PSoC™ 4 interrupts

About this document

Scope and purpose

This application note explains the interrupt architecture in PSoC™ 4 and its configuration in ModusToolbox™ software environment and PSoC™ Creator. This document serves as a guide in developing interrupt-based projects. Advanced interrupt concepts such as latency, vector selection, interrupt code optimization, and debug techniques are also explained.

Intended audience

This document is intended for anyone using interrupts with the PSoC™ 4 family.

Table of contents

1 Introduction

Interrupts are an important part of any embedded application. An interrupt frees the CPU from having to continuously poll for the occurrence of an event; it notifies the CPU only when that event occurs. In system-on-chip (SoC) architectures such as PSoC™, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU.

PSoC™ 4 devices are supported in both the ModusToolbox™ software environment and PSoC™ Creator; this document covers both IDEs. ModusToolbox™ supports only a limited number of devices. To check whether a device is supported, start ModusToolbox™, choose **New Application** > **PSoC™ 4 BSPs**, and under PSoC™ 4 BSPs is a list of supported PSoC™ 4 devices.

The document begins with an explanation of PSoC™ 4 interrupt architecture. If you want to learn about the interrupt support in ModusToolbox™, skip to ModusToolbox™ interrupt support. For the PSoC™ Creator IDE, skip to the PSoC™ Creator interrupt support. For sample code examples, see ModusToolbox™ related code examples and PSoC™ Creator related code examples. If you are debugging an interrupt project, go to Debugging tips, which provides guidance on finding and resolving interrupt issues.

This application note assumes that you are familiar with PSoC™, and the ModusToolbox™ or PSoC™ Creator IDE. If you are new to PSoC™, you can find an introduction in the application note AN79953 - Getting started with PSoC™ 4 MCU or visit the ModusToolbox™ software or PSoC™ Creator home page.

2 PSoC™ 4 interrupt architecture

Figure 1 shows a simplified block diagram of the interrupt architecture in PSoC™ 4.

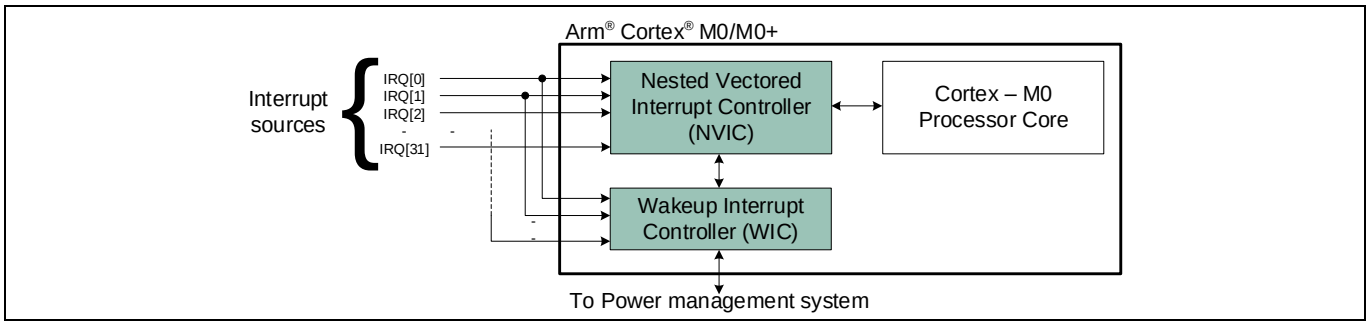

Figure 1 PSoC™ 4 interrupt architecture

There are up to 32 interrupt lines – IRQ[0] to IRQ[31] – each with four priority levels, 0 to 3. Each interrupt line is assigned an interrupt vector address. The CPU branches to this address after receiving an interrupt request, where a special function called an interrupt service routine (ISR) is executed.

Interrupt signals are received by the nested vectored interrupt controller (NVIC). When an interrupt signal becomes active, the NVIC sends the interrupt vector address to the processor core along with the interrupt request signal. In return, the processor core sends an acknowledgment when the ISR is entered and exited. The NVIC is responsible for enabling/disabling an interrupt based on the user configuration. It also resolves interrupt priority when multiple requests occur at the same time, and supports nested interrupts to allow a higher-priority interrupt to be serviced leaving a low-priority ISR.

The wakeup interrupt controller (WIC) block allows the device to wake up from low-power modes – Sleep, Deep Sleep, and Hibernate – using interrupts. The WIC block remains active while the NVIC, processor core, and other device peripherals are shut down. When an interrupt triggers, the WIC activates the power management system, which restores the NVIC and the processor core along with other peripherals. The NVIC then takes over and the processor core executes the ISR. There are several sources in the PSoC™ 4 device that can wake up the device. For example, Figure 1 shows IRQ[0] and IRQ[1] routed to the WIC along with the NVIC. These are the interrupt lines from GPIOs.

Note: Only sources that can wakeup the device from Sleep, Deep Sleep, or Hibernate are routed to the WIC while all interrupt sources are routed to the NVIC.

PSoC™ 4 provides the following interrupt features:

- **Configurable interrupt vector address**: CPU execution can be directly branched to any ISR code when the interrupt occurs.
- **Flexible interrupt sources**: In traditional microcontrollers, the interrupt source is hard-wired to each interrupt line. PSoC™ gives you the flexibility to choose the interrupt source for each interrupt line. This flexible architecture enables any digital signal to be configured as an interrupt source.

2.1 Interrupt sources

PSoC™ 4 interrupt sources are of two types:

1. Fixed-function interrupt sources: These are a predefined set of interrupt sources from on-chip peripherals.
2. Universal digital block (UDB) interrupt sources (available in PSoC™ 4200, 4200 Bluetooth® low-energy (LE), 4200DS, 4200M, and 4200L product lines): UDBs are building blocks for different digital functions such as timer, PWM, UART, SPI, and many more. A UDB consists of programmable logic (PLDs), datapath, and flexible routing. In contrast to fixed-function interrupt sources, any digital signal generated in a UDB can trigger an interrupt. The signals are routed to the interrupt controller through a routing fabric called the digital system interconnect (DSI). See the PSoC™ 4 reference manual for more information.

Note: ModusToolbox™ does not currently support UDBs.

Table 1 shows the interrupt sources. Interrupt sources mentioned in the table are available in all PSoC™ 4 parts unless noted otherwise. For details on each interrupt source, see the PSoC™ Creator Component datasheets or peripheral driver library (PDL) listed in Table 1. Appendix A shows the complete list of interrupt sources depending on the device.

Note: PSoC™ 4 PDL is currently in Alpha and not all peripherals are supported.

Table 1 PSoC™ 4 interrupt sources

ModusToolbox™ PDL	PSoC™ Creator component datasheets	Details	
GPIOs	GPIOs	Each port consists of eight pins. Each pin can generate an interrupt, but the vector address is common for all pins in a port. Firmware must identify the pin that caused the interrupt. PSoC™ 4 enables an interrupt trigger on the rising edge, falling edge, or both edges of the GPIO signal. This interrupt can wake the device from Sleep, Deep Sleep, and Hibernate modes.	
Low Power Comparator (LPCOMP)	Low Power Comparator (LPCOMP)	Like GPIOs, an interrupt can be triggered on the rising edge, falling edge, or both edges of the comparator output signal. The LPCOMP can also wake the device from Sleep, Deep Sleep, and Hibernate modes. LPCOMP is not available in PSoC™ 4000.	
WDT	WDT	The watchdog timer (WDT) is a timer that can reset the device or generate an interrupt. PSoC™ 4000, 4000S, 4100S, 4100S Plus, and 4100PS devices have a 16-bit free-running WDT, whereas other PSoC™ 4 parts have two 16-bit WDTs and one 32-bit WDT. The WDT can wake the device from Sleep and Deep Sleep modes.	
SCB	SCB	PSoC™ 4 has up to five serial communication blocks (SCB), which can be configured as I^2C, SPI, or UART. The exact number of SCB blocks depends on the device family.	
		I^2C	The following events generate an interrupt: arbitration lost, slave address match, start/stop detect, bus error, byte/word transfer complete, TX FIFO not full, TX/RX FIFO empty, RX FIFO not empty, RX FIFO overrun, and RX FIFO full. The slave address

PSoC™ 4 interrupt architecture

ModusToolbox™ PDL	PSoC™ Creator component datasheets	Details	
			match event can wake the device from Sleep and Deep Sleep modes.
		SPI	The following events generate an interrupt: transfer done, idle, TX FIFO not full, TX/RX FIFO empty, byte/Word transfer complete, RX FIFO is not empty, attempt to write to a full RX FIFO, and RX FIFO full.
		UART	The following events generate an interrupt: transmission done, UART TX received a NACK in SmartCard mode, UART arbitration lost in LIN or SmartCard mode, frame error, parity error, LIN baud rate detection complete, and LIN successful break detection. It can also wake up the device from low-power modes[1].
SysTick	SysTick	SysTick is a 24-bit timer built into the Arm® Cortex®-M0/Cortex® M0+ processor. It is generally used by real-time operating systems (RTOS) as a tick timer. However, it can be used as a general-purpose timer. See the ModusToolbox™ SysTick timer and PSoC™ Creator SysTick timer section for more information.	
SAR ADC	SAR ADC	The successive approximation register analog-to-digital converter (SAR ADC) can generate interrupts on end of conversion, data overflow, scan collision, data saturation, and data over-range events.	
CAPSENSE™ (CSD)	CAPSENSE™ (CSD)	CSD, used for touch applications, generates an interrupt when the sensor scan is complete.	
Timer, Counter, and Pulse Width Modulator (TCPWM)	Timer, Counter and Pulse Width Modulator (TCPWM)	The TCPWM block can be configured to work as a 16-bit timer, counter, or PWM. It can generate interrupts on terminal count, input capture signal, or a “compare true” event.	
CAN with Flexible Data-Rate (CAN FD)	Controller Area Network (CAN)	PSoC™ 4200M and PSoC™ 4200L devices have two CAN blocks. PSoC™ 4100S Plus device has one CAN block. The CAN block can generate interrupts on events such as message received, message sent, and various error events. See the CAN chapter of the reference manual for more information.	
Direct Memory Access Controller (DMAC)	Direct Memory Access (DMA)	PSoC™ 4100M/4200M, PSoC™ 4200L, PSoC™ 4100S Plus, and PSoC™ 4100PS devices have DMA to transfer data between peripherals. An interrupt can be generated when the data transfer is completed.	
Not currently supported in ModusToolbox™	Universal Digital Block (UDB)	UDB implementations such as timer, PWM, counter, UART, and so on can generate interrupts on different events similar to their fixed-function	

[1] There are pin limitations; not all ports have dedicated interrupts. If the UART selected pins do not have dedicated port interrupt, it cannot wake up the device. See the "Interrupts" chapter in device *Architecture reference manual* to learn about ports that have dedicated interrupts.

ModusToolbox™ PDL	PSoC™ Creator component datasheets	Details
		counterparts. UDBs are available in PSoC™ 4200, 4200 BLE, 4200DS, 4200M, and 4200L product lines.
USB Full-Speed device (USBFS)	Full Speed USB (USBFS)	PSoC™ 4200L has USB with start-of-frame interrupt and interrupt on completion of the communication over data endpoints.
In Progress	CTB/CTBm	Provides continuous time analog functionality. It generates interrupts on event such as comparator triggers.
In Progress	WCO WDT/WCO	PSoC™ 41000S and PSoC™ 4100S Plus have timers that can be clocked by WCO. These timers can generate interrupts.

2.2 Level- and edge-triggered interrupts

PSoC™ 4 supports level and edge triggering for interrupts. Figure 2 shows the logic to select the trigger type. This logic is present for each interrupt line supported by NVIC. Note that the fixed-function interrupt can only be configured to level, but for the DSI sources, which include the UDB, the interrupt can be rising-edge triggered as well as level-triggered. The rising-edge detect block generates a pulse at every rising edge of the DSI interrupt signal. See the timing diagrams (Figure 3 and Figure 4) to know how the NVIC responds to level- and edge-configured interrupts.

Currently, ModusToolbox™ does not support digital signal interconnect (DSI) or universal digital blocks (UDBs), this means that the only interrupts supported are fixed-function interrupts. PSoC™ Creator does support both fixed function interrupts and UDB based interrupts.

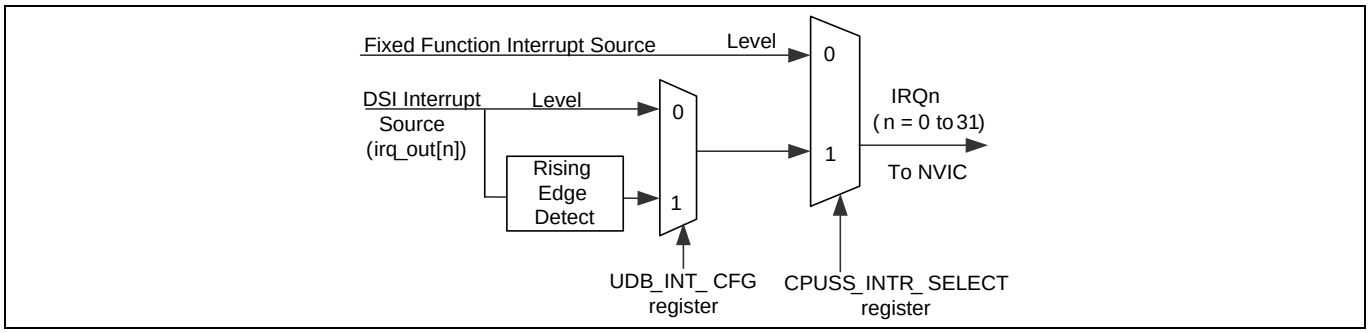

Figure 2 Level trigger and edge trigger

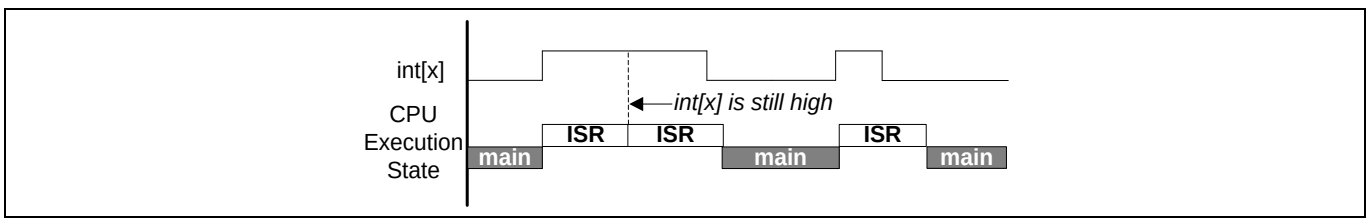

Figure 3 Level-triggered interrupts

PSoC™ 4 interrupt architecture

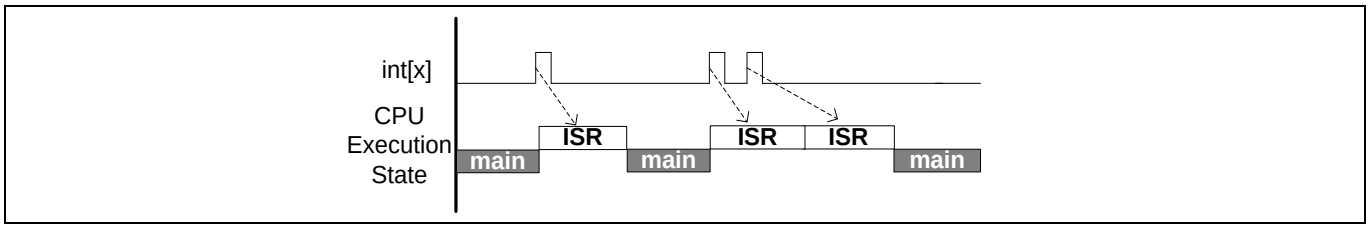

Figure 4 Edge-triggered interrupts

Note: The GPIO interrupt logic has additional circuitry to support interrupts on the rising edge, falling edge, and both edges. See the PSoC™ 4 reference manual for more information.

3 ModusToolbox™ interrupt support

ModusToolbox™ software environment does not use Components as in PSoC™ Creator, but uses a Device Configurator tool. The Device Configurator is used to enable and configure device peripherals such as clocks and pins, as well as standard MCU peripherals that do not require their own tool.

As there is currently no DSI or UDB support in ModusToolbox™, an interrupt can only be connected through a dedicated route. To open the Device Configurator, go to the **Quick Panel** and select the Device Configurator, as shown in Figure 5.

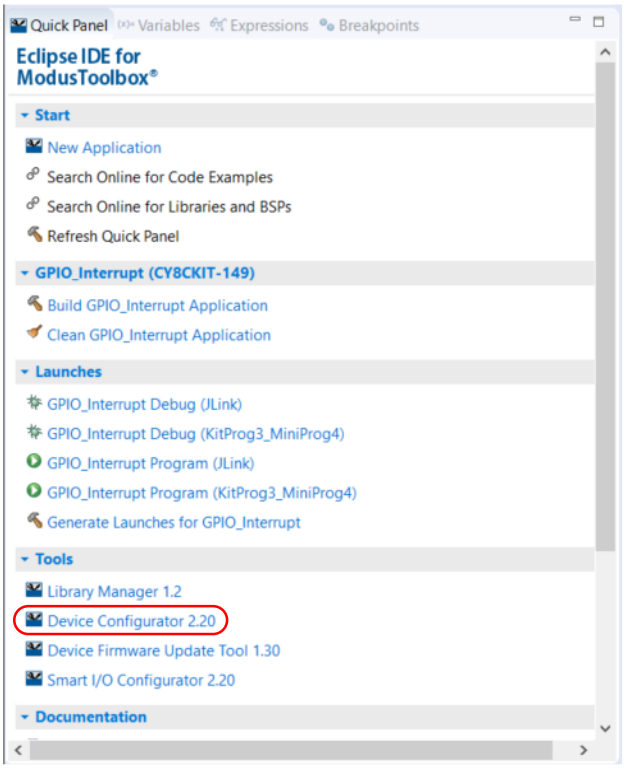

Figure 5 Device Configurator quick start

3.1 Enabling interrupt sources

As mentioned previously, all the interrupts are fixed function, and fixed function interrupts route directly to the NVIC. A peripheral that has an interrupt source has options to enable that interrupt from inside the Device Configurator. For example, the GPIO has a fixed function interrupt that can be configured as shown in Figure 6. The interrupt source parameters vary based on the peripheral.

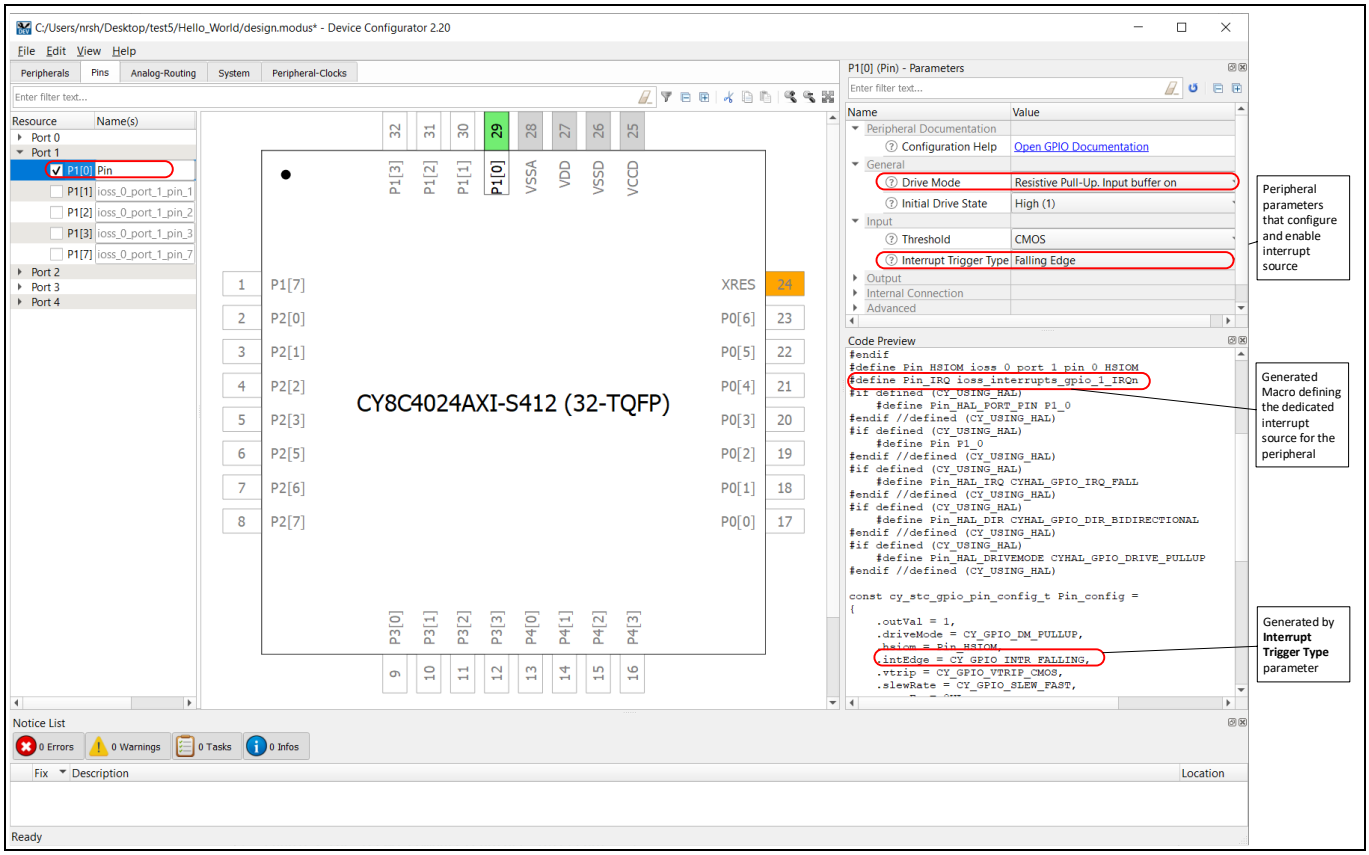

Figure 6 TCPWM fixed function interrupt configuration

Currently the only interrupts that can be used in ModusToolbox™ are fixed-function interrupts, this means that all interrupt types are level driven. Some peripherals have added hardware that can be configured so that an interrupt can be triggered on a rising edge.

An interrupt source does need to be configured in the Device Configurator but can be configured in software through the peripheral driver library (PDL). The peripheral that supplies the interrupt source includes API to enable and disable interrupts. An example of this can be seen in Enabling interrupt sources using PDL.

Based on the configuration, ModusToolbox™ generates the 'C' code to achieve the desired configuration. The code generated can be viewed in the Code Preview pane; it is added to relevant cycfg_xxx.c/h files found under <ApplicationName>/libs/TARGET_<TargetName>/COMPONENT_BSP_DESIGN_MODUS/GeneratedSource folder in the ModusToolbox™ project workspace window. The generated code includes macros defining the interrupt source numbers and any peripheral configuration that is necessary to set up and enable the interrupt source. This simplifies the process of searching for the dedicated interrupt numbers in the device header file. The user application only needs to enable the interrupt vector on the CPU and assign an interrupt handler function as described in Setting up an interrupt.

3.2 Enabling interrupt sources using PDL

An interrupt source does not need to be enabled from the Device Configurator but can be enabled using the peripheral driver library (PDL). The peripheral that supplies the interrupt source has API functions that allow the interrupt source to be enabled/disabled, set/cleared, and allows the status to be read from the peripheral's hardware. An example using GPIO shows how to enable an interrupt source, see Figure 7. This can be done for any peripheral that has a fixed function interrupt.

```
/* Get the interrupt edge setting of P1.0 */
if(CY_GPIO_INTR_RISING == Cy_GPIO_GetInterruptEdge(P1_0_PORT, P1_0_NUM))
{
    /* Set the interrupt trigger type to falling edge for P1.0 */
    Cy_GPIO_SetInterruptEdge(P1_0_PORT, P1_0_NUM, CY_GPIO_INTR_FALLING);
}
```

Figure 7 TCPWM enabling interrupt

3.3 Configuring interrupts using PDL

The peripheral driver library (PDL) is a software development kit (SDK) that enables firmware development for PSoC™ 4 MCU devices. PDL API function calls are used to configure, initialize, enable, and use a peripheral driver. One such driver is system interrupts (SysInt). SysInt provides structures and functions to configure and enable interrupt functionality. PDL also supports the Interrupts and Exceptions (NVIC) functions used for interrupt configuration.

3.3.1 Interrupt API functions

ModusToolbox™ generates an API –*.c* and *.h* files – for each peripheral in the project. These APIs include functions to configure and use each peripheral. The following API functions are associated with an interrupt:

- **Cy_SysInt_Init (const cy_stc_sysint_t *config, cy_israddress userIsr)**
  Initializes the referenced interrupt by setting the priority and the interrupt vector. Use the Cortex® microcontroller system interface standard (CMSIS – Library is supplied by Cypress and is retrieved automatically when required) core function NVIC_EnableIRQ(config.intrSrc) to enable the interrupt.
- **Cy_SysInt_SetVector (IRQn_Type IRQn, cy_israddress userIsr)**
  Changes the ISR vector for the interrupt. This function relies on the assumption that the vector table is relocated to __RAM_VECTOR_TABLE[RAM_VECTORS_SIZE] in SRAM. Otherwise, it returns the address of the default ISR location in the flash vector table.
- **Cy_SysInt_GetVector (IRQn_Type IRQn)**
  Gets the address of the current ISR vector for the interrupt. This function relies on the assumption that the vector table is relocated to __RAM_VECTOR_TABLE[RAM_VECTORS_SIZE] in SRAM. Otherwise, it returns the address of the default ISR location in the flash vector table.

3.3.2 Critical section control functions

ModusToolbox™ also provides a set of generic interrupt functions in the *cy_syslib.h* and *cy_syslib.c* files. The important ones are *Cy_SysLib_EnterCriticalSection* and *Cy_SysLib_ExitCriticalSection*. These two functions are used to avoid the corruption of firmware variables and hardware registers. *Cy_SysLib_EnterCriticalSection* disables interrupts and returns an interrupt state value. *Cy_SysLib_ExitCriticalSection* restores the interrupt state.

To see how this works, consider an example of writing to a timer control register:

```
TCPWM_BLOCK_CONTROL_REG |= TCPWM_MASK;
```

The following sequence of operations occurs while executing the statement above:

1. The CPU reads the control register of the TCPWM and stores it in a temporary register.
2. The CPU executes a logical OR operation of the temporary register with its mask value.
3. The CPU loads the OR result back to the control register.

Between steps 1 and 2, an interrupt may occur, and its ISR may load a new value into the same control register. After executing the ISR, when the CPU resumes executing step 2, it uses the stale control register value, which was in the temporary register– this leads to data corruption.

To avoid this issue, add the following code:

```
uint32_t InterruptState;

InterruptState=Cy_SysLib_EnterCriticalSection();

TCPWM_BLOCK_CONTROL_REG |= TCPWM_MASK;

Cy_SysLib_ExitCriticalSection(InterruptState);
```

The *Cy_SysLib_EnterCriticalSection* and *Cy_SysLib_ExitCriticalSection* functions solve the problem by disabling interrupts while the control register is being written. Use these functions when a shared variable or register is being written.

3.3.3 Setting up an interrupt

These steps use PDL and NVIC APIs to set up an interrupt to trigger on a signal from a peripheral.

1. Configure the peripheral to generate the interrupt. For example, for a GPIO, configure the drive mode (pull up or pull down), interrupt signal generation on falling or rising edge, and unmask the interrupt. Refer to the PDL API reference documentation for your peripheral for this information.
2. Configure the interrupt using the structure provided by the SysInt API.
   The structure is defined in the PDL SysInt driver file *cy_sysint.h*, as shown in Code Listing 1.

Code Listing 1

```
/**
* Initialization configuration structure for a single interrupt channel
*/
typedef struct {
    IRQn_Type       intrSrc;        /**< Interrupt source */
    uint32_t        intrPriority;   /**< Interrupt priority number (Refer to
__NVIC_PRIO_BITS) */
} cy_stc_sysint_t;
```

a) Interrupt Source (intrSrc)
   - These are the dedicated interrupt numbers as defined under the drivers header file (*cy_sysint.h*).
   - Each number represents an interrupt that routes from a peripheral to the NVIC.

b) Interrupt Priority (intrPriority)
   - Sets the priority of the interrupt. PSoC™ 4 supports priorities 0 to 3.

3. Call Cy_SysInt_Init(&SysInt_SW_cfg_1, ISR_1_handler).

   Here, SysInt_SW_cfg_1 is the name of the configured structure from step 2. ISR_1_handler is the name of the interrupt handler that executes when the interrupt triggers. This function applies the routing and priority configuration of the interrupt but does not enable it.
4. Call NVIC_ClearPendingIRQ(SysInt_SW_cfg_1.intrSrc) to clear any pending interrupts.
5. Call NVIC_EnableIRQ(SysInt_SW_cfg_1.intrSrc) to enable the interrupt.
6. Call the __enable_irq() function to enable global interrupts. This is safe to perform as the first step, as individual CPU interrupts have not been enabled yet. You can also perform this later, but interrupts are disabled at startup until this is called.

4 PSoC™ Creator interrupt support

In PSoC™ Creator, properties of interrupts such as the level or edge trigger, vector address, and interrupt priority must be configured using the PSoC™ Creator Interrupt Component. This Component is available under the System tab in the Component Catalog window, as shown in Figure 8.

Each instance of the Interrupt Component uses one interrupt line out of the 32 lines that go to NVIC. In the example shown in Figure 8, the end-of-conversion (eoc) signal from the SAR ADC is connected to the Interrupt Component "isr_1." The SAR ADC has an allotted vector line of the NVIC (see Appendix A). For example, in PSoC™ 4200, IRQ14 is allotted for SAR ADC interrupt. Thus, the Interrupt Component "isr_1" wires the eoc signal to the IRQ14 line through the MUX logic shown in Figure 2.

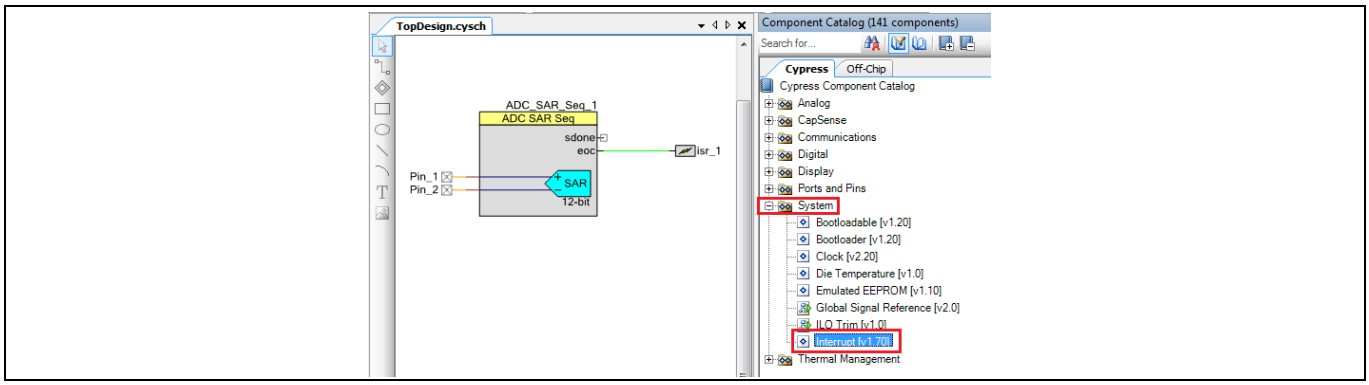

Figure 8 PSoC™ Creator interrupt component

4.1 Interrupt component configuration

Figure 9 shows the interrupt component configuration dialog. There are three options in the Component: DERIVED, RISING_EDGE, and LEVEL.

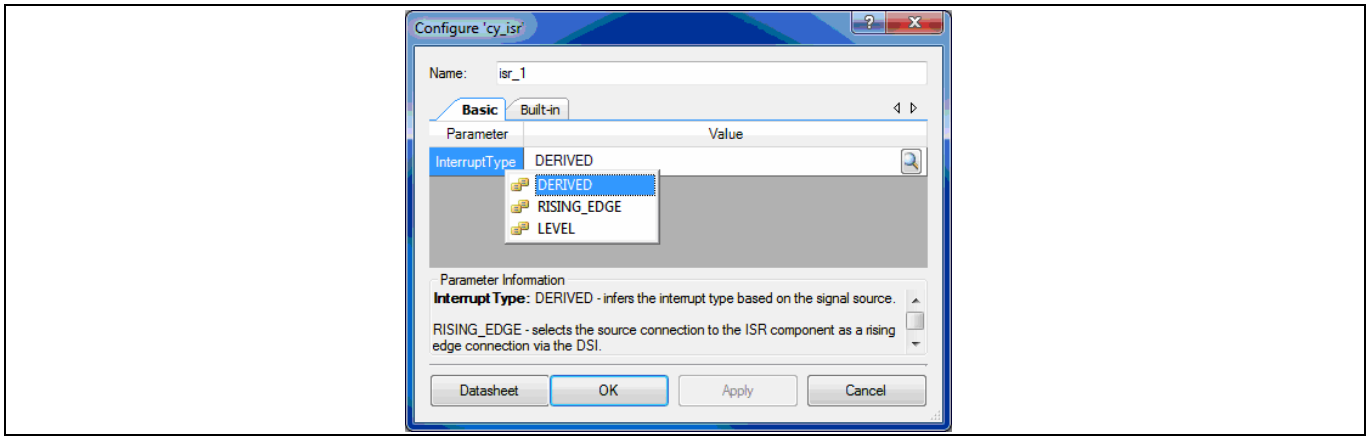

Figure 9 Interrupt component configuration

This setting configures the multiplexers shown in Figure 2. The selection of a particular option depends on the interrupt source (fixed-function or UDB/DSI) and the application requirements.

Fixed-function blocks: The interrupt line from the fixed-function block is always routed through the "dedicated route" as shown by the red line in Figure 10. When configured to this path, the interrupt is level-triggered and the vector number is determined based on the hardware block being used. The Interrupt

PSoC™ Creator interrupt support

Component (isr_1) connected to the interrupt line can only be configured as level-triggered. Setting the interrupt to RISING_EDGE trigger results in a build error. When configured to DERIVED, the tool selects Level interrupt only.

In PSoC™ 4 devices with DSI, other output signals from the fixed-function block can be routed for interrupts. This allows the RISING_EDGE option as shown by the blue line in Figure 10 for the “line” output of a PWM Component. The Interrupt Component (isr_2) connected to the output of the PWM can be configured to Level or RISING_EDGE. When the DERIVED option is selected, the tool selects the level trigger configuration. Level trigger in such cases is usually not useful as it causes the ISR to be repeatedly executed as long as the signal is HIGH, and so in most cases, RISING_EDGE is used.

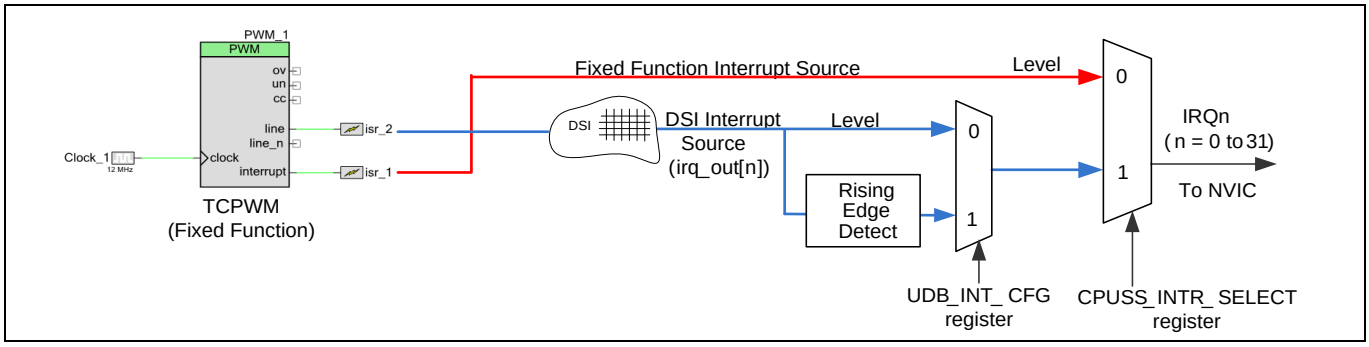

Figure 10 Interrupt routing for fixed-function blocks

UDBs: For UDBs, the DSI is used to route the signal (from the interrupt line of the UDB Component or any output) to the MUX logic as shown in Figure 11. Thus, both LEVEL and RISING_EDGE options are available for any signal from the UDB. When the DERIVED option is selected in the Interrupt Component (isr_1 or isr_2), the RISING_EDGE option is configured. This is in contrast to the case of the DSI signal routing for fixed-function block outputs.

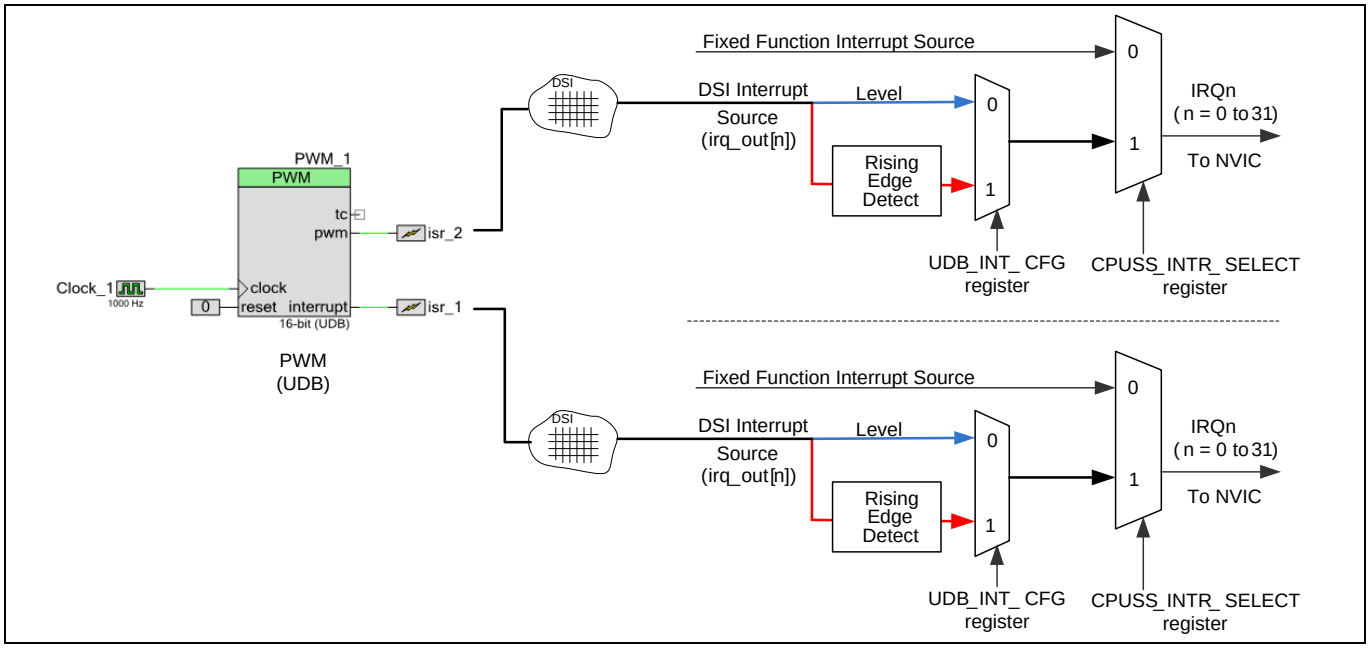

Figure 11 Interrupt routing for UDBs

PSoC™ Creator interrupt support

*Note: PSoC™ 4 Bluetooth® LE, PSoC™ 4200M, and PSoC™ 4200L parts have eight DSI channels with each channel demultiplexed to **4** to spread across 32 **(8x4)** interrupt lines for the Arm Cortex®-M0/M0+ processor. Thus, the maximum number of DSI interrupts is limited to **eight** in a design.*

Table 2 provides guidelines for setting the InterruptType parameter in the Interrupt Component.

Table 2 Interrupt component configuration

Interrupt source	Signal	Interrupt component configuration
Fixed-function	Interrupt	Select LEVEL or DERIVED. RISING_EDGE is not allowed.
	Block output	Select RISING_EDGE; otherwise, the interrupt is repeatedly triggered for the duration of the logic HIGH signal state.
UDB function	Interrupt	Select RISING_EDGE or DERIVED.
	Block output	Select RISING_EDGE; selecting LEVEL causes the interrupt to be repeatedly triggered for the duration of the logic HIGH signal state.

4.1.1 Sticky bits

An interrupt signal may be “sticky”, which means that the interrupt line remains active (HIGH) until it is read or cleared. In this case, if the Interrupt Component is configured to RISING_EDGE, the ISR is executed once. If the Interrupt Component is configured to LEVEL, the ISR is executed repeatedly. To handle this, clear the interrupt source by using the API function provided by the Component. See the Component datasheet of the interrupt source. You can also refer to the section, which provides an example using the timer interrupt.

Note that when the output lines of a fixed-function block or the UDB (for example, the “pwm” line of a PWM Component as shown in Figure 12) are connected to the Interrupt Component instead of the) sectioninterrupt line, there is no need to clear the interrupt. However, the ISR is repeatedly executed as long as the signal is HIGH, if the interrupt Component is configured to LEVEL.

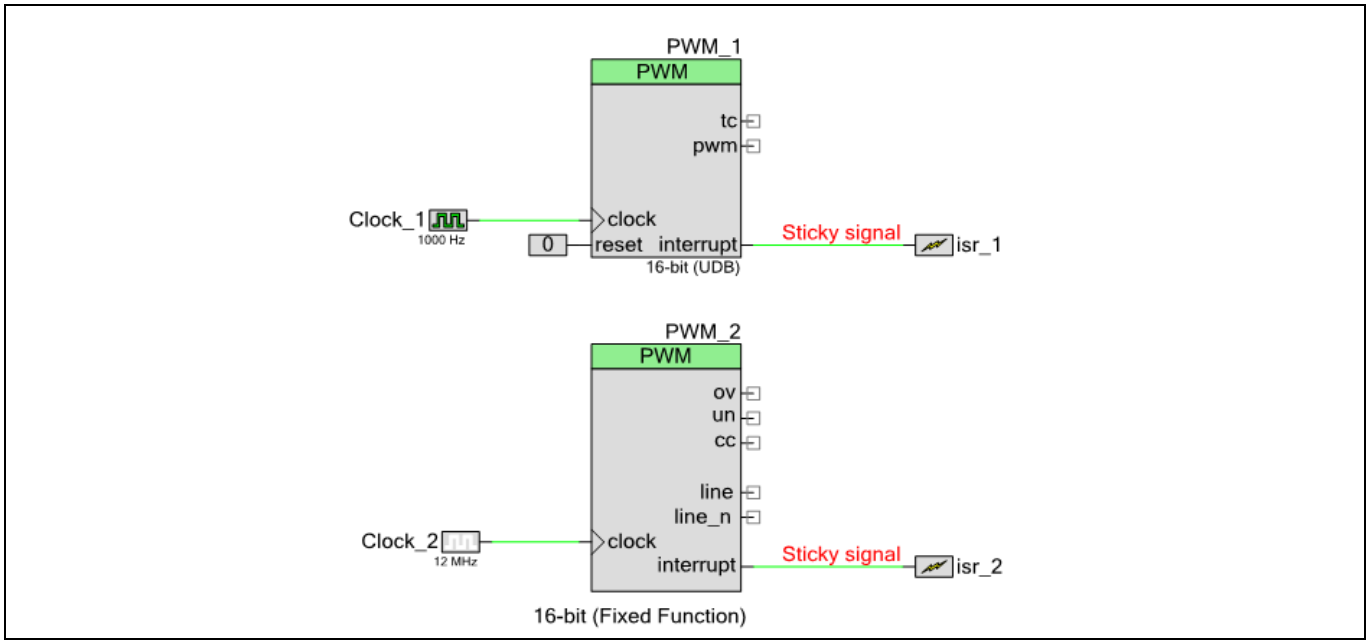

Figure 12 Sticky signal

4.2 Interrupt priority configuration

The design-wide resources window (*project_name.cydwr*) of the PSoC™ Creator project has an Interrupts tab, which displays the Interrupt Component instance names, their priorities, and vector numbers, as Figure 13 shows. isr_1, isr_2, and isr_3 are the Interrupt components used in the design.

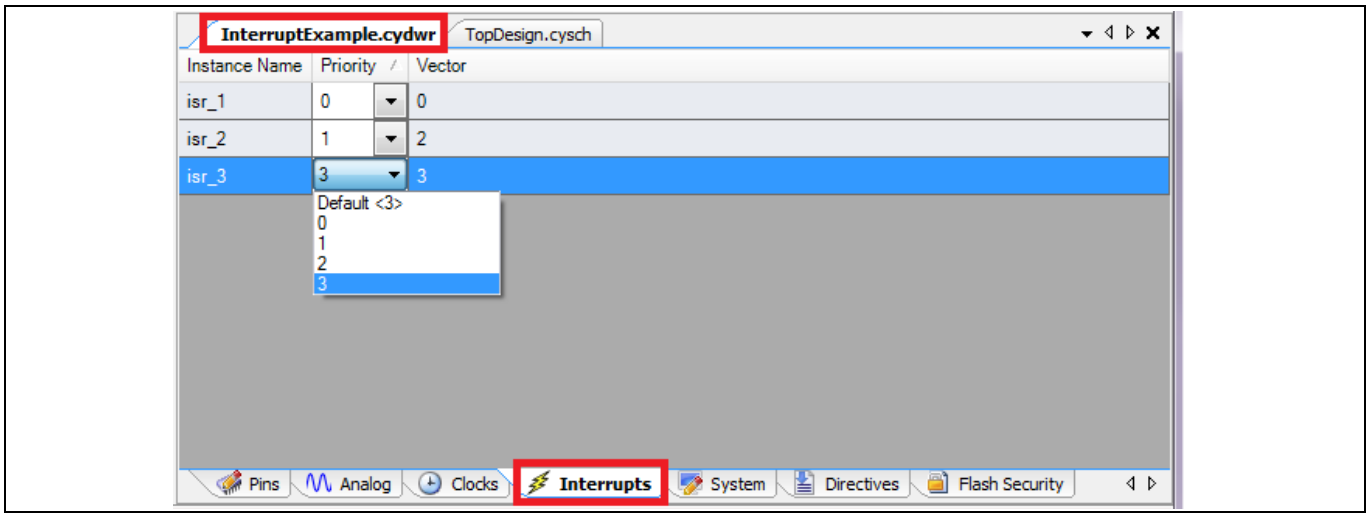

Figure 13 Interrupt tab in *cydwr* window

Use the *cydwr* window to change the priority of an interrupt. Note that 0 is the highest priority; and 3, the lowest priority. The Cortex®-M0/M0+ CPU supports interrupt nesting; see Nested interrupts for details.

The interrupt vector number for each Interrupt Component is automatically assigned by PSoC™ Creator when the project is built, but can be manually changed. See PSoC™ Creator forcing interrupt vector num for details. Also, note that the vector number is shown with an offset in the *.cydwr* window. Vector number 0 corresponds to exception number 16 in Cortex®-M0/Cortex® M0+. See Exceptions for an overview of Cortex®-M0/Cortex® M0+ exceptions.

4.3 Interrupt API functions

PSoC™ Creator generates an API –*.c* and *.h* files – for each Component in the project. These APIs include functions to configure and use the hardware corresponding to the Component. The following API functions are associated with an Interrupt Component:

- `<instance_name> Start()` and `<instance_name>_Stop()`
  `Start()` enables the interrupt, sets its vector to the default ISR, and sets the interrupt priority.
  `Stop()` disables the interrupt.
- `<instance_name>_StartEx()`
  Similar to `Start()`; the only difference is that this function takes a vector address as an input, enabling you to write a custom ISR rather than using the default ISR generated by the Component.
- `<instance_name> Enable()` and `<instance_name>_Disable()`
  These functions are called internally by `Start()` and `Stop()` to enable and disable the interrupt. These functions can be called to dynamically enable and disable an interrupt.
- `<instance_name> SetVector()` and `<instance_name> SetPriority()`

These functions are called internally by `Start()` and `Stop()` to set the interrupt vector address and the interrupt priority. These functions can also be called to dynamically set the vector and the priority. Make sure that the interrupt is disabled before calling these functions.

- `<instance_name> SetPending()`
  Makes the interrupt pending without an interrupt request, that is, under firmware control.
- `<instance_name> ClearPending()`
  Clears the pending status of the interrupt so that it is not serviced. This function does not have any effect on the interrupt source signal; it only clears the pending status bit of the interrupt line in the NVIC.

See the Interrupt component datasheet for a detailed explanation of the API.

4.3.1 Critical section control functions

PSoC™ Creator also provides a set of generic interrupt functions in the *CyLib.h and CyLib.c* files. These files are generated when the project is built. The important ones are `CyEnterCriticalSection` and `CyExitCriticalSection`. These two functions are used to avoid the corruption of firmware variables and hardware registers. `CyEnterCriticalSection` disables interrupts and returns an interrupt state value. `CyExitCriticalSection` restores the interrupt state.

To see how this works, consider an example of writing to a timer control register:

```
TCPWM_BLOCK_CONTROL_REG |= TCPWM_MASK;
```

The following sequence of operations occurs while executing the statement above:

1. The CPU reads the control register of the TCPWM and stores it in a temporary register.
2. The CPU executes a logical OR operation of the temporary register with its mask value.
3. The CPU loads the OR result back to the control register.

Between steps 1 and 2, an interrupt may occur, and its ISR may load a new value into the same control register. After executing the ISR, when the CPU resumes executing step 2, it uses the stale control register value, which was in the temporary register– this leads to data corruption.

To avoid this issue, add the following code:

```
InterruptState = CyEnterCriticalSection();
TCPWM_BLOCK_CONTROL_REG |= TCPWM_MASK;
CyExitCriticalSection(InterruptState);
```

The `CyEnterCriticalSection and CyExitCriticalSection` functions solve the problem by disabling interrupts while the control register is being written. Use these functions when a shared variable or register is being written.

For details on these functions, see the System reference guide (also available under the PSoC™ Creator menu **Help** > **Documentation**).

4.4 Writing interrupt service routine (ISR)

To understand how to write an ISR, consider a timer interrupt as an example. The Interrupt Component “isr_1” is connected to the interrupt terminal of Timer_1, as shown in Figure 14.

After building the project, PSoC™ Creator generates the files associated with all the Components as shown in Figure 15. *isr_1.c* and *isr_1.h* are the files generated for the Interrupt Component isr_1. These files provide the API for configuring and using the Component, including the ISR.

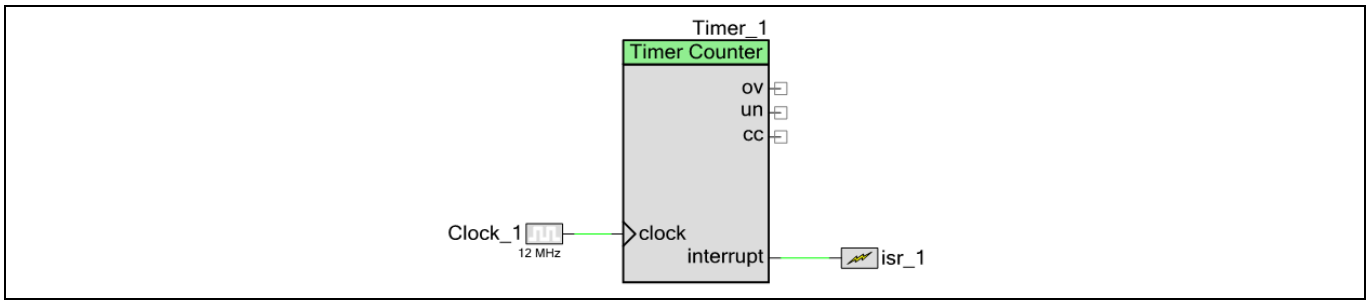

Figure 14 Timer interrupt example

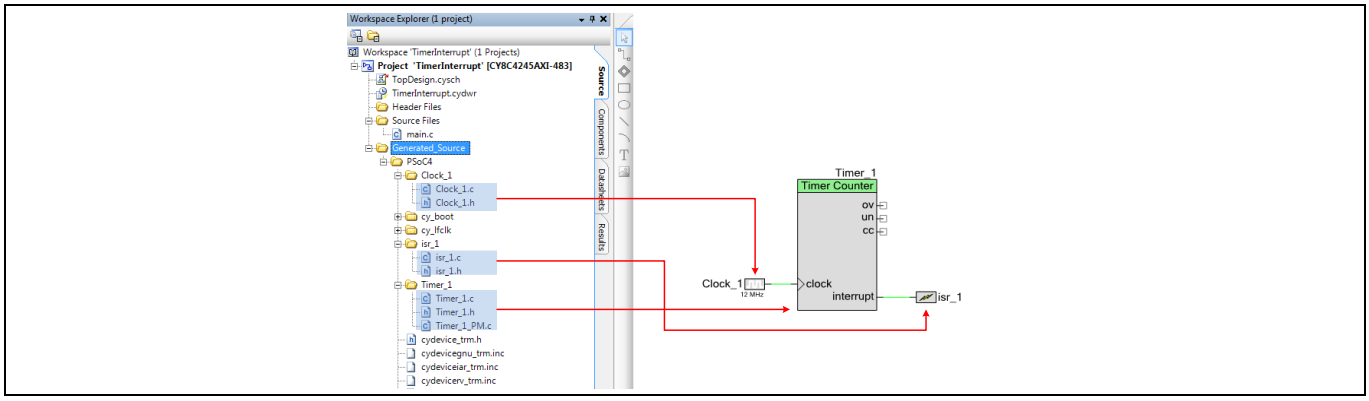

Figure 15 Files generated for interrupt components

There are two ways to write an ISR – using the PSoC™ Creator auto-generated ISR, and creating a custom ISR function.

4.5 Using auto-generated ISR

The following is an ISR generated by default in *isr_1.c*. The ISR function name is in the format - `CY_ISR(<isr_name>_interrupt).`

Code Listing 2

```
CY_ISR(isr_1_Interrupt)
{
      #ifdef isr_1_INTERRUPT_INTERRUPT_CALLBACK
            isr_1_Interrupt_InterruptCallback();
      #endif /* isr_1_INTERRUPT_INTERRUPT_CALLBACK */
      /*  Place your Interrupt code here. */
      /* `#START isr_1_Interrupt` */

      /* `#END` */
   }
```

There are two parts in this function: one to invoke a callback function and another a placeholder for the handler code. The callback function is explained in the next section. You can write the handler code in this auto-generated ISR between the #START and #END markers. Note that code written outside these markers is deleted when the project files are re-generated.

To enable the interrupt, start the isr Component. Code Listing 3 is the *main.c* code to start the interrupt source, that is, the timer and the Interrupt Component.

Code Listing 3

```
int main()
{
      /* Start the timer component */
      Timer_1_Start();

      /* Start the interrupt component */
      isr_1_Start();

      /* Enable global interrupt */
      CyGlobalIntEnable;

      for(;;)
      {
            /* Place your application code here. */
      }
}
```

Note that in addition to enabling the Interrupt Component, you must enable the global interrupt using the `CyGlobalIntEnable` macro. Inside the ISR, clear the interrupts as explained in Sticky bits. In this example, the Timer interrupt is cleared using the following API function:

```
void Timer_1_ClearInterrupt(uint32 interruptMask)
```

The `interruptMask` parameter can be the Timer Component's terminal count interrupt mask or compare/capture count interrupt mask – see the Timer Component datasheet or the *timer_1.h* file. See other Component datasheets to learn about the API and the interrupt mask that clears the interrupt from a particular component.

4.5.1 Using extern keyword

Many times, in the auto-generated ISR, it is required to access variables and call functions defined in user source files. But to use the variables and the function calls in the auto-generated ISR, it needs to be declared in the isr_1 file. An "extern" keyword is used for variable declaration.

Look out for Code Listing 4 in the beginning of the file.

Code Listing 4

```
/*******************************************************************************
*  Place your includes, defines and code here
********************************************************************************/
/* `#START isr_1_intc` */
```

Code Listing 4

```
/* `#END` */
```

You can either declare the variables and functions between the #START and #END markers directly or just include the header file containing the declarations. An example is shown with a variable and a function declaration.

Code Listing 5

```
/***************************************************************************
*  Place your includes, defines and code here
***************************************************************************/
/* `#START isr_1_intc` */

extern uint8 userVariable;
void userFunction(void);
/* `#END` */
```

4.6 Using the callback function

Instead of writing the handler code in the auto-generated ISR, you can invoke your own function from the ISR. This helps to keep a separation between the user code and generated code.

The auto-generated ISR has a code with conditional compilation, controlled by macros, for invoking the callback function.

Code Listing 6

```
CY_ISR(isr_1_Interrupt)
{
    #ifdef isr_1_INTERRUPT_INTERRUPT_CALLBACK
        isr_1_Interrupt_InterruptCallback();
    #endif /* isr_1_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START isr_1_Interrupt` */

    /* `#END` */
}
```

PSoC™ Creator interrupt support

By default, the `isr_1_INTERRUPT_INTERRUPT_CALLBACK` macro is not defined, thereby disabling the call to `isr_1_Interrupt_InterruptCallback()`. This is the callback function that you need to write in the source file.

Enable the callback function. To do this, define `isr_1_INTERRUPT_INTERRUPT_CALLBACK` in the *cyapicallbacks.h* file, which is located under "Header Files" of the project. Also, declare the callback function in the same file as Code Listing 7.

Code Listing 7

```
#ifndef CYAPICALLBACKS_H
#define CYAPICALLBACKS_H



    /*Define your macro callbacks here */
    /*For more information, refer to the Writing Code topic in PSoC™Creator Help.*/

    #define isr_1_INTERRUPT_INTERRUPT_CALLBACK
    void isr_1_Interrupt_InterruptCallback(void);


   #endif /* CYAPICALLBACKS_H */
```

Notice that the function call is enabled in the auto-generated ISR.

Write the callback function in your source file like any other function.

4.7 Creating a custom ISR

The ISR can also be written completely in your own source file instead of modifying the auto-generated code. This method has a benefit of saving time in the function call which occurs in the case of callback function. To make your own function, for example `MyCustomISR`, to be the ISR for an Interrupt Component *isr_1*, do the following:

1. Declare the custom function using the `CY_ISR_PROTO` macro:

   ```
   CY_ISR_PROTO(MyCustomISR);
   ```

2. Define the custom function using the `CY_ISR` macro:

   ```
   CY_ISR(MyCustomISR)
   {
        /* ISR code goes here */
   }
   ```

3. In the startup code of your *main.c* file, add a call to the `isr_1_StartEx()` API function instead of `isr_1_Start()`. The `isr_1_StartEx()` API function is similar to `isr_1_Start()` except that `isr_1_StartEx()` has a parameter for your ISR function: `isr_1_StartEx(MyCustomISR);`

Code Listing 8

```
CY_ISR_PROTO(MyCustomISR);
/*******************************************************************
* Function Name: MyCustomISR
********************************************************************/
CY_ISR(MyCustomISR)
{
      /* Add code here */
}

int main()
{
      /* Start the timer component */
      Timer_1_Start();

      /* Set the custom ISR */
      isr_1_StartEx(MyCustomISR);

      /* Enable global interrupt */
      CyGlobalIntEnable;

      for(;;)
      {
          /* Place your application code here. */
      }
   }
```

4.7.1 Significance of the keyword CY_ISR

The interrupt source file defines the ISR function using the `CY_ISR` macro. This macro is defined in the auto-generated *cytypes.h* file. It is used for compatibility and easy code porting to other PSoC™ device families such as PSoC™ 3 or PSoC™ 5LP.

Similarly, the macro `CY_ISR_PROTO` declares an ISR function prototype. The declaration is in the header file of the Interrupt Component. For example, the *isr_1* Interrupt Component has the following function prototype declaration in the header file *isr_1.h*:

```
CY_ISR_PROTO(isr_1_Interrupt);
```

5 ModusToolbox™ related code examples

Table 3 provides the list of code examples that use the interrupt feature. For more code examples, visit Github.

Note: PSoC™ 4 PDL is currently in Alpha and many features are in progress.

Table 3 ModusToolbox™ interrupt code examples

Code Example	Interrupt Source
CE230654 - PSoC™ 4: GPIO interrupt	GPIO
CE230664 – PSoC™ 4: Periodic interrupt using TCPWM	TCPWM
CE230603 - PSoC™ 4: I2C slave using callbacks	I2C
CE231429 - PSoC™ 4: SCB UART transmit and receive with DMAv	UART
CE230653 - PSoC™ 4: Watchdog timer interrupt and reset	WDT
CE231432 - PSoC™ 4: Watchdog counter interrupts	WDC

All ModusToolbox™ Code Examples

To see all currently released code examples select **New Application** in the ModusToolbox™ **Quick Panel.** A list of board support packages (BSP) are available to choose from, see Figure 16. The BSP corresponds to the specific device that is being used. When a BSP is selected, a list of code examples that correspond with that BSP can be chosen, as shown in Figure 17.

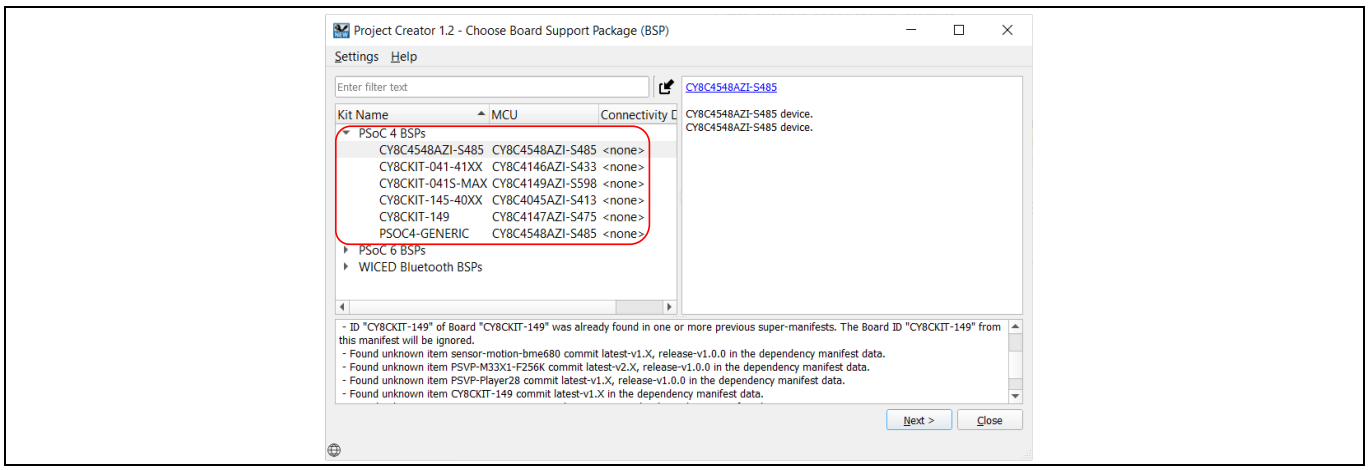

Figure 16 BSP selection wizard

ModusToolbox™ related code examples

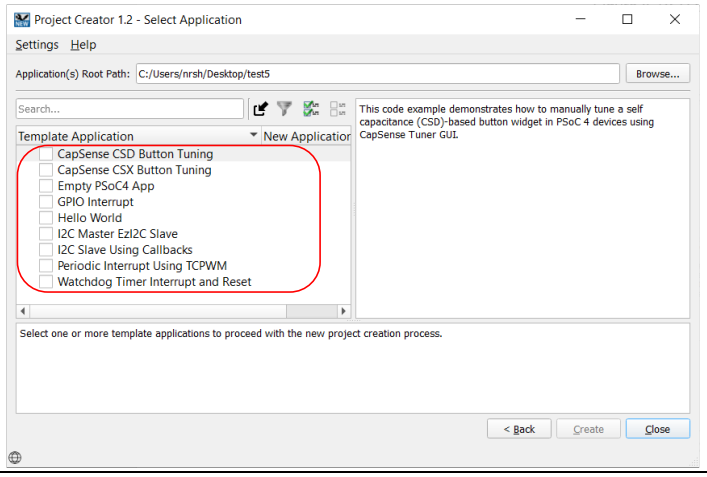

Figure 17 Code example selection

6 PSoC™ Creator related code examples

Table 4 provides the list of code examples that use the interrupt feature.

Table 4 Interrupt code examples

Code Example	Interrupt Source
CE210558 – PSoC™ 4 GPIO interrupt	GPIO
CE95915 – Implementing an RTC with PSoC™ 4100/PSoC™ 4200 Devices	TCPWM
CE95333 – Low Power Comparator with PSoC™ 4	LPCOMP
CE95321 – Hibernate and Stop Power Modes with PSoC™ 4	LPCOMP, GPIO
CE95400 – Watchdog Timer Reset and Interrupt for PSoC™ 41xx/42xx Devices	WDT
CE95275 – Sequencing SAR ADC and Die temperature sensor with PSoC™ 4	SAR ADC
CE97089 – PSoC™ 4 ADC to Memory Buffer DMA Transfer	DMA
CE210741 – UART Full Duplex and printf Support with PSoC™	UART

All PSoC™ Creator code examples

To see all currently released code examples select **Find Code Example** in the **Start Page,** see Figure 18. This opens a code example selection menu that can be filtered by device or by Component.

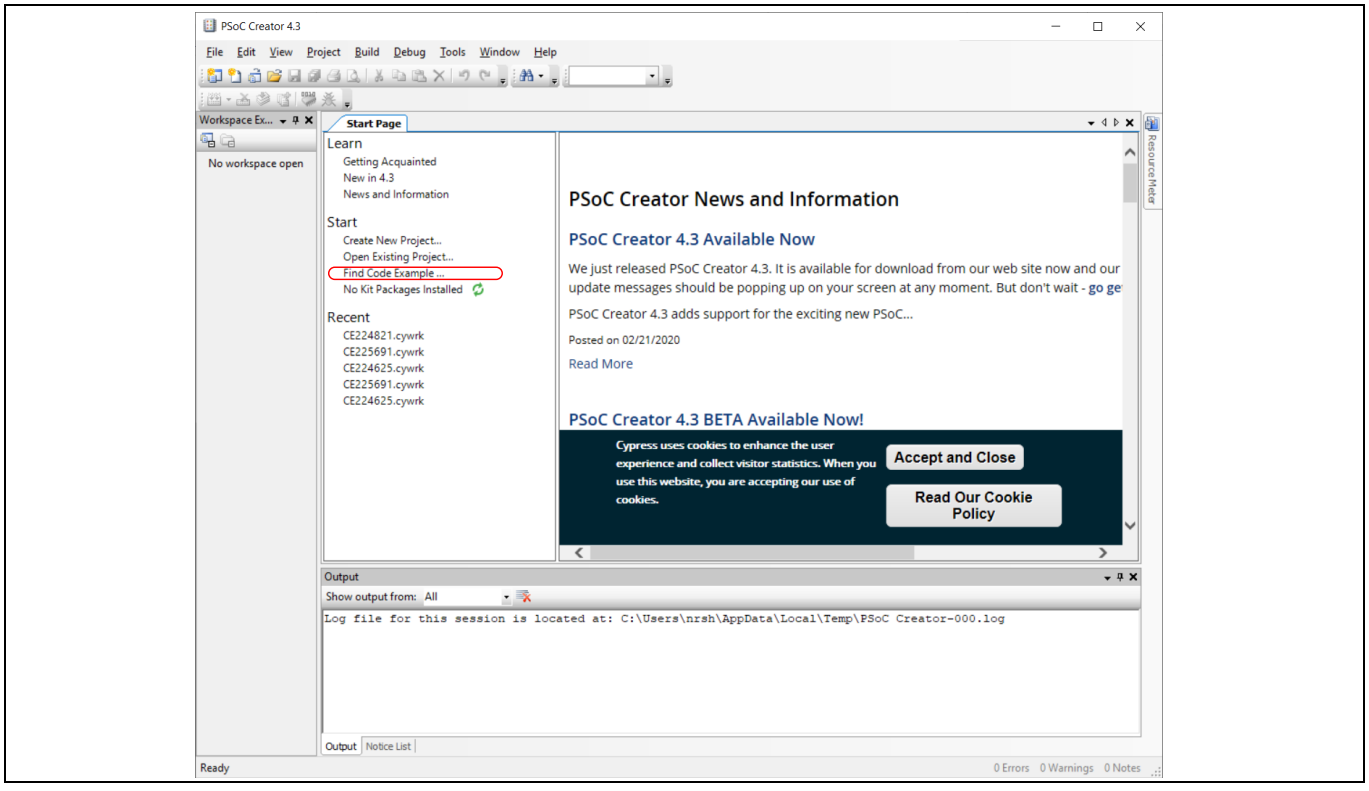

Figure 18 Code example Quick Start

7 Debugging tips

This section provides tips on debugging interrupt projects. The following are some of the frequently encountered cases:

1. **Interrupt does not get triggered**
   - Ensure that the interrupt source and global interrupt are enabled.
   - Check whether the vector is set to the correct ISR. See Writing interrupt service routine (ISR) or Setting up an interrupt for more details on how to write and assign the handler for an interrupt source.
   - Check whether there are other interrupt sources that are getting repeatedly triggered, thus consuming the entire CPU bandwidth.
   - Check whether the interrupt is getting triggered only once. This happens if the Interrupt Component is configured to rising edge and the interrupt source is not cleared.
2. **Interrupt is triggered repeatedly**

   This can happen in multiple cases:
   - The interrupt line from the source Component is connected to the Interrupt Component configured to level type. Clear the interrupt source to resolve this behavior.
   - A digital output from the Component (not the interrupt line) is connected to the Interrupt Component configured to level type. Configure the Interrupt Component to rising edge to get one interrupt per rising edge.

   See Sticky bits for more details.
3. **Interrupt is triggered only once**

   The interrupt line from the source Component is connected to the Interrupt Component configured to rising edge type. Clear the interrupt source to allow the interrupt to be triggered for every rising edge.

   See Sticky bits for more details.
4. **Execution of the interrupt service routine (ISR) is taking longer time than expected**

   This can happen if other high-priority interrupts are being triggered during the execution of the ISR. Increase the priority of the interrupt relative to other interrupt sources.

PSoC™ 4 devices have an on-chip debug capability that uses the serial wire debug (SWD) interface. It allows you to add breakpoints, evaluate and edit variables, view CPU registers, observe assembler instructions, and read and write memory. The debug mode is useful for checking interrupts as given below:

- To check whether the interrupts are getting executed, add a breakpoint in one of the instructions of the ISR.
- Use the Call Stack window of the debugger to locate when an interrupt is getting executed. You can also use it to check whether a high-priority interrupt occurred during the execution of a low-priority ISR.
- Use Breakpoint Hit Count to detect the number of times an interrupt is being triggered. This is particularly useful to check if the interrupt signal has glitches causing the interrupt to trigger multiple times.

For more details on how to use the Debugger, see the “Using the Debugger” section in PSoC™ Creator help. To access the document, press **F1** or use the **Help** > **Topics** menu in PSoC™ Creator.

As an alternative to the debugger, you can also bit bang a pin to do the following:

- Check whether the CPU is entering the ISR.
- Measure the ISR execution time. This can be done by setting the pin in the beginning of the ISR and resetting the pin at the end.

8 Advanced interrupt topics

8.1 Exceptions

Exceptions are events that cause the processor to suspend the currently executing code and branch to a handler. Interrupts are a subset of exceptions. Besides interrupts, exceptions exist for operating system applications and fault handling, as shown in Table 5.

Table 5 Exceptions in Arm® Cortex® M0

Exception	Exception number	Interrupt priority	Description
Reset	1	–3 (Highest)	Triggered on power-on-reset or external reset.
Hard fault	3	–1	Generated on fault conditions such as the detection of undefined opcode.
SVCall (Supervisor call)	11	Programmable	Triggered on a supervisory call (execution of the SVC instruction). It is normally used in operating system applications.
PendSV (pendable service call)	14	Programmable	Similar to SVCall, but the branching to the handler is done only after all high-priority tasks are completed.
SysTick	15	Programmable	SysTick is a 24-bit down-counting timer present in Cortex® M0/M0+. It generates periodic interrupts for use in operating system applications.
IRQ0 to IRQ31	16–47	Programmable	External (Pins) or internal peripheral interrupts.

8.1.1 ModusToolbox™ exceptions

Note that the exception numbers are defined by Arm®. In ModusToolbox™ software environment, interrupt vector numbers are shown in the device header file (example: *cy8c4024axi_s402.h*). For example, interrupt vector 0 is exception number 16 (IRQ0).

Reset is the highest-priority exception in the device followed by Hard Fault. These have a fixed priority, whereas others have programmable priorities. ModusToolbox™ provides a default handler for all exceptions. For reset, the default handler is `Reset_Handler()` in the *startup_psocXXXX.c* file. This function is executed first on startup. For all other exceptions, the `Defualt_Handler()` function is the default handler provided in the *startup_psocXXXX.c* file. However, vector addresses of exceptions that are used including interrupts (defined by the PSoC™ Creator Components or by the user) are loaded into the vector table during program execution. Unused exceptions still use the default handler.

8.1.2 PSoC™ Creator exceptions

Note that the exception numbers are defined by Arm®. In PSoC™ Creator interrupt vector numbers are shown in the interrupt tab of the *.cydwr* window in the PSoC™ Creator project, include an exception offset. For example, interrupt vector 0 is exception number 16 (IRQ0).

Reset is the highest-priority exception in the device followed by Hard Fault. These have a fixed priority, whereas others have programmable priorities. PSoC™ Creator provides a default handler for all exceptions. For reset, the default handler is `Reset()` in the *Cm0Start.c* file. This function is executed first on startup. For all other exceptions, the `IntDefaultHandler()` function is the default handler provided in the *Cm0Start.c* file. However, vector addresses of exceptions that are used including interrupts (defined by the PSoC™ Creator

Components or by the user) are loaded into the vector table during program execution. Unused exceptions still use the default handler.

To identify the exception currently being handled, read the Interrupt Program Status Register (IPSR). This is particularly useful when the default handler is under execution.

For more details on exceptions, see ARM® developer.

8.2 Interrupt latency

Interrupt latency is defined as the time delay between the assertion of an interrupt and the execution of the first instruction in its ISR. The Arm® Cortex®-M0 or Arm® Cortex®-M0+ processor in PSoC™ 4 devices has a latency of 16 and 15 CPU clock cycles (worst-case) respectively with additional CPU cycles because of the synchronizer between peripherals and Cortex®-M0/ Cortex® M0+ interrupt lines. Table 6 provides the synchronizer CPU clock cycle delays in different PSoC™ 4 families for DSI and fixed-function source interrupts.

Table 6 Synchronizer clock cycle delays for DSI and fixed-function source interrupts

Device	DSI Interrupt	Fixed-function interrupt
PSoC™ 4000	NA	Depends on the peripheral: SCB-I2C, GPIO, WDT: 3 CPU cycles SPC, CSD, TCPWM: 0 CPU cycles
PSoC™ 4200 / PSoC™ 4100	0 CPU cycles	3 CPU Cycles
PSoC™ 42x7 BLE / PSoC™ 41x7 BLE	3 CPU cycles	Depends on the peripheral: SCB-I2C, GPIO, WDT, CTBm, LPCOMP, BLE, LVD: 3 CPU cycles SPC, CSD, TCPWM, SAR: 0 CPU cycles
PSoC™ 4200M / PSoC™ 4100M / PSoC™ 4200L	3 CPU cycles	Depends on the peripheral: SCB-I2C, GPIO, WDT, CTBm, LPCOMP, LVD: 3 CPU cycles SPC, CSD, TCPWM, SAR, DMA, CAN, USB (only available in PSoC™ 4200L): 0 CPU cycles
PSoC™ 4000S / PSoC™ 4100S / PSoC™ 4100PS	NA	Depends on the peripheral: SCB, GPIO, WDT, CTBm/CTB, LPCOMP: 2 CPU cycles CSD, TCPWM, SAR: 0 CPU cycles
PSoC™ 4100S Plus	NA	Depends on the peripheral: SCB, GPIO, WDT, CTBm/CTB, LPCOMP: 2 CPU cycles CSD, Crypto, CAN, SAR: 0 CPU cycles

During the 16-cycles latency in Cortex® M0 or 15-cycles latency in Cortex® M0+, the following actions take place:

1. The processor pushes the current Program Counter (PC), Link Register (LR), Program Status Register (PSR), and some of the general-purpose registers to the stack.
2. The processor reads the vector address from the NVIC and updates it to the PC.
3. The processor updates the NVIC registers.

Thus, the latency differs from 16 cycles in Cortex® M0 and 15 cycles in Cortex® M0+ when an ISR is currently in execution or about to begin. To make the process efficient, the Cortex®-M0/ Cortex® M0+ processor implements the following two schemes:

1. *Tail Chaining:* If an interrupt is in the pending state while the processor is executing another interrupt handler, unstacking is skipped when the execution ends for the first interrupt and the handler for the pending interrupt is immediately executed. This saves the time of restoring the registers from the stack and pushing the same registers again to stack. This is useful for reducing the latency of low-priority interrupts.
2. *Late Arrival:* If a higher-priority interrupt occurs during the stacking process of a lower-priority interrupt, the processor jumps to the higher-priority interrupt handler instead of a lower-priority one. The processor reads the vector address of the higher-priority interrupt at the end of the stacking process. Once the higher-priority interrupt handler execution is completed, the vector address for the pending lower-priority interrupt handler is fetched and executed. This reduces the latency for a higher-priority interrupt by eliminating the delay caused by entering the lower-priority ISR and pushing the register values to the stack.

Note that the current instruction in execution when the interrupt is triggered causes an additional delay in the execution of the ISR. In the case of a device wakeup from an interrupt, an additional delay is caused by the voltage stabilization after the power-up sequence. See the device datasheet for specifications.

8.3 Optimizing the interrupt code

One of the important performance requirements in interrupt-based applications is the ISR code execution time. In some applications, the critical code in the ISR must be executed within a particular time of receiving the interrupt request. Also, interrupt execution should not take too much time and stall the main code execution or other interrupts. To meet these requirements, use the following guidelines:

- **Avoid calls to lengthy functions in the ISR.** Functions such as Character LCD display routines take a long time to execute, and thus block the execution of other low-priority interrupts.

  The recommended technique is to move noncritical function calls to the main code and just set a flag variable in the ISR. The main code periodically checks the flag and if set, clears it and calls the function.
- **Assign proper priority to interrupts.** In applications with multiple interrupts, give a higher priority to more time-critical interrupts.

8.4 PSoC™ Creator components internal interrupts

Many PSoC™ Creator Components have an Interrupt Component internally as part of their implementation. Examples include CAPSENSE™, SAR ADC, EZI2C, and Segment LCD.

Similar to Interrupt Components, internal ISRs in these Components provide a placeholder region for writing user code. See the respective Component datasheets and associated code examples provided in PSoC™ Creator to understand the interrupt usage in these Components. Interrupt usage can also be seen in the *cydwr* window as shown in Figure 19.

8.5 PSoC™ Creator forcing interrupt vector num

PSoC™ Creator automatically assigns the vector numbers for Interrupt Components in a project. After building the project, you can view the assigned vector numbers in the Interrupts tab of the *cydwr* window, as shown in Figure 19. You can also select a particular vector number for an interrupt signal when it is routed through the DSI. This section provides a step-by-step procedure to do this.

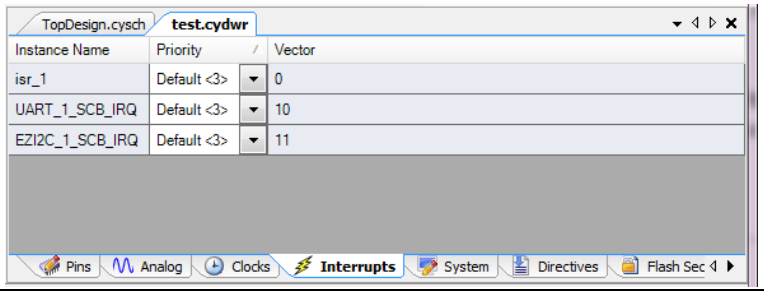

Figure 19 Interrupt vector numbers in *cydwr* Window

To override the vector numbers assigned by PSoC™ Creator and manually assign a vector number, a *Control File* is used. Follow the steps given below:

1. Click the **Components** tab of the Workspace Explorer window.
2. Right-click the **TopDesign** Component and select **Add Component Item...**. The Add Component Item dialog opens.
3. Scroll down to the **Misc** group, select **Control File**, and click **Create New**, as shown in Figure 20.

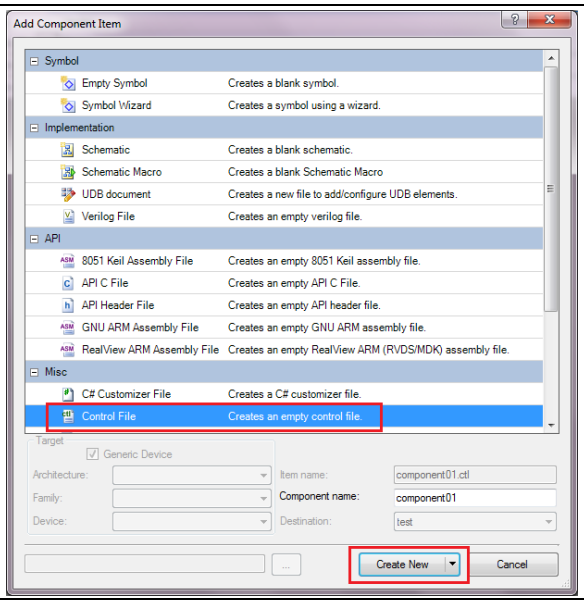

Figure 20 Adding the Control File

A *TopDesign.ctl* file is created and added to the Workspace Explorer window.

4. Double-click the *TopDesign.ctl* file to open it for editing. The *attribute* keyword is used in the control file to specify the interrupt vector number for each Interrupt Component. The method of specifying the interrupt vector number depends on whether you have placed the Interrupt Component on the example schematic or the Interrupt Component is used internally in a PSoC™ Creator Component in the schematic. The two methods are as follows:

   a) For Interrupt Components that you have placed on the schematic, the syntax is:

```
attribute placement_force of instance_name : label is "Intr(0,
DesiredVectorNumber)";
```

Here, `instance_name` refers to the name of the Interrupt Component in the schematic and `DesiredVectorNumber` is the vector number (0 to 31). For example, to assign vector 17 to the Interrupt Component *isr_1*:

```
attribute placement_force of isr_1 : label is "Intr(0, 17)";
```

b) For Components that use interrupts internally such as EZI2C, the syntax is:

```
attribute placement_force of \top_instance_name : InternalInterruptName\: label
is "Intr(0, DesiredVectorNumber)";
```

Here, `top_instance_name` refers to the name of the Component that uses the interrupt internally. `InternalInterruptName` refers to the name assigned for the internal interrupt in the Component. This can be found from the Interrupts tab of the *cydwr* window, where the interrupt name is appended to the top Component instance name. In Figure 19, *SCB_IRQ* is the internal interrupt name for the EZI2C Component and the UART Component. The following statement assigns the vector for EZI2C Component to 11.

```
attribute placement_force of \EZI2C_1:SCB_IRQ\ : label is "Intr(0,11)"
```

5. After assigning the interrupt vector numbers, click **Save** to save the changes made to the control file.
6. **Clean and Build** the example for the new interrupt vector assignments to take effect. The **Interrupts** tab in the *cydwr* window now shows the modified interrupt vector number assignments.

8.6 ModusToolbox™ SysTick timer

SysTick is a 24-bit down-counting timer. Its interrupt is generally used for task switching in a real-time system. It uses the Cortex®-M0/ Cortex® M0+ internal clock for counting. SysTick is configured using the APIs given below:

1. Setting interrupt handler

```
Cy_SysInt_SetVector(SysTick_IRQn, SysTick_ISR);
```

`SysTick_IRQN` is the exception number for the SysTick interrupt, which is 15 for Cortex®–M0.
`SysTick_ISR` is the interrupt handler.

2. Configuring interrupt period

```
Cy_SysTick_Init(CY_SYSTICK_CLOCK_SOURCE_CLK_CPU,
CLOCK_FREQ/INTERRUPT_FREQ);
```

CLOCK_FREQ is the CPU clock frequency. INTERRUPT_FREQ is the derived interrupt rate from SysTick.

Code Listing 9 is the code snippet for SysTick timer usage.

Code Listing 9

```
/* clock and interrupt rates, in Hz */
#define CLOCK_FREQ     24000000u
#define INTERRUPT_FREQ 2u

void SysTick_ISR(void)
{
    /* Interrupt Handler */
}

int main()
{
    /* Point the Systick vector to the ISR */
    CyIntSetSysVector(SysTick_IRQn, SysTick_ISR);

      /* Set the number of ticks between interrupts */
    Cy_SysTick_Init(CY_SYSTICK_CLOCK_SOURCE_CLK_CPU, CLOCK_FREQ/INTERRUPT_FREQ);

    /* Enable Global Interrupts */
    __enable_irq();

    for(;;)
    {
    }
}
```

8.7 PSoC™ Creator SysTick timer

SysTick is a 24-bit down-counting timer. Its interrupt is generally used for task switching in a real-time system. It uses the Cortex®-M0/Cortex® M0+ internal clock for counting. SysTick is configured using the APIs given below:

1. Setting interrupt handler

   ```
   CyIntSetSysVector(SYSTICK_VECTOR_NUMBER, SysTick_ISR);
   ```

   `SYSTICK_VECTOR_NUMBER` is the exception number for the SysTick interrupt, which is 15 for Cortex®–M0. `SysTick_ISR` is the interrupt handler.

2. Configuring interrupt period

   ```
   (void)SysTick_Config(CLOCK_FREQ / INTERRUPT_FREQ);
   ```

   CLOCK_FREQ is the CPU clock frequency. INTERRUPT_FREQ is the derived interrupt rate from SysTick.

The following is the code snippet for SysTick timer usage.

Code Listing 10

```
#define SYSTICK_INTERRUPT_VECTOR_NUMBER 15u /* Cortex®-M0/M0+ hard vector */

/* clock and interrupt rates, in Hz */
#define CLOCK_FREQ     24000000u
#define INTERRUPT_FREQ 2u

CY_ISR(SysTick_ISR)
{
    /* Interrupt Handler */
}

int main()
{
    /* Point the Systick vector to the ISR */
    CyIntSetSysVector(SYSTICK_INTERRUPT_VECTOR_NUMBER, SysTick_ISR);

      /* Set the number of ticks between interrupts */
    (void)SysTick_Config(CLOCK_FREQ / INTERRUPT_FREQ);

      /* Enable Global Interrupt */
    CyGlobalIntEnable;

    for(;;)
    {
    }
}
```

8.8 Nested interrupts

NVIC automatically handles nested interrupts without any software overhead. If a higher-priority interrupt is asserted during the execution of a lower-priority interrupt handler, some of the general-purpose registers are pushed to stack, the processor core reads the vector address from NVIC and jumps to the higher-priority interrupt handler. After the execution is completed, the processor restores the register values and execution resumes for the lower-priority interrupt.

8.9 PSoC™ Creator GlobalSignal component

The GlobalSignal Component helps access interrupt signals from various resources in the system. Some of the interrupt signals accessed are watchdog timer interrupt, combined port interrupt, combined low-power comparator interrupt, System Performance Controller Interface (SPCIF) timer (used in the case of flash write operations), and so on. For details on available interrupt signals, see the Component datasheet. The combined port interrupt is explained below.

8.9.1 Combined port interrupt

In most PSoC™ 4 devices, not all ports have a dedicated interrupt vector (see Table 7). In such a case, it is recommended to use the combined port interrupt feature in the GlobalSignal component. The combined port interrupt uses OR logic to combine port interrupt signals.

For example, if you want to generate an interrupt from a signal on pin P5[0], which doesn't have a dedicated interrupt, in the PSoC™ 4100PS device, do the following:

1. Place a digital input pin component and set it to P5[0].
2. Configure the pin interrupt. Ensure that dedicated interrupt is unchecked, see Figure 21.

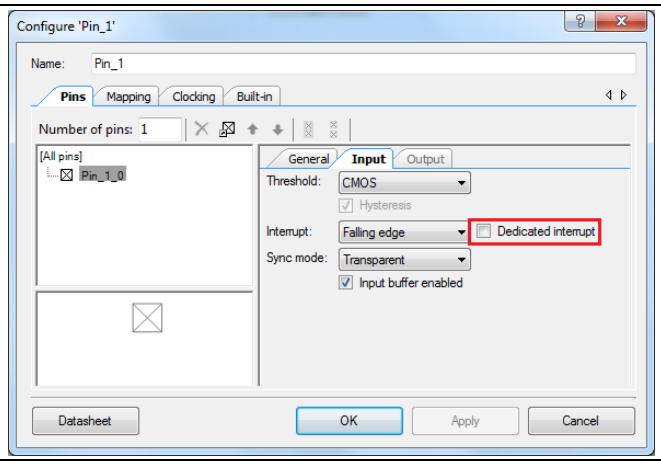

Figure 21 Pin properties for combined interrupt

3. Place a GlobalSignal Component and set it to "Combined port interrupt (AllPortInt)", see Figure 22.

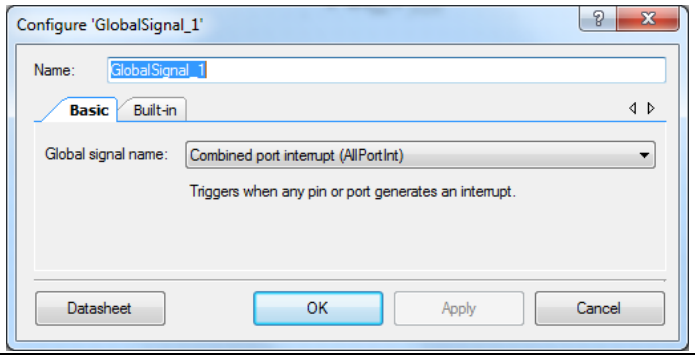

Figure 22 Placing a GlobalSignal component

Connect an interrupt Component to the GlobalSignal Component, see Figure 23.

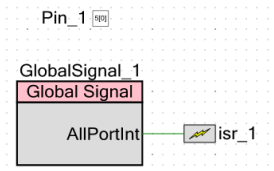

Figure 23 Connecting an interrupt component to GlobalSignal component

The ISR can now be written for the isr_1 Component. Note that if multiple port pins are enabled with interrupt, you should check the GPIO_PRTx_INTR register to identify the port pin that triggered the interrupt. See the *Register Reference Manual* of the device for details on the register.

8.10 Use of volatile for global variables

In interrupts, a common case is to use a variable as a flag that is set in the interrupt handler and polled in the main loop. In such cases, the compiler may optimize out the variable by assuming that there is no code present in the program flow to update the flag; this results in error during run time. To avoid this problem, always declare the global variables that are accessed in both the ISR and the main loop as volatile.

9 Summary

Interrupts are commonly used in embedded applications. For system-on-chip architectures such as PSoC™ 4, interrupts play the critical role of communicating the status of on-chip peripherals to the CPU. This application note has provided the information needed to understand the infrastructure available and create interrupt-based projects.

Appendix A - Interrupt sources and vector numbers

Table 7 lists the interrupt sources for the 32 interrupt vectors in PSoC™ 4.

Table 7 PSoC™ 4 interrupt sources ('–' Indicates function not available)

Fixed function interrupt source	DSI interrupt source (not for PSoC™ 4000/4000S/ 4100S/4100S Plus/ PSoC™ 4100PS)	Interrupt vector													
		PSoC™ 4000	PSoC™ 4100/ 4200	PSoC™ 4 BLE	PSoC™ 4 M	PSoC™ 4 L	PSoC™ 4000S	PSoC™ 4100S	PSoC™ 4100S Plus	PSoC™ 4100PS	PSoC™ 4100S Plus 256k	PSoC™ 4100S Max	PSoC™ 4200DS	PSoC™ 4500S	PSoC™ 4700S
GPIO Interrupt – Port 0	DSI	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0
GPIO Interrupt – Port 1	DSI	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1
GPIO Interrupt – Port 2	DSI	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2
GPIO Interrupt – Port 3	DSI	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3
GPIO Interrupt – Port 4	DSI	–	IRQ4	IRQ4	IRQ4	IRQ4	–	–	–	–	–	–	–	–	–
GPIO Interrupt – Port 5	DSI	–	–	IRQ5	–	–	–	–	–	–	–	–	–	–	–
GPIO Interrupt – Port 13 (USB Wake up)	DSI	–	–	–	–	IRQ5	–	–	–	–	–	–	–	–	–
GPIO Interrupt – All Port*	DSI	–	–	IRQ6	IRQ5	IRQ6	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4
–	DSI	–	IRQ5	–	–	–	–	–	–	–	–	–	–	–	–
–	DSI	–	IRQ6	–	–	–	–	–	–	–	–	–	–	–	–
–	DSI	–	IRQ7	–	–	–	–	–	–	–	–	–	–	–	–
LPCOMP (low-power comparator)	DSI	–	IRQ8	IRQ7	IRQ6	IRQ7	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5
WDT (Watchdog timer)	DSI	IRQ4	IRQ9	IRQ8	IRQ7	IRQ8	IRQ6	IRQ6	IRQ6	IRQ7	IRQ6	IRQ6	IRQ6	IRQ6	IRQ6
SCB0 (Serial Communication Block 0)	DSI	IRQ5	IRQ10	IRQ9	IRQ8	IRQ9	IRQ7	IRQ7	IRQ7	IRQ8	IRQ7	IRQ7	IRQ7	IRQ7	IRQ7

Fixed function interrupt source	DSI interrupt source (not for PSoC™ 4000/4000S/ 4100S/4100S Plus/ PSoC™ 4100PS)	Interrupt vector													
		PSoC™ 4000	PSoC™ 4100/ 4200	PSoC™ 4 BLE	PSoC™ 4 M	PSoC™ 4 L	PSoC™ 4000S	PSoC™ 4100S	PSoC™ 4100S Plus	PSoC™ 4100PS	PSoC™ 4100S Plus 256k	PSoC™ 4100S Max	PSoC™ 4200DS	PSoC™ 4500S	PSoC™ 4700S
SCB1 (Serial Communication Block 1)	DSI	–	IRQ11	IRQ10	IRQ9	IRQ10	IRQ8	IRQ8	IRQ8	IRQ9	IRQ8	IRQ8	IRQ8	IRQ8	IRQ8
SCB2 (Serial Communication Block 2)	DSI	–	–	–	IRQ10	IRQ11	–	IRQ9	IRQ9	IRQ10	IRQ9	IRQ9	IRQ9	IRQ9	–
SCB3 (Serial Communication Block 3)	DSI	–	–	–	IRQ11	IRQ12	–	–	IRQ10	–	IRQ10	IRQ10	–	IRQ10	–
SCB3 (Serial Communication Block 4)	DSI	–	–	–	–	–	–	–	IRQ11	–	IRQ11	IRQ11	–	IRQ11	–
CTBm0 Interrupt (all CTBm0s)	DSI	–	–	IRQ11	IRQ12	IRQ13	–	IRQ10	IRQ12	IRQ6	IRQ12	IRQ12	–	IRQ12	–
CTBm1 Interrupt (all CTBm1s)	DSI	–	–	–	–	–	–	–	–	–	IRQ13	–	–	IRQ13	–
Bluetooth LE Subsystem Interrupt	DSI	–	–	IRQ12	–	–	–	–	–	–	–	–	–	–	–
DMA Interrupt	DSI	–	–	–	IRQ13	IRQ14	–	–	IRQ14	IRQ12	IRQ15	IRQ14	IRQ10	IRQ15	–
SPCIF Interrupt	DSI	IRQ6	IRQ12	IRQ13	IRQ14	IRQ15	IRQ9	IRQ10	IRQ15	IRQ13	IRQ16	IRQ15	IRQ11	IRQ16	IRQ9
SRSS LVD Interrupt	DSI	–	IRQ13	IRQ14	IRQ15	IRQ16	–	–	–	–	–	–	–	–	–
SAR 0 (Successive Approximation ADC)	DSI	–	IRQ14	IRQ15	IRQ16	IRQ17	–	IRQ19	IRQ25	IRQ15	IRQ26	IRQ25	–	IRQ26	–
SAR 1 (Successive Approximation ADC)	DSI	–	–	–	–	–	–	–	–	–	IRQ27	–	–	IRQ27	–
CSD0 (CAPSENSE™)	DSI	IRQ7	IRQ15	IRQ16	IRQ17	IRQ18	IRQ10	IRQ13	IRQ16	IRQ14	IRQ17	IRQ16	–	IRQ17	IRQ10
CSD1 (CAPSENSE™)	DSI	–	–	–	IRQ18	IRQ19	–	–	–	–	–	IRQ29	–	–	–

Fixed function interrupt source	DSI interrupt source (not for PSoC™ 4000/4000S/ 4100S/4100S Plus/ PSoC™ 4100PS)	Interrupt vector													
		PSoC™ 4000	PSoC™ 4100/ 4200	PSoC™ 4 BLE	PSoC™ 4 M	PSoC™ 4 L	PSoC™ 4000S	PSoC™ 4100S	PSoC™ 4100S Plus	PSoC™ 4100PS	PSoC™ 4100S Plus 256k	PSoC™ 4100S Max	PSoC™ 4200DS	PSoC™ 4500S	PSoC™ 4700S
VDAC Interrupt (Both VDACs)	–	–	–	–	–	–	–	–	–	IRQ16	–	–	–	–	–
TCPWM0 (Timer/Counter/ PWM 0)	DSI	IRQ8	IRQ16	IRQ17	IRQ19	IRQ20	IRQ11	IRQ14	IRQ17	IRQ17	IRQ18	IRQ17	IRQ12	IRQ18	IRQ11
TCPWM1 (Timer/Counter/ PWM 1)	DSI	–	IRQ17	IRQ18	IRQ20	IRQ21	IRQ12	IRQ15	IRQ18	IRQ18	IRQ19	IRQ18	IRQ13	IRQ19	IRQ12
TCPWM2 (Timer/Counter/ PWM 2)	DSI	–	IRQ18	IRQ19	IRQ21	IRQ22	IRQ13	IRQ16	IRQ19	IRQ19	IRQ20	IRQ19	IRQ14	IRQ20	IRQ13
TCPWM3 (Timer/Counter/ PWM 3)	DSI	–	IRQ19	IRQ20	IRQ22	IRQ23	IRQ14	IRQ17	IRQ20	IRQ20	IRQ21	IRQ20	IRQ15	IRQ21	IRQ14
TCPWM4 (Timer/Counter/ PWM 4)	DSI	–	–	–	IRQ23	IRQ24	IRQ15	IRQ18	IRQ21	IRQ21	IRQ22	IRQ21	–	IRQ22	IRQ15
TCPWM5 (Timer/Counter/ PWM 5)	DSI	–	–	–	IRQ24	IRQ25	–	–	IRQ22	IRQ22	IRQ23	IRQ22	–	IRQ23	–
TCPWM6 (Timer/Counter/ PWM 6)	DSI	–	–	–	IRQ25	IRQ26	–	–	IRQ23	IRQ23	IRQ24	IRQ23	–	IRQ24	–
TCPWM7 (Timer/Counter/ PWM 7)	DSI	–	–	–	IRQ26	IRQ27	–	–	IRQ24	IRQ24	IRQ25	IRQ24	–	IRQ25	–
CAN0 Interrupt	DSI	–	–	–	IRQ27	IRQ28	–	–	IRQ26	–	–	IRQ26	–	–	–
CAN1 Interrupt	DSI	–	–	–	IRQ28	IRQ29	–	–	–	–	–	IRQ27	–	–	–
USB Start of Frame	DSI	–	–	–	–	IRQ30	–	–	–	–	–	–	–	–	–
USB EP1-EP8 Data	DSI	–	–	–	–	IRQ31	–	–	–	–	–	–	–	–	–
Crypto Interrupt	–	–	–	–	–	–	–	–	IRQ27	–	–	IRQ28	–	–	–
WCO/WDT Interrupt	–	–	–	–	–	–	–	–	IRQ13	IRQ11	IRQ14	IRQ13	–	IRQ14	–
EXCO Interrupt	–	–	–	–	–	–	–	–	–	–	IRQ28	IRQ30	–	IRQ28	–
I2S	–	–	–	–	–	–	–	–	–	–	–	IRQ31	–	–	–

Fixed function interrupt source	DSI interrupt source (not for PSoC™ 4000/4000S/ 4100S/4100S Plus/ PSoC™ 4100PS)	Interrupt vector													
		PSoC™ 4000	PSoC™ 4100/ 4200	PSoC™ 4 BLE	PSoC™ 4 M	PSoC™ 4 L	PSoC™ 4000S	PSoC™ 4100S	PSoC™ 4100S Plus	PSoC™ 4100PS	PSoC™ 4100S Plus 256k	PSoC™ 4100S Max	PSoC™ 4200DS	PSoC™ 4500S	PSoC™ 4700S
–	DSI	–	IRQ20	IRQ21	IRQ29	–	–	–	–	–	IRQ29	–	IRQ16	IRQ29	–
–	DSI	–	IRQ21	IRQ22	IRQ30	–	–	–	–	–	IRQ30	–	IRQ17	IRQ30	–
–	DSI	–	IRQ22	IRQ23	IRQ31	–	–	–	–	–	IRQ31	–	IRQ18	IRQ31	–
–	DSI	–	IRQ23	IRQ24	–	–	–	–	–	–	–	–	IRQ19	–	–
–	DSI	–	IRQ24	IRQ25	–	–	–	–	–	–	–	–	IRQ20	–	–
–	DSI	–	IRQ25	IRQ26	–	–	–	–	–	–	–	–	IRQ21	–	–
–	DSI	–	IRQ26	IRQ27	–	–	–	–	–	–	–	–	IRQ22	–	–
–	DSI	–	IRQ27	IRQ28	–	–	–	–	–	–	–	–	IRQ23	–	–
–	DSI	–	IRQ28	IRQ29	–	–	–	–	–	–	–	–	IRQ24	–	–
–	DSI	–	IRQ29	IRQ30	–	–	–	–	–	–	–	–	IRQ25	–	–
–	DSI	–	IRQ30	IRQ31	–	–	–	–	–	–	–	–	IRQ26	–	–
–	DSI	–	IRQ31	–	–	–	–	–	–	–	–	–	IRQ27	–	–
–	DSI	–	–	–	–	–	–	–	–	–	–	–	IRQ28	–	–
–	DSI	–	–	–	–	–	–	–	–	–	–	–	IRQ29	–	–
–	DSI	–	–	–	–	–	–	–	–	–	–	–	IRQ30	–	–
–	DSI	–	–	–	–	–	–	–	–	–	–	–	IRQ31	–	–

References

The wealth of information available on the Infineon webpage can help you select the right PSoC™ device and, additionally, integrate the device into your designs efficiently and effectively. The following is an abbreviated list for PSoC™ 4:

- Overview: PSoC™ portfolio
- Product selectors: PSoC™ 4. In addition, PSoC™ Creator includes a device selection tool.
- Datasheets describe and provide electrical specifications for each family.
- Application notes cover a broad range of topics, from basic to advanced level, and include the following:
  - AN88619: PSoC™ 4 MCU hardware design considerations
  - AN73854: PSoC™ Creator - Introduction to bootloaders
  - AN89610: PSoC™ Arm® Cortex® code optimization
  - AN86233: PSoC™ 4 low-power modes and power reduction techniques
  - AN57821: PSoC™ 3, PSoC™ 4, and PSoC™ 5LP mixed-signal circuit board layout considerations
  - AN89056: PSoC™ 4 - IEC 60730 class B and IEC 61508 SIL safety software library
  - AN64846: Getting started with CAPSENSE™
  - AN85951: PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide
- Code examples demonstrate product features and usage.
- Reference manuals provide detailed descriptions of the architecture and registers in each PSoC™ 4 device family.
- PSoC™ 4 programming specification provides the information necessary to program PSoC™ 4 nonvolatile memory.
- Development tools:
  - CY8CKIT-040, CY8CKIT-042, CY8CKIT-044, CY8CKIT-046, CY8CKIT-042-BLE-A, CY8CKIT-045S, and CY8CKIT-041S-MAX PSoC™ 4 pioneer kits are easy-to-use and inexpensive development platforms. These include connectors for Arduino-compatible shields and Digilent Pmod daughter cards.
  - CY8CKIT-043, CY8CKIT-145-40XX, CY8CKIT-147, and CY8CKIT-149 are very low-cost prototyping platforms for sampling PSoC™ 4 devices.
  - CY8CKIT-040T is a low-cost evaluation kit showing the low power CAPSENSE™, low power wake on touch and liquid tolerant features of the PSoC™ 4000T device.
  - The MiniProg3 (CY8CKIT-002) or MiniProg4 (CY8CKIT-005) kit provides an interface for flash programming and debug.
  - Integrated development environment (IDE): There are two development platforms that can be used for application development with PSoC™ 4 – ModusToolbox™ software and PSoC™ Creator.
  - PSoC™ 4 CAD libraries provide footprint and schematic support for common tools. IBIS models are also available.
- Training videos are available in infineon website on a wide range of topics including the PSoC™ MCUs.
- Infineon community enables connection with fellow PSoC™ developers around the world, 24 hours a day, 7 days a week, and hosts a dedicated PSoC™ 4 MCU community.

Other resources:

- ModusToolbox™ software
- ModusToolbox™ software help on GitHub

Revision history

Document Revision	Date	Description of changes
**	2014-05-22	New Application note
*A	2015-05-14	Updated for PSoC™ 4 Bluetooth® LE and PSoC™ 4 M Added section on Writing interrupt handlers Added details on interrupts latency Provided links for PSoC™ Creator code examples Updated projects with PSoC™ Creator 3.2 Updated Appendix B with development kits Added information on CyEnterCriticalSection and CyExitCriticalSection APIs Updated template
*B	2016-02-02	Updated for PSoC™ 4200L. Added Exceptions and Debugging tips. Updated Introduction, PSoC™ 4 interrupt architecture and Interrupt priority configuration. Removed projects from the application note and moved to code examples CE210557 and CE210558.
*C	2017-04-19	Updated logo and copyright
*D	2017-12-13	Updated for PSoC™ 4000S, 4100S, PSoC™ 4100S Plus, and PSoC™ Analog Coprocessor. Updated Table 1, Table 3, Table 5, and Table 6.
*E	2018-09-14	Updated for PSoC™ 4100PS. Added Using extern keyword, Using the callback function, PSoC™ Creator GlobalSignal component and Use of volatile for global variables. Updated Using auto-generated ISR.
*F	2021-03-23	Updated to Infineon template Updated document to support ModusToolbox™ software environment and PDL Created new sections: 3 ModusToolbox™ interrupt support 3.1 Enabling interrupt sources 3.2 Enabling interrupt sources using PDL 3.3 Configuring interrupts using PDL 3.3.1 Interrupt API functions 3.3.2 Critical section control functions 3.3.3 Setting up an interrupt 5 ModusToolbox™ related code examples 8.1.1 ModusToolbox™ exceptions 8.6 ModusToolbox™ SysTick timer Updated Appendix A for new PSoC™ 4 devices
*G	2024-03-06	Fixed broken links. Updated References section.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.
The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc., and any use of such marks by Infineon is under license.

Edition 2024-03-06

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

001-90799 Rev. *G

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.