Interrupt
1.50

Features



- Defines hardware-triggered interrupts
- Provides a software method to pend interrupts

General Description

The Interrupt component defines hardware triggered interrupts. It is an integral part of the Interrupt Design-Wide Resource system (see PSoC Creator Help, Design-Wide Resources section).

There are three types of system interrupt waveforms that can be processed by the interrupt controller:

- **Level** – IRQ source is sticky and remains active until firmware clears the source of the request with an action (for example, clear on read). Most fixed-function peripherals have level-sensitive interrupts, including the UDB FIFOs and status registers.
- **Pulse** – Ideally, a pulse IRQ is a single bus clock, which logs a pending action and ensures that the ISR action is only executed once. No firmware action to the peripheral is required.
- **Edge** – An arbitrary synchronous waveform is the input to an edge-detect circuit and the positive edge of that waveform becomes a synchronous one-cycle pulse (Pulse mode).

Note These interrupt waveform types are different from the settings made in the **Configure** dialog for the **InterruptType** parameter. The parameter only configures the multiplexer select lines. It processes the “IRQ” signal to be sent to the interrupt controller based on the multiplexer selection (Level, Edge).

In other words, regardless of the **InterruptType** multiplexer selection, the interrupt controller is still able to process level, edge, or pulse waveforms. Refer to the applicable TRM document for more details.

When to Use an Interrupt Component

Use an Interrupt component whenever a hardware-triggered interrupt is required. Interrupts are indispensable because they use hardware support to reduce both the latency and overhead of event detection, when compared to polling.

Input/Output Connections

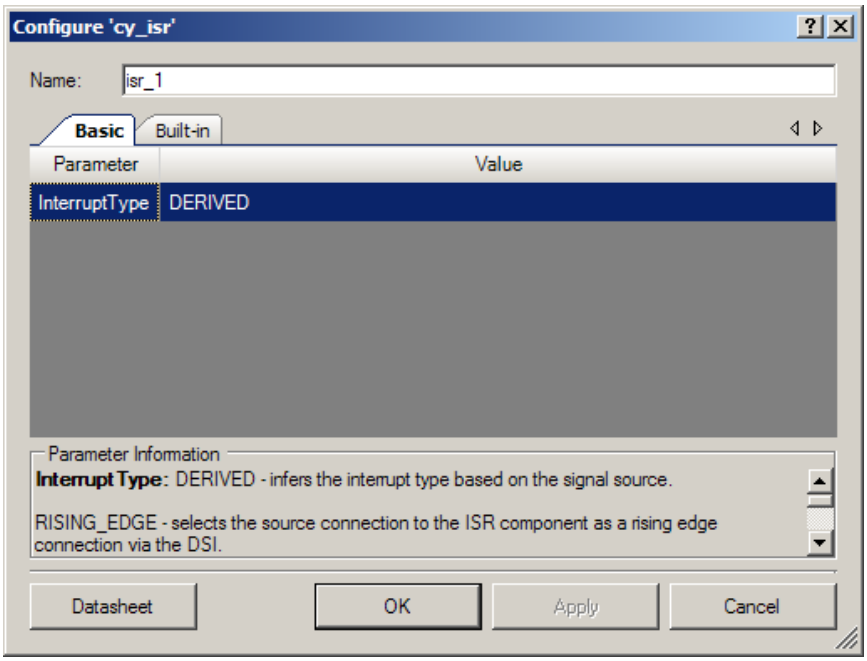
This section describes the various input and output connections for the Interrupt component.

int_signal – Input

Connect the signal that generates the interrupt to this input. When the signal value becomes logic high, the interrupt is triggered.

Component Parameters

Drag an Interrupt component onto your design and double-click it to open the **Configure** dialog.



The Interrupt component provides the following parameters:

InterruptType

This parameter configures which type of waveform the component will process to trigger the interrupt. There are three possible values for this parameter:

- **RISING_EDGE** – Triggers the interrupt on the rising edge of the source signal. If this option is selected, a rising edge on the “int_signal” input is converted into a pulse of period “bus_clk” and is sent to the interrupt controller.
- **LEVEL** – Selects the source connected to the interrupt as a level-sensitive connection through the DSI. If this option is selected, the “int_signal” input is directly passed to the interrupt controller. See the General Description section for more details.

- **DERIVED** – This is the default setting. It inspects the driver of the “int_signal” and, when connected to a fixed-function block (I^2C, USB, CAN, and so on), derives the interrupt type based on what it is connected to. This automatic assignment is based on information found in the device datasheet.

When connected to fixed-function interrupt outputs, the type should be set to DERIVED. For other interrupt sources, you should usually choose RISING_EDGE to capture an event (for example, periodic clock) and LEVEL for a state (for example, FIFO fill levels). For DMA NRQ signals, any of the settings produce the same result of a single interrupt for each NRQ event.

Placement

Each Interrupt component consumes one entry in the device's interrupt vector memory.

Resources

The Interrupt component uses the following device resources:

- Flash space for interrupt service routine (ISR) code
- An entry in the device interrupt vector memory

Analog Blocks	Digital Blocks					API Memory (Bytes)		Pins (per External I/O)
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
N/A	N/A	N/A	N/A	N/A	N/A	189	0	N/A

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name “ISR_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “ISR.”

Function	Description
ISR_Start()	Sets up the interrupt to function.
ISR_StartEx()	Sets up the interrupt to function and sets address as the ISR vector for the interrupt.

Function	Description
ISR_Stop()	Disables and removes the interrupt.
ISR_Interrupt()	The default interrupt handler for ISR.
ISR_SetVector()	Sets address as the new ISR vector for the Interrupt.
ISR_GetVector()	Gets the address of the current ISR vector for the interrupt.
ISR_SetPriority()	Sets the priority of the interrupt.
ISR_GetPriority()	Gets the priority of the interrupt.
ISR_Enable()	Enables the interrupt to the interrupt controller.
ISR_GetState()	Gets the state (enabled, disabled) of the interrupt.
ISR_Disable()	Disables the interrupt.
ISR_SetPending()	Causes the interrupt to enter the pending state, a software method of generating the interrupt.
ISR_ClearPending()	Clears a pending interrupt.

void ISR_Start(void)

Description: Sets up the interrupt and enables it. This function disables the interrupt, sets the default interrupt vector, sets the priority from the value in the Design Wide Resources Interrupt Editor, then enables the interrupt to the interrupt controller.

Parameters: void

Return Value: void

Side Effects: None

void ISR_StartEx(cyisraddress address)

Description: Sets up the interrupt and enables it. This function disables the interrupt, sets the interrupt vector based on the address passed in, sets the priority from the value in the Design Wide Resources Interrupt Editor, then enables the interrupt to the interrupt controller.

Parameters: address: Address of the ISR to set in the interrupt vector table

Return Value: void

Side Effects: None

void ISR_Stop(void)

Description: Disables and removes the interrupt.

Parameters: void

Return Value: void

Side Effects: None

void ISR_Interrupt(void)

Description: The default ISR for the component. Locate this function in the corresponding C file and add code between the START and END comments.

Parameters: void

Return Value: void

Side Effects: None

void ISR_SetVector(cyisraddress address)

Description: Changes the ISR vector for the interrupt. Use this function to change the ISR vector to the address of a different interrupt service routine. Note that calling ISR_Start() overrides any effect this method would have had. To set the vector before the component has been started, use ISR_StartEx() instead.

Parameters: address: Address of the ISR to set in the interrupt vector table

Return Value: void

Side Effects: Disable the interrupt before calling this function and re-enable it after.

cyisraddress ISR_GetVector(void)

Description: Gets the address of the current ISR vector for the interrupt.

Parameters: void

Return Value: cyisraddress: Address of the current ISR

Side Effects: None

void ISR_SetPriority(uint8 priority)

Description: Sets the priority of the interrupt.

Note Calling ISR_Start() or ISR_StartEx() overrides any effect this method would have had. This method should only be called after ISR_Start() or ISR_StartEx() has been called. To set the initial priority for the component, use the Design-Wide Resources Interrupt Editor.

Parameters: priority: Priority of the interrupt. 0 to 7, 0 is the highest

Return Value: void

Side Effects: None

uint8 ISR_GetPriority(void)

Description: Gets the priority of the interrupt.

Parameters: void

Return Value: Priority of the interrupt. 0 to 7, 0 is the highest

Side Effects: None

void ISR_Enable(void)

Description: Enables the interrupt to the interrupt controller. Do not call this function unless ISR_Start() has been called or the functionality of the ISR_Start() function, which sets the vector and the priority, has been called.

Parameters: void

Return Value: void

Side Effects: None

uint8 ISR_GetState(void)

Description: Gets the state (enabled, disabled) of the interrupt.

Parameters: void

Return Value: 1 if enabled, 0 if disabled.

Side Effects: None

void ISR_Disable(void)

Description: Disables the interrupt to the interrupt controller.

Parameters: void

Return Value: void

Side Effects: None

void ISR_SetPending(void)

Description: Causes the interrupt to enter the pending state; a software method of generating the interrupt.

Parameters: void

Return Value: void

Side Effects: If interrupts are enabled and the interrupt is set up properly, the ISR is entered (depending on the priority of this interrupt and other pending interrupts).

void ISR_ClearPending(void)

Description: Clears a pending interrupt to the interrupt controller.

Parameters: void

Return Value: void

Side Effects: Some interrupt sources also need to be cleared with the appropriate block API (GPIO, UART, and so on) or they will just re-pend the interrupt. Entering the ISR clears the pending bit for some interrupt sources.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

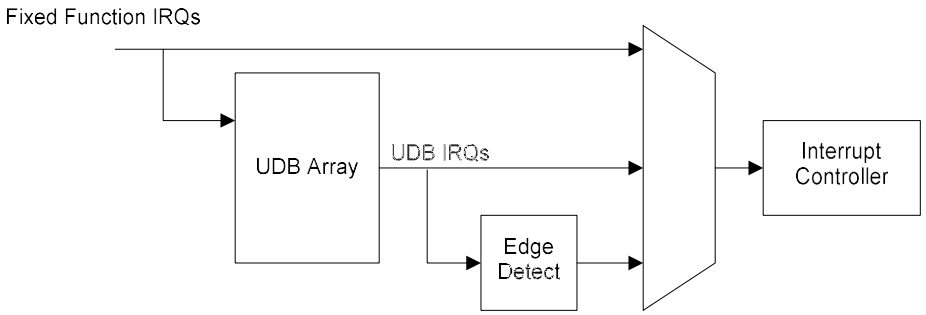
Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Functional Description

Interrupt routing is flexible in the PSoC 3 architecture. In addition to the fixed-function peripherals, any data signal in the UDB array routing can be used to generate an interrupt. A

high-level view of the interrupt mux (IDMUX) routing is shown in Figure 1. The IDMUX selects from the available sources of interrupt requests.

Figure 1. IDMUX Routing



Design-Wide Resources

The use of an Interrupt component in a design results in an entry in the Design-Wide Resources editor. The **Interrupts** tab contains the following parameters:

Instance Name	Priority	ES2 Patch	Vector
isr_1	Default <7>	☐	1
isr_2	Default <7>	☐	3

- **Instance Name** – Shows the component instance names in your design.
- **Priority** – Shows and allows you to set the instance's priority.
- **ES2 Patch** –This is a temporary workaround that fixes a problem with PSoC 3 interrupts. A high-priority interrupt arrives at the same time a lower priority interrupt is returning (RETI), and the higher priority interrupt gets lost. Select this option if the application uses more than one interrupt. Note that this option is temporary until the silicon is updated.
- **Vector** – Indicates the interrupt vector.

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.50.c	Improved explanation of the Derived option in the datasheet	
1.50.b	Datasheet corrections	
1.50.a	Minor datasheet edits and updates	

Version	Description of Changes	Reason for Changes / Impact
1.50	Added InterruptType parameter.	The old functionality (equivalent to selecting "DERIVED" in the new version) can't determine the desired interrupt type in all situations, so the ability to specify it manually was added.
	Don't redefine CYINT_VECTORS and CYINT_IRQ_BASE if they already exist.	These macros were already defined in *CyLib.h*. The redefinition caused a warning with some versions of cy_boot. This change affects PSoC 5 only.
	Declare ISR with CY_ISR.	This causes the compiler to generate code that ensures correct stack alignment on PSoC 5.
	Use *cydevice_trm.h* instead of *cydevice.h*.	*cydevice.h* is obsolete and should only be used for compatibility with old components and firmware. If the code in the Interrupt API function requires *cydevice.h*, then include *cydevice.h* in the "Place your includes, defines, and code here" section.
	Added ISR_StartEx	Allows for the setting of the address of the ISR to set in the interrupt vector table before the interrupt has been started so that it is used as the default instead of ISR_Interrupt.
	Added `=ReentrantKeil($INSTANCE_NAME . "_...")` to the following functions: void ISR_Stop() void ISR_SetVector() cyisraddress ISR_GetVector() void ISR_SetPriority() uint8 ISR_GetPriority() void ISR_Enable() uint8 ISR_GetState() void ISR_Disable() void ISR_SetPending() void ISR_ClearPending()	Allows users to make these APIs reentrant if reentrancy is desired.
1.20	ES2 ISR patch.	

© Cypress Semiconductor Corporation, 2010-2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.