Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

Interrupt
1.71



Features

- Defines hardware-triggered interrupts
- Provides a software API to pend (hardware-connected) 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

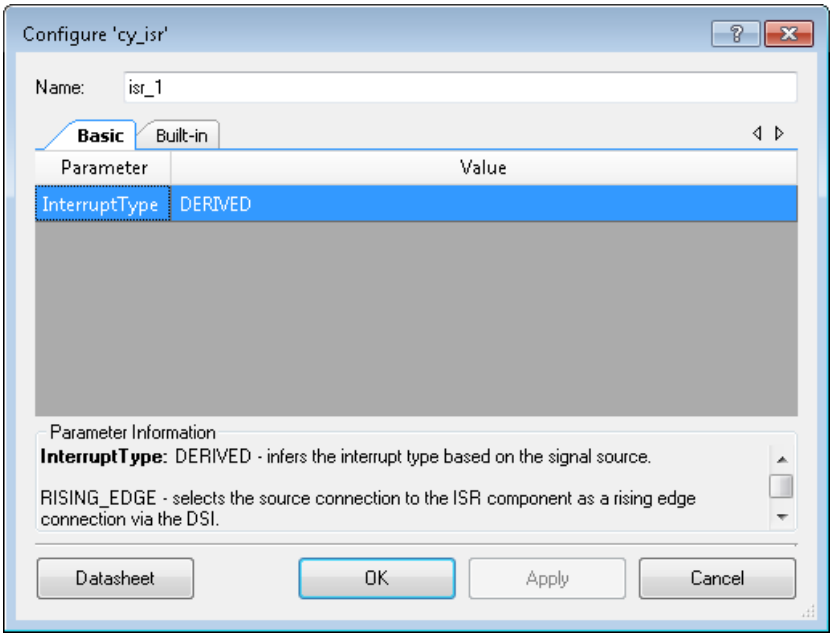
This section describes the various input and output connections for the Interrupt Component.

int_signal – Input

Connect the signal that generates the interrupt to this input. When the signal value becomes logic high, the interrupt is triggered. For a Level type interrupt, the interrupt will continue to trigger as long as the signal remains logic high.

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.

  Note For PSoC 4 devices, RISING_EDGE cannot be used to connect to fixed-function peripheral interrupt sources. On PSoC 5LP, if the interrupt Component is connected to a wakeup source used to wake the part up from the sleep or hibernate low-power mode, RISING_EDGE may not be used.

- **LEVEL** – Selects the source connected to the interrupt as a level-sensitive connection. If this option is selected, the “int_signal” input is directly passed to the interrupt controller.

  Note Most fixed-function peripherals have level-sensitive interrupts. However, you can use RISING_EDGE on that signal source if UDB routing is available (not possible for PSoC 4 devices). UDB-based peripherals may use either RISING_EDGE or LEVEL triggers depending on the usage scenario.

- **DERIVED** – This is the default setting. It inspects the driver of the “int_signal” and derives the interrupt type based on what it is connected to. For most fixed-function blocks, the interrupt type is LEVEL. For UDB signal sources, the interrupt type is RISING_EDGE.

As a guideline, you should use RISING_EDGE when capturing a signal change (for example, periodic clock), and use LEVEL when capturing a state change of a peripheral (for example, FIFO fill levels). The DERIVED interrupt type should be used sparingly as it does not give you full control.

For PSoC 3 and PSoC 5LP DMA NRQ signals, the interrupt path is dedicated and any setting will produce the same result of a single interrupt (edge-triggered) for each NRQ event.

For Component specific interrupt usage information, refer to that Component’s datasheet.

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.

PSoC Creator provides two methods for configuring interrupts.

- For peripheral interrupts via schematic entry, the interrupt Component API allows instance-specific configuration as assigned by the PSoC Creator tool.
- For configuring processor core exceptions and for software-triggered interrupts, the *cy_boot* design-wide Component (always included in every project) provides an interrupt API in *CyLib.c*. Refer to the *System Reference Guide* for details on the *cy_boot* Component.

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.”

Functions

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.
ISR_Stop()	Disables and un-configures 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 in the interrupt controller.

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 in the interrupt controller.

When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be used to provide consistent definition across compilers:

Function definition example:

```
CY_ISR(MyISR)
{
  /* ISR Code here */
}
```

Function prototype example:

```
CY_ISR_PROTO(MyISR);
```

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

void ISR_Stop(void)

Description: Disables and removes the interrupt.

void ISR_Interrupt(void)

Description: The default ISR for the Component. Add custom code between the START and END comments to keep the next version of this file from over-writing your code.

Note You may use either the default ISR by using this API, or you may define your own separate ISR through ISR_StartEx().

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 API would have had. To set the vector before the Component has been started, use ISR_StartEx() instead.

When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be used to provide consistent definition across compilers:

Function definition example:

```
CY_ISR(MyISR)
{
  /* ISR Code here */
}
```

Function prototype example:

```
CY_ISR_PROTO(MyISR);
```

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

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.

Return Value: cyisraddress: Address of the current ISR

void ISR_SetPriority(uint8 priority)

Description: Sets the priority of the interrupt.

Note Calling ISR_Start() or ISR_StartEx() overrides any effect this API would have had. This API 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 being the highest priority

PSoC 3 and PSoC 5LP: Priority is from 0 to 7.

PSoC 4: Priority is from 0 to 3.

uint8 ISR_GetPriority(void)

Description: Gets the priority of the interrupt.

Return Value: Priority of the interrupt, 0 being the highest priority.

PSoC 3 and PSoC 5LP: Priority is from 0 to 7.

PSoC 4: Priority is from 0 to 3.

void ISR_Enable(void)

Description: Enables the interrupt in 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.

uint8 ISR_GetState(void)

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

Return Value: 1 if enabled, 0 if disabled.

void ISR_Disable(void)

Description: Disables the interrupt in the interrupt controller.

void ISR_SetPending(void)

Description: Causes the interrupt to enter the pending state; a software API to generate the interrupt.

void ISR_ClearPending(void)

Description: Clears a pending interrupt in the interrupt controller.

Note Some interrupt sources are clear-on-read and require the block interrupt/status register to be read/cleared with the appropriate block API (GPIO, UART, and so on). Otherwise the ISR will continue to remain in pending state even though the interrupt itself is cleared using this API.

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for the more details.

In order to add code to the macro callback present in the Component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will "uncomment" the function call from the Component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.
- Write the function implementation (in any user file).

Callback Function [1]	Associated Macro	Description
ISR_Interrupt_InterruptCallback	ISR_INTERRUPT_INTERRUPT_CALLBACK	Used in the ISR_Interrupt() interrupt handler to perform additional application-specific actions.

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.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the Component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator Components
- specific deviations – deviations that are applicable only for this Component.

This section provides information on Component specific deviations. The project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

1 The callback function name is formed by component function name optionally appended by short explanation and “Callback” suffix.

The Interrupt Component is MISRA compliant, except for the following specific deviations:

MISRA-C: 2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
1.2	R	No reliance shall be placed on undefined or unspecified behavior.	Based on the PSoC Creator design the identifiers are generated based on component name. This insure that global object names are unique and self-documented. There are no drawbacks as current compilers support much greater number of significant characters than 32.
8.8	R	An external object or function shall be declared in one and only one file.	IntDefaultHandler (PSoC 5LP/PSoC 4) and CyRamVectors (PSoC 4) are being declared with external linkage, but these declarations are not in a header file.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Applicable for PSoC 5LP only. Pointer arithmetic used for working with ram vector table.

API Memory Usage

The Component memory usage varies significantly, depending on the compiler, device, number of APIs used and Component configuration. The following table provides the memory usage for all APIs available in the given Component configuration.

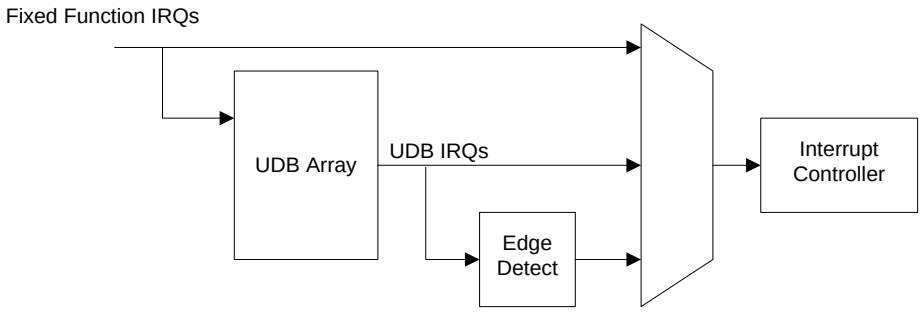
The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 4 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Default	141	0	214	0	196	0

Functional Description

Interrupt routing is flexible in the PSoC 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

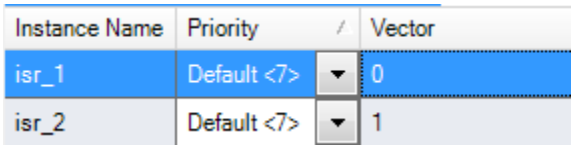


Depending on your choice of Interrupt type, several scenarios are possible.

Source	Interrupt Type	Routing	Description
Fixed-function block interrupt	LEVEL	Direct	Interrupt is a dedicated connection to the interrupt controller.
		UDB	Routed through the UDB array and connected to the interrupt controller. This path is possible in PSoC 3 and PSoC 5LP only when UDB logic is placed between the interrupt and the interrupt controller.
	RISING_EDGE	UDB	Routed through UDB array, edge-detect circuit, and then connected to the interrupt controller (not available in PSoC 4).
	DERIVED	Direct	Interpreted as LEVEL.
UDB block interrupt	LEVEL	UDB	UDB based Components frequently use LEVEL. These are routed directly from the UDB array to the interrupt controller.
	RISING_EDGE	UDB	The signal can route through an edge-detect circuit to the interrupt controller, if the interrupt is set as RISING_EDGE.
	DERIVED	UDB	Interpreted as RISING_EDGE
Signal on Schematic	LEVEL	UDB	This mode is not often used. The signal is routed through the UDB array to the interrupt controller.
	RISING_EDGE	UDB	Used to detect a signal transition. Routes through edge-detect to the interrupt controller.
	DERIVED	UDB	Interpreted as RISING_EDGE
DMA NRQ (PSoC 3, PSoC 5LP)	N/A	Direct	Direct connection to the interrupt controller as an edge-triggered interrupt.

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** – Shows the Component instance names in your design.
- **Priority** – Shows and allows you to set the instance's priority, where 0 is the highest priority.
  - The range is from 0 to 7 in PSoC 3 and PSoC 5LP.
  - The range is from 0 to 3 in PSoC 4.
- **Vector** – Indicates the interrupt vector.

Resources

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

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
1.71.a	Updated datasheet.	Added rule 1.2 to MISRA Compliance table.
1.71	Updated the MISRA Compliance table.	Fixed the deviation for MISRA rule 10.5 for PSoC 4.
1.70.g	Datasheet update	Added information about cy_boot interrupt API
1.70.f	Minor datasheet update.	
1.70.e	Datasheet update.	Added Macro Callbacks section.
1.70.d	Datasheet update.	Updated source code comments and API descriptions.
		Updated InterruptType parameter description and Functional Description section.
1.70.c	Added notes about CY_ISR/CY_ISR_PROTO macros to SetVector and StartEx API descriptions	Clarify usage of CY_ISR/CY_ISR_PROTO macros.
	Minor edits and updates	

Version	Description of Changes	Reason for Changes / Impact
1.70.b	Updated the MISRA-C Rule table.	
1.70.a	Added note about interrupt priority for different devices.	PSoC 4 support.
1.70	Added MISRA Compliance section.	The Component has one specific deviation.
1.60	Minor datasheet edits and updates	
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	
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.

Version	Description of Changes	Reason for Changes / Impact
	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 (an Infineon company), 2020-2021. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical Components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical Component is any Component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.