How to Use Interrupt in Traveo II

About this document

Scope and purpose

This application note explains how to set up and use interrupts for Traveo™ II family MCU. This document serves as a guide for developing interrupt-based projects. It also explains the interrupt structure, vector selection, and provides guidance on coding interrupt handlers. In addition, it explains how to set up and use the fault report structure that capture the fault information.

Intended audience

This document is intended for anyone who uses the Traveo II family to use interrupt function.

Table of contents

1 Introduction

This application note describes interrupts in Traveo II family series MCUs. 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.

The CYT2 series has one Arm Cortex-M4F-based CPU (CM4) and Cortex-M0+-based CPU (CM0+). The CYT4 series has two Arm Cortex-M7-based CPUs (CM7) and CM0+, and the CYT3 series has one Arm Cortex-M7-based CPUs (CM7) and CM0+.

Interrupts are an important part of any embedded application. They free the CPU from having to continuously poll for the occurrence of a specific event; it notifies the CPU only when that event occurs. The occurrence of an interrupt results in the current program flow being stopped and the interrupt service routine (ISR) being executed by the CPU.

In embedded applications, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU. They are also used for notifying when errors are detected from 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 **[1]**.

The fault report structures capture faults that occur while software is running. The Traveo II platform uses a centralized fault report structure, which allows for a 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” chapter of the Architecture Technical Reference Manual (Architecture TRM) **[2]**.

2 Interrupt Overview

2.1 Interrupt Structure

Figure 1 shows a simplified block diagram of the interrupt architecture in CYT2B series.

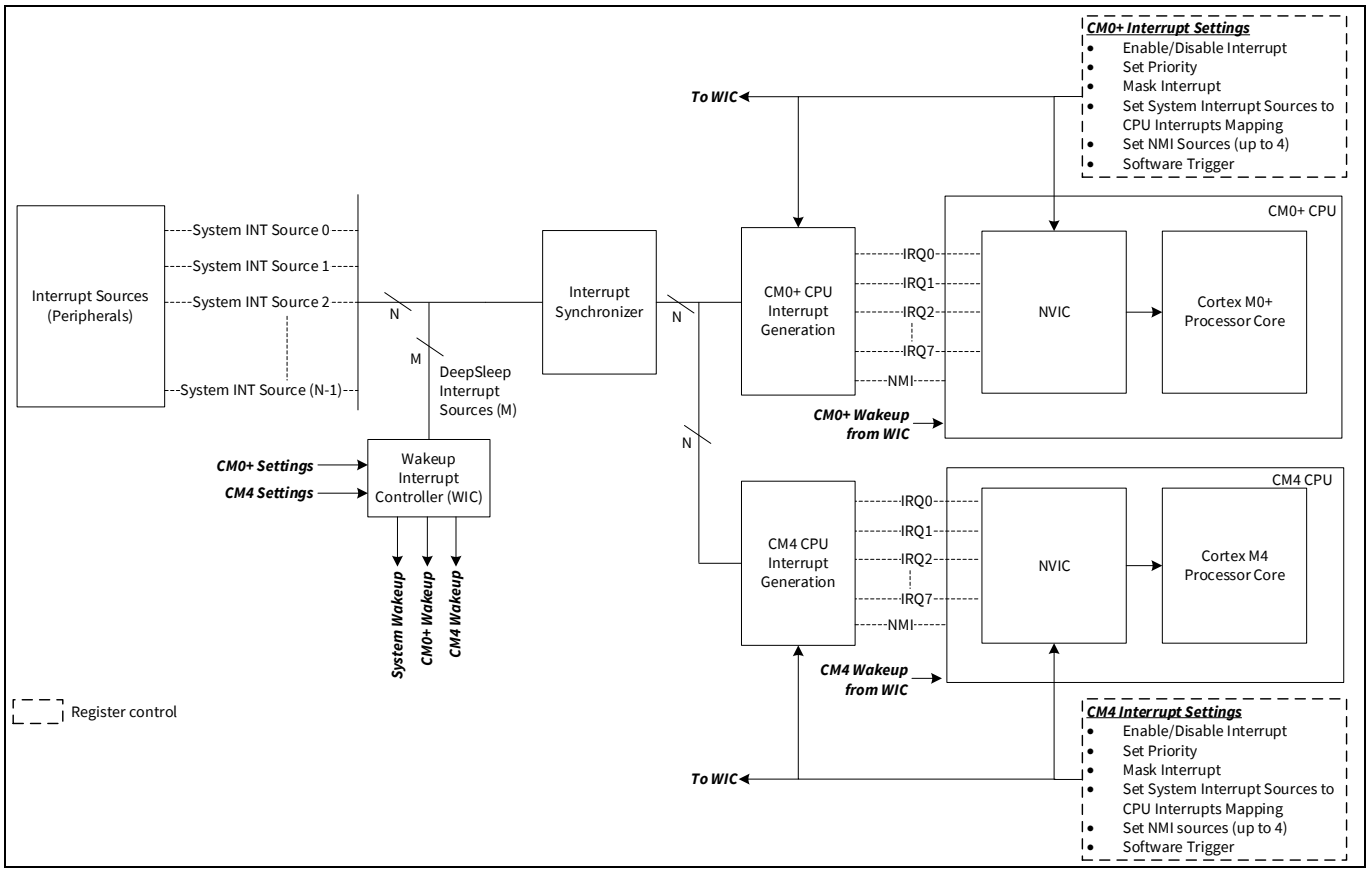

Figure 1 CYT2B Series Interrupt Architecture

See the Architecture TRM [2] for other series interrupt architecture.

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

In the Traveo II interrupt architecture, each CPU can use eight CPU interrupts IRQ[7:0] and any of the 'N' system interrupts can be mapped to any of the IRQ[7:0] of each CPU. All system interrupts can be mapped onto any CPU interrupt simultaneously and the interrupt can be mapped independent for both CPUs.

Multiple system interrupts can be mapped on the same CPU interrupt. Therefore, an active CPU interrupt may indicate one or multiple active system interrupts.

For each interrupt, there are eight levels of configurable priority levels for CM4 and CM7, and four levels for CM0+.

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

The WIC provides detection of DeepSleep interrupts in the DeepSleep CPU power mode. When interrupt signal of one or more wakeup sources detected, the WIC generates a wakeup signal, and causes the CPU to enter Active mode. See the Device Datasheet [1] for the interrupt sources available for DeepSleep wakeup.

Interrupt Overview

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.

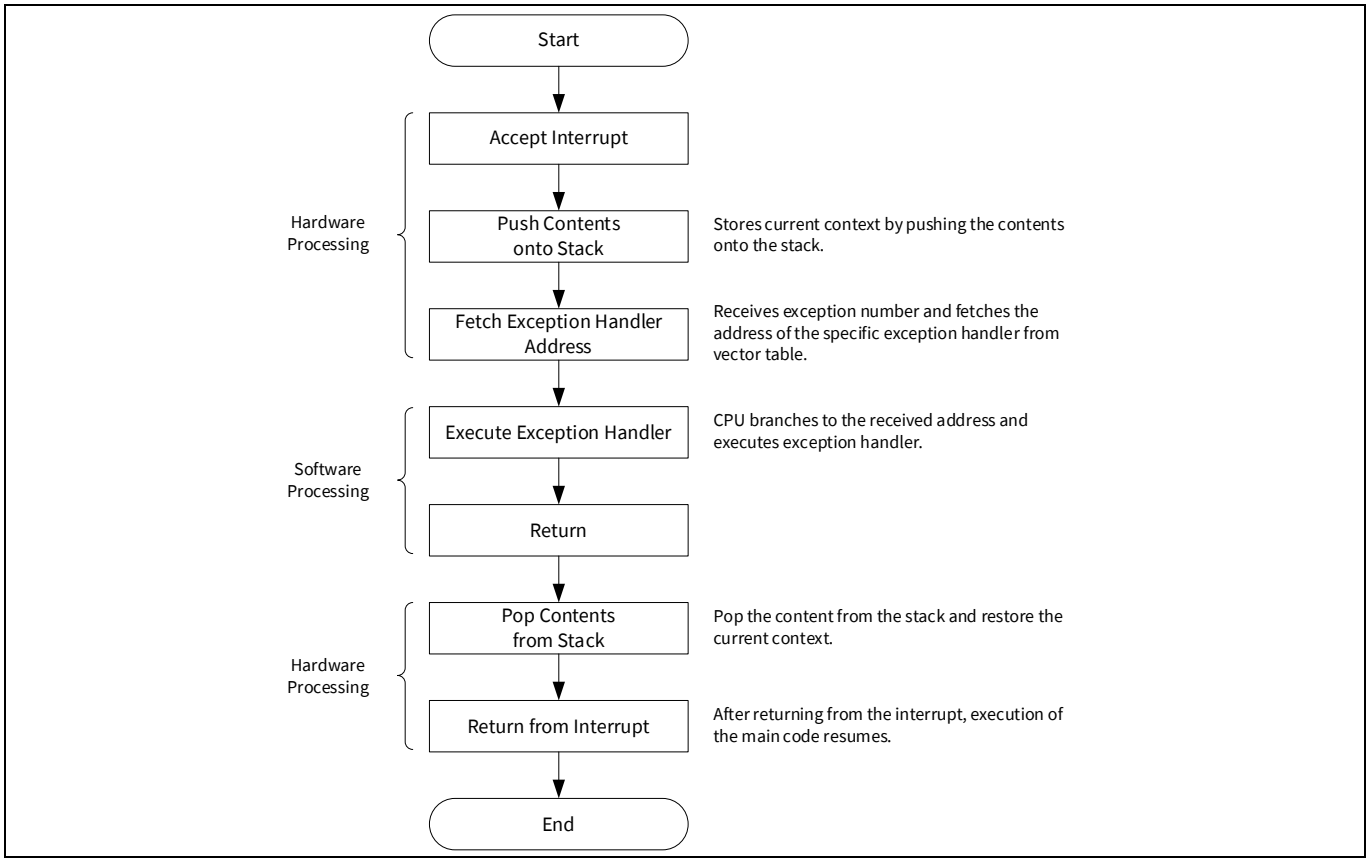

Figure 2 Interrupt Sequence

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 these interrupts, and sends the exception number of the highest-priority interrupt to the CPU. Therefore, a higher priority interrupt can block the execution of a lower-priority Interrupt processing.

Figure 3 shows the interrupt operation by interrupt priority.

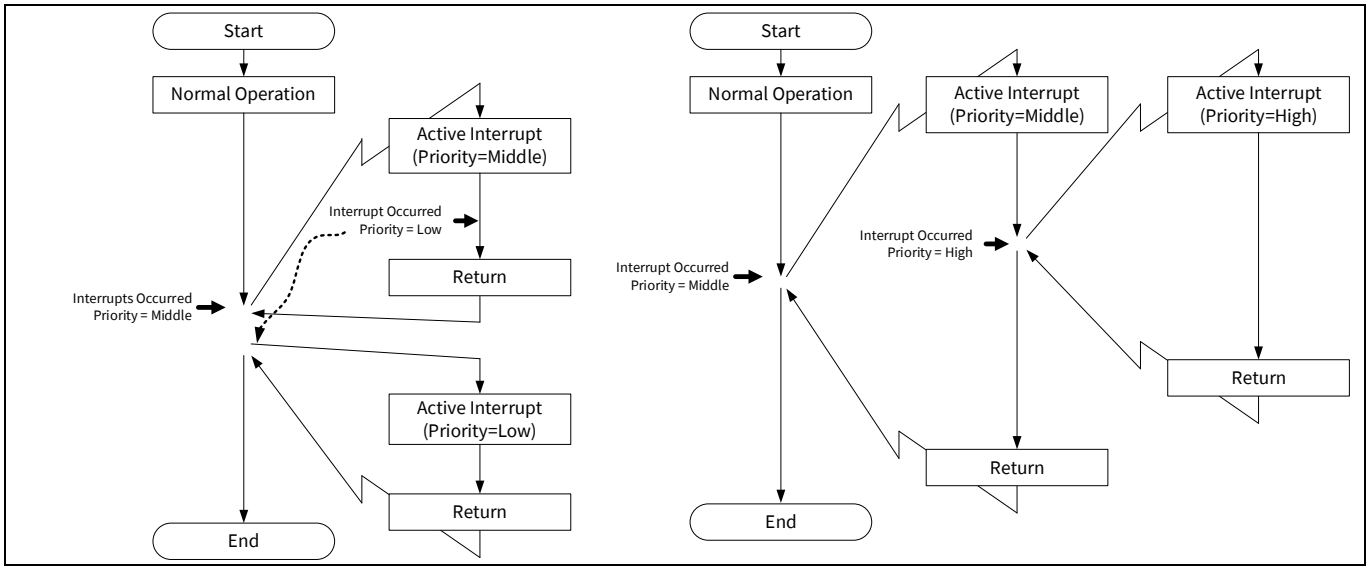

Figure 3 Operation when Interrupt Occurs During Interrupt Processing

For more details, see the Arm documentation sets for **CM4**, **CM7**, and **CM0+**.

2.3 Initial Setting

This section describes how to initialize interrupts using 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 4** shows the initializing interrupts.

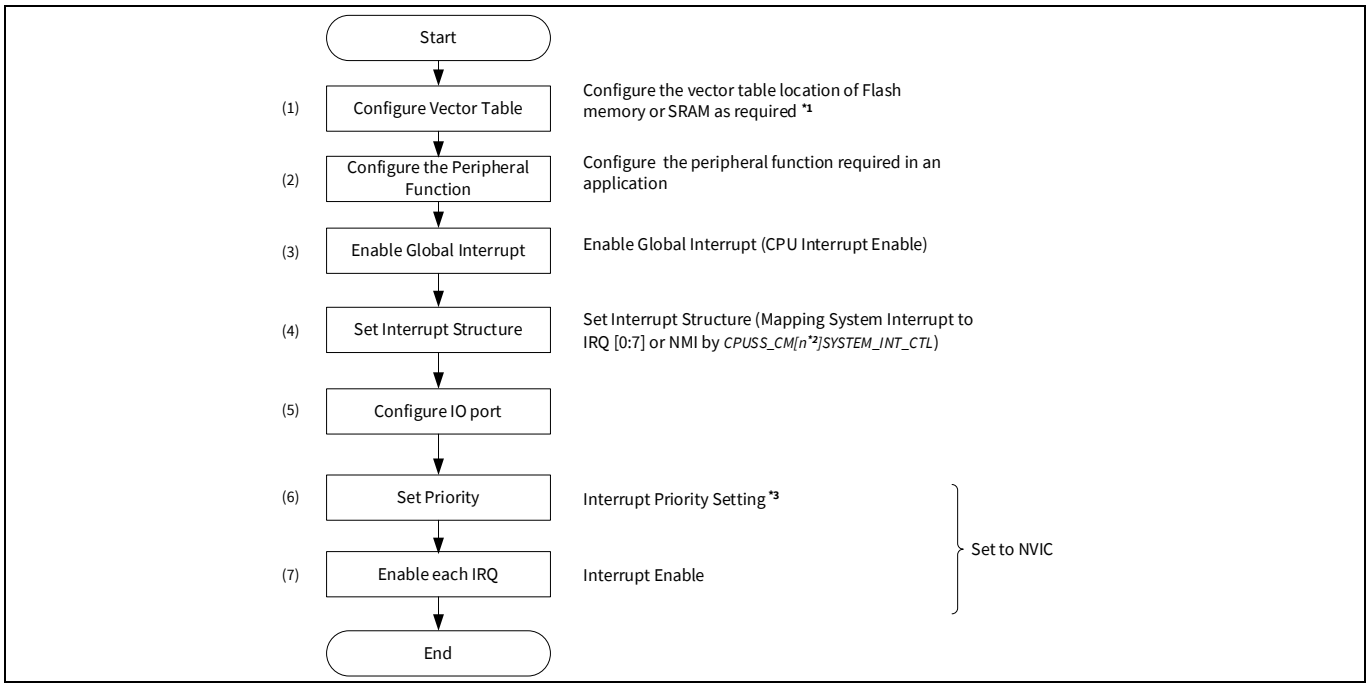

Figure 4 Interrupt Initial Setting Flow

Interrupt Overview

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

*Note: (*2) "n" in **Figure 4** indicates 0, 4. For example, CPUSS_CM4_SYSTEM_INT_CTL shows system interrupt control register for CM4. The CYT2 series have CPUSS_CM4_SYSTEM_INT_CTL and CPUSS_CM0_SYSTEM_INT_CTL registers. The CYT3 and CYT4 series have CPUSS_CM7_0_SYSTEM_INT_CTL, CPUSS_CM7_1_SYSTEM_INT_CTL (only CYT4 series) and CPUSS_CM0_SYSTEM_INT_CTL registers. See the Registers TRM [2] for more information.*

*Note: (*3) Interrupt priority can set with the 8-bit PRI_N field in the NVIC_IPR register. The correspondence between interrupt numbers and register bit fields can be expressed by the following equation:*
*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 flash to SRAM, thus allowing the change of interrupt handlers dynamically. When VTOR was set to SRAM area, vector table need to place in SRAM.

2.3.1 Use case

This section explains an example of the configure interrupt structure using the following use case.

In this configuration, GPIO Port 8_0 is connected to the button switch. An interrupt is generated by pressing the switch. IDX numbers used this section denote the system interrupt index numbers. In this usage, the IDX number is that of the CYT2B series. See the Device Datasheet [1] for the actual index number to use. See the "I/O System" chapter in Architecture TRM [2] and Application Note [3] for GPIO setting.

- System Interrupt source: GPIO Port Interrupt#8 (IDX: 29)
- Mapped to CPU Interrupt: IRQ3
- CPU Interrupt Priority: 3

2.3.2 Configuration

Table 1 lists the parameters and functions of the configuration part in SDL for interrupt initialization.

Table 1 List of Interrupt Initialization Parameters and Functions

Parameters	Description	Value
irq_cfg.sysIntSrc	Set system interrupt index number [0: 1022]	CY_BUTTON3_IRQN It is assigned to Port 8 (Index number = 29)
irq_cfg.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4, CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7,	CPUIntIdx3_IRQn
irq_cfg.isEnabled	Set interrupt enable False: Disabled True: Enabled	True
Function	**Description**	**Value**
Cy_SysInt_InitIRQ(irq_cfg)	Configure interrupt structure Irq_cfg: Interrupt structure parameters address	&irq_cfg
Cy_SysInt_SetSystemIrqVector (sysintSrc, Handler)	Set system interrupt handler to vector table sysintSrc: System interrupt index number Handler: Interrupt handler address	sysintSrc: CY_BUTTON3_IRQN Handler: ButtonIntHandler, See **Code Listing 6**.
NVIC_SetPriority(intSrc, intPriority)	Set interrupt priority: intSrc: CPU interrupt number intPriority: Interrupt priority Level	intSrc: CPUIntIdx3_IRQn intPriority: 3ul
NVIC_ClearPendingIRQ(intSrc)	Clear pending flag: intSrc: CPU interrupt number	CPUIntIdx3_IRQn
NVIC_EnableIRQ(intSrc)	Set interrupt enable intSrc: CPU interrupt number	CPUIntIdx3_IRQn

Interrupt Overview

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 SDL:

- ***CPUSS***->un***CM4_SYSTEM_INT_CTLx***.u32Register is the CPUSS_CM4_SYSTEM_INT_CTLx register mentioned in the Registers TRM **[2]**. Other registers are also described in the same manner. "***x***" signifies the system interrupt index number.
- Performance improvement measures

  For register setting performance improvement, the SDL writes a complete 32-bit data to the register. Each bit field is generated in advance in a bit writable buffer and written to the register as the final 32-bit data.

```
unIntCtl.stcField.u3CPU_INT_IDX = (uint8_t)config->intIdx;
unIntCtl.stcField.u1CPU_INT_VALID = config->isEnabled ? 1ul : 0ul;
CPUSS->unCM4_SYSTEM_INT_CTL[config->sysIntSrc].u32Register =
unIntCtl.u32Register;
```

See *cyip_cpuss_v2.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

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;
}
```

(1) Configure Vector Table

SRAM vector table address

Flash memory vector table address

Copy vector table from Flash memory to SRAM

Set VTOR to SRAM

This code is called during start up.

Interrupt Overview

Code Listing 2 Example of Interrupt Configuration

```
#define CY_BUTTON3_PORT                 GPIO_PRT8
#define CY_BUTTON3_PIN                  0

cy_stc_gpio_pin_config_t user_button3_port_pin_cfg =
{
    .outVal    = 0ul,
    .driveMode = CY_GPIO_DM_HIGHZ,
    .hsiom     = CY_BUTTON3_PIN_MUX,
    .intEdge   = CY_GPIO_INTR_FALLING,
    .intMask   = 1ul,
    .vtrip     = 0ul,
    .slewRate  = 0ul,
    .driveSel  = 0ul,
    .vregEn    = 0ul,
    .ibufMode  = 0ul,
    .vtripSel  = 0ul,
    .vrefSel   = 0ul,
    .vohSel    = 0ul,
};

void ButtonIntHandler(void)
{
:
}

int main(void)
{
    SystemInit();

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


    Cy_GPIO_Pin_Init(CY_BUTTON3_PORT, CY_BUTTON3_PIN, &user_button3_port_pin_cfg);

    /* Setup GPIO for BUTTON3 interrupt */
    cy_stc_sysint_irq_t irq_cfg =
    {
        .sysIntSrc  = CY_BUTTON3_IRQN,
        .intIdx     = CPUIntIdx3_IRQn,
        .isEnabled  = true,
    };

    Cy_SysInt_InitIRQ(&irq_cfg);
    Cy_SysInt_SetSystemIrqVector(irq_cfg.sysIntSrc, ButtonIntHandler);

    NVIC_SetPriority(CPUIntIdx3_IRQn, 3ul);
    NVIC_EnableIRQ(CPUIntIdx3_IRQn);

    for(;;)
    {
    }
}
```

IO port definition

IO port configuration

System interrupt handler

(2) Peripheral function setting

(3) Enable global interrupt (*1)

(4) Configure IO Port

System interrupt handler

(5) Set interrupt structure. See **Code Listing 3** and **Code Listing 4** for details of each function.

(6) Set priority (*1)

(7) Interrupt Enable (*1)

(*1) Programming hint: The CM4/CM0+ interrupt enable and NVIC operation instructions are provided by Cortex microcontroller software interface standard (CMSIS) with intrinsic functions.

Code Listing 3 Cy_SysInt_InitIRQ() Function

```
cy_en_sysint_status_t Cy_SysInt_InitIRQ(const cy_stc_sysint_irq_t* config)
{
    cy_en_sysint_status_t status = CY_SYSINT_SUCCESS;

    un_CPUSS_CM4_SYSTEM_INT_CTL_t unIntCtl = { 0ul };

    if(NULL != config)
    {
      unIntCtl.stcField.u3CPU_INT_IDX = (uint8_t)config->intIdx;
      unIntCtl.stcField.u1CPU_INT_VALID = config->isEnabled ? 1ul : 0ul;
      CPUSS->unCM4_SYSTEM_INT_CTL[config->sysIntSrc].u32Register = unIntCtl.u32Register;
    }
    else
    {
        status = CY_SYSINT_BAD_PARAM;
    }
    return(status);
}
```

Check if configuration parameter values are valid

Set the parameters to interrupt structure

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;
    }
}
```

Check the vector location

Set the system interrupt handler

2.4 Interrupt Handling

In this series MCUs, multiple system interrupts can be mapped to the same CPU interrupt. Therefore, they share a CPU interrupt handler.

To identify the system interrupt occurred, each CPU interrupt has a CMn_INT_STATUS register. This register has the following fields:

- CMn_INTx_STATUS.SYSTEM_INT_VALID: specifies if any system interrupt is active for the CPU interrupt.
- 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 MCUs, these interrupts are level interrupts, so you should clear the peripheral interrupt in return processing. Also, you should read back the register to ensure the completion of register write access, return from the interrupt after clearing the pending register. **Figure 5** shows the interrupt handling example in CYT2B series.

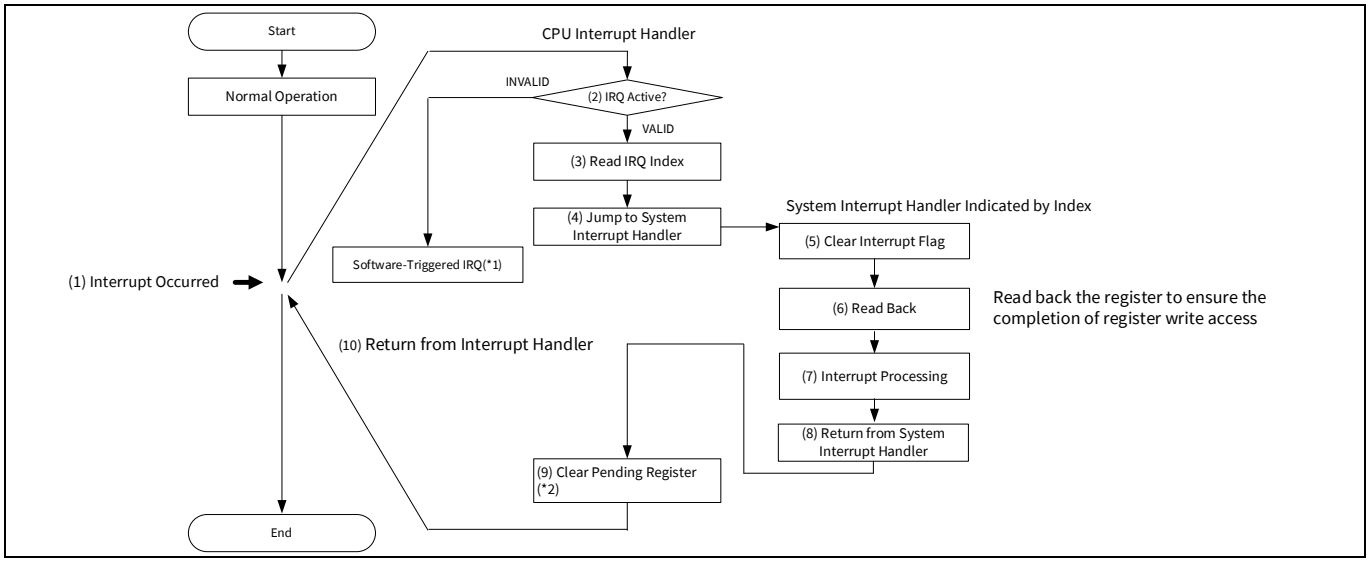

Figure 5 Interrupt Handling

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

Code Listing 5 CPU Interrupt Handler

```
void CpuUserInt3_Handler(void)
{
    un_CPUSS_CM0_INT3_STATUS_t system_int_status = {0};

    system_int_status.u32Register = CPUSS->unCM4_INT3_STATUS.u32Register;

    if(system_int_status.stcField.u1SYSTEM_INT_VALID)
    {
    // jump to system interrupt handler
      Cy_SystemIrqUserTable[system_int_status.stcField.u10SYSTEM_INT_IDX]();
    }
    else
    {
     // Triggered by SW or due to SW clear error (SW cleared a peripheral
     // interrupt flag but didn't clear the Pending flag at NVIC)
    }

    NVIC_ClearPendingIRQ(CPUIntIdx3_IRQn);
}
```

(2) Check if IQR is Active

(3) Read interrupt index
(4) Jump to system interrupt handler

Case of Software Trigger (*1)

(9) Clear the IRQ pending flag executed by the Pending register in NVIC (*2)

Code Listing 6 System Interrupt Handler

```
void ButtonIntHandler(void)
{
    uint32_t intStatus;

    /* If button3 falling edge detected */
    intStatus = Cy_GPIO_GetInterruptStatusMasked(CY_BUTTON3_PORT, CY_BUTTON3_PIN);
    if (intStatus != 0ul)
    {
        Cy_GPIO_ClearInterrupt(CY_BUTTON3_PORT, CY_BUTTON3_PIN);

        :
    }
}
```

Interrupt handler registered in the configuration part

Port number

Pin number in Port

Clear the peripheral interrupt flag that became the interrupt source. See **Code Listing 7**.

(7) Interrupt processing

Code Listing 7 Cy_GPIO_ClearInterrupt() Function

```
__STATIC_INLINE void Cy_GPIO_ClearInterrupt(volatile stc_GPIO_PRT_t* base, uint32_t pinNum)
{
    /* Any INTR MMIO registers AHB clearing must be preceded with an AHB read access */
    (void)base->unINTR.u32Register;

    base->unINTR.u32Register = CY_GPIO_INTR_STATUS_MASK << pinNum;   (5) Clear interrupt flag

    /* This read ensures that the initial write has been flushed out to the hardware */
    (void)base->unINTR.u32Register;                                  (6) Read back
}
```

*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) The flag on the pending register (pending flag) is cleared when the CPU interrupt is accepted. Therefore, normally it is not necessary to clear it in the CPU interrupt handler. However, note that when handling multiple interrupts with one interrupt handler. In the following case, 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.*

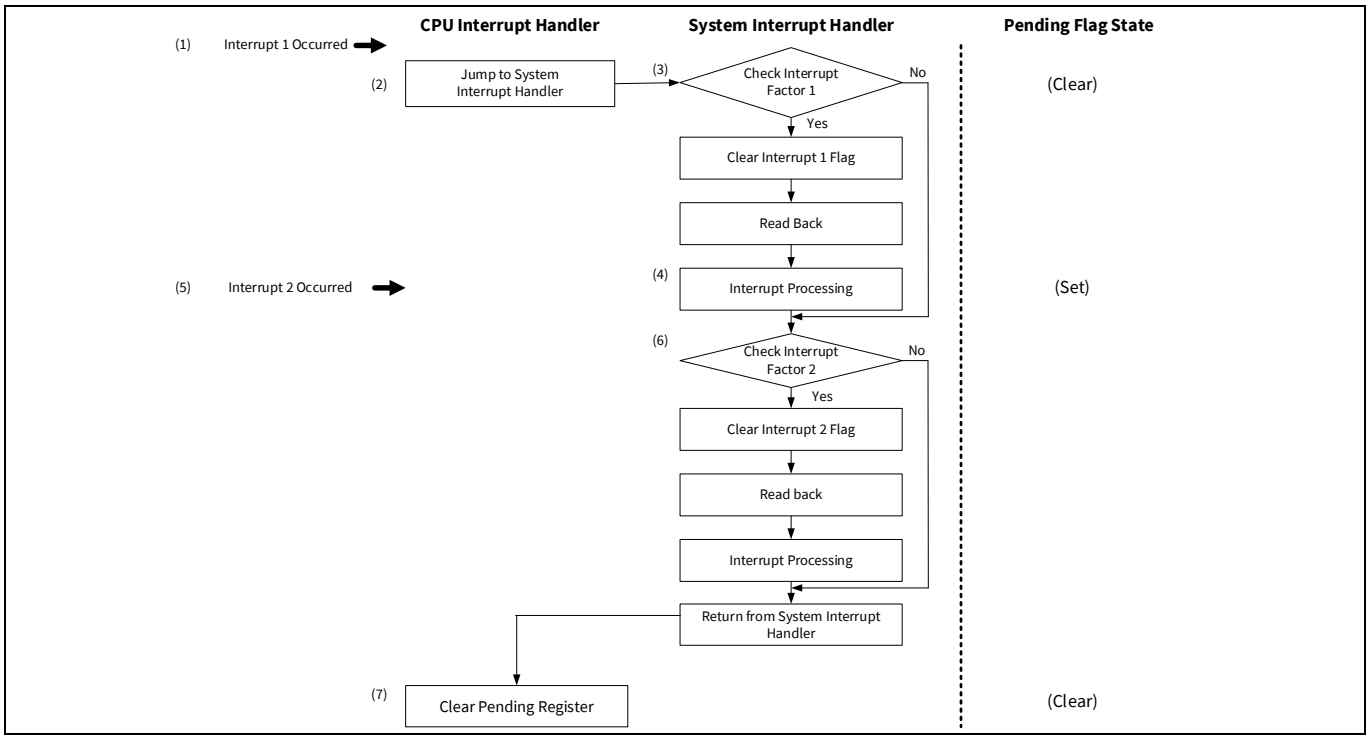

Figure 6 Example of Pending Flag Clear

1. Interrupt 1 occurs.
2. The CPU interrupt handler checks the interrupt factor and jumps to the corresponding system interrupt handler. Here, 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. Interrupt 2 occurs during processing of interrupt 1. Therefore, 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 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. Software can clear these by writing '1' to these bits.
- INTR_SET: Sets an interrupt. Software can write '1' to these bits to set the corresponding INTR bit.
- INTR_MASK: Masks an interrupt. 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.

IDX numbers in this section denote system interrupt index numbers. IDX numbers are the IDX number of CYT2B series. See the Device Datasheet **[1]** for the actual index number to use.

2.6.1 Peripheral Interrupt Handling

This section explains general interrupts from an external pin (GPIO Port#0,1,2), and shows its operation, initial setting, and interrupt handling. Each interrupt is processed by CM4. The following shows the use case.

2.6.1.1 Use case

- Interrupt Setting 1
  - System Interrupt source: GPIO Port Interrupt#0 (IDX: 21) rising-edge detection
  - Mapped to CPU Interrupt: IRQ4
  - CPU Interrupt Priority: 2
- Interrupt Setting 2
  - System Interrupt source: GPIO Port Interrupt#1 (IDX: 22) rising-edge detection
    GPIO Port Interrupt#2 (IDX: 23) rising-edge detection
  - Mapped to CPU Interrupt: IRQ5
  - CPU Interrupt Priority: 3

Figure 7 shows interrupt operation for this configuration.

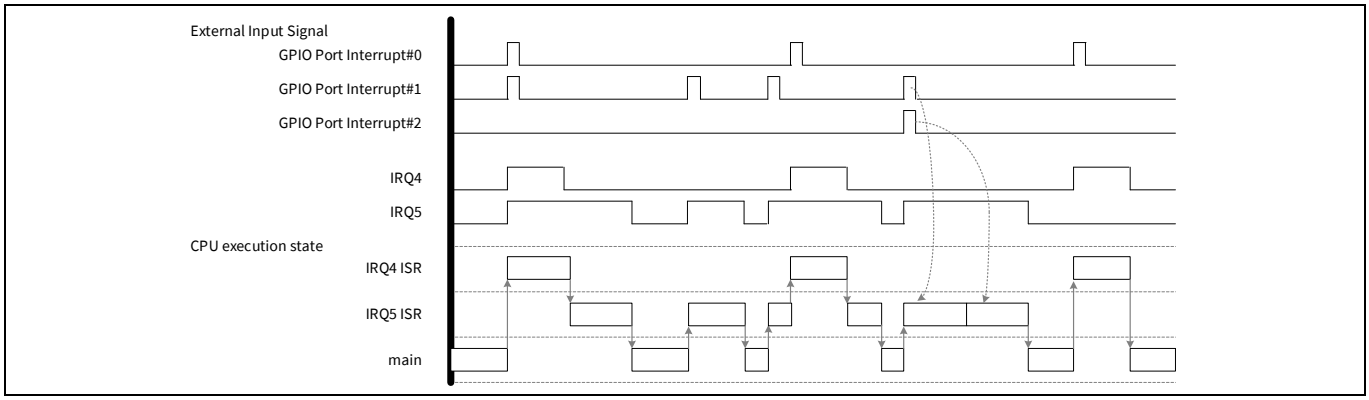

Figure 7 Example Interrupt Operation Usage

If GPIO Port Interrupt#0 and GPIO Port Interrupt#1 interrupts occur at the same time, then the GPIO Port Interrupt#0 interrupt takes precedence because it is mapped to IRQ4, which has higher priority than IRQ5 to which GPIO Port Interrupt#1 interrupt is mapped.

If GPIO Port Interrupt#0 interrupts occur during GPIO Port Interrupt#1 processing, then the GPIO Port Interrupt#0 interrupt takes precedence because it is mapped to IRQ4, which has higher priority than IRQ5 to which GPIO Port Interrupt#1 interrupt is mapped.

If GPIO Port Interrupt#1 and GPIO Port Interrupt#2 interrupts occur at the same time, then the GPIO Port Interrupt#1 interrupt takes precedence (CM0/4_INT5_STATUS register) because both the system interrupts are mapped to IRQ5 but GPIO Port Interrupt#1 has a lower system interrupt index (22) than the system interrupt index (23) of GPIO Port Interrupt#2.

2.6.1.2 Configuration

- **Initial Setting Procedure**

  See **Initial Setting** for the initial setting procedure. After each peripheral resource (GPIO Port Interrupt#0,1,2) is set up, each resource interrupt is routed to the CPU interrupt. Next, set the level and permission of each CPU interrupt to the NVIC. After setting up interrupt connection, start each resource. See the “I/O System” chapter of the Architecture TRM **[2]** for GPIO setting.

- **Interrupt Handler Operation**

  See **Interrupt Handling** for the interrupt handler configuration. When an interrupt occurs, the CPU jumps to the interrupt handler specified in the vector table. Within the CPU interrupt handler, it identifies the system interrupt that triggered the CPU interrupt and jumps 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 8 and **Figure 9** show the software flow when single and multiple interrupts occur.

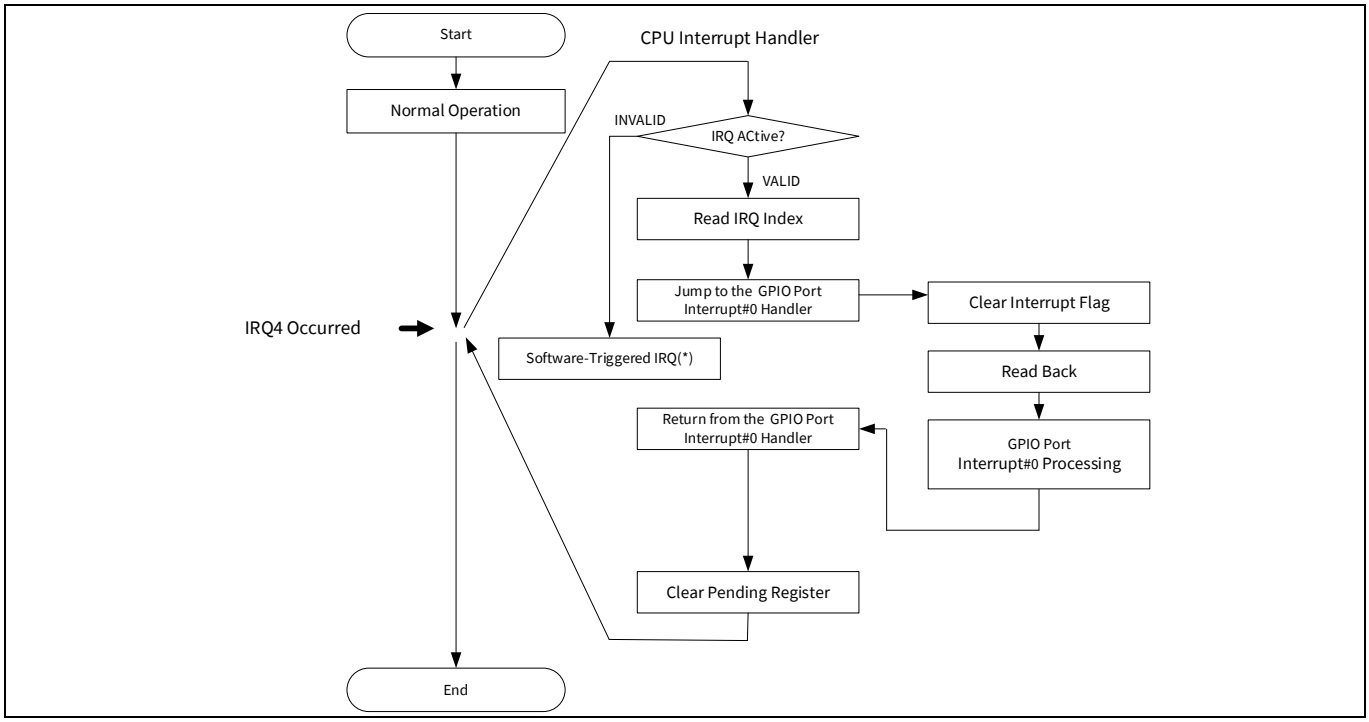

Figure 8 Interrupt Operation Flow

*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 current ISR processing and execute the ISR with the higher priority.

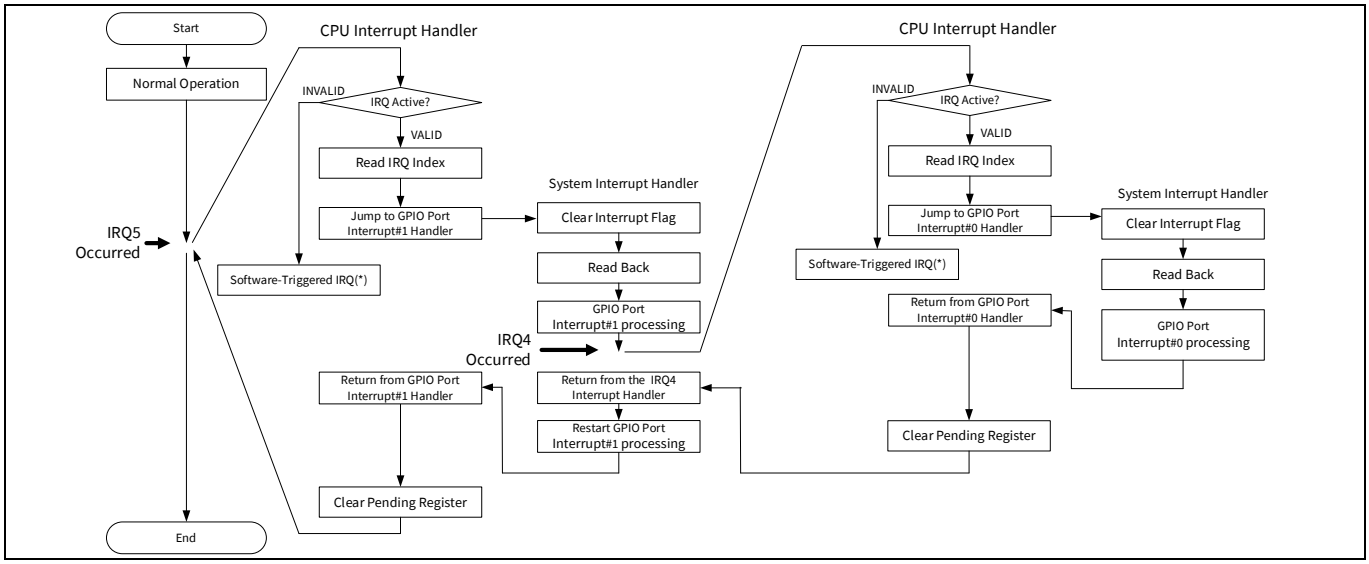

Figure 9 Nested Interrupt Operation Flow

*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.*

2.6.2 NMI Generation of System Interrupts

In this series, up to four of all the available system interrupts can be mapped to the NMI of each CPU. This section explains the Timing Protection configuration using NMI, shows its operation, initial setting, and interrupt handling. Each interrupt is processed by CM4.

2.6.2.1 Use case

In this configuration, monitor the processing time of the periodic interrupt routine (500 counts) called by TCPWM Group#0 Counter#1.

TCPWM Group#0 Counter#0 is used to monitor the processing time of the interrupt routine. TCPWM Group#0 Counter#0 is a counter that counts upwards, and generates an interrupt when the count value is more than the compare value (100 counts). 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 TCPWM Group#0 Counter#0 underflow interrupt.

- TCPWM Group#0 Counter#0
  - System Interrupt source: TCPWM Group#0 Counter#0(IDX: 274) TC interrupt
  - Mapped to CPU NMI
  - Monitor of IRQ5 ISR operation time
  - Compare value: 100 (Monitoring cycle)
- TCPWM Group#0 Counter#1
  - System Interrupt source: TCPWM Group#0 Counter#1(IDX: 275) TC interrupt
  - Mapped to CPU Interrupt: IRQ5
  - CPU Interrupt Priority: 3
  - Compare value: 500 (Interrupt period for system operation)

Figure 10 shows interrupt operation for this configuration.

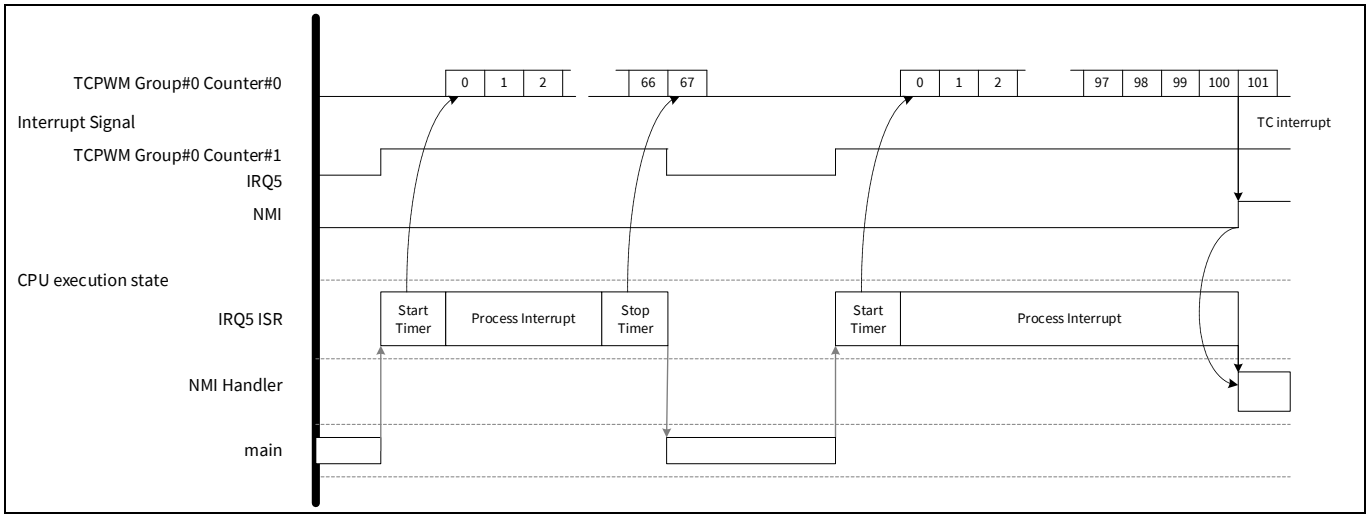

Figure 10 Timing Protection Operation Example

2.6.2.2 Configuration

- Initial Setting Procedure

  See **Initial Setting** for the initial setting procedure. After each peripheral resource (TCPWM Group#0 Counter#0 and #1) is set up, each resource interrupt is routed to the CPU interrupt. Next, set the level and permission of each CPU interrupt to the NVIC. After setting up interrupt connection, start each resource.

Figure 11 shows the initializing interrupts for NMI generation.

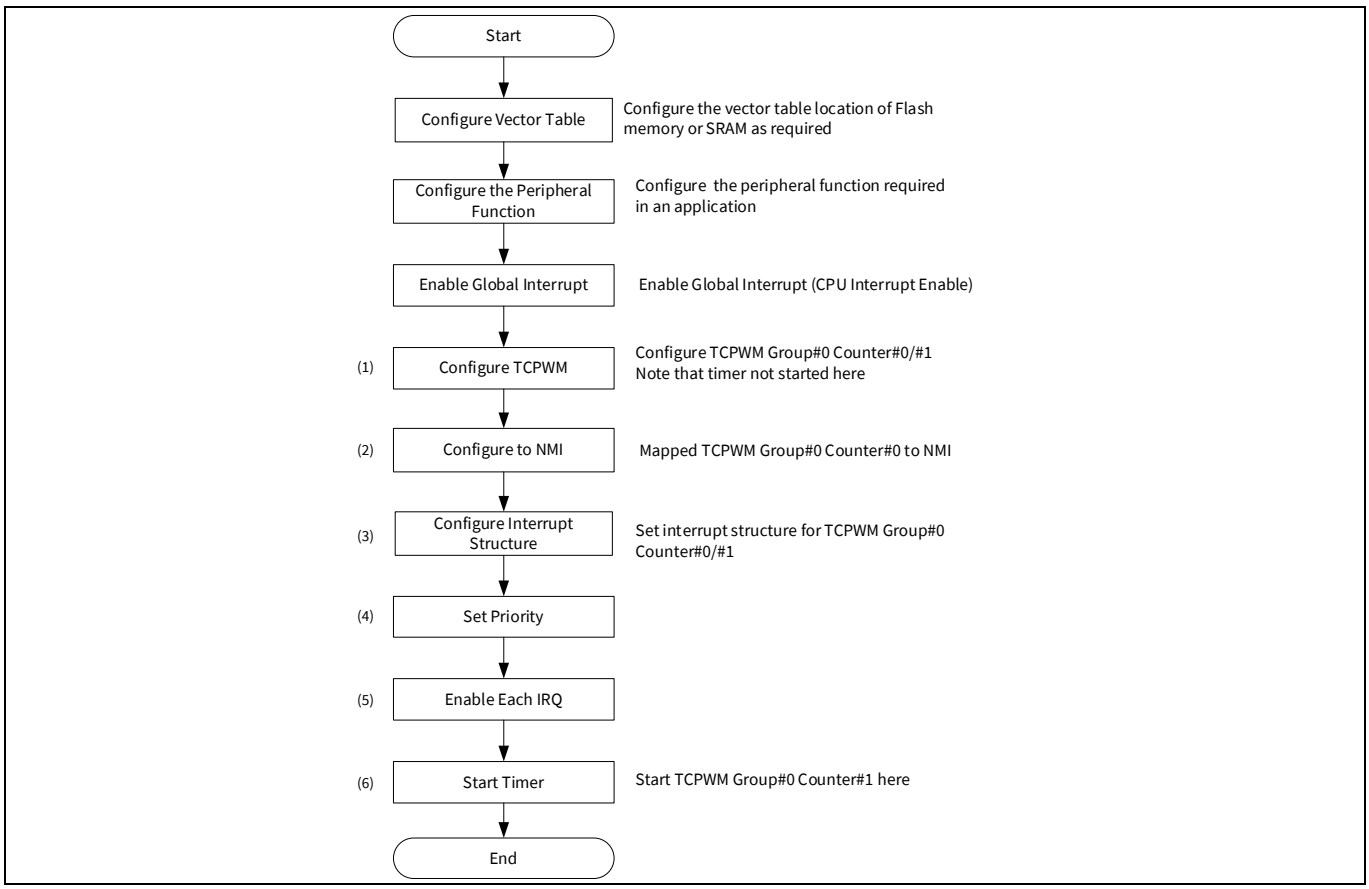

Figure 11 Interrupt Structure Initial Setting Procedure for NMI

Table 2 lists the parameters and functions of the configuration part in SDL for interrupt initialization.

Table 2 List of Interrupt Initialization Parameters and Functions

Parameters	Description	Value
irq_cfg_0.sysIntSrc	Set system interrupt index number [0: 1022]	tcpwm_0_interrupts_0_IRQn It is assigned to TCPWM Group#0 Counter#0 (Index number = 274)
irq_cfg_0.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4,	CPUIntIdx4_IRQn

Interrupt Overview

Parameters	Description	Value
	CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7	
irq_cfg_0.intIdx.isEnabled	Set interrupt enable False: Disabled True: Enabled	True
irq_cfg_1.sysIntSrc	Set system interrupt index number [0: 1022]	tcpwm_0_interrupts_1_IRQn It is assigned to TCPWM Group#0 Counter#1 (Index number = 275)
irq_cfg_1.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4, CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7	CPUIntIdx5_IRQn
irq_cfg_1.intIdx.isEnabled	Set interrupt enable False: Disabled True: Enabled	True
Function	**Description**	**Value**
Cy_SysInt_SetIntSourceNMI (Reg_num, sysintSrc)	Set NMI regster Reg_num: CM4 NMI control Register number CPUSS_CM4_NMI_CTLx register (x = 0 to 3) sysintSrc: System interrupt index number	Reg_nim: 0 Use CPUSS_CM4_NMI_CTL0 register sysintSrc: tcpwm_0_interrupts_0_IRQn
Cy_SysInt_InitIRQ(irq_cfg)	Configure interrupt structure Irq_cfg: Interrupt structure parameters address	&irq_cfg0/1
Cy_SysInt_SetSystemIrq Vector (sysintSrc, Handler)	Set system interrupt handler to vector table sysintSrc: System interrupt index number Handler: Interrupt handler address	sysintSrc: irq_cfg_0/1.sysIntSrc Handler: NMI_Handler / Counter_Handler See **Code Listing 11** and **Code Listing 13**.
NVIC_SetPriority(intSrc, intPriority)	Set interrupt priority: intSrc: CPU interrupt number intPriority: Interrupt priority Level	intSrc: irq_cfg_1.intIdx intPriority: 3ul
NVIC_EnableIRQ(intSrc)	Set interrupt enable intSrc: CPU interrupt number	irq_cfg_0/1.intIdx

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 [2] and Application Note [3] for TCPWM setting details.

Code Listing 8 Example of Interrupt Configuration 1

```
#define COMPARE0_NUM (100ul)
#define COMPARE1_NUM (500ul)

cy_stc_tcpwm_counter_config_t MyCounter_config =
{
    .period             = 0xFFFFul,
    .clockPrescaler     = CY_TCPWM_COUNTER_PRESCALER_DIVBY_4,
    .runMode            = CY_TCPWM_COUNTER_CONTINUOUS,
    .countDirection     = CY_TCPWM_COUNTER_COUNT_UP,
    .debug_pause        = false,
    .CompareOrCapture   = CY_TCPWM_COUNTER_MODE_COMPARE,
    .compare0           = COMPARE0_NUM,                    // Set the comp value
    .compare0_buff      = 0ul,                             // do not care
    .compare1           = 0ul,                             // do not care
    .compare1_buff      = 0ul,                             // do not care
    .enableCompare0Swap = false,                           // do not care
    .enableCompare1Swap = false,                           // do not care
    .interruptSources   = 0ul,                             // do not care
    .capture0InputMode  = 3ul,                             // do not care
    .capture0Input      = 0ul,                             // do not care
    .reloadInputMode    = 3ul,                             // do not care
    .reloadInput        = 0ul,                             // do not care
    .startInputMode     = 3ul,                             // do not care
    .startInput         = 0ul,                             // do not care
    .stopInputMode      = 3ul,                             // do not care
    .stopInput          = 0ul,                             // do not care
    .capture1InputMode  = 3ul,                             // do not care
    .capture1Input      = 0ul,                             // do not care
    .countInputMode     = 3ul,                             // do not care
    .countInput         = 1ul,                             // do not care
    .trigger1           = CY_TCPWM_COUNTER_OVERFLOW,       // do not care
};

cy_stc_sysint_irq_t irq_cfg_0 =
{
    .sysIntSrc  = tcpwm_0_interrupts_0_IRQn,
    .intIdx     = CPUIntIdx4_IRQn,
    .isEnabled  = true,
};
cy_stc_sysint_irq_t irq_cfg_1 =
{
    .sysIntSrc  = tcpwm_0_interrupts_1_IRQn,
    .intIdx     = CPUIntIdx5_IRQn,
    .isEnabled  = true,
};
```

Counter period parameter definition

TCPWM configuration

Count direction

Compare value

Configure interrupt structure parameters for TCPWM Group#0 counter#0

Configure interrupt structure parameters for TCPWM Group#0 counter#1

Code Listing 9 Example of Interrupt Configuration 2

```
void Counter_Handler(void)
{
:
}

void NMI_Handler(void)
{
:
}

int main(void)
{
    :
    /* Initialize TCPWM0_GRPx_CNTx_COUNTER as Counter & Enable */
    Cy_Tcpwm_Counter_Init(TCPWMx_GRPx_CNTx_COUNTER_0, &MyCounter_config);
    MyCounter_config.compare0 = COMPARE1_NUM;
    Cy_Tcpwm_Counter_Init(TCPWMx_GRPx_CNTx_COUNTER_1, &MyCounter_config);
    Cy_Tcpwm_Counter_Enable(TCPWMx_GRPx_CNTx_COUNTER_0);
    Cy_Tcpwm_Counter_Enable(TCPWMx_GRPx_CNTx_COUNTER_1);

    /* Enable Interrupt */
    Cy_Tcpwm_Counter_SetCC0_IntrMask(TCPWMx_GRPx_CNTx_COUNTER_0);
    Cy_Tcpwm_Counter_SetCC0_IntrMask(TCPWMx_GRPx_CNTx_COUNTER_1);

    /* Set NMI mapping of CM4 */
    Cy_SysInt_SetIntSourceNMI(0ul, tcpwm_0_interrupts_0_IRQn);

    /*  Interrupt setting for TCPWMs  */
    Cy_SysInt_InitIRQ(&irq_cfg_0);
    Cy_SysInt_InitIRQ(&irq_cfg_1);
    Cy_SysInt_SetSystemIrqVector(irq_cfg_0.sysIntSrc, NMI_Handler);
    Cy_SysInt_SetSystemIrqVector(irq_cfg_1.sysIntSrc, Counter_Handler);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(irq_cfg_1.intIdx, 3ul);
    NVIC_EnableIRQ(irq_cfg_1.intIdx);

    /* Start TCPWM Group#0 Counter#1 */
    Cy_Tcpwm_TriggerStart(TCPWMx_GRPx_CNTx_COUNTER_1);

    for(;;)
    {
    }
}
```

System interrupt handler

(1) Configure TCPWM

(2) Configure NMI. See **Code Listing 10**.

NMI handler

System interrupt handler

(3) Set interrupt structure. See **Code Listing 3** and **Code Listing 4** for details of each function.

(4) Set priority

(5) Interrupt Enable

(6) Start TCPWM Group#0 counter#1 timer

Code Listing 10 Cy_SysInt_SetIntSourceNMI () Function

```
void Cy_SysInt_SetIntSourceNMI(uint8_t nmiNum, cy_en_intr_t sysIntSrc)
{
    CPUSS->unCM4_NMI_CTL[nmiNum].stcField.u10SYSTEM_INT_IDX = sysIntSrc;
}
```

Set the fault structure interrupt to NMI

Interrupt Overview

- Interrupt Handler Operation

  See **Interrupt Handling** for the interrupt handler configuration. 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 NMI handler is executed.

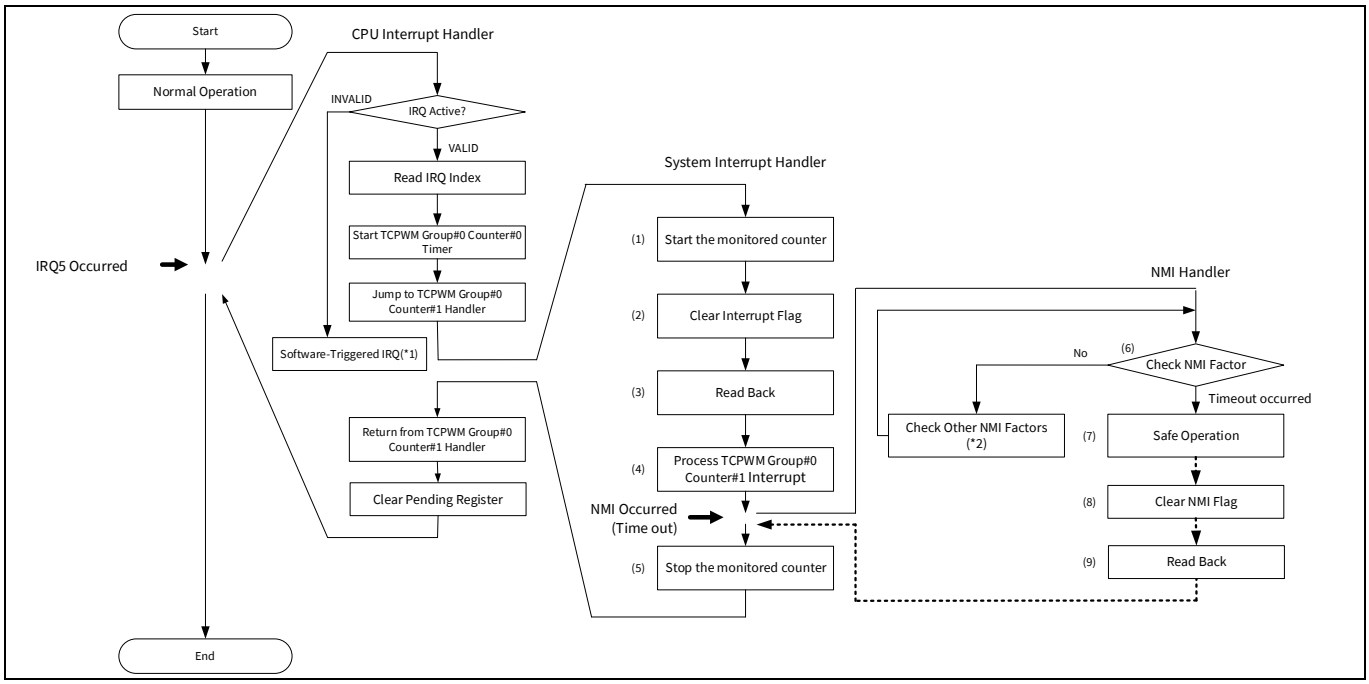

Figure 12 Timing Protection Violation Operation Flow Example

*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 appropriate safe operation such as reset generation. If recovery to normal operation is possible after safe operation, return from the interrupt.

Code Listing 11 System Interrupt Handler for TCPWM Group#0 Counter#1

```
void Counter_Handler(void)
{
    if(Cy_Tcpwm_Counter_GetCC0_IntrMasked(TCPWMx_GRPx_CNTx_COUNTER_1))
    {
        /* Start TCPWM Group#0 Counter#0 */
        Cy_Tcpwm_TriggerStart(TCPWMx_GRPx_CNTx_COUNTER_0);

        /* Clear TCPWM CC0 interrupt flag Group#0 Counter#1 */
        Cy_Tcpwm_Counter_ClearCC0_Intr(TCPWMx_GRPx_CNTx_COUNTER_1);

        /* user program here.. */

        /* Stop TCPWM Group#0 Counter#0 */
        Cy_Tcpwm_TriggerStopOrKill(TCPWMx_GRPx_CNTx_COUNTER_0);
    }
  }
```

Interrupt handler registered in the configuration part

Check the source that generated the interrupt

(1) Start the monitored counter (TCPWM Group#0 Counter#0)

(2) Clear the peripheral interrupt flag. See **Code Listing 12**.

(4) Interrupt processing

(5) Stop the monitored counter (TCPWM Group#0 Counter#0)

Code Listing 12 Cy_Tcpwm_Counter_ClearCC0_Intr() Function

```
void Cy_Tcpwm_Counter_ClearCC0_Intr(volatile stc_TCPWM_GRP_CNT_t *ptscTCPWM)
{
    ptscTCPWM->unINTR.stcField.u1CC0_MATCH = 1ul;
    ptscTCPWM->unINTR.u32Register;
}
```

(2) Clear interrupt flag

(3) Read back

Code Listing 13 NMI Handler for TCPWM Group#0 Counter#0

```
void NMI_Handler(void)
{
    if(Cy_Tcpwm_Counter_GetCC0_IntrMasked(TCPWMx_GRPx_CNTx_COUNTER_0))
    {
        /* Safe Operation (Counter#0 Stop)*/
        {
            /* Stop TCPWM Group#0 Counter#0 */
            Cy_Tcpwm_TriggerStopOrKill(TCPWMx_GRPx_CNTx_COUNTER_0);
        }

        /* Clear TCPWM CC0 interrupt flag Group#0 Counter#0 */
        Cy_Tcpwm_Counter_ClearCC0_Intr(TCPWMx_GRPx_CNTx_COUNTER_0);
    }
  }
```

NMI handler registered in the configuration part

(6) Check NMI factor

(7) Safe Operation (Stop the monitored counter)

(8) Clear the NMI flag, and (9) Read back. See **Code Listing 12**.

2.6.3 Wakeup Interrupt

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

A subset of the system interrupts can wake up the CPU from DeepSleep; this subset can be used in Active power mode, and to wake up the CPU from Sleep and DeepSleep power modes. See the Device Datasheet **[1]** for available interrupts details.

This section explains an interrupt from the Backup domain that is caused by an interrupt from the real-time clock (RTC) and shows how to set and return processing from DeepSleep 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: IRQ7
  - CPU Interrupt Priority: 6

Figure 13 shows interrupt operation for this configuration.

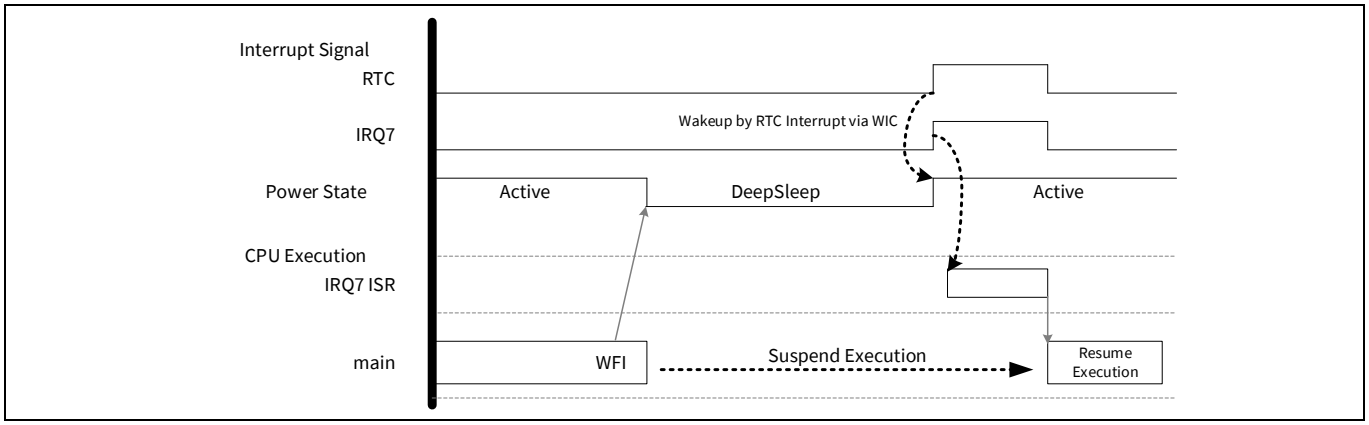

Figure 13 Example Wakeup Interrupt Operation

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

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 normal interrupt setting. See **Initial Setting** for the initial setting procedure and the “Real-Time Clock” chapter of the Architecture TRM **[2]** for 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.

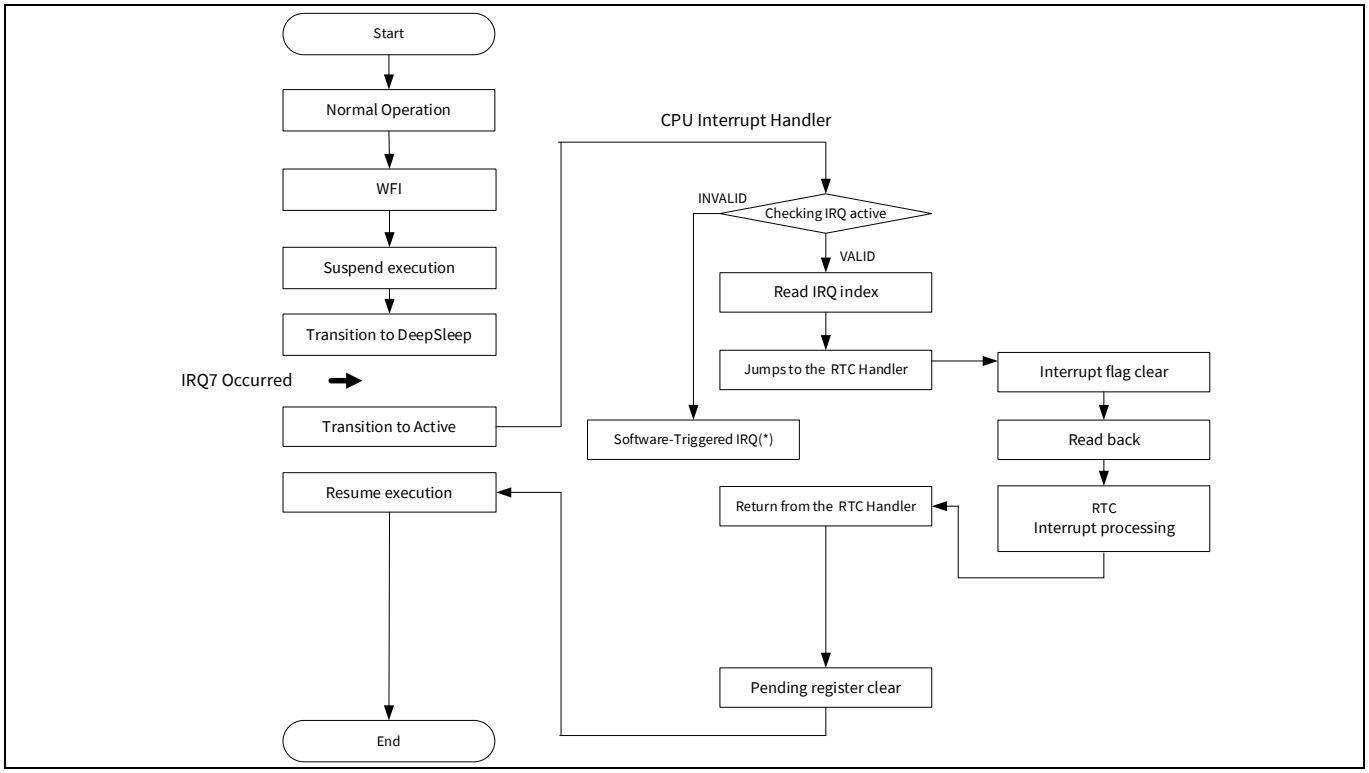

Figure 14 Wakeup Interrupt Handling Example

*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 IRQ8 to IRQ15 for software interrupt. A software interrupt can be generated by writing '1' to the software triggered interrupt register (STIR) of the corresponding CPU interrupts IRQ8 to IRQ15. IRQ0 to IRQ7 can also trigger software interrupts with STIR.

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

2.6.4.1 Use Case

- Interrupt Setting
  - Mapped to CPU Interrupt: IRQ8
  - CPU Interrupt Priority: 3

Figure 15 shows interrupt operation for this configuration.

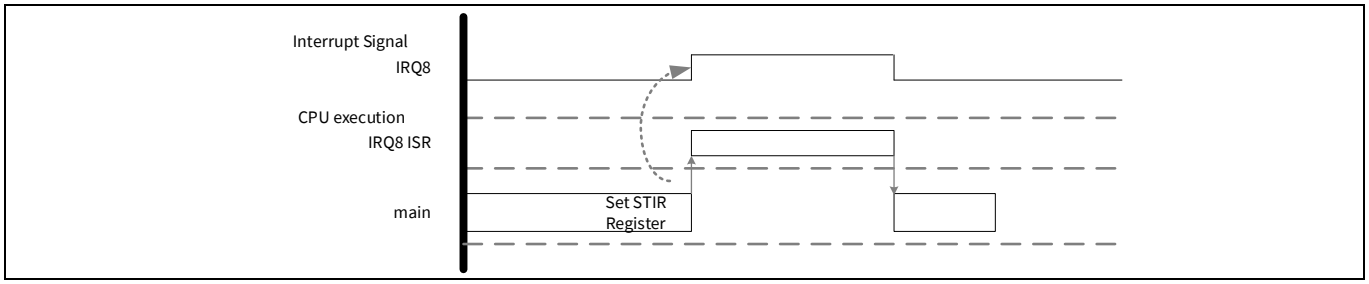

Figure 15 Software Interrupt Operation

2.6.4.2 Configuration

- Initial Setting

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

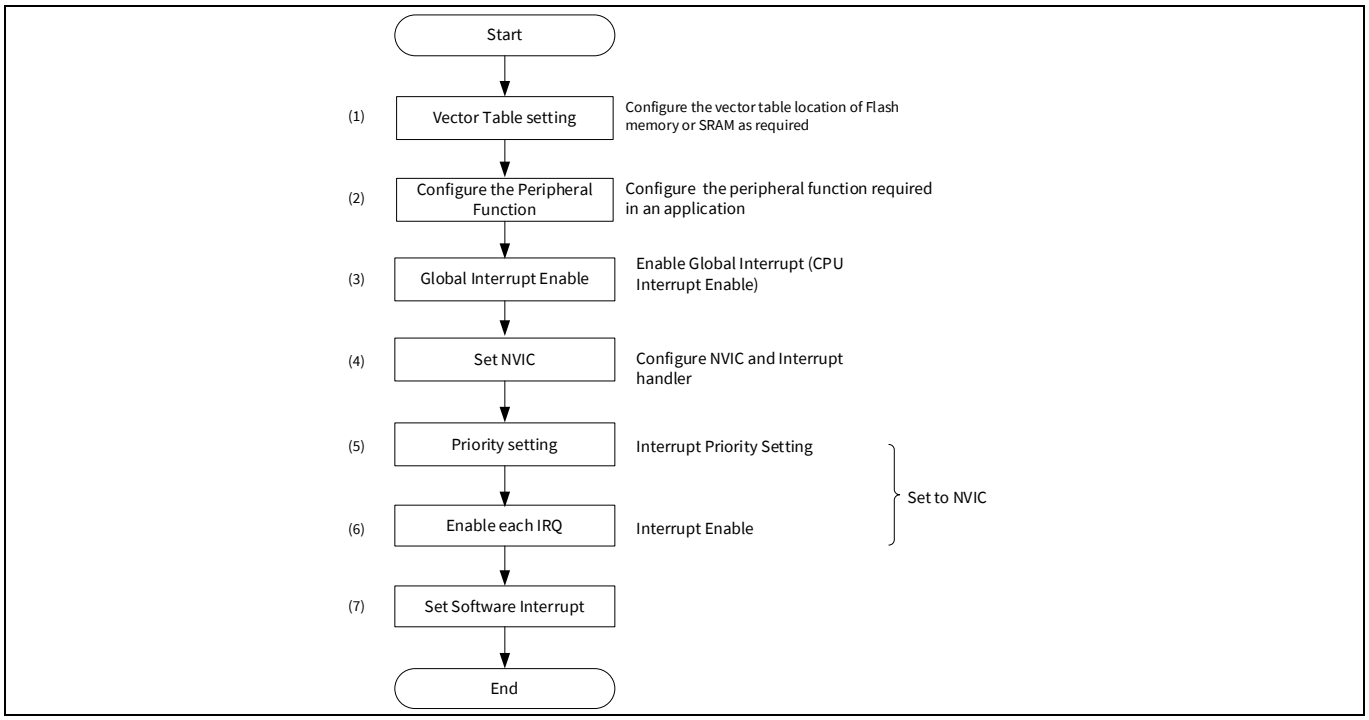

Figure 16 Initial Setting Procedure for Software Interrupt (Using CPU Register)

Table 3 lists the parameters and functions of the configuration part in SDL for interrupt initialization.

Table 3 List of Interrupt Initialization Parameters and Functions

Parameters	Description	Value
cfg.intrSrc	Set 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,	Internal0_IRQn

Interrupt Overview

Parameters	Description	Value
	Internal6_IRQn is assigned to IRQ 1 4, Internal7_IRQn is assigned to IRQ 15	
cfg.intrPriority	Set Interrupt Priority	3ul
Function	**Description**	**Value**
Cy_SysInt_Init (cfg, Handler)	Configure interrupt structure cfg: Software interrupt parameters address Handler: Interrupt handler address	cfg: cfg address Handler: SoftwareInteruptHandler, See **Code Listing 18**.
NVIC_SetPriority(intrSrc, intPriority)	Set 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 14 shows an example program of the interrupt configuration part for software interrupt. See **Initial Setting** for Configuration of (1).

Code Listing 14 Example for Software Interrupt Configuration

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

int main(void)
{
    SystemInit();

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

    /* Software Interrupt 0 */
    {
        /* At first change software interrupt 0 handler */
        /* The default handler is defined at startup_tviibe1m_cm4 */
        cy_stc_sysint_t cfg =
        {
            .intrSrc = Internal0_IRQn,
            .intrPriority = 3ul,
        };
        Cy_SysInt_Init(&cfg, SoftwareInteruptHandler);

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

    for(;;)
    {
    }
}
```

System interrupt handler

(2) Peripheral function setting

(3) Enable global interrupt

Set configuration parameters for NVIC

(4) Set NVIC registers and interrupt handler. See **Code Listing 15**.

(6) Interrupt Enable

(7) Set software interrupt. See **Code Listing 17**.

Code Listing 15 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)
    {
        /* Only set the new vector if it is CPU interrupt (not internal SW interrupt) */
        if ((config->intrSrc < CPUIntIdx0_IRQn) || (config->intrSrc >= Internal0_IRQn))
        {
            /* 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.
        }
        NVIC_SetPriority(config->intrSrc, config->intrPriority);
    }
    else
    {
        status = CY_SYSINT_BAD_PARAM;
    }

    return(status);
}
```

Check if configuration parameter values are valid

Set the software interrupt handler. See **Code Listing 16**.

(5) Set priority

Code Listing 16 Cy_SysInt_SetVector() Function

```
void  Cy_SysInt_SetVector(IRQn_Type intrSrc, cy_israddress userIsr)
{
    /* Only set the new vector if it was moved to __ramVectors */
    if (SCB->VTOR == (uint32_t)&__ramVectors)
    {
        __ramVectors[CY_INT_IRQ_BASE + intrSrc] = userIsr;
    }
}
```

Set the interrupt handler

Code Listing 17 Cy_SysInt_SoftwareTrig() Function

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

Set STIR

- Interrupt Handling

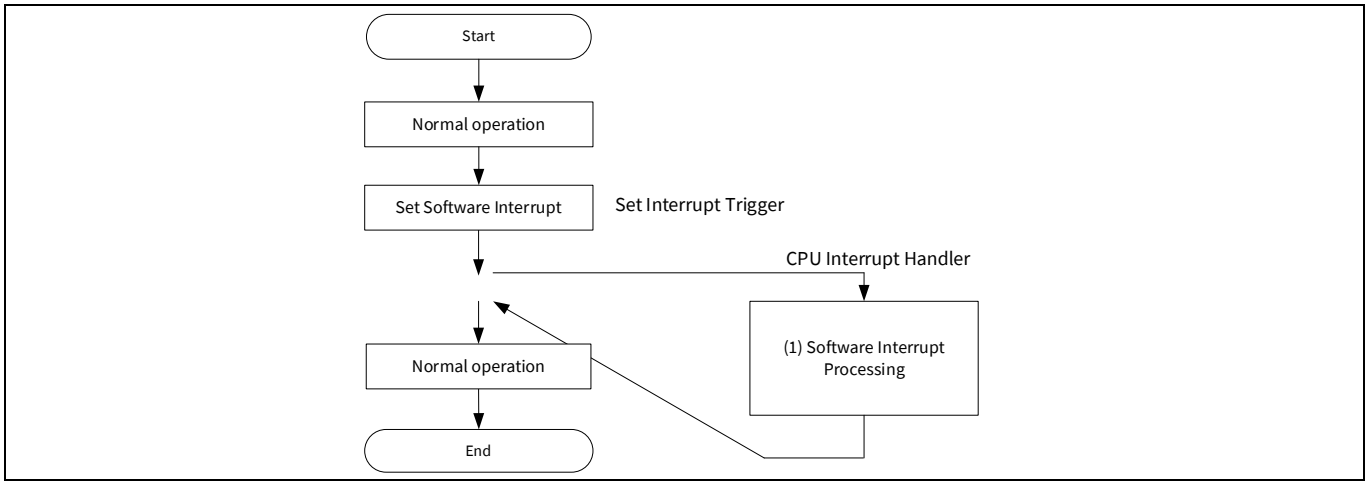

Figure 17 Software Interrupt Handling Flow Example

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 at return from the interrupt handler.

Code Listing 18 shows an example of CPU interrupt handler for software interrupt.

Code Listing 18 CPU Interrupt Handler for Software Interrupt

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

  }
```

Software interrupt processing

2.6.5 Software Interrupt (Using Peripheral)

This series can generate 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 software interrupt generation by using peripherals and shows the setting procedure and return processing. The interrupt is processed by CM4.

2.6.5.1 Use case

- Interrupt Setting
  - System Interrupt source: TCPWM Group#0 Counter#1(IDX: 275) TC interrupt
  - Mapped to CPU Interrupt: IRQ4
  - CPU Interrupt Priority: 3

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

Figure 18 shows interrupt operation for this configuration.

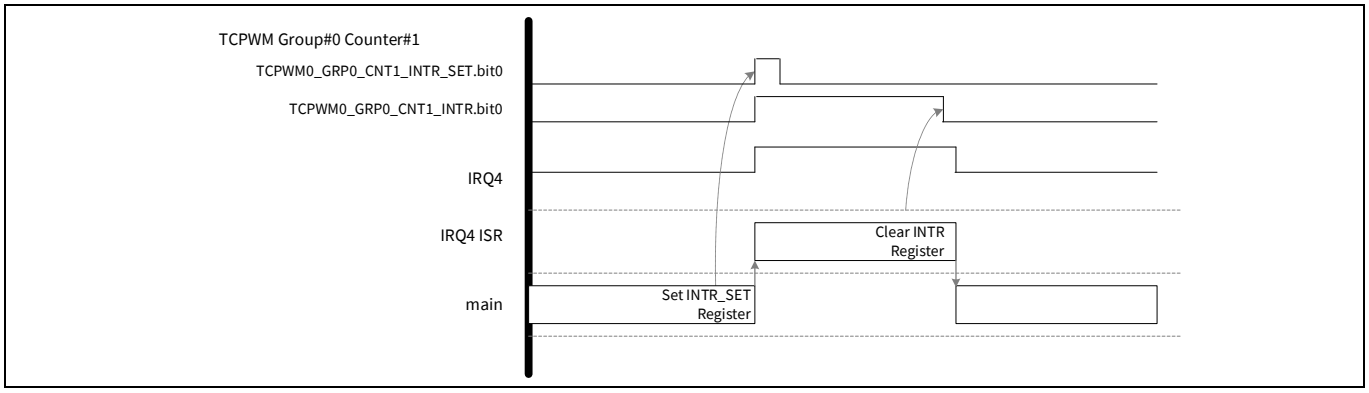

Figure 18 Software Interrupt Operation Example by Utilizing Unused TCPWM

2.6.5.2 Configuration

- Initial Setting Procedure

  See **Initial Setting** for the initial setting procedure. After each peripheral resource (TCPWM Group#0 Counter#1) is set up, each resource interrupt is routed to the CPU interrupt. Next, set the level and permission of each CPU interrupt to the NVIC. After setting up interrupt connection, start each resource. See the “Timer, Counter, and PWM” chapter of the Architecture TRM [2] for TCPWM setting.

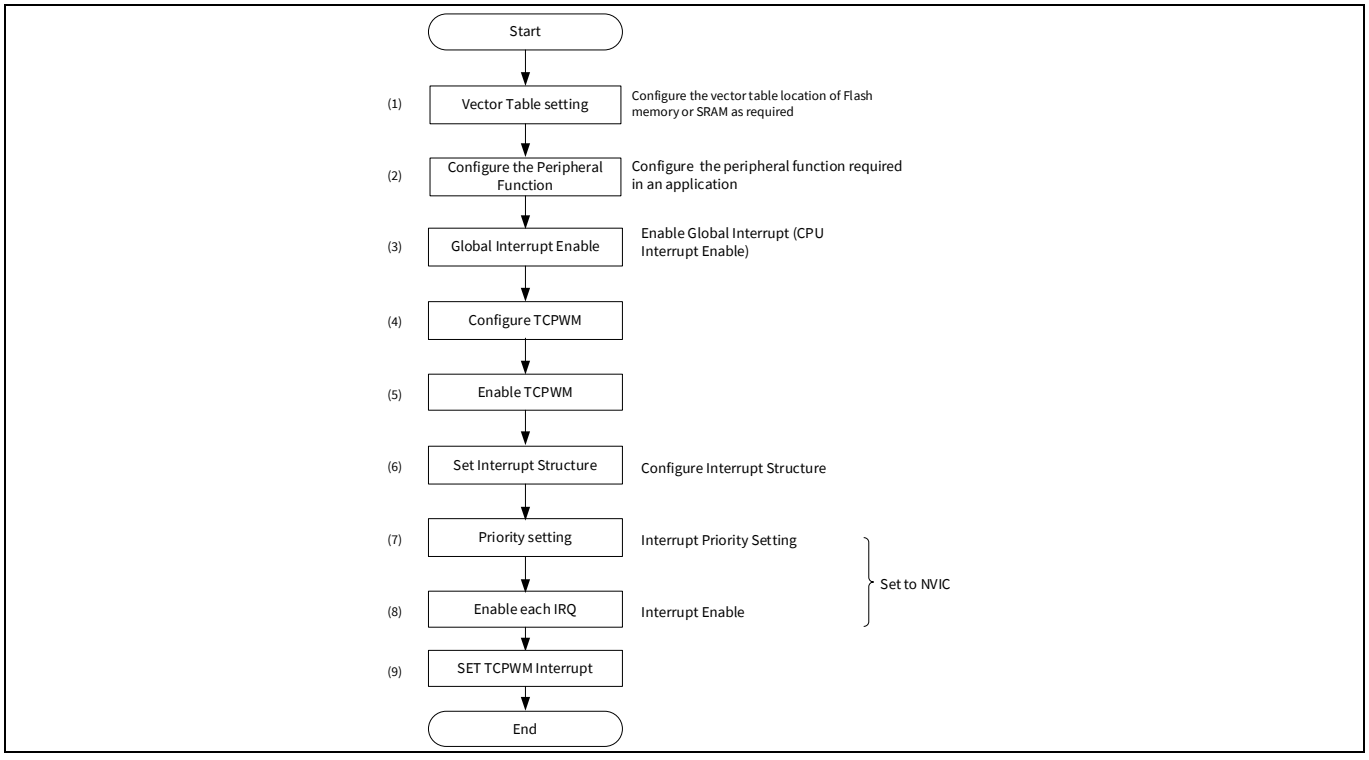

Figure 19 Initial Setting Procedure for Software Interrupt (Using Peripheral)

Table 4 lists the parameters and functions of the configuration part in SDL for interrupt initialization.

Interrupt Overview

Table 4 List of Interrupt Initialization Parameters and Functions

Parameters	Description	Value
irq_cfg_1.sysIntSrc	Set system interrupt index number [0: 1022]	tcpwm_0_interrupts_1_IRQn It is assigned to TCPWM Group#0 Counter#1 interrupt (Index number = 275)
irq_cfg_1.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4, CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7	CPUIntIdx4_IRQn
irq_cfg_1.isEnabled	Set interrupt enable False: Disabled True: Enabled	True
Function	**Description**	**Value**
Cy_Tcpwm_Counter_ SetTC_IntrMask (Count_num)	TCPWM interrupt enable Count_num: Interrupt enabled timer Counter number	TCPWMx_GRPx_CNTx_COUNTER_1 It is assigned to Counter#1
Cy_SysInt_InitIRQ(irq_cfg)	Configure interrupt structure Irq_cfg: Interrupt structure parameters address	&irq_cfg_1
Cy_SysInt_SetSystemIrq Vector (sysintSrc, Handler)	Set system interrupt handler to vector table sysintSrc: System interrupt index number Handler: Interrupt handler address	sysintSrc: irq_cfg_1.sysIntSrc Handler: Counter_Handler, See **Code Listing 21**.
NVIC_SetPriority(intrSrc, intPriority)	Set interrupt priority: intrSrc: CPU interrupt number intPriority: Interrupt priority level	intrSrc: irq_cfg_1.intIdx intPriority: 3ul
NVIC_EnableIRQ(intSrc)	Set interrupt enable intSrc: CPU interrupt number	intrSrc: irq_cfg_1.intIdx

Code Listing 19 shows an example program of the interrupt configuration part for software interrupt. See **Initial Setting** for Configuration of (1).

Code Listing 19 Example of Interrupt Configuration

```
cy_stc_sysint_irq_t irq_cfg_1 =
{
    .sysIntSrc  = tcpwm_0_interrupts_1_IRQn,
    .intIdx     = CPUIntIdx4_IRQn,
    .isEnabled  = true,
};

void Counter_Handler(void)
{
:
}

int main(void)
{
    SystemInit();

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

    /* Configure TCPWM Group#0 Counter#1 */
     :

    /* Enable Interrupt */
    Cy_Tcpwm_Counter_SetTC_IntrMask(TCPWMx_GRPx_CNTx_COUNTER_1);

    /*  Interrupt setting for TCPWMs  */
    Cy_SysInt_InitIRQ(&irq_cfg_1);
    Cy_SysInt_SetSystemIrqVector(irq_cfg_1.sysIntSrc, Counter_Handler);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(irq_cfg_1.intIdx, 3ul);
    NVIC_EnableIRQ(irq_cfg_1.intIdx);

    /* Set INTR_SET register, TCPWM Group#0 Counter#1 */
    TCPWMx_GRPx_CNTx_COUNTER_1->unINTR_SET.u32Register = 0x1ul;

    for(;;)
    {
    }
}
```

Interrupt structure configuration parameters

System interrupt handler

(2) Peripheral function setting

(3) Enable global interrupt (*1)

(4) Configure the TCPWM. See Architecture TRM [2].

(5) Enable the TCPWM counter#1 interrupt. See **Code Listing 20**.

(6) Configure interrupt structure.

(7) Set priority (*1)

(8) Interrupt Enable (*1)

(9) Set TCPWM Interrupt

Code Listing 20 Cy_Tcpwm_Counter_SetTC_IntrMask () Function

```
void Cy_Tcpwm_Counter_SetTC_IntrMask(volatile stc_TCPWM_GRP_CNT_t *ptscTCPWM)
{
    ptscTCPWM->unINTR_MASK.stcField.u1TC = 1ul;
}
```

Interrupt enable

- Interrupt Handling

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

  When an interrupt occurs, the CPU jumps to the interrupt handler specified in the vector table. Within the CPU interrupt handler, it identifies the system interrupt that triggered the CPU interrupt and jumps 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.

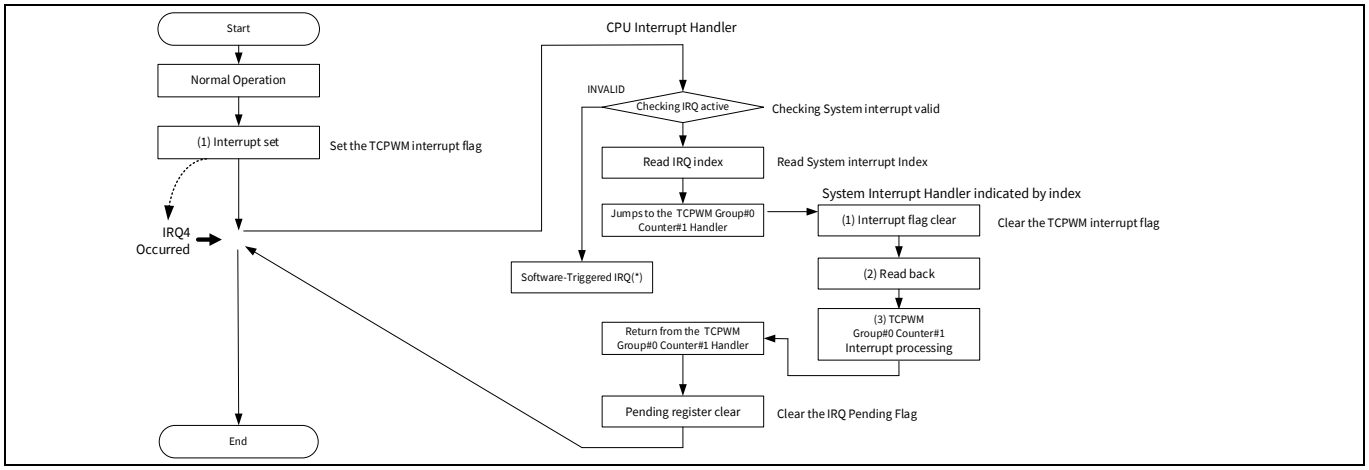

Figure 20 Software Interrupt Handling by Utilizing Unused TCPWM

*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 21 shows an example of system interrupt handler. See **Code Listing 5** for CPU Interrupt handler.

Code Listing 21 Example of System Interrupt Handler

```
void Counter_Handler(void)
{
    if(Cy_Tcpwm_Counter_GetTC_IntrMasked(TCPWMx_GRPx_CNTx_COUNTER_1))
    {
        /* Clear TCPWM TC interrupt flag Group#0 Counter#1 */
        Cy_Tcpwm_Counter_ClearTC_Intr(TCPWMx_GRPx_CNTx_COUNTER_1);

        /* User program here.. */

    }
}
```

Interrupt handler registered in the configuration part

Check the source that generated the interrupt

Clear the peripheral interrupt flag that became the interrupt source. See **Code Listing 22**.

(3) Interrupt Processing

Code Listing 22 Cy_Tcpwm_Counter_ClearTC_Intr()

```
void Cy_Tcpwm_Counter_ClearTC_Intr(volatile stc_TCPWM_GRP_CNT_t *ptscTCPWM)
{
    ptscTCPWM->unINTR.stcField.u1TC = 1ul;
    ptscTCPWM->unINTR.u32Register;
}
```

(1) Clear interrupt flag

(2) Read back

3 Fault Report Structure Overview

3.1 Fault Report Structure

Fault report structures encapsulate information about transient faults that occur while software is running. Fault report structures store the information about the faults and can cause reset. The platform uses centralized fault report structures. This centralized nature allows for a 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 21 shows a block diagram of the fault report structure.

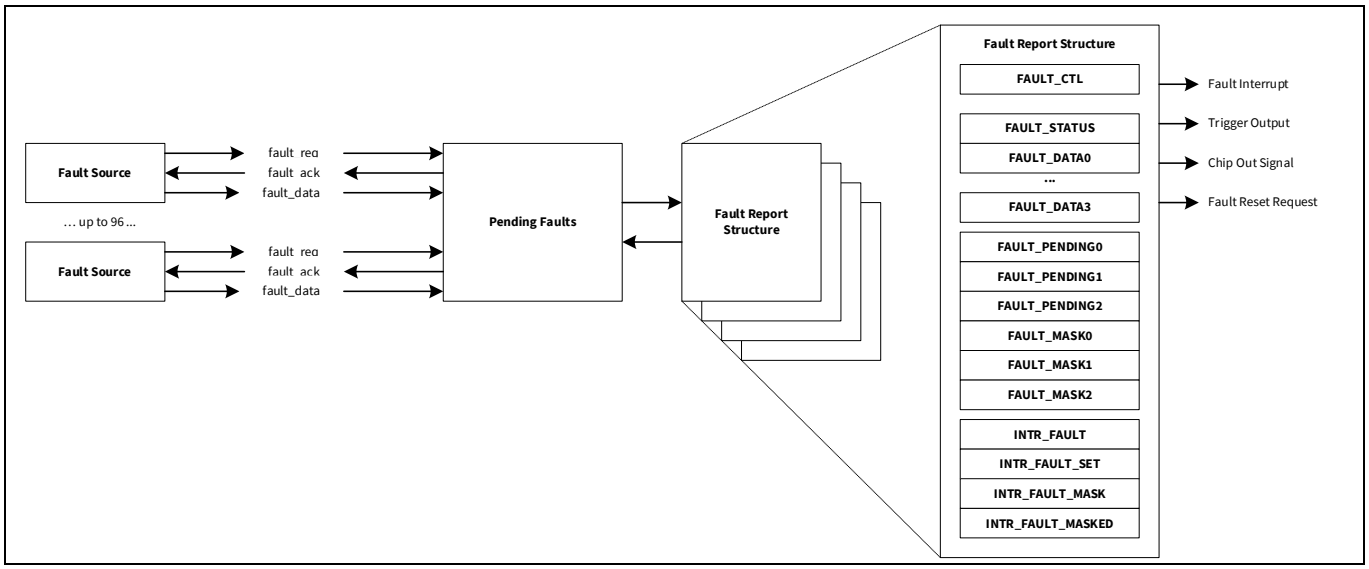

Figure 21 Fault Report Structure

This 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 [2].

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 signals 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. When a pending fault is captured by a fault report structure, the associated pending bit is cleared to '0'. The fault report structure functionality is only available in Active and Sleep power modes.

3.2 Initial Setting Procedure

This section describes how to initialize interrupts using 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 22 shows the procedure for initializing the fault report structure. Initialize the fault report structure, and set the fault connection and action when detecting a fault.

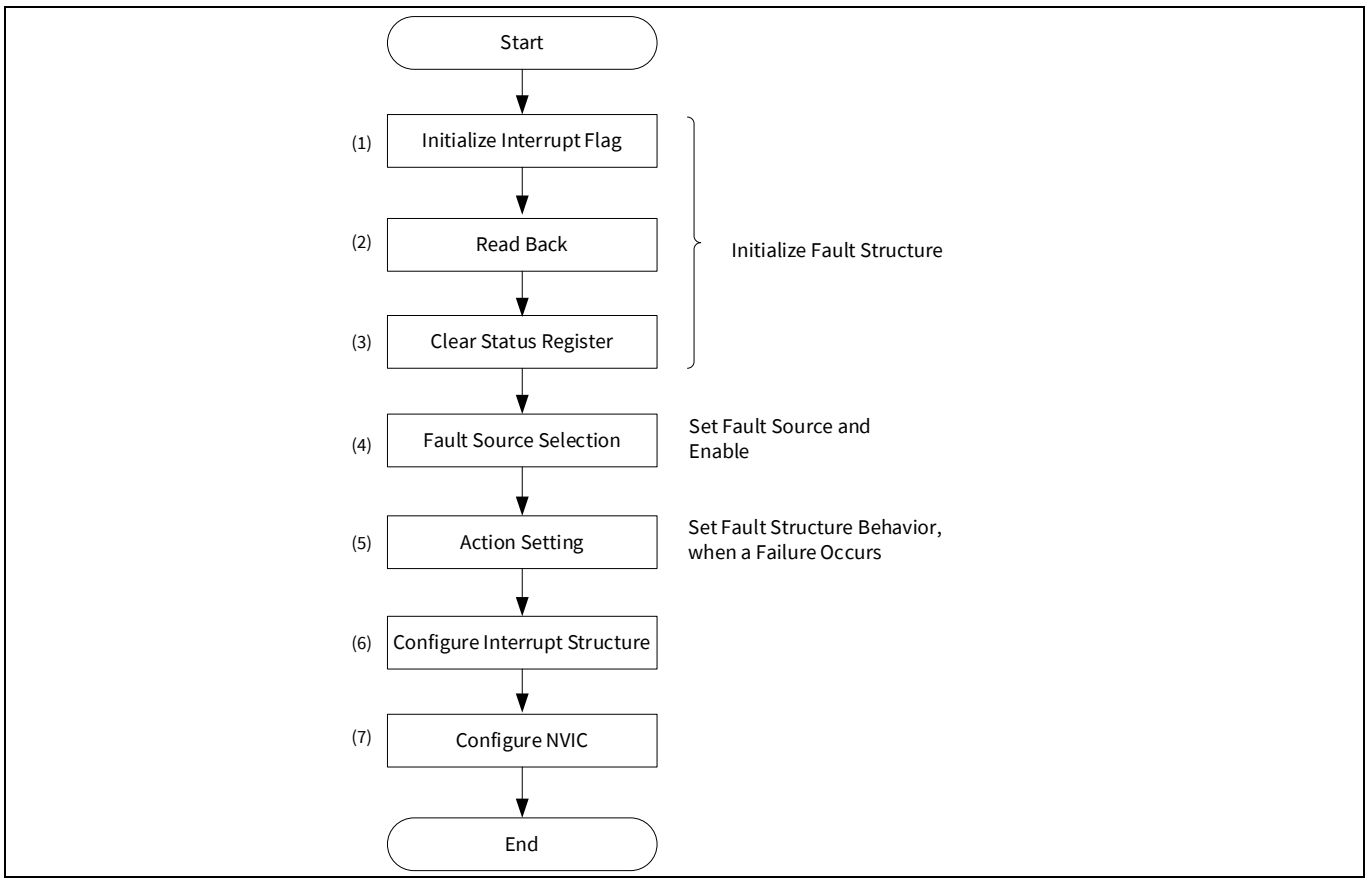

Figure 22 Fault Report Structure Initial Setting Flow

3.2.1 Use case

In this configuration, the CPU is notified of the interrupt via interrupt structure, when SRAM0 correctable ECC violation is detected.

IDX numbers used this section denote system interrupt index numbers. IDX numbers are the IDX number of CYT2B series. See the Device Datasheet [1] for the actual index number to use.

- Using fault report structure: #0
- Fault source: CPUSS_RAM0_C_ECC (IDX: 58)
- Fault action: Interrupt generation
- Mapped to CPU Interrupt: IRQ2
- CPU Interrupt Priority: 0

3.2.2 Configuration

Table 5 lists the parameters and functions of the configuration part in SDL for fault report structure Initialization.

Table 5 List of Fault Report Structure Initialization Parameters and Functions

Parameters	Description	Value
irq_cfg.sysIntSrc	Set system interrupt index number [0: 1022]	cpuss_interrupts_fault_0_IRQn It is assigned to Fault structure #0 interrupt (Index number = 8)
irq_cfg.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4, CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7	CPUIntIdx2_IRQn
irq_cfg.isEnabled	Set interrupt enable False: Disabled True: Enabled	True
tFlt_Temp.ResetEnable	Set reset request enable: False: Disabled True: Enabled	False
tFlt_Temp.OutputEnable	Set IO output signal enable: False: Disabled True: Enabled	False
tFlt_Temp.TriggerEnable	Set Trigger output enable: False: Disabled True: Enabled	False

Fault Report Structure Overview

Function	Description	Value
Cy_SysFlt_ClearStatus (FAULT_STRUCT)	Clear fault structure status: FAULT_STRUCT: Fault report structure number FAULT_STRUCT0 = Fault report structure 0 FAULT_STRUCT1 = Fault report structure 1 FAULT_STRUCT2 = Fault report structure 2 FAULT_STRUCT3 = Fault report structure 3	FAULT_STRUCT0
Cy_SysFlt_ClearInterrupt (FAULT_STRUCT)	Clear fault structure interrupt flag:	FAULT_STRUCT0
Cy_SysFlt_SetInterruptMask (FAULT_STRUCT)	Set interrupt enable: FAULT_STRUCT: Fault report structure number	FAULT_STRUCT0
Cy_SysFlt_SetMaskByIdx (FAULT_STRUCT, faultSrc)	Select fault source: FAULT_STRUCT: Fault report structure number faultSrc: Fault Source	FAULT_STRUCT: FAULT_STRUCT0 faultSrc: CY_SYSFLT_RAMC0_C_ECC It is assigned CPUSS_RAM0_C_ECC (Fault number = 58)
Cy_SysFlt_Init (FAULT_STRUCT, tFlt_Temp)	Set fault structure action: FAULT_STRUCT: Fault report structure number tFlt_Temp: Action parameters	FAULT_STRUCT0

Code Listing 23 show an example program of the fault report configuration part.

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

- ptsc***SYSFLT*** signifies the pointer to the fault report structure register base address.
- ptsc***SYSFLT***->un***STATUS***.u32Register is the FAULT_STRUCTx_STATUS register mentioned in the Registers TRM [2]. Other registers are also described in the same manner. "***x***" signifies the fault report structure number.

See *cyip_fault_v2.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

Fault Report Structure Overview

Code Listing 23 Example of Fault Report Structure Configuration

```
cy_stc_sysint_irq_t irq_cfg =
{
    .sysIntSrc  = cpuss_interrupts_fault_0_IRQn,
    .intIdx     = CPUIntIdx2_IRQn,
    .isEnabled  = true,
};

int main(void)
{
    cy_stc_sysflt_t tFlt_Temp = {0ul};
    :
    /* Clear Status register */
    Cy_SysFlt_ClearInterrupt(FAULT_STRUCT0);
    FAULT_STRUCT0->unINTR.u32Register;
    Cy_SysFlt_ClearStatus(FAULT_STRUCT0);

    /* Set the signal interface (Interrupt generation) */
    Cy_SysFlt_SetInterruptMask(FAULT_STRUCT0);

    /* Fault source enable 58 System memory controller 0 correctable ECC violation */
    Cy_SysFlt_SetMaskByIdx(FAULT_STRUCT0, CY_SYSFLT_RAMC0_C_ECC);

    /* Init Sysflt */
    tFlt_Temp.ResetEnable = false;
    tFlt_Temp.OutputEnable = false;
    tFlt_Temp.TriggerEnable = false;
    Cy_SysFlt_Init(FAULT_STRUCT0, &tFlt_Temp);

    /*  Interrupt setting */
    Cy_SysInt_InitIRQ(&irq_cfg);
    Cy_SysInt_SetSystemIrqVector(irq_cfg.sysIntSrc, irqFaultReport0Handler);

    /* Set the Interrupt Priority & Enable the Interrupt */
    NVIC_SetPriority(irq_cfg.intIdx, 0ul);
    NVIC_EnableIRQ(irq_cfg.intIdx);
    :
    for(;;)
    {
    }
}
```

Configure interrupt structure parameters

(1) Initialize interrupt flag See **Code Listing 24**.

(2) Read back

(3) Initialize status register. See **Code Listing 25**.

(4) Fault source selection. See **Code Listing 26**.

(5) Action Setting. See **Code Listing 27**.

(6) Configure interrupt setting. See **Code Listing 3** and **Code Listing 4**.

Set interrupt handler

(7) Configure NVIC. Priority sets to “0”.

Code Listing 24 Cy_SysFlt_ClearInterrupt () Function

```
void Cy_SysFlt_ClearInterrupt(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    ptscSYSFLT->unINTR.stcField.u1FAULT = true;
}
```

Fault structure interrupt flag clear

Code Listing 25 Cy_SysFlt_ClearStatus () Function

```
void Cy_SysFlt_ClearStatus(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    ptscSYSFLT->unSTATUS.u32Register = 0x00000000UL;
}
```

Fault structure status clear

Fault Report Structure Overview

Code Listing 26 Cy_SysFlt_SetMaskByIdx() and Cy_SysFlt_SetInterruptMask() Functions

```
void Cy_SysFlt_SetMaskByIdx(volatile stc_FAULT_STRUCT_t *ptscSYSFLT, cy_en_sysflt_source_t idx)
{
    switch(idx / 32)
    {
    case 0:
        ptscSYSFLT->unMASK0.u32Register |= (1UL << (uint32_t)idx);
        break;
    case 1:
        ptscSYSFLT->unMASK1.u32Register |= (1UL << ((uint32_t)idx - 32UL));
        break;
    case 2:
        ptscSYSFLT->unMASK2.u32Register |= (1UL << ((uint32_t)idx - 64UL));
        break;
    default:
        break;
    }
    return;
}

void Cy_SysFlt_SetInterruptMask(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    ptscSYSFLT->unINTR_MASK.stcField.u1FAULT = true;
}
```

Fault source select

Fault structure interrupt enable

Code Listing 27 Cy_SysFlt_Init() Function

```
void Cy_SysFlt_Init(volatile stc_FAULT_STRUCT_t *ptscSYSFLT,const cy_stc_sysflt_t * config)
{
    if (config->TriggerEnable)
    {
        ptscSYSFLT->unCTL.stcField.u1TR_EN =  true;
    }
    else
    {
        ptscSYSFLT->unCTL.stcField.u1TR_EN = false;
    }

    if (config->OutputEnable)
    {
        ptscSYSFLT->unCTL.stcField.u1OUT_EN = true;
    }
    else
    {
        ptscSYSFLT->unCTL.stcField.u1OUT_EN = false;
    }

    if (config->ResetEnable)
    {
        ptscSYSFLT->unCTL.stcField.u1RESET_REQ_EN = true;
    }
    else
    {
        ptscSYSFLT->unCTL.stcField.u1RESET_REQ_EN = false;
    }
}
```

Fault action select

Trigger output enable

IO output signal enable

Reset request enable

3.3 Fault Handling

As mentioned earlier, when the fault report structure captures a fault, it can notify the system of the occurrence of the fault. In addition, the fault report structure captures additional information for failure analysis.

Figure 23 shows an example of the handling when an interrupt is notified.

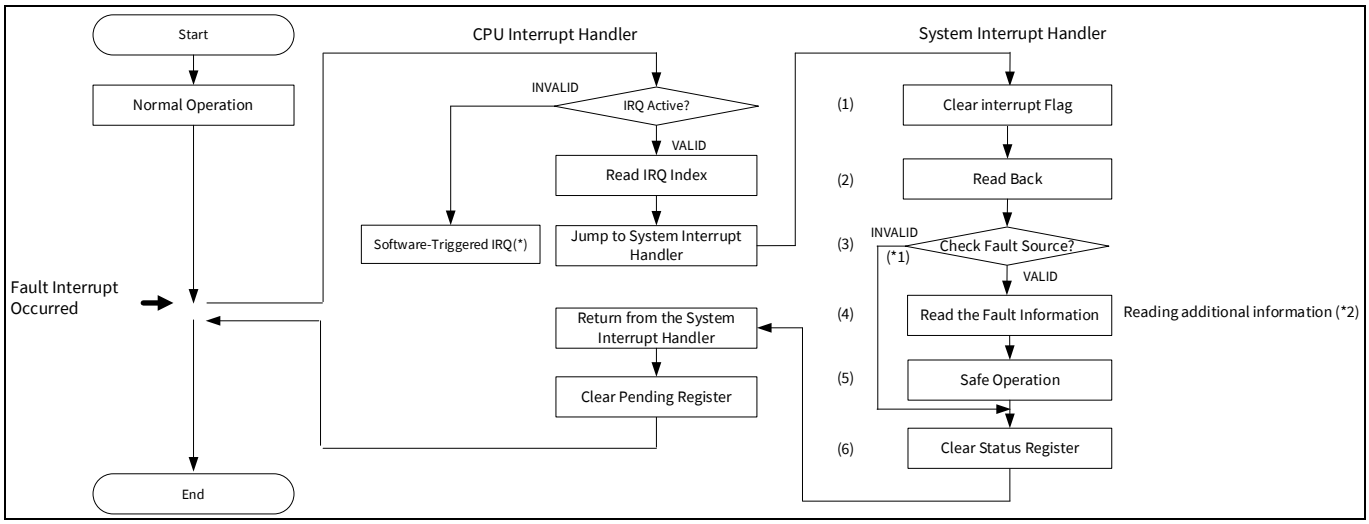

Figure 23 Fault Handling Example

*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.*

*Note: (*1) If a fault is not active (FAULT_STRUCT_STATUS.VALID = 0), FAULT_STRUCT_STATUS_IDX and FAULT_STRUCT_DATA are invalid.*

*Note: (*2) The number of registers used as additional information depends on the detected fault. See the Architecture TRM [2] and Registers TRM [2] for more information.*

The software checks the cause of the fault from the additional information, and executes appropriate safety operation or diagnostic programs according to fault situations. If recovery to normal operation is possible after safe operation, return from the interrupt.

Code Listing 28 shows an example program of the fault interrupt handler.

Fault Report Structure Overview

Code Listing 28 Fault Interrupt Handler

```
void irqFaultReport0Handler(void)
{
    cy_en_sysflt_source_t status;

    /* Clear Interrupt flag */
    Cy_SysFlt_ClearInterrupt(FAULT_STRUCT0);
    FAULT_STRUCT0->unINTR.u32Register; /* Read Back */

    status = Cy_SysFlt_GetErrorSource(FAULT_STRUCT0);

    if(status == CY_SYSFLT_RAMC0_C_ECC)
    {

        violatingAddr = Cy_SysFlt_GetData0(FAULT_STRUCT0);
        violatingInfo.u32 = Cy_SysFlt_GetData1(FAULT_STRUCT0);

        /* User program here.. */

    }
    /* Clear Status register */
    Cy_SysFlt_ClearStatus(FAULT_STRUCT0);
}
```

Interrupt handler registered in the configuration part

(1) Clear interrupt flag. See **Code Listing 24**.

(2) Read back

(3) Check fault source. See **Code Listing 29**.

(4) Read the fault information. See **Code Listing 30**.

(5) Safe Operation. (Fault handling)

(6) Clear status register. See **Code Listing 25**.

Code Listing 29 CySysFlt_GetErrorSource() Function

```
cy_en_sysflt_source_t Cy_SysFlt_GetErrorSource(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    if(ptscSYSFLT->unSTATUS.stcField.u1VALID == 1u)
    {
        return((cy_en_sysflt_source_t)(ptscSYSFLT->unSTATUS.stcField.u7IDX));
    }
    else
    {
        return CY_SYSFLT_NO_FAULT;
    }
}
```

Check Fault is valid

Read Fault Index

Code Listing 30 Cy_SysFltGetData0 and 1 () Function

```
uint32_t Cy_SysFlt_GetData0(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    return(ptscSYSFLT->unDATA[0].u32Register);
}

uint32_t Cy_SysFlt_GetData1(volatile stc_FAULT_STRUCT_t *ptscSYSFLT)
{
    return(ptscSYSFLT->unDATA[1].u32Register);
}
```

Read additional information of fault

Read additional information of fault

3.4 Usage Example of Fault Report Structure

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

IDX numbers in this section denote faults index numbers. IDX numbers are the IDX number of CYT2B series. See the Device Datasheet **[1]** for the actual index number to use.

3.4.1 Reset Generation

This section explains the generation of a reset signal when a fault is detected, and shows its operation and initial setting.

3.4.1.1 Use case

In this configuration, reset will be issued, when CM0+ violates SMPU protection.

- Using fault report structure: #0
- Fault source: CPUSS_MPU_VIO_0 (IDX: 0)
- Fault Action: Reset generation

*Note: (*) See the Architecture TRM **[2]** and Registers TRM **[2]** for fault assignment and additional information.*

A fault reset causes a warm/soft reset. For failure analysis, fault information such as FAULT_STATUS and FAULT_DATA registers are retained during a warm/soft reset.

3.4.1.2 Configuration

- Initial Setting Procedure

  Set according to the **Initial Setting Procedure**. Initialize the fault report structure to be used (Fault Report Structure#0), set the action (Reset) at detection and the fault source (MPU_VIO_0). **Table 6** lists the parameters and functions of the configuration part in SDL for fault report structure Initialization.

Table 6 List of Fault Report Structure Initialization Parameters and Functions

Parameters	Description	Value
tFlt_Temp.ResetEnable	Set reset request enable: False: Disabled True: Enabled	True
tFlt_Temp.OutputEnable	Set IO output signal enable: False: Disabled True: Enabled	False
tFlt_Temp.TriggerEnable	Set Trigger output enable: False: Disabled True: Enabled	False
Function	**Description**	**Value**
Cy_SysFlt_SetMaskByIdx (FAULT_STRUCT, faultSrc)	Select fault source: FAULT_STRUCT: Fault report structure number faultSrc: Fault Source	FAULT_STRUCT: FAULT_STRUCT0 faultSrc: CY_SYSFLT_MPU_0 It is assigned MPU_VIO_0 (Fault number = 0)

Code Listing 31 shows an example program of the Fault Report Structure configuration part for reset generation.

Code Listing 31 Example of Fault Report Structure Configuration

```
:
 /* Fault source enable 58 System memory controller 0 correctable ECC violation */
 Cy_SysFlt_SetMaskByIdx(FAULT_STRUCT0, CY_SYSFLT_MPU_0);
 :
/* Init Sysflt */
 tFlt_Temp.ResetEnable = true;
 tFlt_Temp.OutputEnable = false;
 tFlt_Temp.TriggerEnable = false;
 Cy_SysFlt_Init(FAULT_STRUCT0, &tFlt_Temp);
 :
```

Fault source selection. See **Code Listing 24**.

Action Setting. See **Code Listing 27**.

3.4.2 NMI Generation via Interrupt Structure

This section explains the generation of NMI to the CPU when a fault is detected, and shows its operation, initial setting, and fault processing.

Figure 24 shows the procedure for initializing the fault report structure. Initialize the fault report structure, and set the fault connection and action when detecting a fault.

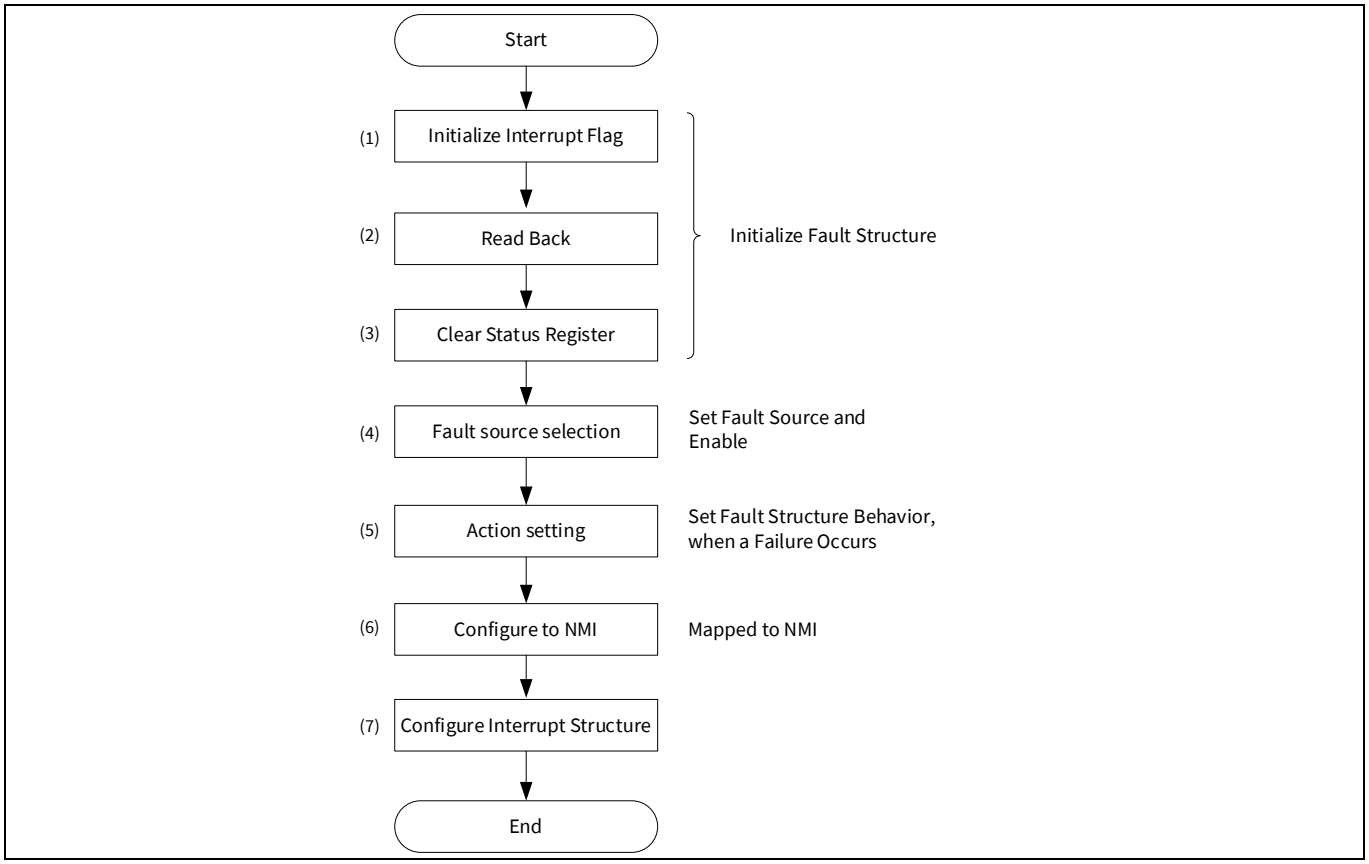

Figure 24 Fault Report Structure Initial Setting Flow for NMI Generation

3.4.2.1 Use case

Note: In this configuration, reset will be issued, when CM0+ violates SMPU protection. PPU violation can cause a bus error depending on the setting of the CPUSS_BUFF_CTL register. See the Architecture TRM [2] and Registers TRM [2] for more details.

- Using fault report structure: #0
- Fault source: PERI_MS_VIO_1 (IDX: 29) CM4 peripheral master interface, PPU violation (*)
- Additional information: DATA0[31:0] is indicated the violating address (*)
  DATA1[0] is User read (*)
  DATA1[1] is User write (*)
  DATA1[2] is User execute (*)
  DATA1[3] is Privileged read (*)
  DATA1[4] is Privileged write (*)
  DATA1[5] is Privileged execute (*)
  DATA1[6] is Non-secure (*)
  DATA1[11:8] is Master identifier (*)
  DATA1[15:12] is Protection context identifier (*)
  DATA1[31:28] is fault identification code (*)
- Interrupt generation
- Mapped to CPU NMI

Note: (*) See the Architecture TRM [2] and Registers TRM [2] for fault assignment and additional information.

3.4.2.2 Configuration

Set according to the **Initial Setting Procedure**. **Table 7** lists the parameters and functions of the configuration part in SDL for fault report structure Initialization.

Table 7 List of Fault Report Structure Initialization Parameters and Functions

Parameters	Description	Value
irq_cfg.sysIntSrc	Set system interrupt index number [0: 1022]	cpuss_interrupts_fault_0_IRQn It is assigned to Fault structure #0 interrupt (Index number = 8)
irq_cfg.intIdx	Set CPU interrupt number [0: 7] CPUIntIdx0_IRQn is assigned to IRQ 0, CPUIntIdx1_IRQn is assigned to IRQ 1, CPUIntIdx2_IRQn is assigned to IRQ 2, CPUIntIdx3_IRQn is assigned to IRQ 3, CPUIntIdx4_IRQn is assigned to IRQ 4, CPUIntIdx5_IRQn is assigned to IRQ 5, CPUIntIdx6_IRQn is assigned to IRQ 6, CPUIntIdx7_IRQn is assigned to IRQ 7,	CPUIntIdx4_IRQn
irq_cfg.isEnabled	Set interrupt enable	True

Fault Report Structure Overview

Parameters	Description	Value
	False: Disabled True: Enabled	
tFlt_Temp.ResetEnable	Set reset request enable: False: Disabled True: Enabled	False
tFlt_Temp.OutputEnable	Set IO output signal enable: False: Disabled True: Enabled	False
tFlt_Temp.TriggerEnable	Set Trigger output enable: False: Disabled True: Enabled	False
Function	**Description**	**Value**
Cy_SysFlt_ClearStatus (FAULT_STRUCT)	Clear fault structure status: FAULT_STRUCT: Fault report structure number FAULT_STRUCT0 = Fault report structure 0 FAULT_STRUCT1 = Fault report structure 1 FAULT_STRUCT2 = Fault report structure 2 FAULT_STRUCT3 = Fault report structure 3	FAULT_STRUCT0
Cy_SysFlt_ClearInterrupt (FAULT_STRUCT)	Clear fault structure interrupt flag:	FAULT_STRUCT0
Cy_SysFlt_SetInterruptMask (FAULT_STRUCT)	Set interrupt enable: FAULT_STRUCT: Fault report structure number	FAULT_STRUCT0
Cy_SysFlt_SetMaskByIdx (FAULT_STRUCT, faultSrc)	Select fault source: FAULT_STRUCT: Fault report structure number faultSrc: Fault Source	FAULT_STRUCT: FAULT_STRUCT0 faultSrc: CY_SYSFLT_MS_PPU_1 It is assigned PERI_MS_VIO_1 (Fault number = 29)
Cy_SysFlt_Init (FAULT_STRUCT, tFlt_Temp)	Set fault structure action: FAULT_STRUCT: Fault report structure number tFlt_Temp: Action parameters	FAULT_STRUCT0
Cy_SysInt_SetIntSourceNMI (Reg_num, sysintSrc)	Set NMI regster Reg_num: CM4 NMI control Register number CPUSS_CM4_NMI_CTLx register (x = 0 to 3) sysintSrc: System interrupt index number	Reg_nim: 0ul Use CPUSS_CM4_NMI_CTL0 register sysintSrc: irq_cfg.sysIntSrc

Code Listing 32 shows an example program of the Fault Report Structure configuration part for NMI generation.

Code Listing 32 Example of Fault Report Structure Configuration

```
cy_stc_sysint_irq_t irq_cfg =
{
    .sysIntSrc  = cpuss_interrupts_fault_0_IRQn,
    .intIdx     = CPUIntIdx4_IRQn,
    .isEnabled  = true,
};

int main(void)
{
    cy_stc_sysflt_t tFlt_Temp = {0ul};
    :
    /* Clear Status register */
    Cy_SysFlt_ClearInterrupt(FAULT_STRUCT0);
    FAULT_STRUCT0->unINTR.u32Register;
    Cy_SysFlt_ClearStatus(FAULT_STRUCT0);

    /* Set the signal interface (Interrupt generation) */
    Cy_SysFlt_SetInterruptMask(FAULT_STRUCT0);

    /* Fault source enable 29 CM4 Peripheral Master Interface PPU violation */
    Cy_SysFlt_SetMaskByIdx(FAULT_STRUCT0, CY_SYSFLT_MS_PPU_1);

    /* Init Sysflt */
    tFlt_Temp.ResetEnable = false;
    tFlt_Temp.OutputEnable = false;
    tFlt_Temp.TriggerEnable = false;
    Cy_SysFlt_Init(FAULT_STRUCT0, &tFlt_Temp);

    /* Set NMI mapping of CM4 */
    Cy_SysInt_SetIntSourceNMI(0ul, irq_cfg.sysIntSrc);

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

    /* CPUSS_BUFF_CTL register setting */
    CPUSS->unBUFF_CTL.stcField.u1WRITE_BUFF = 1ul;
    :
    for(;;)
    {
    }
}
```

Configure interrupt structure parameters.

(1) Initialize interrupt flag. See **Code Listing 24**.

(2) Read back

(3) Initialize status register. See **Code Listing 25**.

(4) Fault source selection. See **Code Listing 26**.

(5) Action Setting. **See Code Listing 27**.

(6) Configure NMI See **Code Listing 10**.

(7) Configure interrupt setting. See **Code Listing 3** and **Code Listing 4**.

Set NMI handler

- Fault Handling Example

  When access violation to the PPU#1 master occurs, the fault report structure captures the fault information and outputs an interrupt signal to the interrupt structure. The interrupt structure routes an interrupt from the fault report structure to the NMI, and notifies the NMI to the CPU.

Fault Report Structure Overview

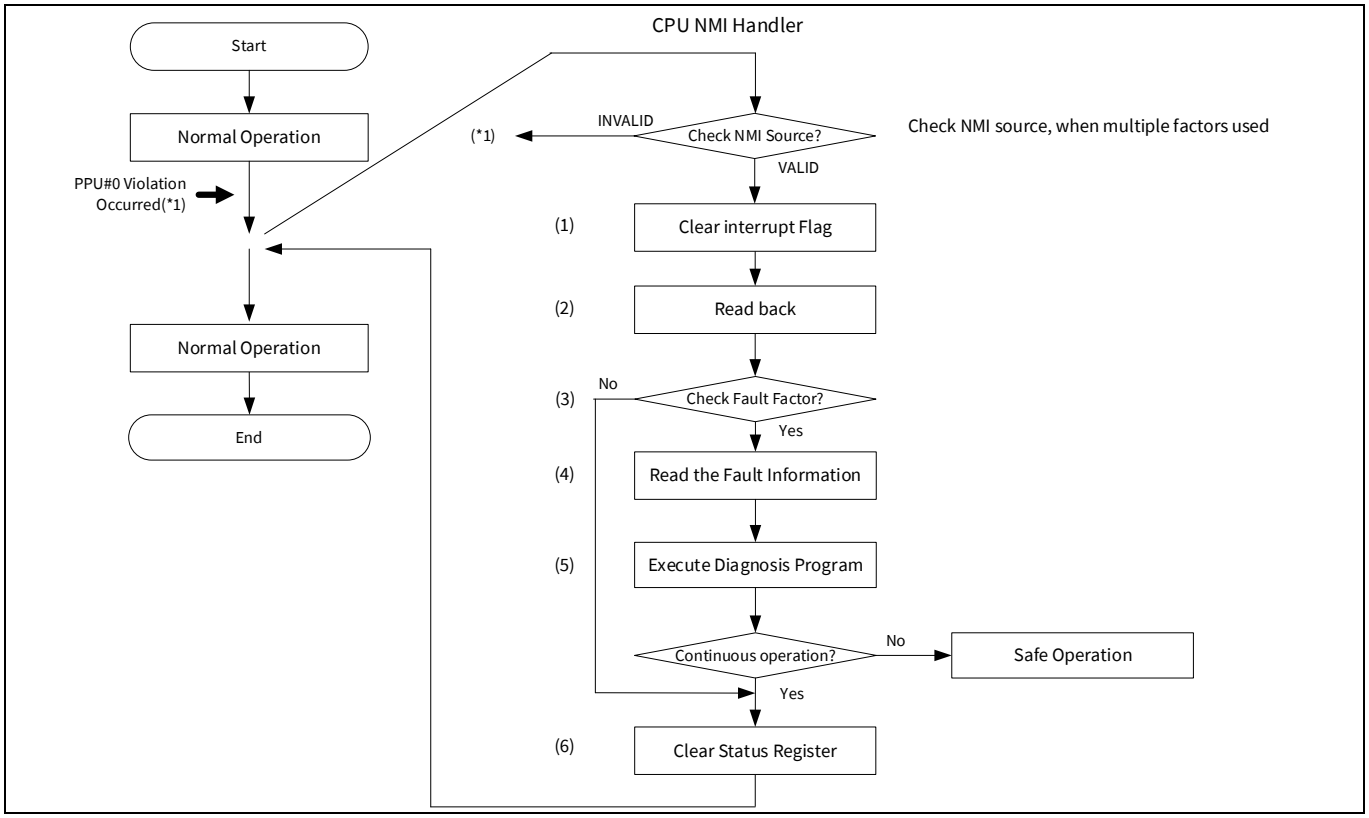

Figure 25 Fault Handling Flow Example

*Note: (*1) 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.*

The software checks the cause of the fault from the additional information, and executes appropriate safety operation or diagnostic programs according to fault situations. If recovery to normal operation is possible after safe operation, return from the interrupt.

Code Listing 33 shows an example program of the fault interrupt handler.

Fault Report Structure Overview

Code Listing 33 Fault Interrupt Handler for NMI Generation

```
void NMI_Handler(void)
{
    cy_en_sysflt_source_t status;

    /* Clear Interrupt flag */
    Cy_SysFlt_ClearInterrupt(FAULT_STRUCT0);
    FAULT_STRUCT0->unINTR.u32Register; /* Read Back */

    status = Cy_SysFlt_GetErrorSource(FAULT_STRUCT0);

    if(status == CY_SYSFLT_MS_PPU_1)
    {
        violatingAddr = Cy_SysFlt_GetData0(FAULT_STRUCT0);
        violatingInfo.u32 = Cy_SysFlt_GetData1(FAULT_STRUCT0);

            /* User program here.. */

    }
    /* Clear Status register */
    Cy_SysFlt_ClearStatus(FAULT_STRUCT0);
}
```

NMI handler registered in the configuration part

(1) Clear interrupt flag. See **Code Listing 24**.

(2) Read back

(3) Check fault factor. See **Code Listing 29**.

(4) Read the fault information. See **Code Listing 30**.

(5) Execute diagnosis and Safe Operation (Fault handling)

(6) Clear status register. See **Code Listing 25**.

4 Glossary

Table 8

Terms	Description
CPU	Central Processing Unit
NVIC	Nested vectored interrupt controller
WIC	Wakeup interrupt controller
NMI	Non-maskable interrupt
DeepSleep	Low Power mode in Traveo II family series. See the “Device Power Mode” chapter of the Architecture TRM [2] 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 [2] for details.
SMPU	Shared Memory Protection Unit. See the “Protection Unit” chapter of the Architecture TRM [2]for details.
PPU	Peripheral Protection Unit. See the “Protection Unit” chapter of the Architecture TRM [2] for details.
GPIO	General purpose input/output. See the “IO System” chapter of the Architecture TRM [2] for details.
TCPWM	Timer, Counter, and Pulse Width Modulator. See the “Timer, Counter, and PWM” chapter of the Architecture TRM [2] for details.
RTC	Real-time clock. See the “Real-time Clock” chapter of the Architecture TRM [2]for details.

References

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
- CYT3BB/4BB Datasheet 32-Bit Arm® Cortex®-M7 Microcontroller Traveo™ II Family

[2] Technical Reference Manuals

- Body Controller Entry Family
  - 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
- Body Controller High Family
  - Traveo™ II Automotive Body Controller High Family Architecture Technical Reference Manual (TRM)
  - Traveo™ II Automotive Body Controller High Registers Technical Reference Manual (TRM) for CYT4BF
  - Traveo™ II Automotive Body Controller High Registers Technical Reference Manual (TRM) for CYT3BB/4BB
- Cluster 2D Family
  - Traveo™ II Automotive Cluster 2D Family Architecture Technical Reference Manual (TRM)
  - Traveo™ II Automotive Cluster 2D Registers Technical Reference Manual (TRM)

[3] Application Note

- **AN220193 - GPIO USAGE SETUP IN TRAVEO II FAMILY**
- AN220224 – HOW TO USE TIMER, COUNTER, AND PWM (TCPWM) 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
**	03/09/2018	New Application Note.
*A	11/29/2018	Changed target parts number (CYT2B series). Added Note (*1) and VTOR explanation in **Initial Setting** Changed IDX from 17, 18, and 19 to 21, 22, and 23 in **Peripheral Interrupt Handling** Changed IDX from 274 and 275 to 273 and 274 in **NMI Generation of System Interrupts** Changed IDX from 275 to 274 in **Software Interrupt (Using Peripheral)** Changed IDX from 47 to 8 in **NMI Generation via Interrupt Structure**
*B	02/22/2019	Added target parts number (CYT4B series).
*C	07/25/2019	Added target parts number (CYT4D series).
*D	02/14/2020	Changed target parts number (CYT2/ CYT4 series) Added target parts number (CYT3 series)
*E	2020-10-22	Changed **Figure 3** Updated sections **2.3**, **2.4**, **2.6**, **3.2**, **3.3**, **3.4** - Changed Flowchart - Change Use case - Added Configuration and Code Updated **References** Added information on the Sample Driver Library in **Other References**

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

Edition 2020-10-22
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Go to www.cypress.com/support

Document reference
002-19842 Rev. *E

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

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

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

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

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