn
#define TCPWM_COUNTER2_INPUT_DISABLED 0x7U

const cy_stc_tcpwm_counter_config_t TCPWM_COUNTER2_config =
{
       .period = 32768,
       .clockPrescaler = CY_TCPWM_COUNTER_PRESCALER_DIVBY_1,
       .runMode = CY_TCPWM_COUNTER_CONTINUOUS,
       .countDirection = CY_TCPWM_COUNTER_COUNT_UP,
       .compareOrCapture = CY_TCPWM_COUNTER_MODE_COMPARE,
       .compare0 = 16384,
       .compare1 = 16384,
       .enableCompareSwap = false,
       .interruptSources = (CY_TCPWM_INT_ON_TC & 0U) | (CY_TCPWM_INT_ON_CC0 ) |
(CY_TCPWM_INT_ON_CC1 & 0U),
       .captureInputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .captureInput = CY_TCPWM_INPUT_0,
       .reloadInputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .reloadInput = CY_TCPWM_INPUT_0,
       .startInputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .startInput = CY_TCPWM_INPUT_0,
       .stopInputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .stopInput = CY_TCPWM_INPUT_0,
       .countInputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .countInput = CY_TCPWM_INPUT_1,
       .capture1InputMode = TCPWM_COUNTER2_INPUT_DISABLED & 0x3U,
       .capture1Input = CY_TCPWM_INPUT_0,
       .compare2 = 16384,
       .compare3 = 16384,
       .enableCompare1Swap = false,
       .trigger0Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,

       .trigger1Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,

};

#define INTERRUPT_PRIORITY1 (6u)
#define INTERRUPT_PRIORITY2 (5u)

cy_stc_sysint_t irq_cfg_0 =
{
    .intrSrc = ((NvicMux4_IRQn << 16) | TCPWM_COUNTER1_IRQ),    /* Interrupt source
is tcpwm_0_interrupts_0_IRQn */
    .intrPriority = INTERRUPT_PRIORITY1                         /* Interrupt
priority is 6 */
};

cy_stc_sysint_t irq_cfg_1 =
{
    .intrSrc = ((NvicMux5_IRQn << 16) | TCPWM_COUNTER2_IRQ),    /* Interrupt source
is tcpwm_0_interrupts_0_IRQn */
    .intrPriority = INTERRUPT_PRIORITY2                         /* Interrupt
priority is 5 */
};
```

Interrupt overview

Code Listing 9 Example of interrupt configuration 2

```
void Counter_Handler()
{
    /*toggle user LED1*/
    Cy_GPIO_Inv(CYBSP_USER_LED2_PORT, CYBSP_USER_LED2_PIN);

    /*clear the counter CC0 compare interrupt*/
    Cy_TCPWM_ClearInterrupt(TCPWM_COUNTER_HW, TCPWM_COUNTER_NUM,
CY_TCPWM_INT_ON_CC0);
  }
void NMIException_Handler()
{
    /*toggle user LED1*/
    Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN);

    /*clear the counter CC0 compare interrupt*/
    Cy_TCPWM_ClearInterrupt(TCPWM_COUNTER_HW, TCPWM_COUNTER_NUM,
CY_TCPWM_INT_ON_CC0);
  }
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /*Initialize the User LED*/
    Cy_GPIO_Pin_Init(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN,
&CYBSP_USER_LED1_config );
    Cy_GPIO_Pin_Init(CYBSP_USER_LED2_PORT, CYBSP_USER_LED2_PIN,
&CYBSP_USER_LED2_config );

    /*TCPWM Counter Mode initial*/
    if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(TCPWM_COUNTER1_HW,
TCPWM_COUNTER1_NUM, &TCPWM_COUNTER1_config))
    {
        CY_ASSERT(0);
    }

    /*TCPWM Counter Mode initial*/
    if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(TCPWM_COUNTER2_HW,
TCPWM_COUNTER2_NUM, &TCPWM_COUNTER2_config))
    {
        CY_ASSERT(0);
    }

    /* Enable the initialized counter */
    Cy_TCPWM_Counter_Enable(TCPWM_COUNTER1_HW, TCPWM_COUNTER1_NUM);
    Cy_TCPWM_Counter_Enable(TCPWM_COUNTER2_HW, TCPWM_COUNTER2_NUM);
```

Interrupt overview

Code Listing 9 Example of interrupt configuration 2

```
    /* Set NMI mapping of CM7 */
    Cy_SysInt_SetNmiSource(1ul, TCPWM_COUNTER1_IRQ);

    /* Configure and enable Counter CC0 interrupt */
    Cy_SysInt_Init(&irq_cfg_0, NMIException_Handler);
    NVIC_ClearPendingIRQ((IRQn_Type)irq_cfg_0.intrSrc);
    NVIC_EnableIRQ((IRQn_Type) NvicMux4_IRQn);

    /* Configure and enable Counter CC0 interrupt */
    Cy_SysInt_Init(&irq_cfg_1, Counter_Handler);
    NVIC_ClearPendingIRQ((IRQn_Type)irq_cfg_1.intrSrc);
    NVIC_EnableIRQ((IRQn_Type) NvicMux5_IRQn);

    /* Set INTR_SET register */
    Cy_TCPWM_TriggerStart_Single(TCPWM_COUNTER1_HW, TCPWM_COUNTER1_NUM);
    Cy_TCPWM_TriggerStart_Single(TCPWM_COUNTER2_HW, TCPWM_COUNTER2_NUM);

    for (;;)
    {
      /*empty loop*/
    }
}
```

Interrupt overview

Code Listing 10 Cy_SysInt_SetNmiSource () function

```
void Cy_SysInt_SetNmiSource(cy_en_sysint_nmi_t nmiNum, cy_en_intr_t intrSrc)
{
    CY_ASSERT_L3(CY_SYSINT_IS_NMI_NUM_VALID(nmiNum));

#if (defined(CY_IP_M4CPUSS) && (CY_CPU_CORTEX_M0P))
    CY_ASSERT_L1(CY_SYSINT_IS_PC_0);
#endif

#if defined (CY_IP_M4CPUSS)
    if (CY_CPUSS_V1)
    {
        nmiNum = CY_SYSINT_NMI1; /* For CPUSS_ver1 the NMI number is 1 */
    }
#endif

    #if (CY_CPU_CORTEX_M0P)
        CPUSS_CM0_NMI_CTL((uint32_t)nmiNum - 1UL) = (uint32_t)intrSrc;
    #elif (CY_CPU_CORTEX_M4)
        CPUSS_CM4_NMI_CTL((uint32_t)nmiNum - 1UL) = (uint32_t)intrSrc;
    #else
        if(0UL != CY_IS_CM7_CORE_0)
        {
            CPUSS_CM7_0_NMI_CTL((uint32_t)nmiNum - 1UL) = (uint32_t)intrSrc;
        }
        else
        {
            CPUSS_CM7_1_NMI_CTL((uint32_t)nmiNum - 1UL) = (uint32_t)intrSrc;
        }
    #endif
}
```

- Interrupt handler operation:

For the interrupt handler configuration, see Interrupt handling. If interrupt processing is completed within a specific time, NMI will not occur. However, if interrupt processing is not completed within a specific time, NMI is notified, and the NMI handler is executed.

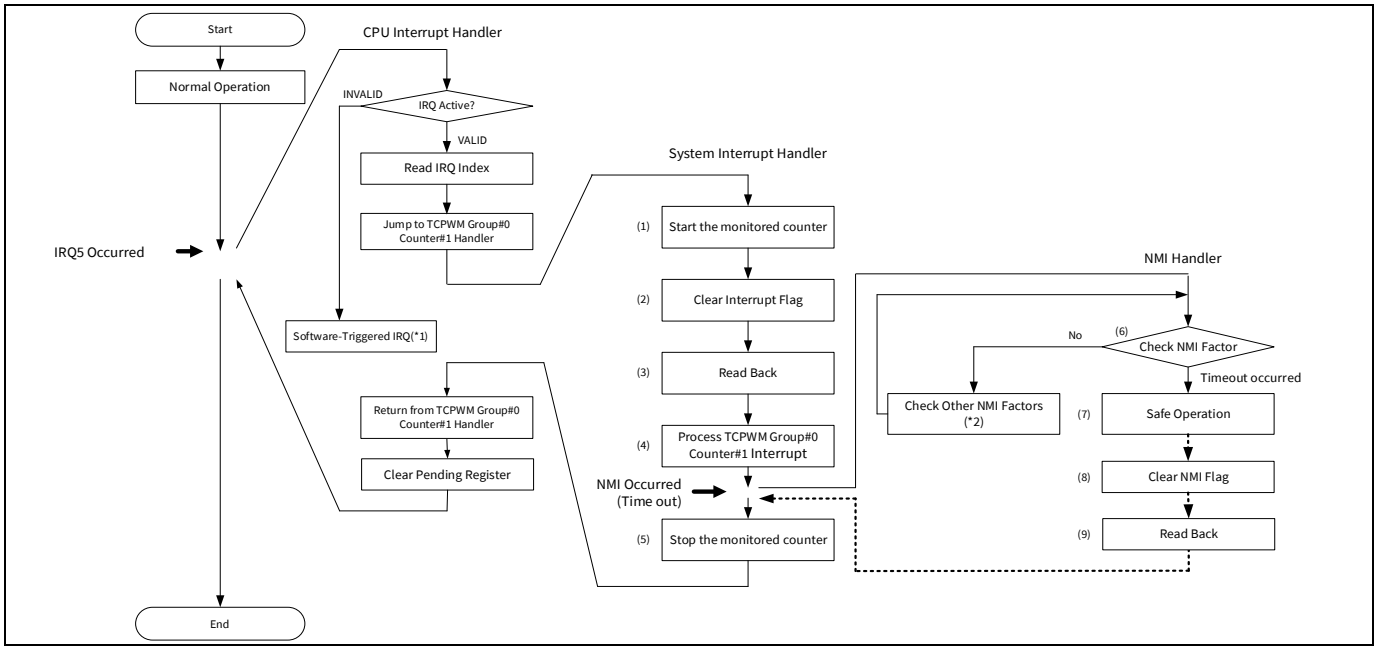

Figure 12 Timing protection violation operation flow example

*Note: (*1) The SYSTEM_INT_VALID bit is not set when a software interrupt occurs using the software-triggered interrupt register (STIR). See Software interrupt (using CPU register) for software interrupts.*

*Note: (*2) If there are multiple (four) NMI factors depending on the system, the NMI handler may need to investigate all selected system interrupt sources to identify the source of the CPU NMI.*

Programming hint: When an NMI occurs, execute the appropriate safe operation such as reset generation. If recovery to normal operation is possible after a safe operation, return from the interrupt.

2.6.3 Wakeup interrupt

All available system interrupts can be used in Active power mode and for wakeup from Sleep power mode.

A subset of the system interrupts can wake up the CPU from Deep Sleep; this subset can be used in Active power mode and to wake up the CPU from Sleep and Deep Sleep power modes. See the device datasheets for available interrupts details.

This section explains an interrupt from the backup domain caused by an interrupt from the real-time clock (RTC) and shows how to set and return processing from Deep Sleep mode through an interrupt.

2.6.3.1 Use case

- Interrupt setting:
  - System interrupt source: Backup domain (IDX: 12) ALARM1 interrupt of RTC
  - Mapped to CPU Interrupt: NvicMux7_IRQn
  - CPU interrupt priority: 6

Figure 13 shows interrupt operation for this configuration.

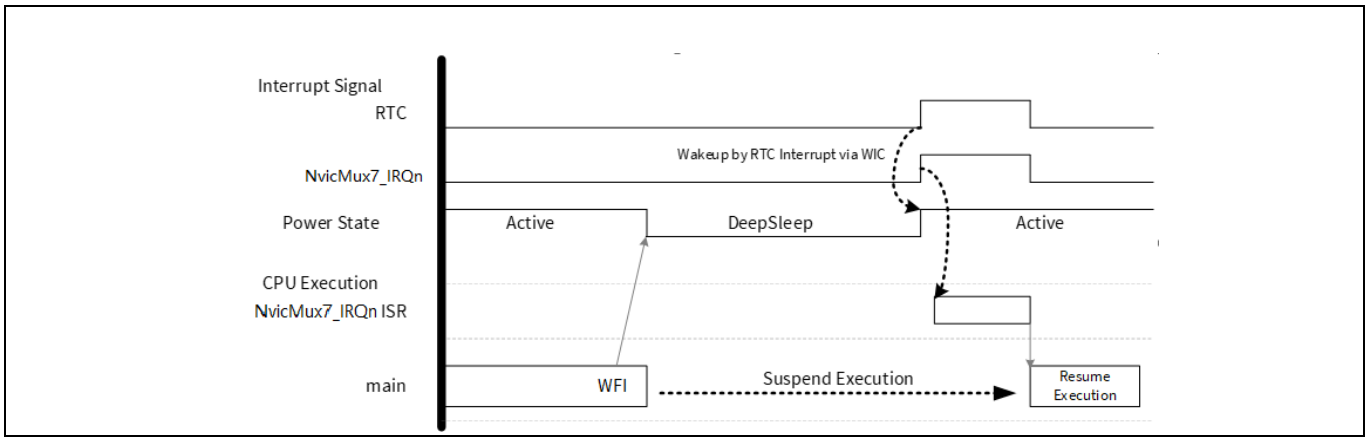

Figure 13 Example wakeup interrupt operation

The wait for interrupt (WFI) executed by the CPU triggers the transition into Sleep and Deep Sleep modes. WFI suspends the processor execution until an interrupt event occurs. When an interrupt from one or more wakeup sources occurs, the WIC generates a wakeup signal and places the CPU in Active mode.

Interrupt overview

When the CPU enters Active mode, the ISR is executed, and the suspended program is resumed after returning from the ISR.

2.6.3.2 Configuration

- Initial setting procedure:

  The initial setting for generating a wakeup interrupt using the WIC is the same as the normal interrupt setting. See the Initial setting for the initial setting procedure and the “Real-Time Clock” chapter of the architecture TRM for the RTC setting.
- Interrupt handler operation:

  See Interrupt handling for interrupt handler configuration. After returning from the Wakeup interrupt, the CPU resumes from the program suspend point.

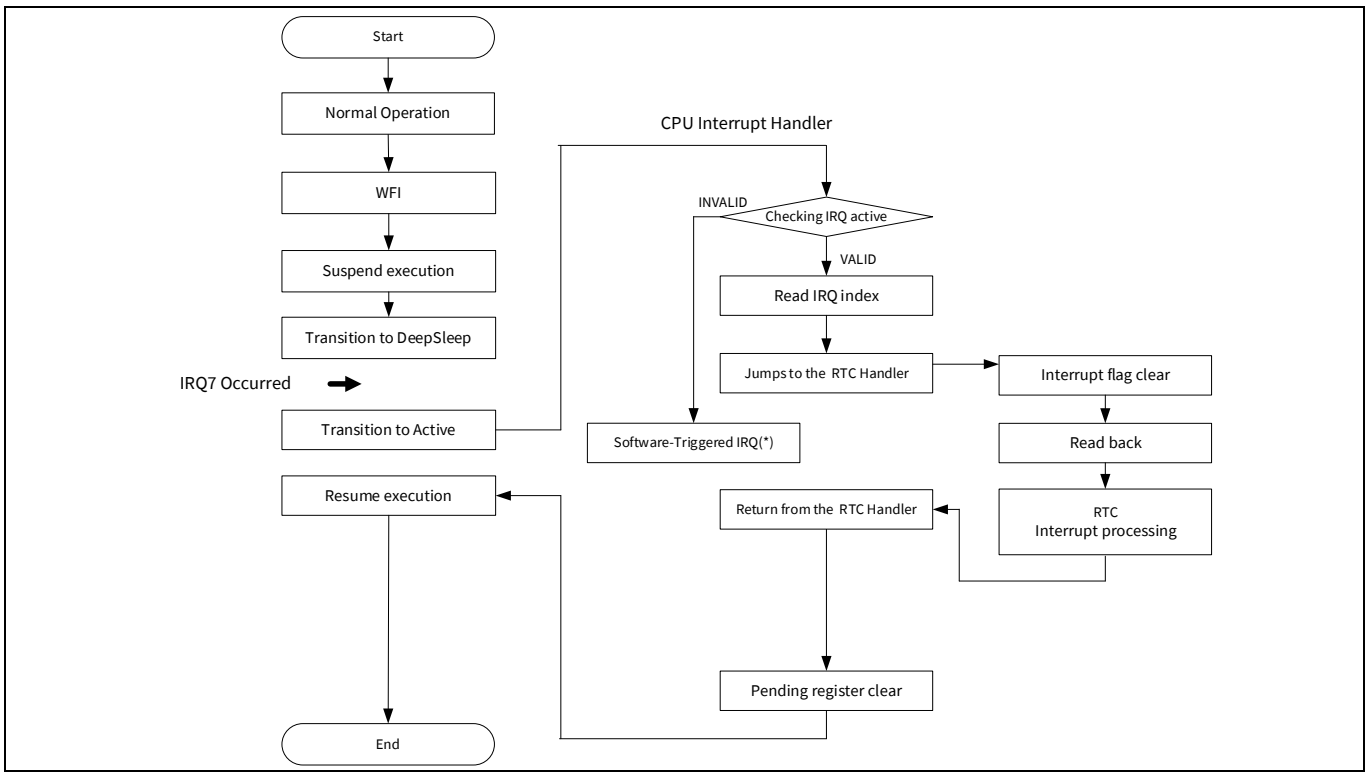

Figure 14 Wakeup interrupt handling example

Note: (*) The `SYSTEM_INT_VALID` bit is not set when a software interrupt occurs using the software-triggered interrupt register (STIR). See Software interrupt (using CPU register) for software interrupts.

2.6.4 Software interrupt (using CPU register)

This series can use Internal0_IRQn to Internal7_IRQn for a software interrupt. A software interrupt can be generated by writing ‘1’ to the software triggered interrupt register (STIR) of the corresponding CPU interrupts Internal0_IRQn to Internal7_IRQn. NvicMux0_IRQn to NvicMux7_IRQn can also trigger software interrupts with STIR.

This section explains software interrupt generation using the STIR register and shows the setting procedure and return processing.

2.6.4.1 Use case

- Interrupt setting:
  - Mapped to CPU interrupt: Internal0_IRQn
  - CPU interrupt priority: 3

Figure 15 shows interrupt operation for this configuration.

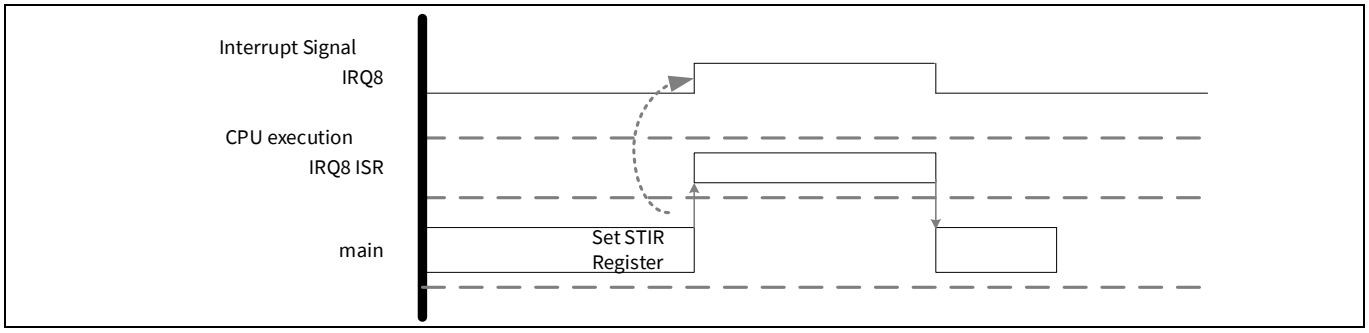

Figure 15 Software interrupt operation

2.6.4.2 Configuration

- Initial setting:

Software interrupt does not use interrupt structure, set only NVIC.

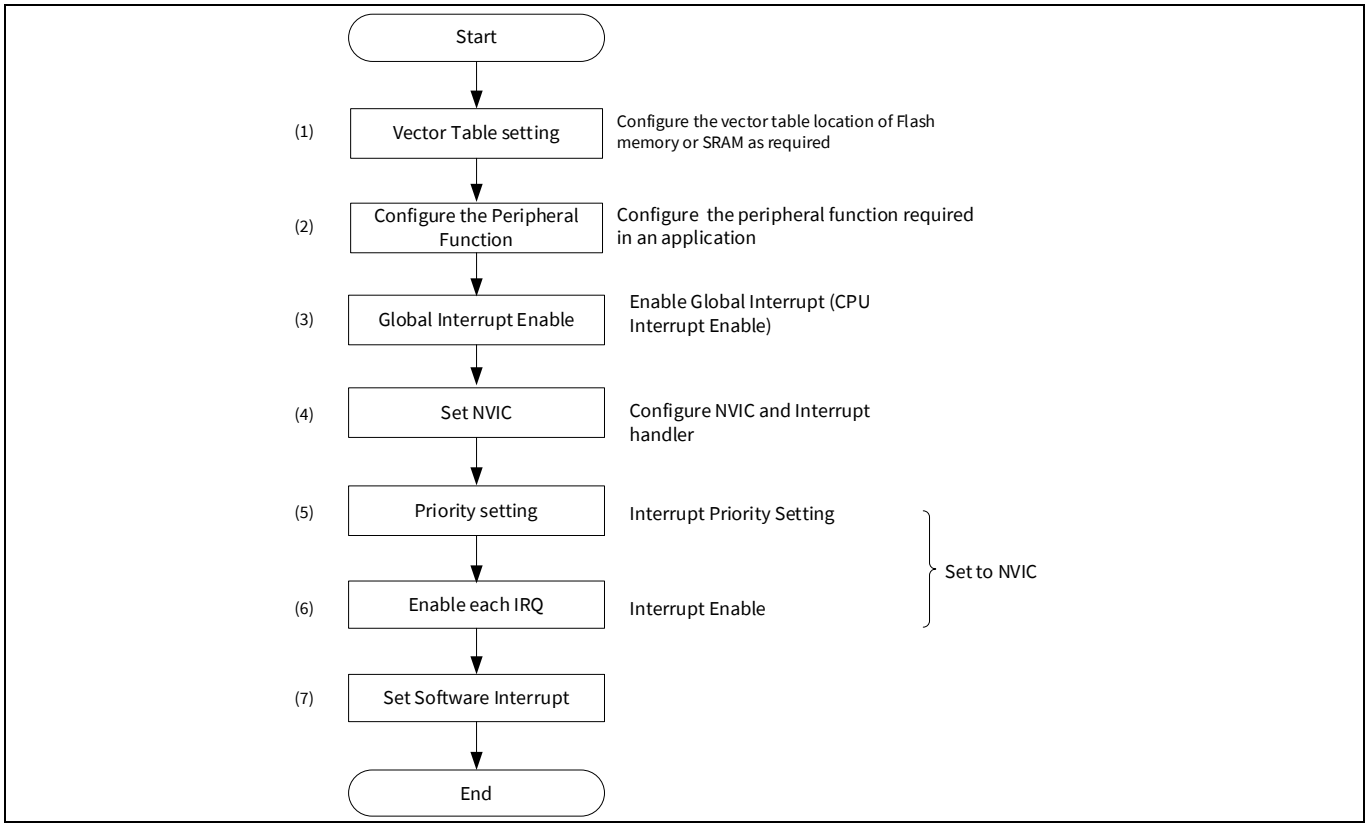

Figure 16 Initial setting procedure for software interrupt (using CPU register)

Table 5 and Table 6 list the parameters and functions of the configuration part in PDL for interrupt initialization.

Interrupt overview

Table 5 Interrupt initialization parameters

Parameters	Description	Value
`cfg.intrSrc`	Set the CPU interrupt number [8: 15]. Internal0_IRQn is assigned to IRQ 8, Internal1_IRQn is assigned to IRQ 9, Internal2_IRQn is assigned to IRQ 10, Internal3_IRQn is assigned to IRQ 11, Internal4_IRQn is assigned to IRQ 12, Internal5_IRQn is assigned to IRQ 13, Internal6_IRQn is assigned to IRQ 1 4, Internal7_IRQn is assigned to IRQ 15	Internal0_IRQn
`cfg.intrPriority`	Set Interrupt Priority	3ul

Table 6 Interrupt initialization functions

Function	Description	Value
`Cy_SysInt_InitIntIRQ (cfg, Handler)`	Configure the interrupt structure. cfg: Software interrupt parameters address Handler: Interrupt handler address	cfg: cfg address Handler: SoftwareInterupt0Handler, See Code Listing 15.
`NVIC_SetPriority(intrSrc, intPriority)`	Set the interrupt priority. intrSrc: CPU interrupt number intPriority: Interrupt priority level	intrSrc: Internal0_IRQn intPriority: 3ul
`NVIC_EnableIRQ(intSrc)`	Set interrupt enable. intSrc: CPU interrupt number	`Internal0_IRQn`
`Cy_SysInt_SoftwareTrig(intSrc)`	Set software interrupt. intSrc: CPU interrupt number	`Internal0_IRQn`

Code Listing 11 shows an example program of the interrupt configuration part for software interrupt. See the Initial setting for configuration of (1).

Code Listing 11 Example for software interrupt configuration

```
void SoftwareInteruptOHandler(void)
{
    __NOP();
}

int main(void)
{
    SystemInit();

    __enable_irq(); /* Enable global interrupts. */

    /* Software Interrupt 0 */
    cy_stc_sysint_t cfg =
    {
        .intrSrc = Internal0_IRQn,
        .intrPriority = 3ul,
    };
    Cy_SysInt_InitIntIRQ(&cfg, SoftwareInteruptOHandler);

        /* Enable software interrupt IRQ */
    NVIC_EnableIRQ(Internal0_IRQn);
    /* Force set pending status in the NVIC */
    Cy_SysInt_SoftwareTrig(Internal0_IRQn);

    for(;;)
    {
    }
}
```

Code Listing 12 Cy_SysInt_InitIntIRQ() function

```
cy_en_sysint_status_t Cy_SysInt_Init(const cy_stc_sysint_t * config,
cy_israddress userIsr)
{
    cy_en_sysint_status_t status = CY_SYSINT_SUCCESS;

    if(NULL != config)
    {
        CY_ASSERT_L3(CY_SYSINT_IS_PRIORITY_VALID(config->intrPriority));

        /* Only set the new vector if it is CPU interrupt (not internal SW
interrupt) */
        if (config->intrSrc >= Internal0_IRQn && config->intrSrc <=
Internal7_IRQn)
        {
            NVIC_SetPriority(config->intrSrc, config->intrPriority);

            /* Only set the new vector if it was moved to __ramVectors */
            if (SCB->VTOR == (uint32_t)&__ramVectors)
            {
                (void)Cy_SysInt_SetVector(config->intrSrc, userIsr);
            }
        }
        else
        {
            return CY_SYSINT_BAD_PARAM; // Vector of system handler have to be
set at ROM table.
        }

    }
    else
    {
        status = CY_SYSINT_BAD_PARAM;
    }

   return(status);
}
```

Code Listing 13 Cy_SysInt_SetVector() function

```
cy_israddress  Cy_SysInt_SetVector(IRQn_Type IRQn, cy_israddress userIsr)
{
    cy_israddress prevIsr;

    /* Set the new vector only if it was moved to __ramVectors */
    if (SCB->VTOR == (uint32_t)&__ramVectors)
    {
        CY_ASSERT_L1(CY_SYSINT_IS_VECTOR_VALID(userIsr));

        prevIsr = __ramVectors[CY_INT_IRQ_BASE + (uint32_t)IRQn];
        __ramVectors[CY_INT_IRQ_BASE + (uint32_t)IRQn] = userIsr;

        #if defined (CY_IP_M7CPUSS)
            #if (CY_CPU_CORTEX_M7)
```

Code Listing 13 Cy_SysInt_SetVector() function

```
                // Ensure that above change in the vector table is cleaned from
Data Cache,
                // and Instruction Cache is invalidated, so that CPU fetches the
correct address
                SCB_CleanDCache_by_Addr((uint32_t*)&__ramVectors[CY_INT_IRQ_BASE
+ IRQn], 4);
                SCB_InvalidateICache();
            #endif
        #endif
    }
    else
    {
        prevIsr = __Vectors[CY_INT_IRQ_BASE + (uint32_t)IRQn];
    }

    return (prevIsr);
  }
```

Code Listing 14 Cy_SysInt_SoftwareTrig() function

```
 void Cy_SysInt_SoftwareTrig(IRQn_Type IRQn)
{
    NVIC->STIR = (uint32_t)IRQn & CY_SYSINT_STIR_MASK;
}
```

- Interrupt handling:

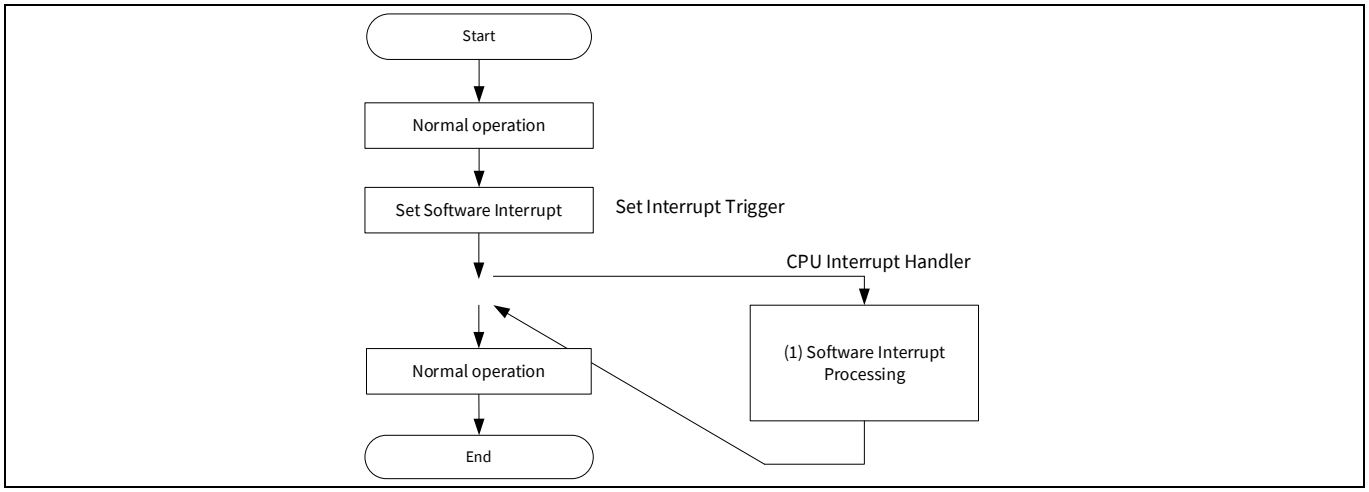

Figure 17 Software interrupt handling flow example

The pending flag is cleared by handler transition. Therefore, when a software interrupt is generated using the set-pending registers, you do not need to clear the flag after the interrupt handler occured.

Code Listing 15 shows an example of a CPU interrupt handler for a software interrupt.

Code Listing 15 CPU interrupt handler for software interrupt

```
void SoftwareInterupt0Handler(void)
{
    __NOP();
  }
```

Software interrupt processing

2.6.5 Software interrupt (using peripheral)

This series can generate the software interrupts by utilizing unused peripherals. A software interrupt with peripherals uses the INTR_SET register in the peripheral interrupt structure. INTR_SET sets the corresponding bit in the INTR register by software writing ‘1’.

This section explains the software interrupt generation by using peripherals and shows the setting procedure and return processing. CM7 processes the interrupt.

2.6.5.1 Use case

- Interrupt setting:
  - System interrupt source: TCPWM0 Group#0 Counter#1(IDX:520) TC interrupt
  - Mapped to CPU interrupt: IRQ4
  - CPU interrupt priority: 3

In this type of interrupt, the timer function of TCPWM0 Group#0 Counter#1 is unused.

Figure 18 shows interrupt operation for this configuration.

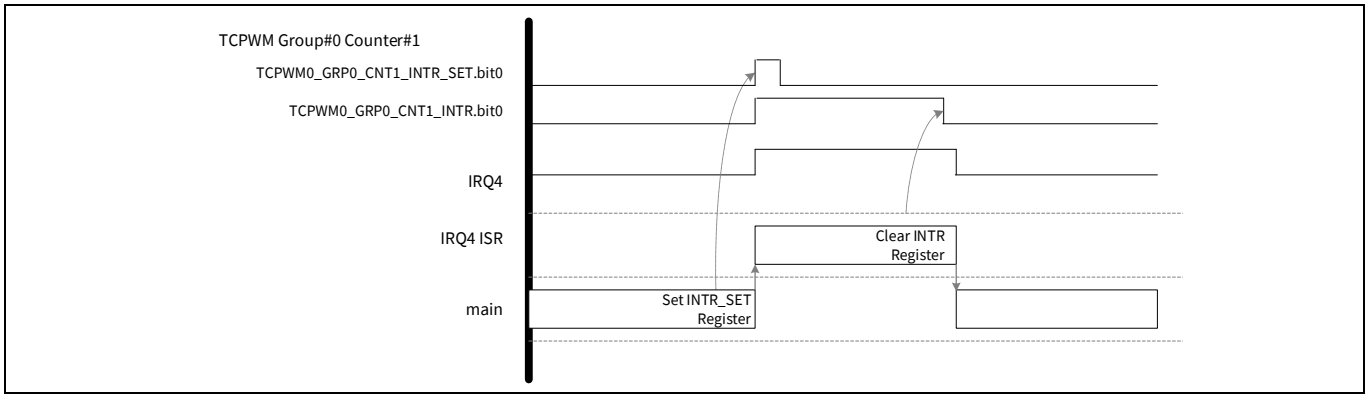

Figure 18 Software interrupt operation example by utilizing unused TCPWM

2.6.5.2 Configuration

- Initial setting procedure:

  For the initial setting procedure, see the Initial setting. After each peripheral resource (TCPWM0 Group#0 Counter#1) is set up, each resource interrupt is routed to the CPU interrupt. Set the level and permission of each CPU interrupt to the NVIC. After setting up the interrupt connection, start each resource. See the “Timer, Counter, and PWM” chapter of the architecture TRM for the TCPWM setting.

Interrupt overview

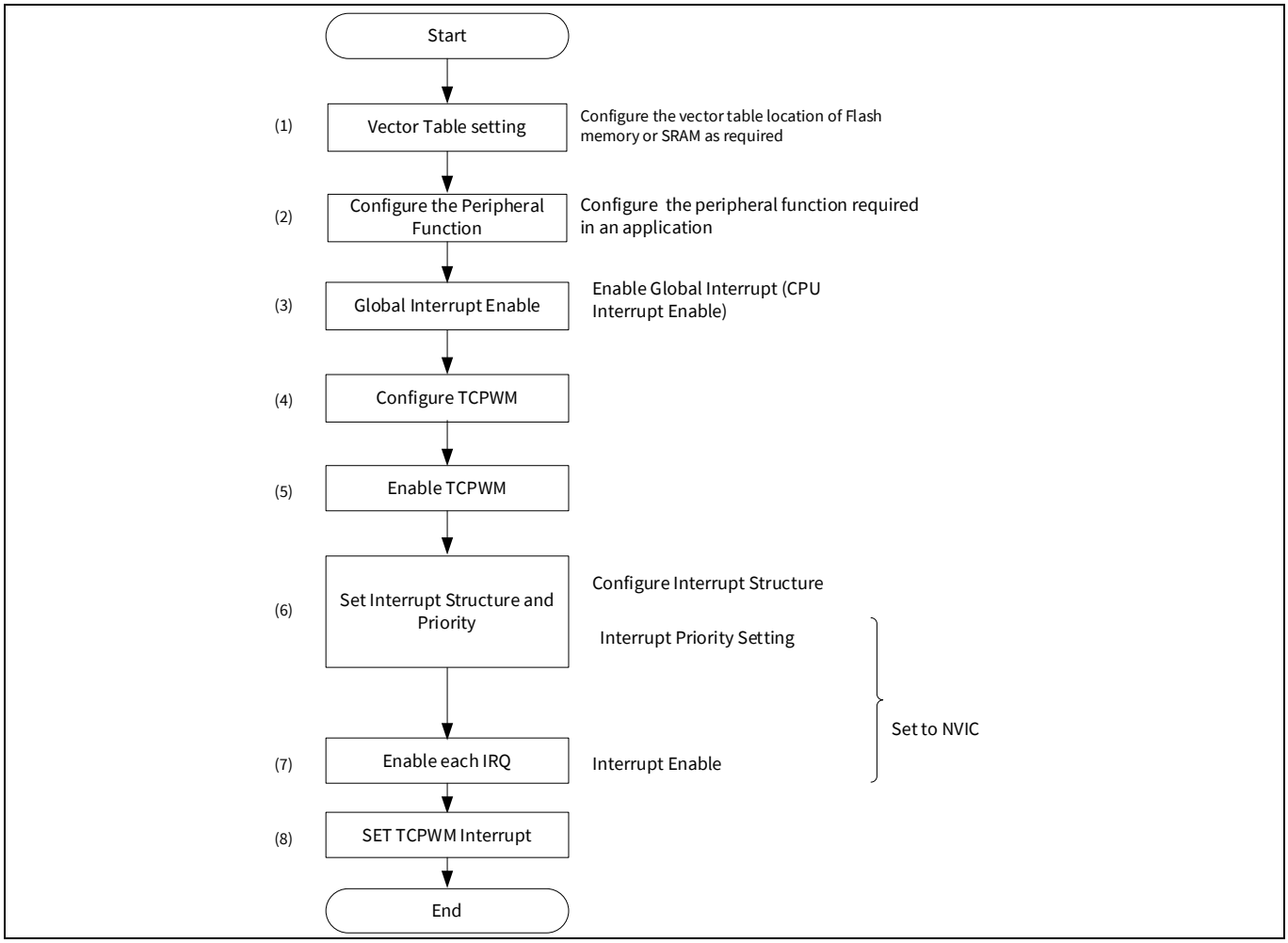

Figure 19 Initial setting procedure for software interrupt (using peripheral)

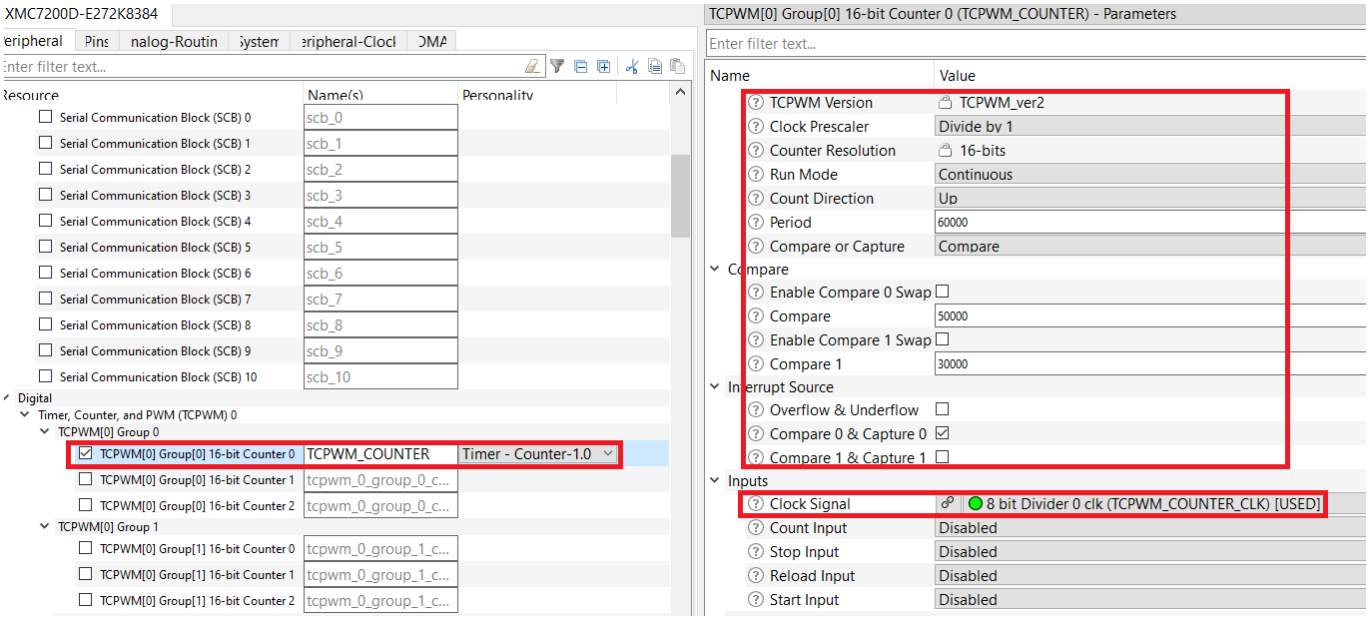

Figure 20 TCPWM timer configuration in device configurator tool

Interrupt overview

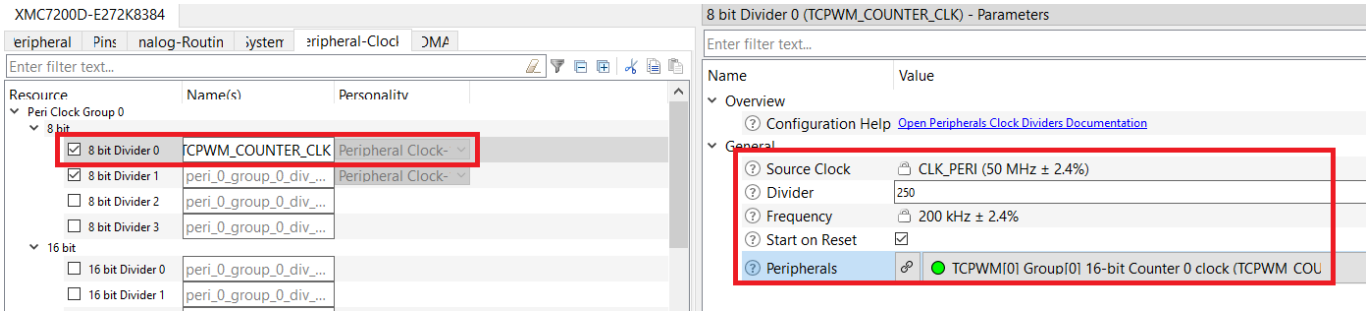

Figure 21 TCPWM timer clock configuration in device configurator tool

Code Listing 16 XMC7200: Example of TCPWM counter configuration

```
const cy_stc_tcpwm_counter_config_t TCPWM_COUNTER_config =
{
    .period = 60000,
    .clockPrescaler = CY_TCPWM_COUNTER_PRESCALER_DIVBY_1,
    .runMode = CY_TCPWM_COUNTER_CONTINUOUS,
    .countDirection = CY_TCPWM_COUNTER_COUNT_UP,
    .compareOrCapture = CY_TCPWM_COUNTER_MODE_COMPARE,
    .compare0 = 50000,
    .compare1 = 16384,
    .enableCompareSwap = false,
    .interruptSources = (CY_TCPWM_INT_ON_TC & 0U) | (CY_TCPWM_INT_ON_CC0 ) |
    (CY_TCPWM_INT_ON_CC1 & 0U),
    .captureInputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .captureInput = CY_TCPWM_INPUT_0,
    .reloadInputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .reloadInput = CY_TCPWM_INPUT_0,
    .startInputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .startInput = CY_TCPWM_INPUT_0,
    .stopInputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .stopInput = CY_TCPWM_INPUT_0,
    .countInputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .countInput = CY_TCPWM_INPUT_1,
    .capture1InputMode = TCPWM_COUNTER_INPUT_DISABLED & 0x3U,
    .capture1Input = CY_TCPWM_INPUT_0,
    .compare2 = 30000,
    .compare3 = 16384,
    .enableCompare1Swap = false,
    .trigger0Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,
    .trigger1Event = CY_TCPWM_CNT_TRIGGER_ON_DISABLED,
 };
```

Code Listing 17 shows an example program of the interrupt configuration part for software interrupt. See the Initial setting for Configuration of (1).

Code Listing 17 XMC7200: Example of interrupt configuration

```
#define CC0_INTERRUPT_PRIORITY (7u)

static void cc0_interrupt_handler_pdl();

cy_stc_sysint_t intrCfg =
{
    .intrSrc = ((NvicMux3_IRQn << 16) | TCPWM_COUNTER_IRQ),    /* Interrupt source
is tcpwm_0_interrupts_0_IRQn */
```

Code Listing 17 XMC7200: Example of interrupt configuration

```
    .intrPriority = CC0_INTERRUPT_PRIORITY                    /* Interrupt priority
is 7 */
};

int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */    __enable_irq();

    /*Initialize the User LED*/
    Cy_GPIO_Pin_Init(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN,
&CYBSP_USER_LED1_config );

    /*TCPWM Counter Mode initial*/
    if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(TCPWM_COUNTER_HW,
                            TCPWM_COUNTER_NUM, &TCPWM_COUNTER_config))
    {
        CY_ASSERT(0);
    }

    /* Enable the initialized counter */
    Cy_TCPWM_Counter_Enable(TCPWM_COUNTER_HW, TCPWM_COUNTER_NUM);

    /* Configure and enable Counter CC0 interrupt */
    Cy_SysInt_Init(&intrCfg, cc0_interrupt_handler_pdl);
    NVIC_ClearPendingIRQ((IRQn_Type)intrCfg.intrSrc);
    NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);

    /* Set INTR_SET register */
    Cy_TCPWM_SetInterrupt(TCPWM_COUNTER_HW,TCPWM_COUNTER_NUM, TCPWM_COUNTER_IRQ);
    for (;;)
    {
      /*empty loop*/
    }
}
```

Code Listing 18 Cy_TCPWM_SetInterruptMask () function

```
__STATIC_INLINE void Cy_TCPWM_SetInterruptMask(TCPWM_Type *base, uint32_t cntNum,
uint32_t mask)
{
#if    (CY_IP_MXTCPWM_VERSION == 1U)

        TCPWM_CNT_INTR_MASK(base, cntNum) = mask;
#else
        TCPWM_GRP_CNT_INTR_MASK(base, TCPWM_GRP_CNT_GET_GRP(cntNum), cntNum) =
mask;
#endif
}
```

- Interrupt handling:

  In this case, the start of interrupts is triggered by the software. However, interrupt handling is similar to that of peripheral interrupts. See the Interrupt handling for the interrupt handler configuration.

  When an interrupt occurs, the CPU jumps to the interrupt handler specified in the vector table. The CPU interrupt handler identifies the system interrupt that triggered the CPU interrupt and jumped to the corresponding ISR. After clearing the system interrupt flag in the CPU interrupt handler and executing the ISR, clear the relevant CPU Pending register bit and return to the main routine.

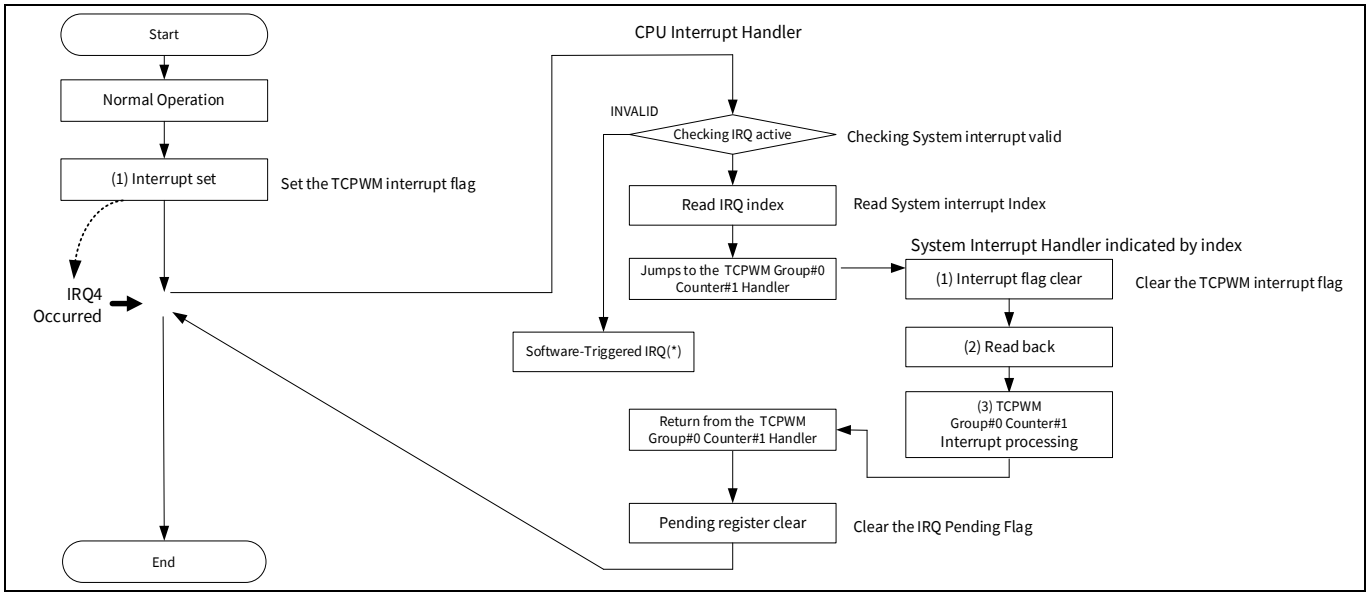

Figure 22 Software interrupt handling by utilizing unused TCPWM

Note: (*) The SYSTEM_INT_VALID bit is not set when a software interrupt occurs using the software-triggered interrupt register (STIR). See Software interrupt (using CPU register) for software interrupts.

Code Listing 19 shows an example of the system interrupt handler. See Code Listing 5 for the CPU Interrupt handler.

Interrupt overview

Code Listing 19 Example of system interrupt handler

```
static void cc0_interrupt_handler_pdl()
{
    /*toggle user LED1*/
    Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN);

    /*clear the counter CC0 compare interrupt*/
    Cy_TCPWM_ClearInterrupt(TCPWM_COUNTER_HW, TCPWM_COUNTER_NUM,
CY_TCPWM_INT_ON_CC0);
  }
```

Code Listing 20 Cy_TCPWM_ClearInterrupt ()

```
__STATIC_INLINE void Cy_TCPWM_ClearInterrupt(TCPWM_Type *base, uint32_t cntNum,
uint32_t source)
{
#if    (CY_IP_MXTCPWM_VERSION == 1U)

        TCPWM_CNT_INTR(base, cntNum) = source;
        (void)TCPWM_CNT_INTR(base, cntNum);
#else
        TCPWM_GRP_CNT_INTR(base, TCPWM_GRP_CNT_GET_GRP(cntNum), cntNum) = source;
        (void)TCPWM_GRP_CNT_INTR(base, TCPWM_GRP_CNT_GET_GRP(cntNum), cntNum);
#endif
}
```

3 Fault report structure overview

3.1 Fault report structure

The fault report structures encapsulate information about transient faults while the software is running. Fault report structures store the information about the faults and can cause a reset. The platform uses centralized fault report structures. This centralized nature allows for system-wide, consistent handling of faults, which simplifies software development because of the following reasons:

- Only a single fault interrupt handler is required
- A fault report structure provides the fault source and additional fault-specific information from a single set of registers; that is, no iterative search for the fault source and fault information is required.
- All pending faults are available from a single set of registers

Fault report structures capture faults such as MPU/SMPU/PPU protection violations, memory-specific errors such as ECC errors, peripheral-specific errors, and so on. Fault report structures capture only faults.

Figure 23 shows a block diagram of the fault report structure.

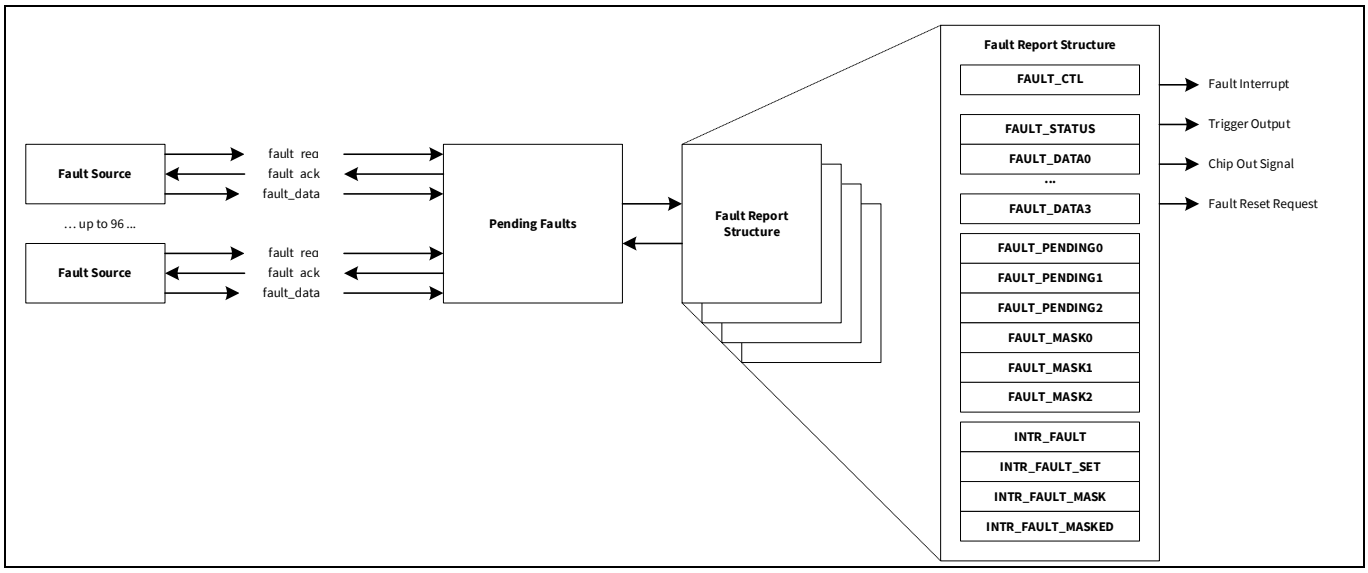

Figure 23 Fault report structure

This MCU series have four-fault report structures. Each fault report structure has a dedicated set of control and status registers and captures a single fault.

When the fault report structure captures fault, it provides the following information:

- A flag that indicates a fault is captured
- A fault index that identifies the fault source
- Additional fault information describing fault specifics

For the fault and additional information captured by fault report structures, see the architecture TRM.

When the fault report structure captures a fault, it generates the following notification signals:

- Fault interrupt
- Trigger output

Fault report structure overview

- Chip output signal
- Fault reset request

Each output signal can be selected as enabled or disabled with a register. A fault interrupt can be notified to the CPU as a system interrupt.

The pending structure holds the faults that are not captured by the fault report structure. The associated pending bit is cleared to ‘0’ when a fault report structure captures a pending fault. The fault report structure functionality is available only in Active and Sleep power modes.

4 Glossary

Table 7 Glossary

Terms	Description
CPU	Central processing unit
NVIC	Nested vectored interrupt controller
WIC	Wakeup interrupt controller
NMI	Non-maskable interrupt
Deep Sleep	Low-power mode in XMC7000 family series. See the “Device Power Mode” chapter of the architecture TRM for details.
ISR	Interrupt service routine
IRQ	Interrupt request
WFI	Wait for interrupt
MMIO	Memory-mapped I/O
MPU	Memory protection unit. See the “Protection Unit” chapter of the architecture TRM for details.
SMPU	Shared memory protection unit. See the “Protection Unit” chapter of the architecture TRM for details.
PPU	Peripheral protection unit. See the “Protection Unit” chapter of the architecture TRM for details.
GPIO	General-purpose input/output. See the “I/O System” chapter of the architecture TRM for details.
TCPWM	Timer, counter, and pulse width modulator. See the “Timer, Counter, and PWM” chapter of the architecture TRM for details.
RTC	Real-time clock. See the “Real-time Clock” chapter of the architecture TRM for details.

5 Related documents

- Contact Technical support to obtain XMC7000 family references documents.
- **Device datasheets:**
  - XMC7100 series 32-bit Arm® Cortex®-M7 microcontroller datasheet
  - XMC7200 series 32-bit Arm® Cortex®-M7 microcontroller datasheet
- **Technical reference manuals:**
  - XMC7000 MCU architecture technical reference manual
  - XMC7100 MCU registers Technical reference manual
  - XMC7200 MCU registers Technical reference manual
- **Application notes:**
  - AN234118 - GPIO usage setup in XMC7000 family
  - AN234119– How to use Timer, Counter, and PWM (TCPWM) usage in XMC7000 family

Revision history

Document version	Date of release	Description of changes
**	2021-12-20	Initial release.
*A	2022-06-15	Updated code style with the latest PDL.
*B	2023-04-10	• Updated links • Compared code with PDL 3.3.0, updated TCPWM code example • Design for verifying and updating NMI, wakeup, and software interrupt (using peripheral) code examples • Removed bubble boxes from all images for a unified style

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

Edition 2023-04-10

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-34226 Rev. *B

Important notice

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

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

Warnings

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

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