AN90799

PSoC® 4 Interrupts

Author: Rajiv Badiger

Associated Project: Code Examples

Associated Part Family: All PSoC 4 Parts

Software Version: PSoC Creator™ 4.2 or later

To get the latest version of this application note, please visit http://www.cypress.com/AN90799.

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, visit our code examples web page. You can also explore the PSoC 4 video library here.

AN90799 explains the interrupt architecture in PSoC 4 and its configuration in 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.

Contents

1 Introduction

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. In system-on-chip (SoC) architectures such as PSoC, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU.

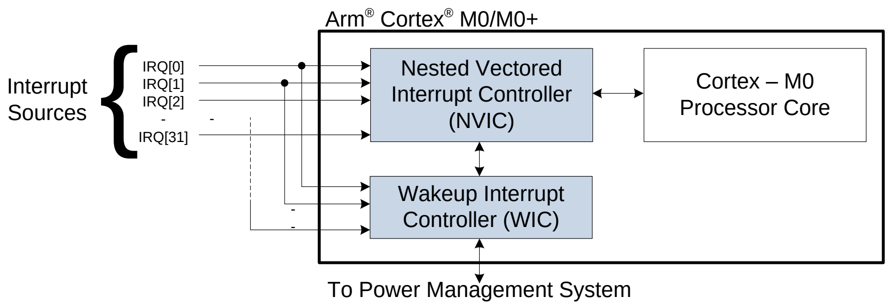
The document begins with the explanation of PSoC 4 interrupt architecture. If you want to learn about the interrupt support in the PSoC Creator IDE, skip to the Interrupt Support in PSoC Creator. For sample code examples, see Code Examples. If you are debugging the interrupt project, go to the Debugging Tips section, which provides a few tips on finding and resolving interrupt issues. The Advanced Interrupt Topics section covers advanced topics on interrupts.

This application note assumes that you are familiar with PSoC and the PSoC Creator IDE. If you are new to PSoC, you can find an introduction in the *Getting Started with PSoC* application note AN79953 and visit the 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


Arm® Cortex® M0/M0+
Interrupt Sources
IRQ[0]
IRQ[1]
IRQ[2]
IRQ[31]
Nested Vectored Interrupt Controller (NVIC)
Cortex – M0 Processor Core
Wakeup Interrupt Controller (WIC)
To Power Management System

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, NVIC sends the interrupt vector address to the processor core along with the interrupt request signal. In return, the processor core sends an acknowledgement when the ISR is entered and exited. The NVIC is responsible for enabling/disabling of any interrupt based on the user configuration. It also resolves the 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 sends the vector address to the processor core to execute the ISR. There are several sources in the PSoC 4 device that has the capability to 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.

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, thus reducing the latency.
- **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:

Fixed-function interrupt sources: These are the predefined set of interrupt sources from on-chip peripherals.

Universal Digital Block (UDB) interrupt sources (available in PSoC 4200, PSoC 42x7_BLE, PSoC 4200M and PSoC 4200L parts): UDBs are the basic building blocks for different digital functions such as Timer, PWM, UART, SPI, and many more. It 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 the routing fabric known as Digital System Interconnect (DSI). See the PSoC 4 Technical Reference Manual for more information.

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 listed in Table 1. Appendix A shows the complete list of interrupt sources depending on the device.

Table 1. PSoC 4 Interrupt Sources

Interrupt Source	Details	
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 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)	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	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	PSoC 4 has up to five Serial Communication Blocks (SCB), which can be configured as I^{2}C, SPI, or UART. The exact number of SCB blocks depends on the device family.	
	I^{2}C	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 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].
System Performance Controller (SPC)	The SPC block controls flash write operations. It triggers an interrupt when the flash write operation is complete.	
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 SysTick Timer section for more information.	
Power Manager(2)	This block generates a low-voltage detect (LVD) interrupt when the device supply voltage drops below a threshold.	
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)	CSD, used for touch applications, generates an interrupt when the sensor scan is complete.	
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.	
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 Technical Reference Manual for more information.	
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.	
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 counterparts. UDBs are available in PSoC 4200, PSoC 42xx_BLE, PSoC 4200M, and PSoC 4200L.	
USB	PSoC 4200L has USB with start-of-frame interrupt and interrupt on completion of the communication over data endpoints.	
Voltage DAC (VDAC)	It is part of the programmable analog sub system. VDACs generate interrupts on events such as comparator triggers, or when VDAC results are ready. This is available only in PSoC 4100PS.	
CTB/CTBm	Provides continuous time analog functionality. It generates interrupts on event such as comparator triggers.	
WCO WDT/WCO	PSoC 41000S and PSoC 4100S Plus have timers that can be clocked by WC0. These timers can generate interrupts.	

(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 Technical Reference Manual (TRM) to learn about ports that have dedicated interrupts.

(2) Not available in PSoC 4000 / PSoC 4000S/ PSoC 4100S/4100S Plus parts

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.

Figure 2. Level Trigger and Edge Trigger

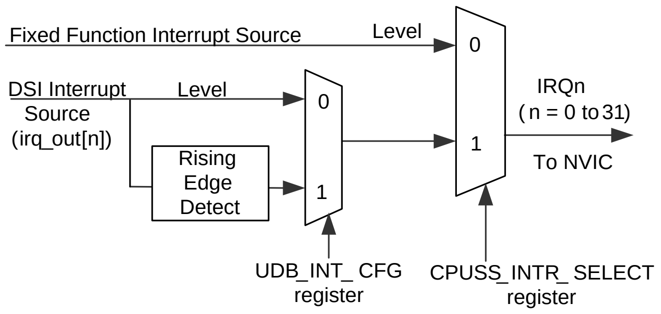

Fixed Function Interrupt Source
Level
0
DSI Interrupt Source (irq_out[n])
Level
0
Rising Edge Detect
1
1
IRQn (n = 0 to 31)
To NVIC
UDB_INT_ CFG register
CPUSS_INTR_ SELECT register

Figure 3. Level-Triggered Interrupts

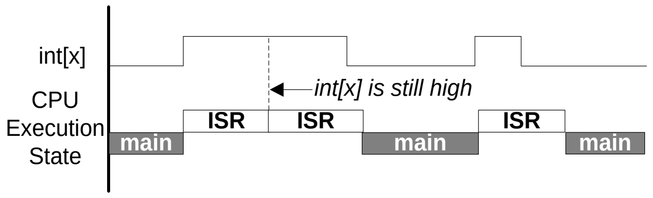
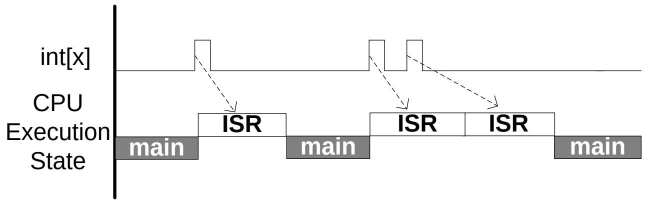


Figure 4. Edge-Triggered Interrupts



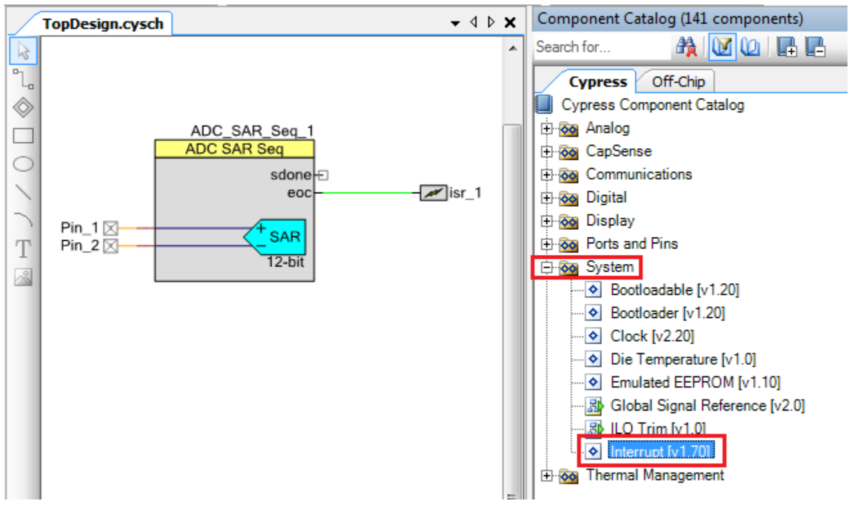
Note: The GPIO interrupt logic has additional circuitry to support interrupts on the rising edge, falling edge, and both edges. See the PSoC 4 Technical Reference Manual for more information.

3 Interrupt Support in PSoC Creator

The previous sections show that some properties of interrupts such as the level or edge trigger, vector address, and interrupt priority must be configured. Configuration is facilitated by the PSoC Creator Interrupt Component. This Component is available under the System tab in the Component Catalog window, as Figure 5 shows.

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 5, 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.

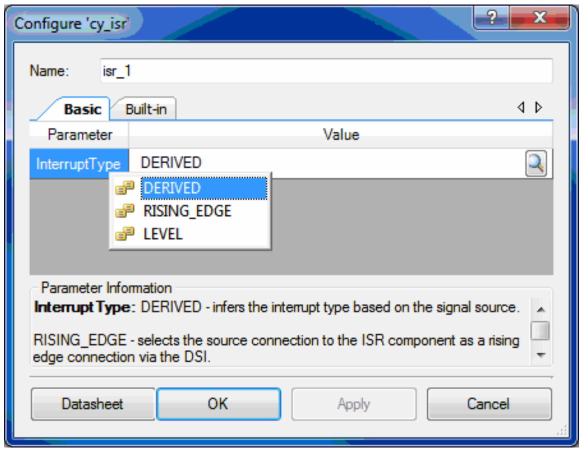
Figure 5. PSoC Creator Interrupt Component



3.1 Interrupt Component Configuration

Figure 6 shows the Interrupt Component configuration dialog. There are three options in the Component: DERIVED, RISING_EDGE, and LEVEL.

Figure 6. Interrupt Component Configuration



Name: isr_1

InterruptType: DERIVED

Interrupt Type: DERIVED - infers the interrupt type based on the signal source.

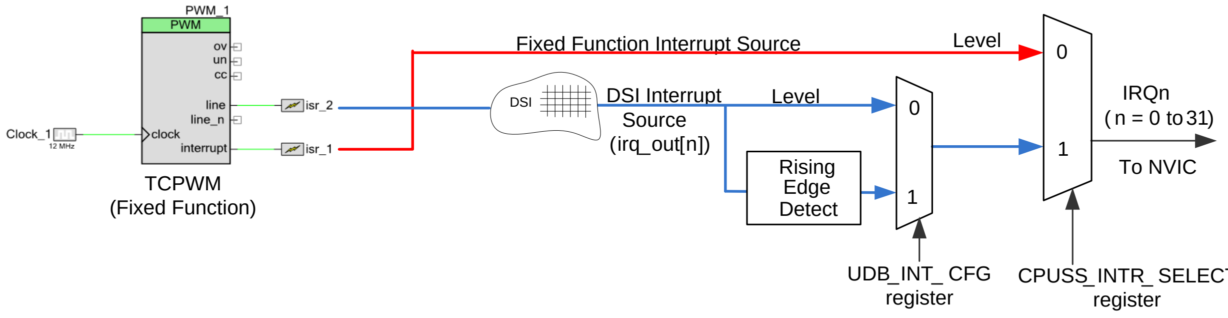
RISING_EDGE - selects the source connection to the ISR component as a rising edge connection via the DSI.

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 7. 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 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.

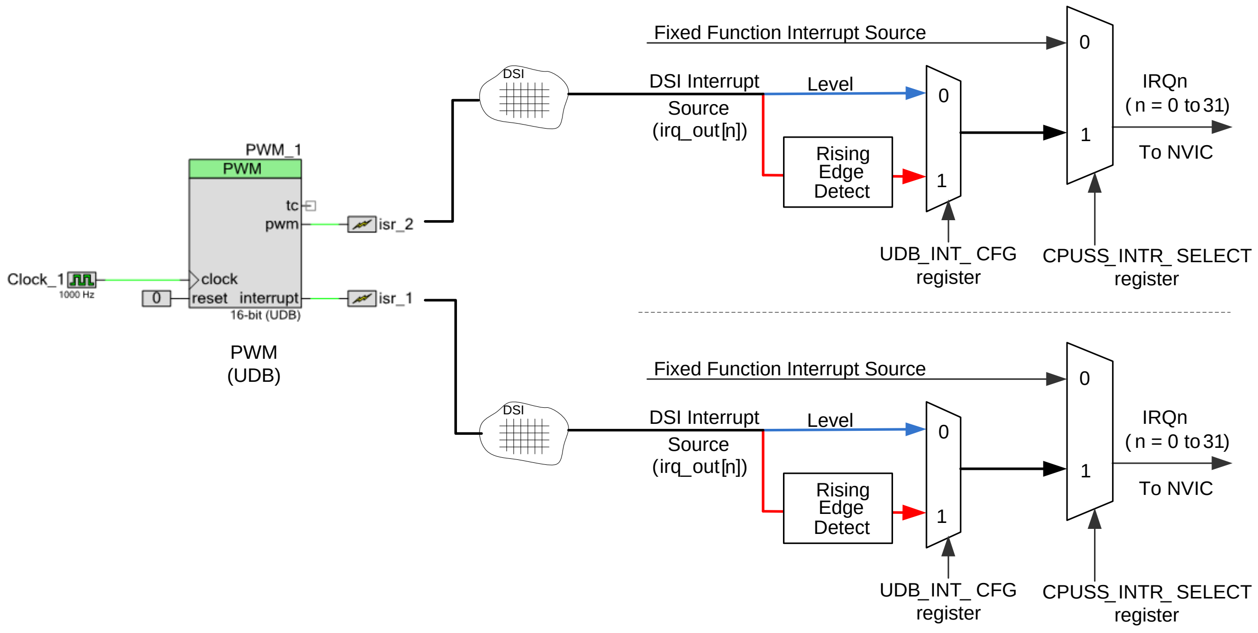
However, in the PSoC part that has DSI, other output signals from the fixed-function block can be routed for interrupts. This allows the RISING_EDGE option as shown by the blue line in Figure 7 for the "line" output of a PWM Component. The Interrupt Component (isr_2) connected to the output of the PWM 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 signal is HIGH, and so in most cases, RISING_EDGE is used.

Figure 7. 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 8. 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 8. Interrupt Routing for UDBs



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

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

Table 2. Interrupt Component Configuration

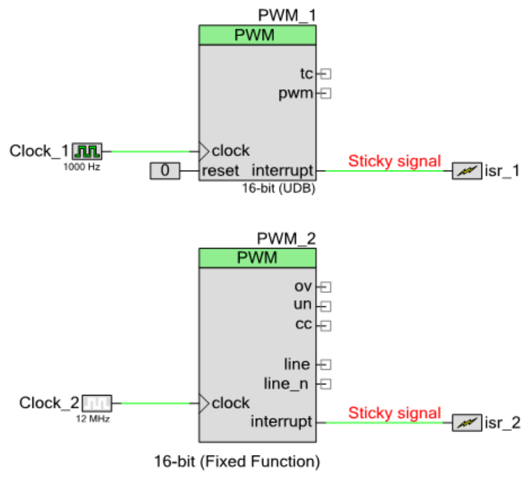
Interrupt Source	Signal	Interrupt Component Configuration
Fixed-Function	Interrupt	Select LEVEL or DERIVED. RISING_EDGE is not allowed.
	Block output	Select RISING_EDGE; otherwise, the interrupt will be repeatedly triggered for the duration of the logic HIGH signal state.
UDB Function	Interrupt	Select 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.

3.1.1 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 provided by the Component. See the Component datasheet of the interrupt source. You can also refer the Writing an Interrupt Service Routine (ISR) 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, "pwm" line of a PWM Component as shown in Figure 9) are connected to the Interrupt Component instead of the interrupt 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.

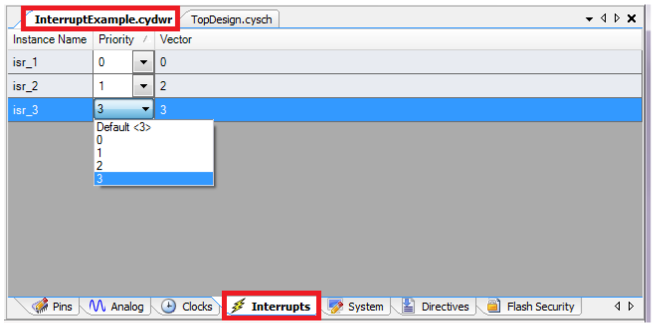
Figure 9. Sticky Signal



3.2 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 10 shows. isr_1, isr_2, and isr_3 are the Interrupt Components used in the design.

Figure 10. Interrupt Tab in *cydwr* Window



Instance Name	Priority	Vector
isr_1	0	0
isr_2	1	2
isr_3	3	3

Use the *cydwr* window to change the priority of an interrupt. Remember that 0 is the highest priority; and 3, the lowest priority. The Cortex-M0/Cortex 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 Forcing the Interrupt Vector Number for details. Also, note that the vector number is shown with an offset in the *.cydwr* window. Vector number of 0 corresponds to exception number 16 in Cortex-M0/Cortex M0+. See Exceptions for an overview of Cortex-M0/Cortex M0+ exceptions.

3.3 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 each 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, i.e., 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.

3.3.1 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.

The CPU executes a logical OR operation of the temporary register with its mask value.

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 Writing an 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 Figure 11 shows.

After building the project, PSoC Creator generates the files associated with all the Components as shown in Figure 12. *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 11. Timer Interrupt Example

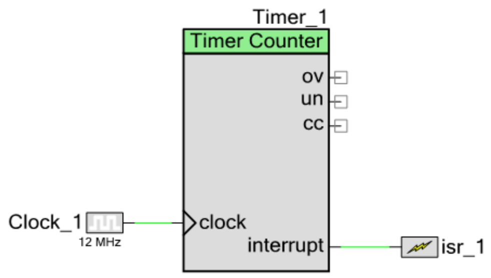
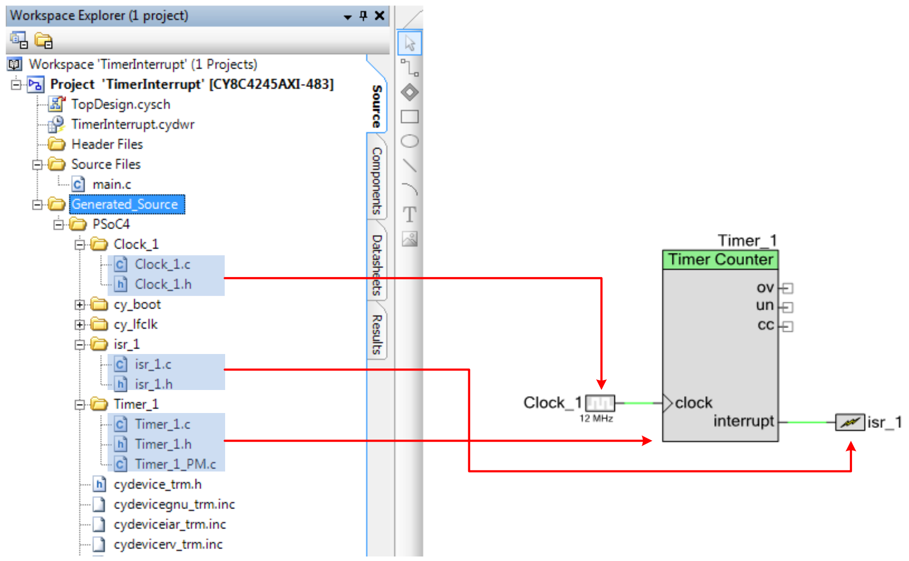


Figure 12. Files Generated for Interrupt Components



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

4.1 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)`.

```
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. The following is the *main.c* code to start the interrupt source, that is, the timer and the Interrupt Component.

```
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 interrupt 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.

4.1.1 Using extern keyword

Many times, in the auto-generated ISR, it is required to access the variables and call the functions defined in the 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 following code in the beginning of the file-

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

/* `#END` */
```

You can either declare the variables and functions between #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-

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

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

4.2 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.

```
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` */
}
```

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.

1. 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 follows:

```
#ifndef CYAPICALLBACKS_H
#define CYAPICALLBACKS_H


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

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

4.3 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);`

Define the custom function using the `CY_ISR` macro:

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

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);`

```
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. */
      }
}
```

4.3.1 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 Code Examples

Table 3 provides the list of code examples that uses interrupt feature.

Table 3. Interrupt Code Examples

Code Example	Interrupt Source
CE210557 – PSoC 4 Timer Interrupt	Timer
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
CE95298 – Switch Debouncer with PSoC 3/4/5LP	GPIO, Debouncer
CE97089 – PSoC 4 ADC to Memory Buffer DMA Transfer	DMA
CE210741 – UART Full Duplex and printf Support with PSoC	UART

6 Debugging Tips

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

a. **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 an Interrupt Service Routine (ISR) 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.

b. **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.

c. **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.

d. **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 **Help** > **Topics** menu in PSoC Creator.

As an alternative to 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.

7 Advanced Interrupt Topics

7.1 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 Table 4 shows.

Table 4. 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. It generates periodic interrupts for use in operating system applications.
IRQ0 to IRQ31	16–47	Programmable	External (Pins) or internal peripheral interrupts.
Reset	1	–3 (Highest)	Triggered on power-on-reset or external reset.

Note that the exception numbers are defined by Arm. Interrupt vector numbers, 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 http://infocenter.arm.com/help/index.jsp.

7.2 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 5 provides the synchronizer CPU clock cycle delays in different PSoC 4 families for DSI and fixed-function source interrupts.

Table 5. 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

Device	DSI Interrupt	Fixed-Function Interrupt
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.

7.3 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 long time to execute, 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.

7.4 Components with Inbuilt Interrupts

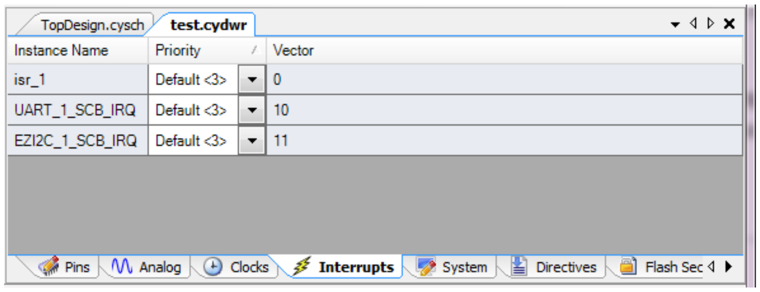
Many PSoC Creator Components have an Interrupt Component internally as part of their implementation. Examples include CapSense®, SAR ADC, EZI2C, and Segment LCD.

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.

7.5 Forcing the Interrupt Vector Number

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 Figure 13 shows. 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 13. Interrupt Vector Numbers in *cydwr* Window



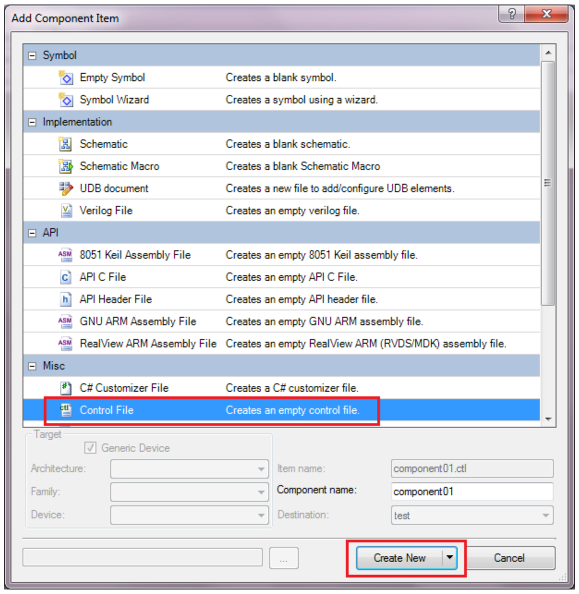
Instance Name	Priority	Vector
isr_1	Default <3>	0
UART_1_SCB_IRQ	Default <3>	10
EZI2C_1_SCB_IRQ	Default <3>	11

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.

Right-click the **TopDesign** Component and select **Add Component Item…**. The Add Component Item dialog opens. Scroll down to the **Misc** group, select **Control File**, and click **Create New**, as Figure 14 shows.

Figure 14. Adding the Control File



Symbol	
Empty Symbol	Creates a blank symbol.
Symbol Wizard	Creates a symbol using a wizard.
Implementation	
Schematic	Creates a blank schematic.
Schematic Macro	Creates a blank Schematic Macro
UDB document	Creates a new file to add/configure UDB elements.
Verilog File	Creates an empty verilog file.
API	
8051 Keil Assembly File	Creates an empty 8051 Keil assembly file.
API C File	Creates an empty API C File.
API Header File	Creates an empty API header file.
GNU ARM Assembly File	Creates an empty GNU ARM assembly file.
RealView ARM Assembly File	Creates an empty RealView ARM (RVDS/MDK) assembly file.
Misc	
C# Customizer File	Creates a C# customizer file.
Control File	Creates an empty control file.

Item name: component01.ctl
Component name: component01
Destination: test

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

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 13, *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)"
```

After assigning the interrupt vector numbers, click **Save** to save the changes made to the control file.

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.

7.6 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.

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:

```
#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(;;)
    {
    }
}
```

7.7 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, 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.

7.8 Use of the 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), etc. For details on available interrupt signals, see the Component datasheet. The combined port interrupt is explained below.

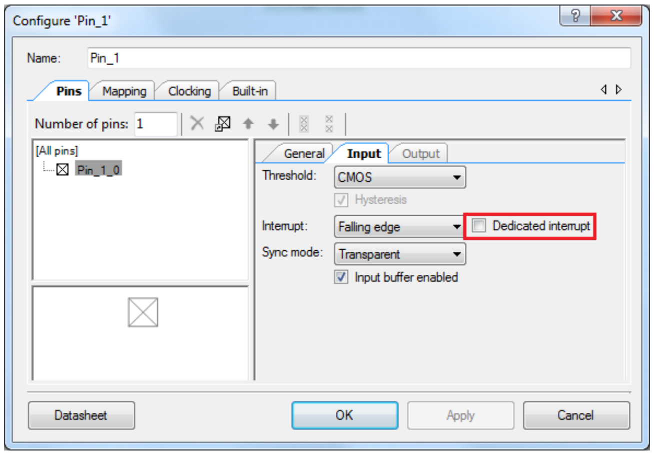
7.8.1 Combined Port Interrupt

In most of the PSoC 4 devices, not all ports have a dedicated interrupt vector (see Table 6). In such a case, it is recommended to use the combined port interrupt feature in the GlobalSignal Component. The combined port interrupt uses an 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:

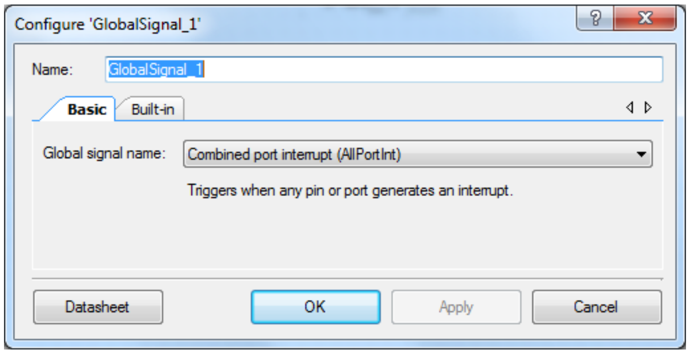
1. Place a Digital Input Pin Component and set it to P5[0].
2. Configure the pin interrupt. Ensure that dedicated interrupt is unchecked.

Figure 15. Pin Properties for Combined Interrupt



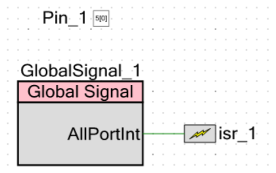
3. Place a GlobalSignal Component and set it to "Combined Port Interrupt (AllPortInt)".

Figure 16. Placing a GlobalSignal Component


Configure 'GlobalSignal_1'
Name: GlobalSignal_1
Basic
Built-in
Global signal name: Combined port interrupt (AllPortInt)
Triggers when any pin or port generates an interrupt.
Datasheet
OK
Apply
Cancel

4. Connect an interrupt Component to the GlobalSignal Component.

Figure 17. 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 Technical Reference Manual (TRM)* of the device for details on the register.

7.9 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 will 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.

8 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.

About the Author

Name:	Rajiv Badiger
Title:	Staff Applications Engineer
Background	BE Electronics and Communication

Appendix A. Interrupt Sources and Vector Numbers in PSoC 4

Table 6 lists the interrupt sources for the 32 interrupt vectors in PSoC 4.

Table 6. 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
GPIO Interrupt – Port 0	DSI	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0
GPIO Interrupt – Port 1	DSI	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1
GPIO Interrupt – Port 2	DSI	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2	IRQ2
GPIO Interrupt – Port 3	DSI	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
–	DSI	–	IRQ5	–	–	–	–	–	–	–
–	DSI	–	IRQ6	–	–	–	–	–	–	–
–	DSI	–	IRQ7	–	–	–	–	–	–	–
LPCOMP (low-power comparator)	DSI	–	IRQ8	IRQ7	IRQ6	IRQ7	IRQ5	IRQ5	IRQ5	IRQ5
WDT (Watchdog timer)	DSI	IRQ4	IRQ9	IRQ8	IRQ7	IRQ8	IRQ6	IRQ6	IRQ6	IRQ7
SCB0 (Serial Communication Block 0)	DSI	IRQ5	IRQ10	IRQ9	IRQ8	IRQ9	IRQ7	IRQ7	IRQ7	IRQ8
SCB1 (Serial Communication Block 1)	DSI	–	IRQ11	IRQ10	IRQ9	IRQ10	IRQ8	IRQ8	IRQ8	IRQ9
SCB2 (Serial Communication Block 2)	DSI	–	–	–	IRQ10	IRQ11	–	IRQ9	IRQ9	IRQ10
SCB3 (Serial Communication Block 3)	DSI	–	–	–	IRQ11	IRQ12	–	–	IRQ10	–
SCB3 (Serial Communication Block 4)	DSI	–	–	–	–	–	–	–	IRQ11	–
CTBm Interrupt (all CTBms)	DSI	–	–	IRQ11	IRQ12	IRQ13	–	IRQ10	IRQ12	–
CTB Interrupt	–	–	–	–	–	–	–	–	–	IRQ6
BLE Subsystem Interrupt	DSI	–	–	IRQ12	–	–	–	–	–	–
DMA Interrupt	DSI	–	–	–	IRQ13	IRQ14	–	–	IRQ14	IRQ12
SPCIF Interrupt	DSI	IRQ6	IRQ12	IRQ13	IRQ14	IRQ15	IRQ9	IRQ10	IRQ15	IRQ13

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
SRSS LVD Interrupt	DSI	–	IRQ13	IRQ14	IRQ15	IRQ16	–	–	–	–
SAR (Successive Approximation ADC)	DSI	–	IRQ14	IRQ15	IRQ16	IRQ17	–	IRQ19	IRQ25	IRQ15
CSD0 (CapSense)	DSI	IRQ7	IRQ15	IRQ16	IRQ17	IRQ18	IRQ10	IRQ13	IRQ16	IRQ14
CSD1 (CapSense)	DSI	–	–	–	IRQ18	IRQ19	–	–	–	–
VDAC Interrupt (Both VDACs)	–	–	–	–	–	–	–	–	–	IRQ16
TCPWM0 (Timer/Counter/PWM 0)	DSI	IRQ8	IRQ16	IRQ17	IRQ19	IRQ20	IRQ11	IRQ14	IRQ17	IRQ17
TCPWM1 (Timer/Counter/PWM 1)	DSI	–	IRQ17	IRQ18	IRQ20	IRQ21	IRQ12	IRQ15	IRQ18	IRQ18
TCPWM2 (Timer/Counter/PWM 2)	DSI	–	IRQ18	IRQ19	IRQ21	IRQ22	IRQ13	IRQ16	IRQ19	IRQ19
TCPWM3 (Timer/Counter/PWM 3)	DSI	–	IRQ19	IRQ20	IRQ22	IRQ23	IRQ14	IRQ17	IRQ20	IRQ20
TCPWM4 (Timer/Counter/PWM 4)	DSI	–	–	–	IRQ23	IRQ24	IRQ15	IRQ18	IRQ21	IRQ21
TCPWM5 (Timer/Counter/PWM 5)	DSI	–	–	–	IRQ24	IRQ25	–	–	IRQ22	IRQ22
TCPWM6 (Timer/Counter/PWM 6)	DSI	–	–	–	IRQ25	IRQ26	–	–	IRQ23	IRQ23
TCPWM7 (Timer/Counter/PWM 7)	DSI	–	–	–	IRQ26	IRQ27	–	–	IRQ24	IRQ24
CAN0 Interrupt	DSI	–	–	–	IRQ27	IRQ28	–	–	IRQ26	–
CAN1 Interrupt	DSI	–	–	–	IRQ28	IRQ29	–	–	–	–
USB Start of Frame	DSI	–	–	–	–	IRQ30	–	–	–	–
USB EP1-EP8 Data	DSI	–	–	–	–	IRQ31	–	–	–	–
Crypto Interrupt	–	–	–	–		–	–	–	IRQ27	–
WCO/WDT Interrupt	–	–	–	–	–	–	–	–	IRQ13	IRQ11
–	DSI	–	IRQ20	IRQ21	IRQ29	–	–	–	–	–
–	DSI	–	IRQ21	IRQ22	IRQ30	–	–	–	–	–
–	DSI	–	IRQ22	IRQ23	IRQ31	–	–	–	–	–
–	DSI	–	IRQ23	IRQ24	–	–	–	–	–	–
–	DSI	–	IRQ24	IRQ25	–	–	–	–	–	–
–	DSI	–	IRQ25	IRQ26	–	–	–	–	–	–
–	DSI	–	IRQ26	IRQ27	–	–	–	–	–	–

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
–	DSI	–	IRQ27	IRQ28	–	–	–	–	–	–
–	DSI	–	IRQ28	IRQ29	–	–	–	–	–	–
–	DSI	–	IRQ29	IRQ30	–	–	–	–	–	–
–	DSI	–	IRQ30	IRQ31	–	–	–	–	–	–
–	DSI	–	IRQ31	–	–	–	–	–	–	–

Document History

Document Title: AN90799 – PSoC® 4 Interrupts

Document Number: 001-90799

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4371470	RJVB	05/22/2014	New Application Note
*A	4765616	RJVB	05/14/2015	Updated for PSoC 4 BLE 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	4968362	RJVB	02/02/2016	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	5687926	BENV	04/19/2017	Updated logo and copyright
*D	5966337	JSLN	12/13/2017	Updated for PSoC 4000S, 4100S, PSoC 4100S Plus, and PSoC Analog Coprocessor. Updated Table 1, Table 3, Table 5, and Table 6.
*E	6281196	DIMA/ RJVB	09/14/2018	Updated for PSoC 4100PS Added Using extern keyword, Using the Callback Function, Use of the GlobalSignal Component and Use of Volatile for Global Variables Updated Using Auto-Generated ISR

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709