AN217666 explains the interrupt architecture in PSoC® 6 MCU and its configuration using PSoC Creator™, ModusToolbox™, and PDL APIs. This document serves as a guide in developing projects that use interrupts. Advanced interrupt concepts such as interrupt latency, code optimization, and debug techniques are also explained.

Contents
1 Introduction .................................................................................1
  1.1 How to Use this Document ..............................................2
2 PSoC 6 MCU Interrupt Architecture ...................................2
  2.1 CY8C62x6/7, CY8C63xx Interrupt Architecture ........3
  2.2 CY8C62x8/A Interrupt Architecture .......................4
  2.3 Types of Interrupts ......................................................4
  2.4 Interrupts and Power modes ....................................5
  2.5 CPU Sleep and Wakeup ..........................................6
3 Interrupt Configuration .................................................................7
  3.1 Configuring Interrupts Using PDL ...........................7
  3.2 Configuring Interrupts Using PSoC Creator ............10
  3.3 Using the Design-Wide Resource Window (CyDWR) ....11
  3.4 Configuring Interrupts Using ModusToolbox .......13
  4 Debugging Tips .......................................................................13
  5 Advanced Interrupt Topics .............................................15
    5.1 Exceptions ...........................................................15
    5.2 Interrupt Latency ....................................................16
    5.3 Nested Interrupts ....................................................17
    5.4 Code Optimization ..................................................17
  6 Related Resources .................................................................17
Appendix A. Interrupt Sources in PSoC 6 MCU .............18
Worldwide Sales and Design Support ..............................21

1 Introduction

An interrupt is a hardware signal or an event that transfers the execution of a program from the normal flow to an alternate set of instructions. An interrupt frees the CPU from continuously polling for a specific event, and only notifies and engages the CPU when the event occurs. The alternate program flow is referred to as an interrupt service routine or ISR. An ISR is also called an interrupt handler. After the interrupt is serviced, the program flow is reverted back to the flow that was interrupted. In system-on-chip (SoC) architectures such as PSoC, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU.

While interrupts refer to those events generated by peripherals external to the CPU such as timers, serial communication blocks, and port pin signals, an exception is an event generated by the CPU such as memory access faults and internal system timer events. PSoC 6 MCU supports interrupts and exceptions on both its ARM® Cortex®-M4 (CM4) and ARM Cortex-M0+ (CM0+) CPUs.

More code examples? We heard you.
To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC video library here.
1.1 How to Use this Document

This document assumes that you are familiar with the PSoC 6 MCU architecture, and application development for PSoC devices using the Cypress PSoC Creator™ integrated design environment (IDE) and Peripheral Driver Library (PDL). For an introduction to PSoC 6 MCU, see AN210781 - Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity. If you are new to PSoC Creator, ModusToolbox, or PDL, see the Related Resources section for links to some of the available resources.

Note: Use PSoC Creator version 4.2 or higher for PSoC 6 MCU-based designs.

This document begins with a brief explanation of the PSoC 6 MCU interrupt architecture, with more details available in the PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual (TRM). To skip to an overview of writing firmware that uses interrupts, see Configuring Interrupts Using PDL or Configuring Interrupts Using PSoC Creator or Configuring Interrupts Using ModusToolbox sections respectively. Code examples that show how to use interrupts for various peripherals are listed in the Related Resources section.

The Debugging Tips section provides a few tips on finding and resolving common issues encountered while using interrupts. More complex topics are covered in Advanced Interrupt Topics.

2 PSoC 6 MCU Interrupt Architecture

PSoC 6 MCU contains two CPUs: CM4 and CM0+. Interrupt signals to each CPU are handled by the respective Nested Vectored Interrupt Controller (NVIC). The NVIC enables/disables 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 before a lower-priority ISR.

PSoC 6 MCU also supports a wakeup interrupt controller (WIC) and multiple synchronization blocks. The WIC block allows the CPU to wake up from Sleep or Deep Sleep low-power modes using interrupts. The WIC block remains active while the NVIC, processor core, and other device peripherals 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. Each CPU has independent WIC settings.

Natively, CM4 supports up to 240 interrupts, while CM0+ supports 32 interrupts. The number of CPU interrupts available to the user varies depending on the device, see Table 1.

CM4 supports configurable interrupt priority from 0 to 7. CM0+ supports priority from 0 to 3.

There are up to 168 interrupt sources (also referred to as system interrupts) in a PSoC 6 MCU device. System interrupts can trigger either or both CPUs.

The WIC block supports up to 41 interrupts that can wake up a CPU from Deep Sleep power mode. Table 2 lists the interrupt sources that can wake a CPU from Deep Sleep.

One or more system interrupts can be selected as the source for the CPU NMI, see Table 1.

Table 1. Interrupt features in PSoC 6 MCU

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CY8C62x6/7, CY8C63xx</th>
<th>CY8C62x8/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of system interrupts</td>
<td>147</td>
<td>168</td>
</tr>
<tr>
<td>Number of Deep Sleep-capable system interrupt sources</td>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>Number of CM0+ interrupt vectors available</td>
<td>32 (8 Deep Sleep-capable)</td>
<td>8 hardware (Deep Sleep-capable) 8 software triggered</td>
</tr>
<tr>
<td>Number of system interrupts that can be connected to a CM0+ multiplexer/vector</td>
<td>1</td>
<td>All (168)</td>
</tr>
<tr>
<td>Number of CM4 interrupt vectors available</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>Number of system interrupts that can be connected to a CM4 multiplexer/vector</td>
<td>1 (1:1 mapping)</td>
<td>1 (1:1 mapping)</td>
</tr>
<tr>
<td>Number of system interrupts that can be connected to CM0+/CM4 NMI interrupt</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
2.1 CY8C62x6/7, CY8C63xx Interrupt Architecture

Figure 1. CY8C62x6/7, CY8C63xx Interrupt Architecture

For CM4, the 147 interrupt sources are directly mapped to its first 147 IRQ lines, i.e., INT source \( n \) is connected to IRQ \( n \), where \( n = 0 \) to 146. For CM0+, a 240:1 multiplexer is present in front of each of 32 IRQs and redirects any of the 147 interrupts to one of CM0+ IRQ lines. This enables any interrupt source to trigger any CM0+ IRQ.
2.2 CY8C62x8/A Interrupt Architecture

There are 168 interrupt sources in a CY8C62x8/A device. For CM4, the 168 interrupt sources are directly mapped to its first 168 IRQ lines. For CM0+, there are 16 interrupts, of which 8 are software-only interrupts and the remaining 8 are interrupts that can be triggered from a peripheral interrupt source. One or more system interrupts (168 total) can be assigned as the interrupt source for each of the 8 IRQ lines. This allows multiple interrupt sources to be connected to the same CPU interrupt simultaneously.

In CY8C62x8/A, the WIC block supports up to 39 interrupts that can wake up a CPU from Deep Sleep power mode. Table 2 lists the interrupt sources that can wake a CPU from Deep Sleep.

Note: When using Cypress software (PDL or PSoC Creator), certain software restrictions apply on the number of CPU interrupts available to user and interrupt priorities. See Configuring Interrupts Using PDL for details.

2.3 Types of Interrupts

There are two kinds of interrupt sources in PSoC 6 MCU:

- Fixed-function interrupt sources
  These are predefined interrupt sources from on-chip peripherals such as GPIO, TCPWM, SCB, and BLE Radio. Interrupts from fixed-function sources are generated from configurable events: for example, an interrupt on a rising edge signal on an input pin (GPIO), or an interrupt on a counter overflow (TCPWM).

- Universal Digital Block (UDB) interrupt sources
  UDBs consist of programmable logic devices (PLDs), datapaths, and flexible routing, which can be used to synthesize different digital functions such as Timer, PWM, UART, SPI and many more. 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).

For a complete list of interrupt sources in PSoC 6 MCU, see Appendix A.

---

1 UDB sources are not available in CY8C62x8/A.
2.3.1 Level and Pulse Interrupts
Both CM0+ and CM4 NVICs support level and pulse signals on IRQ lines. The classification of an interrupt as level or pulse is based on the interrupt source. A fixed-function interrupt is treated as level-sensitive. For the DSI sources, which include the UDB, the interrupt can be configured as either rising-edge-triggered or level-triggered. For more details on selecting the interrupt type, refer to the PSoC Creator Component datasheet or PDL API reference for the interrupt source.

For level interrupts, if the interrupt signal is still HIGH after completing the ISR, the interrupt is still pending and the ISR is executed again. Figure 3 illustrates the timing diagram for level-triggered interrupts, where the ISR is executed as long as the interrupt signal is HIGH.

For pulse interrupts, while the ISR is being executed by the CPU, one or more rising edges of the interrupt signal are logged as a single pending request. The pending interrupt is serviced again after the current ISR execution is complete. Figure 4 illustrates the timing diagram for pulse interrupts.

Figure 3. Level Interrupts
Figure 4. Pulse Interrupts

Note: The GPIO interrupt logic has additional circuitry to support interrupts on the rising edge, falling edge, and both edges. See the I/O System chapter in PSoC 6 MCU Architecture TRM for more information.

2.4 Interrupts and Power modes
PSoC 6 MCU has the following power modes: Active, Low-Power Active (LPACTIVE), Sleep, Low-Power Sleep (LPSLEEP), Deep Sleep, and Hibernate.

In Active and LPACTIVE modes, CPUs execute code; all memory blocks and peripherals are available. LPACTIVE is similar to Active mode, with performance reductions for lower power consumption; peripherals and their interrupts are still available.

In all other power modes, CPU clocks are turned off and CPUs are in Sleep or Deep Sleep mode.

All peripherals available in Active and LPACTIVE modes are also available in the Sleep and LPSLEEP modes. Any peripheral interrupt, masked to the CPU, wakes up the CPU to Active mode.

Only a subset of peripherals operate in Deep Sleep mode. Interrupts from these peripherals cause a CPU to wake up to Active mode. Table 2 lists the peripherals. Each CPU has a Wakeup Interrupt Controller (WIC) to wake up the CPU from its Deep Sleep mode. Deep Sleep wakeup functionality is supported only on the first 8 IRQs (0 to 7) on CM0+ and first 41 IRQs (0 to 40) on CM4 for PSoC 6 MCU and 39 IRQs (0 to 38) on CM4 for CY8C62x8/A.

During Hibernate mode, all peripherals and clocks are turned off and only certain sources like Low Power Comparator, RTC, a dedicated WAKEUP pin, or an XRES event can wake up the device. The wakeup action is a device reset instead of an interrupt to the CPU. For more details on device power modes, refer to AN219528 - PSoC 6 MCU Low-Power Modes and Power Reduction Techniques or PSoC 6 MCU Architecture TRM.

---

Footnote: This configuration is available only in PSoC Creator. ModusToolbox doesn’t have DSI implementation.
2.5 CPU Sleep and Wakeup

There are two instructions that can cause the CPU to enter its sleep modes: the “Wait-for-Interrupt” \([\text{WFI}()]\) and “Wait-for-Event” \([\text{WFE}()]\). When a WFI instruction is executed, the CPU enters Sleep or Deep Sleep (depending on the SLEEPDEEP bit of the SCR register) and wakes up on an interrupt request (with a higher priority than the current priority level) or on debug requests. The WFE instruction is similar to WFI but wakes up on the next interrupt or on events like Send Event (SEV instruction), external event, or debug signals. See AN219528 for more details on Sleep and Wakeup instructions.

Normally, when an ISR is done executing, CPU execution returns to where it was before the ISR. PSoC 6 MCU supports the “Sleep-on-Exit” feature where the CPU enters or returns to Sleep or Deep Sleep (a state similar to WFI) as soon as it completes ISR execution. As seen in Figure 5, when this feature is enabled, only one WFI instruction is needed to enter a sleep mode; the CPU returns to sleep after each ISR instead of the execution returning to main. The Sleep-on-Exit feature reduces the active cycles of the CPU and reduces the energy consumed by the stacking (PUSH to stack) and unstacking (POP from stack) of processes between interrupts. Nested interrupts are also supported when Sleep-on-Exit is enabled.

The Sleep-on-Exit feature is enabled by setting SLEEPONEXIT bit of the SCR register. There is also a PDL function available \(\text{Cy_SysPm_SleepOnExit}\); see Configuring Interrupts Using PDL for details.

Figure 5. Sleep-on-Exit Function
3 Interrupt Configuration

This section lists the steps needed to set up interrupts on a PSoC 6 MCU device, without going into details of the software used to do them. These steps are common to both CM0+ and CM4 unless specified otherwise, and must be done for each CPU separately.

1. Out of device reset, all interrupts are disabled and interrupt priorities are set to zero.
2. Configure the priority level of the required IRQ in the NVIC.
3. Configure the interrupt path.
   Choose which interrupt source is connected to the desired IRQ of the CPU. For CM0+, select the appropriate peripheral interrupt to be connected to the CPU. For CM4, this is not configurable. Interrupt source \( n \) is always connected to \( IRQn \).
4. Configure the interrupt source (peripheral) and enable its interrupt.
5. Configure the vector table with the address of the ISR (vector).
   The vector table stores the entry addresses for each exception handler; see Exception Vector Table in Interrupts chapter of PSoC 6 MCU Architecture TRM.
6. Optional: Clear pending interrupt states in the NVIC.
   If enabling a previously disabled interrupt, it is a good practice to clear the pending state of the NVIC before enabling the interrupt. This prevents any false trigger caused by previous interrupts that created a pending state.
7. Enable the interrupt in the NVIC.
8. Enable global interrupts. Interrupt configuration is complete.

An enabled interrupt is triggered when the hardware signal from the interrupt source is active and there is no higher priority interrupt that is executing. When this happens, CPU execution jumps to the location in its vector table that corresponds to the triggered interrupt. This location contains the address of the ISR associated with that interrupt.

The ISR executes the tasks required to handle the interrupt. Typically, the first thing an ISR does is clearing the interrupt source to avoid re-entering the ISR. When the ISR terminates, the CPU returns to the address it was executing before it was interrupted. The following sections describe the software tools available for performing the steps described.

3.1 Configuring Interrupts Using PDL

Peripheral Driver Library (PDL) is a software development kit (SDK) that enables firmware development for PSoC 6 MCU devices. PDL API function calls are used to configure, initialize, enable, and use a peripheral driver. One such driver is System Interrupts (SysInt). SysInt provides structures and functions to configure and enable interrupt functionality. PDL also supports the CMSIS-Core libraries which include NVIC functions used for interrupt configuration.

**Note:** CY8C62x8/A is only supported on PDL v3.1.x and later, which contains SysInt driver version 1.20. It is recommended to use SysInt v1.20 or newer for development with all PSoC 6 devices.

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

1. Configure the peripheral to generate the interrupt. For example, for a GPIO, configure the drive mode (pull up or pull down), interrupt signal generation on falling or rising edge, and unmask the interrupt. Refer to the PDL API reference documentation for your peripheral for this information.
2. Configure the interrupt using the structure provided by the SysInt API.

   The structure is defined in the PDL SysInt driver file `cy_sysint.h`:

   ```c
   typedef struct {
       IRQn_Type  intrSrc;  /**< Interrupt source */
       #if (CY_CPU_CORTEX_M0P)
       cy_en_intr_t  cm0pSrc;  /**< (CM0+ only) Maps cm0pSrc device interrupts to intrSrc */
       #endif
       uint32_t    intrPriority;  /**< Interrupt priority number (Refer to _NVIC_PRIO_BITS) */
   } cy_stc_sysint_t;
   ```
This structure is used to configure the following (see Figure 6 for a quick summary):

a. Interrupt Source (intrSrc)
   - These are the dedicated interrupt numbers as defined in the device header file (example: cy8c6247bzi_d44.h)
   - This selection depends on which CPU you want to assign the interrupt to.
     - For CM4, this number represents both the interrupt number of the source as well as the CPU IRQ number. Select the interrupt number of the peripheral interrupt you wish to route to the CPU. For example, to route Port 0 GPIO interrupt, assign a value of ioss_interrupts_gpio_0_IRQn (=0).
     - For CM0+, this number represents one of the 32 multiplexers available for routing an interrupt to CM0+. Because each multiplexer is connected to a dedicated CM0+ IRQ line, use this to select the target CM0+ IRQ number. For example, to use multiplexer #4 (CM0+ IRQ#4), use "NvicMux4_IRQn" (=4).

b. CM0+ interrupt number (cm0pSrc)
   - This parameter is applicable only for CM0+.
   - This represents the interrupt number of the source, which is to be routed to the multiplexer/CM0+ interrupt generator logic, selected using the intrSrc parameter. Select the interrupt number of the peripheral interrupt you wish to route to the CPU; for example, to route Port 0 GPIO interrupt, assign a value of "loss_interrupts_gpio_0_IRQn" (=0).

c. Interrupt priority (intrPriority)
   - Set the priority of the interrupt. For CM4, supported priorities are 0 to 7. For CM0+, supported priorities are 0 to 3.

Notes:
- On CM0+, some IRQs are reserved for use by software and not available to the user. See “Configuration Considerations” under SysInt driver in PDL API reference documentation for the list of reserved IRQs.
- On CM0+, the interrupt priority 0 is reserved for system calls.
Figure 6. SysInt PDL Structure Parameters (highlighted in red) Used for Interrupt Configuration.

A sample configured path is highlighted in blue.

3. Call `Cy_SysInt_Init(&SysInt_SW_cfg_1, ISR_1_handler)`.
   Here, `SysInt_SW_cfg_1` is the name of the configured structure, `ISR_1_handler` is the name of the interrupt handler that executes when the interrupt triggers. This function applies the routing and priority configuration of the interrupt but does not enable it.

4. Call `NVIC_ClearPendingIRQ(SysInt_SW_cfg_1.intrSrc)` to clear any pending interrupts.
5. Call NVIC_EnableIRQ(SysInt_SW_cfg_1.intrSrc) to enable the interrupt.

6. Call the __enable_irq() function to enable global interrupts. This is safe to perform as the first step, as individual CPU interrupts have not been enabled yet. You can also perform this later but interrupts are disabled at startup unless this is called.

In addition to the PDL SysInt driver, the system power modes (SysPm) driver API enables the Sleep-on-Exit feature. If Sleep or Deep Sleep mode is used in the application along with interrupts, this feature enables the firmware to keep the system in a sleep mode almost all the time, only wake up to execute the interrupt and then immediately go back to the same sleep mode. The program does not return to the main function and stays either in the interrupt handler or in the same sleep state unless the Sleep-on-Exit feature is disabled again.

```c
Cy_SysPm_SleepOnExit(true);
```

### 3.2 Configuring Interrupts Using PSoC Creator

PSoC Creator provides a graphical interface for routing signals from peripherals to a CPU IRQ line. PSoC Creator provides an Interrupt (SysInt) Component. This component is a UI element on top of the SysInt PDL driver discussed in the previous section. Based on the configuration in the Component, PSoC Creator generates code to initialize peripherals, route interrupts, and populate the interrupt configuration structure. This reduces the amount of code you must write when setting up interrupts.

The following section shows steps to use PSoC Creator to configure an interrupt. See the Related Resources section for code examples.

#### 3.2.1 Using the Schematic (TopDesign)

Drag and drop a Component from the Component Catalog onto the TopDesign. Use TopDesign to place and configure peripherals that provide a source of interrupt. Consult the Component datasheet for information on the particular peripheral’s interrupt configuration. Some peripherals provide an interrupt terminal (e.g., TCPWM). Place an instance of the SysInt Component and connect it to the interrupt terminal of the peripheral.

![Figure 7. TopDesign with Interrupt Component](image)

Some peripherals do not have an external interrupt terminal (e.g., SCB has interrupts built-in) or may have an option to expose it (e.g., UART).
The Interrupt Component has two configurable options as seen in Figure 8:

3.2.2 Deep Sleep Capable
Enable this checkbox if you want the interrupt to be assigned to a CPU IRQ line that is Deep-Sleep-capable. You must ensure that the interrupt source is also active and capable of providing the interrupt signal during Deep Sleep, failing which PSoC Creator throws an error when the project is built. Note that this option is significant only in case the interrupt is assigned to CM0+ which has 8 (IRQ 0-7) Deep Sleep slots to route to. The checkbox is provided only for guidance in automatically assigning an IRQ for the interrupt and can be overridden by manual assignment from the CyDWR window. For CM4, if the interrupt source is Deep-Sleep-capable (IRQ 0-40), disabling the checkbox has no effect on the Deep Sleep functionality of the interrupt.

3.2.3 Interrupt Type
There are three options available for Interrupt type in the Interrupt Component configuration: Auto-Select Trigger, Rising-Edge Triggered, and Level Triggered. The selection of a particular option depends on the interrupt source (fixed-function or UDB/DSI) and the application requirements. In most cases, leave the option to Auto-Select to let PSoC Creator derive the interrupt type from the nature of the interrupt source.

Choose only level-triggered for Fixed-function interrupt sources. Choose Level-triggered or Rising-Edge for UDB sources.

3.3 Using the Design-Wide Resource Window (CyDWR)
The design-wide resources window (.cydwr file) of the PSoC Creator project has an Interrupts tab. This tab lists the instance names of all interrupts used in the TopDesign schematic along with their interrupt numbers.

Each interrupt can be allocated to either CM0+ or CM4 or both the CPUs using the ‘ARM CMx Enable’ checkbox. Unless specified otherwise, all interrupts are assigned to CM4 by default. Though possible, it is not advised to assign an interrupt to both CPUs unless an application requires it. A warning icon appears in the instance name column if both CPUs handle the same interrupt. A tooltip description of the warning can be viewed on hovering the mouse pointer over the icon.

For CM0+, also assign a CPU IRQ line using the ‘ARM CM0+ Vector’ column. Note that some CM0+ IQRs are reserved. PSoC Creator does not allow assigning to these IQRs and will display a warning if done so. There is no option to select the vector for CM4 as these are directly mapped to the corresponding interrupt numbers.

Once assigned to the CPU, assign the priority using the corresponding priority field. CM0+ priority is in the range of 1 to 3. (priority 0 is reserved for system calls). CM4 priority is in the range 0 to 7. For both CPUs, priority 0 corresponds to the highest priority and higher numbers denote lower priorities.

A Deep-Sleep-capable interrupt source or IRQ is indicated using an icon. An info icon appears if a non-Deep-Sleep-capable interrupt is assigned to a Deep-Sleep-capable IRQ line. A build is required to refresh the interrupt numbers and icons.
3.3.1 Using PSoC Creator Generated Code and PDL

Building the project generates code for use in the application. The Pins and Interrupts folder contains files with code generated using the information entered in the Interrupts tab in CyDWR.

cyfitter_sysint.h contains macros with information on interrupt number, its CPU assignment, and priority.

cyfitter_sysint_cfg.c/h declares and pre-populates instances of SysInt PDL configuration structure using the CyDWR information.

The configuration structure for each interrupt is conditionally defined based on the CPU assignment.

The steps to enable interrupts in firmware are similar to the ones listed in the PDL section but fewer in number.

1. Call the __enable_irq() API to enable global interrupts.
2. Call Cy_SysInt_Init(&SysInt_1_cfg, ISR_1_handler)
   - Where SysInt_1_cfg is the name of the auto-generated structure from the cyfitter_sysint_cfg.c file. ISR_1_handler is the name of the interrupt handler that executes when the interrupt triggers. The handler function can reside in the respective CPU's main.c to which the interrupt is assigned. If the handler exists outside main.c, that file must be compiled and linked into the executable for the CPU that handles the ISR.
   - This step configures the interrupt (routing, priority, and interrupt handler assignment) but does not enable it.
3. Call NVIC_ClearPendingIRQ(SysInt_1_cfg.intrSrc) to clear any pending interrupts.
4. Call NVIC_EnableIRQ(SysInt_1_cfg.intrSrc) to enable the interrupt.

You can use PSoC Creator to generate code, and import that into a preferred IDE. AN219434 -- Importing PSoC Creator Code into an IDE for a PSoC 6 MCU Project describes how to do that. It is recommended that you use PSoC Creator to set up and configure interrupts in PSoC 6 MCU, export the project to the IDE you prefer and continue developing firmware code with the IDE preferred.
3.4 Configuring Interrupts Using ModusToolbox

Interrupts are configured in ModusToolbox similar to the steps described in Section 3.1: Configuring Interrupts Using PDL.

ModusToolbox provides a graphical interface for configuring peripherals and their interrupt parameters. This simplifies the step 1 mentioned in section 3.1.

Figure 11 shows the configuration of a GPIO pin to generate interrupt on a falling edge.

![ModusToolbox Peripheral Configuration](image)

Based on the configuration, ModusToolbox generates the 'C' code to achieve the desired configuration. The code generated can be viewed in the Code Preview pane; it is added to relevant cycfg_XXX.c/h files found under <ApplicationName>_mainapp > GeneratedSource folder in the ModusToolbox project workspace window. The generated code includes macro(s) defining the interrupt source numbers and any peripheral configuration that is necessary to set up and enable the interrupt source. This simplifies the process of searching for the dedicated interrupt numbers in the device header file. The user application only needs to enable the interrupt vector on the CPU and assign an interrupt handler function as described in Section 3.1. Refer to the CE219521 (ModusToolbox).zip file in CE219521 – PSoC 6 MCU - GPIO Interrupt page for a code example that demonstrates how to use interrupts using ModusToolbox.

4 Debugging Tips

This section provides tips on trouble-shooting and debugging interrupts. The following are some of the frequently encountered cases:

a. Interrupt is not triggered
   - Ensure that the interrupt source and global interrupt are enabled.
   - Ensure that the interrupt vector is initialized with correct ISR.
   - Check whether other interrupt sources are triggered repeatedly, thus consuming the entire CPU bandwidth.
b. **Interrupt is triggered repeatedly**

This can happen in multiple cases: Insert breakpoints in the ISR and elsewhere in the program which is expected to execute repeatedly (for example, the super-loop in the main function). If the program is not entering the main function, interrupt is triggered repeatedly.

- The interrupt line from a fixed-function source
  Resolution: Clear the interrupt source to resolve this behavior.
- A digital output from the Component (not the interrupt line) is connected to a SysInt Component configured to level type in PSoC Creator. Resolution: Configure the Interrupt Component to rising edge to get one interrupt per rising edge.

b. **Execution of the ISR is taking longer than expected**

This can happen if other high-priority interrupts are triggered during the execution of the ISR.

Resolution: Increase the priority of the interrupt relative to other interrupt sources.

The PSoC 6 BLE Pioneer Kit has the KitProg2 onboard programmer/debugger.

The CY8CPROTO-062-4343W PSoC 6 Wi-Fi BT Prototyping Kit has the KitProg3 onboard programmer/debugger. This kit is supported only on ModusToolbox.

PSoC Creator supports debugging one CPU at a time (either CM0+ or CM4). ModusToolbox IDE supports debugging of both CPUs simultaneously.

The debug mode is useful for checking interrupts as given below:

- To check if an interrupt is executing, add a breakpoint at one of the instructions in the ISR.
- Use Breakpoint Hit Count/breakpoint condition to detect the number of times an interrupt is triggered. This is particularly useful to check if the interrupt signal has glitches causing the interrupt to trigger multiple times. To see Breakpoint Hit Count, right-click on the breakpoint, select Hit Count and observe current hit count.
- Use the Call Stack window of the debugger to check program flow to learn when a particular ISR is executed. You can also use it to check if a high-priority interrupt occurred during the execution of a low-priority ISR As an alternative to the debugger, you can also use a pin to do the following:
  - Check if the CPU is entering the ISR.
  - Measure the ISR execution time. This can be done, for example, by asserting the pin in the beginning of the ISR and de-asserting the pin before returning from the ISR. The time for which the pin is HIGH can be measured using an oscilloscope to give the duration of ISR execution.
5 Advanced Interrupt Topics

5.1 Exceptions

Exceptions are the 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.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Exception Number</th>
<th>Exception Priority</th>
<th>CPUs Supporting the Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>1</td>
<td>-3</td>
<td>Both CM0+ and CM4</td>
<td>This exception can occur due to multiple reasons, such as power-on-reset (POR), external reset signal on XRES pin, or watchdog reset. Cortex-M4 execution begins only after CM0+ de-asserts the M4 reset. The reset exception address in the SRAM vector table will never be used because the device comes out of reset with the flash vector table selected. The register configuration to select the SRAM vector table can be done only as part of the startup code in flash after the reset is de-asserted.</td>
</tr>
<tr>
<td>Nonmaskable Interrupt (NMI)</td>
<td>2</td>
<td>-2</td>
<td>Both CM0+ and CM4</td>
<td>Both CPUs have their own NMI exception. NMI can be triggered by the following: Any of the interrupt sources, by setting NMIPENDSET bit or using System Calls. PSoc 6 BLE supports routing of only one system interrupt as the source for NMI. CY8C62x8/A supports four system interrupt sources for NMI. The four selected interrupt sources are logically ORed into a single CPU NMI input. NMI exception handler address is automatically initialized to the system call API located in SROM (at 0x00000000D by the boot code. The value should be retained by the user during vector table relocations; otherwise, no system call will be executed.</td>
</tr>
<tr>
<td>HardFault Exception</td>
<td>3</td>
<td>-1</td>
<td>Both CM0+ and CM4</td>
<td>HardFault exception occurs when executing an undefined instruction or accessing an invalid memory addresses.</td>
</tr>
<tr>
<td>SVC call Exception</td>
<td>11</td>
<td>Configurable</td>
<td>Both CM0+ and CM4</td>
<td>Supervisor Call (SVC call) is an always-enabled exception caused when the CPU executes the SVC instruction as part of the application code. The SVC instruction enables the application to issue a supervisor call that requires privileged access to the system.</td>
</tr>
<tr>
<td>PendSV</td>
<td>14</td>
<td>Configurable</td>
<td>Both CM0+ and CM4</td>
<td>PendSV exception is normally software-generated. PendSV is another supervisor call related exception similar to SVC call.</td>
</tr>
<tr>
<td>SysTick Exception</td>
<td>15</td>
<td>Configurable</td>
<td>Both CM0+ and CM4</td>
<td>SysTick is a 24-bit decrementing counter that generates periodic interrupts.</td>
</tr>
<tr>
<td>Memory Management Fault Exception</td>
<td>4</td>
<td>Configurable</td>
<td>Only CM4</td>
<td>A memory management fault is an exception that occurs because of a memory protection-related fault.</td>
</tr>
<tr>
<td>Bus Fault Exception</td>
<td>5</td>
<td>Configurable</td>
<td>Only CM4</td>
<td>A Bus Fault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction.</td>
</tr>
<tr>
<td>Usage Fault Exception</td>
<td>6</td>
<td>Configurable</td>
<td>Only CM4</td>
<td>A Usage Fault is an exception that occurs because of a fault related to instruction execution.</td>
</tr>
</tbody>
</table>

Notes:

- Exception priority that are configurable can be configured from priority 0-3 for CM0+ and 0-7 for CM4.
- Interrupts are also part of exceptions. Interrupt vector number 0 (i.e., IRQ 0) corresponds to the exception number 16, and so on.
5.2 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. CM0+ has a latency of 15 clock cycles (worst case); CM4 has a latency of 12 clock cycles (worst case). Some peripherals generate additional cycles due to synchronization circuit between the peripherals and CPUs. Table 3 provides the number of CPU clock cycle delays for various peripherals in PSoC 6 MCU.

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Synchronization Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCPWM, DMA, USB, I2S, PDM – PCM, CDS</td>
<td>0 clock cycles</td>
</tr>
<tr>
<td>SCB, GPIO, LPCmp, RTC, WDT, SMIF, BLE</td>
<td>2 clock cycles</td>
</tr>
</tbody>
</table>

When both CPUs are in Sleep/Deep Sleep power mode, there is a need for additional two clock cycles required for synchronization.

Context switching affects the latency and involves the following steps:

1. Current instruction execution is completed.
2. The processor pushes the current Program Counter (PC), Link Register (LR), Program Status Register (PSR), and some of the general-purpose registers (Program and Status Register (PSR), Return Address, Link Register (LR or R14), R12, R3, R2, R1, and R0) to the stack.
3. The processor reads the vector address from the NVIC and updates it to the PC.
4. The processor updates the NVIC registers.

Thus, the latency varies depending on the current instruction being executed. To make the process efficient, both CM0+ and CM4 processors implement the following two schemes:

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 nested interrupts, as seen in the following section, and for reducing the latency of low-priority interrupts.

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 entering the lower priority ISR and pushing the register values to the stack.
5.3 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. CPU 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.

5.4 Code Optimization

An important performance requirement 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 or printing long strings to a UART terminal takes long time to execute, thus blocking the execution of other low-priority interrupts. The recommended technique is to move non-critical 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 the interrupts. In applications with multiple interrupts, give a higher priority to more time-critical interrupts.

Although AN89610 – PSoC 4 and PSoC 5LP ARM Cortex Code Optimization targets a different CPU architecture, it is a useful reference for general compiler topics.

6 Related Resources

For a complete and updated list of PSoC 6 MCU code examples, visit our code examples web page. For more PSoC 6 MCU-related documents, please visit our PSoC 6 MCU product web page.

Table 4. Documents Related to PSoC 6 MCU Features

<table>
<thead>
<tr>
<th>Document</th>
<th>Document Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Notes</strong></td>
<td></td>
</tr>
<tr>
<td>AN221774</td>
<td>Getting Started with PSoC 6 MCU</td>
</tr>
<tr>
<td>AN210781</td>
<td>Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity</td>
</tr>
<tr>
<td>AN215656</td>
<td>PSoC 6 MCU Dual-CPU System Design</td>
</tr>
<tr>
<td>AN219434</td>
<td>Importing PSoC Creator Code into an IDE for a PSoC 6 MCU Project</td>
</tr>
<tr>
<td><strong>Code Examples</strong></td>
<td></td>
</tr>
<tr>
<td>CE221773</td>
<td>PSoC 6 MCU Hello World Example</td>
</tr>
<tr>
<td>CE219521</td>
<td>PSoC 6 MCU GPIO Interrupt</td>
</tr>
<tr>
<td>CE216795</td>
<td>PSoC 6 MCU Dual-Core Basics</td>
</tr>
<tr>
<td>CE218129</td>
<td>PSoC 6 MCU Wake up from Hibernate Using Low-Power Comparator</td>
</tr>
<tr>
<td>CE218542</td>
<td>PSoC 6 Custom Tick Timer Using RTC Alarm Interrupt</td>
</tr>
<tr>
<td>CE219339</td>
<td>PSoC 6 MCU MCWDT and RTC Interrupts (Dual Core)</td>
</tr>
<tr>
<td>CE220061</td>
<td>PSoC 6 MCU Multi-Counter Watchdog Interrupts</td>
</tr>
<tr>
<td>CE220169</td>
<td>PSoC 6 MCU Periodic Interrupt Using TCPWM</td>
</tr>
<tr>
<td><strong>Software/IDE</strong></td>
<td></td>
</tr>
<tr>
<td>PSoC Creator</td>
<td>PSoC Creator User Guide</td>
</tr>
<tr>
<td>ModusToolbox</td>
<td>ModusToolbox User Guide</td>
</tr>
<tr>
<td>Peripheral Driver Library</td>
<td>PDL API Reference (installed with PDL documentation)</td>
</tr>
</tbody>
</table>
### Appendix A. Interrupt Sources in PSoC 6 MCU

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Details</th>
<th>Interrupt Number (CY8C62x6/7, CY8C63xx)</th>
<th>Interrupt Number (CY8C62x8/A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOs</td>
<td>Each port consists of a maximum 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 6 MCU 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 modes. There is a GPIO All Ports interrupt that allows combining all port interrupts into a single vector. Firmware must identify the port that caused the interrupt. There is a GPIO Supply Detect Interrupt that can be used to detect the supply ramping up or ramping down.</td>
<td>0 to 14</td>
<td>0 to 14</td>
</tr>
<tr>
<td>LPComp</td>
<td>Like GPIOs, an interrupt can be triggered on the rising edge, falling edge, or both edges of the comparator output signal. LPComp can also wake the device from Sleep, Deep Sleep, and Hibernate power modes.</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>SCB (Deep Sleep)</td>
<td>SCB interrupt that can wake up CPU/system from Deep Sleep</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Multi Counter Watchdog Timer (MCWDT) Interrupt</td>
<td>MCWDT configures two 16-bit counters and one 32-bit counter capable of generating periodic interrupts. MCWDT can wake the CPU from Deep Sleep power mode.</td>
<td>19, 20</td>
<td>19, 20</td>
</tr>
<tr>
<td>Backup Domain Interrupt</td>
<td>Backup domain interrupt includes the RTC ALARM1, RTC ALARM2, and RTC century overflow interrupt. This can be used to wake the CPU from Sleep, Deep Sleep, and Hibernate power modes.</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Other Combined Interrupts for SRSS</td>
<td>The following cases generate this interrupt: WDT interrupt, Low Voltage Detect (LVD) interrupt, and clock calibration interrupt. WDT interrupt occurs when the watchdog counter value matches the preset Counter Match value. Missing two interrupts will cause a watchdog reset. Low-voltage detect (LVD) interrupt when the device supply voltage drops below a threshold. Clock calibration interrupt is triggered when clock calibration is complete. These are capable of waking the CPU from Deep Sleep.</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>CTBm Interrupt (all CTBms)</td>
<td>This block provides continuous time analog functionality. It generates interrupts on event such as comparator triggers.</td>
<td>23</td>
<td>-</td>
</tr>
<tr>
<td>Bluetooth Radio Interrupt</td>
<td>Bluetooth sub-system interrupt</td>
<td>24</td>
<td>-</td>
</tr>
<tr>
<td>IPC Interrupt</td>
<td>IPC interrupts could be triggered when an IPC release or notify event occurs.</td>
<td>25 to 40</td>
<td>23 to 38</td>
</tr>
<tr>
<td>SCB</td>
<td>PSoC 6 MCU supports SCBs which can be configured as SPI, I²C or UART. One SCB interrupt amongst the 8 SCBs is Deep-Sleep-capable. The following events generate an interrupt in a SCB. TX FIFO has less entries than specified. TX FIFO is not full/full/overflow/underflow. RX FIFO has more entries than the value specified, RX FIFO is full/not empty. SPI: SPI interrupts are triggered when SPI master transfer done, SPI Bus Error, SPI slave deselected after any EZSPI transfer occurred. I²C: I²C master lost arbitration, received NACK, received ACK, sent STOP, I²C bus error, I²C slave lost arbitration, received NACK, received ACK, received STOP, received START, address matched. UART Interrupts: TX received a NACK in SmartCard mode, TX done, Arbitration lost, frame error in received data frame, parity error in received data frame, LIN baud rate detection is completed, LIN break detection is successful.</td>
<td>41 to 48</td>
<td>39 to 50</td>
</tr>
<tr>
<td>CSD (CapSense®) Interrupt</td>
<td>CSD, used for touch applications, generates an interrupt when the sensor scan is complete.</td>
<td>49</td>
<td>51</td>
</tr>
<tr>
<td>CPUSS DMAC, Channel #0 - 3</td>
<td>DMA Controller (DMAC) interrupts on DMA events like transfer completion, bus errors, address misalignments, current pointer being NULL, active channel disabled and descriptor error.</td>
<td>-</td>
<td>52 to 55</td>
</tr>
<tr>
<td>DMA Interrupt</td>
<td>DMA interrupt can be generated when the data transfer is completed.</td>
<td>50 to 81</td>
<td>56 to 113</td>
</tr>
</tbody>
</table>
## PSoC 6 MCU Interrupts

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Details</th>
<th>Interrupt Number (CY8C62x6/7, CY8C63xx)</th>
<th>Interrupt Number (CY8C62x8/A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUSS Fault Structure Interrupt #0</td>
<td>This interrupt occurs when there is a protection unit access violation.</td>
<td>82, 83</td>
<td>114,115</td>
</tr>
<tr>
<td>CRYPTO Accelerator Interrupt</td>
<td>Crypto Interrupt is generated in the following cases: a FIFO event is activated, FIFO overflows, true random number generator is initialized, true random number generator has generated a data value of the specified bit size, pseudo random number generator has generated a data value, instruction decoder encounters an instruction with a non-defined operation code, instruction decoder encounters an instruction with a non-defined condition code, when a AHB-Lite bus error is observed, true random number generator monitor adaptive proportion test detects a repetition of a specific bit value, true random number generator monitor adaptive proportion test detects a disproportionate occurrence of a specific bit value.</td>
<td>84</td>
<td>116</td>
</tr>
<tr>
<td>FLASH Macro Interrupt</td>
<td>Flash controller has a timer that generates interrupts.</td>
<td>85</td>
<td>117</td>
</tr>
<tr>
<td>CM0+ CTI #0</td>
<td>CTI triggers are used to communicate events between debug components.</td>
<td>86, 87</td>
<td>119,120</td>
</tr>
<tr>
<td>CM4 CTI #0</td>
<td>CTI triggers are used to communicate events between debug components.</td>
<td>88, 89</td>
<td>137,138</td>
</tr>
<tr>
<td>TCPWM</td>
<td>The TCPWM block can be configured to work as a 16- or 32-bit timer, counter, or PWM. It can generate interrupts on terminal count, input capture signal, or a compare true event.</td>
<td>90 to 121</td>
<td>139 to 154</td>
</tr>
<tr>
<td>UDB Interrupt #0</td>
<td>Any digital signal generated in a UDB can trigger an interrupt. Signals are routed to the interrupt controller through the routing fabric known as Digital System Interconnect (DSI).</td>
<td>122 to 137</td>
<td>-</td>
</tr>
<tr>
<td>I2S Audio Interrupt</td>
<td>Interrupt can be generated in the following cases. Less entries in the TX FIFO than the value specified, TX FIFO is not full, TX FIFO is empty, attempt to write to a full TX FIFO, attempt to read from an empty TX FIFO, triggers when the Tx watchdog event occurs, more entries in the RX FIFO than the value specified, RX FIFO is not empty, RX FIFO is full, attempt to write to a full RX FIFO, attempt to read from an empty RX FIFO, triggers when the Rx watchdog event occurs.</td>
<td>139</td>
<td>156</td>
</tr>
<tr>
<td>PDM/PCM Audio interrupt</td>
<td>More entries in the RX FIFO than the value specified, RX FIFO is not empty, attempt to write to a full RX FIFO, attempt to read from an empty RX FIFO</td>
<td>140</td>
<td>157</td>
</tr>
<tr>
<td>Energy Profiler Interrupt</td>
<td>This interrupt occurs on a profiling counter overflow.</td>
<td>141</td>
<td>159</td>
</tr>
<tr>
<td>Serial Memory Interface Interrupt</td>
<td>This interrupt is activated when TX data FIFO is activated, RX data FIFO is activated, alignment error, FIFO overflow.</td>
<td>142</td>
<td>160</td>
</tr>
<tr>
<td>USB Interrupt</td>
<td>The USB block has a predefined set of 13 interrupt trigger events that can be mapped to either one of the three interrupts. Events such as USB Start of Frame (SOF), USB bus reset, data endpoint events, control endpoint events, Arbiter Interrupt Event, and Link Power Management (LPM) event generate interrupts.</td>
<td>143-145</td>
<td>161-163</td>
</tr>
<tr>
<td>Consolidated Interrupt for all DACs</td>
<td>Interrupt can be generated when DAC buffer is empty. This interrupt can be used by the CPU to transfer the next value to the DAC.</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>SDIO wakeup Interrupt for SDHC</td>
<td>SDIO wakeup interrupt triggered on events such as card insertion, removal, and SDIO card interrupt. This doesn't wakeup the system from Deep Sleep.</td>
<td>-</td>
<td>164</td>
</tr>
<tr>
<td>Consolidated Interrupt for SDHC</td>
<td>Consolidated interrupt on all other normal / error events related to SDHC</td>
<td>-</td>
<td>165</td>
</tr>
<tr>
<td>EEMC Wakeup Interrupt for mxsadhc, not used</td>
<td>EEMC wakeuup interrupt for SDHC block (reserved)</td>
<td>-</td>
<td>166</td>
</tr>
<tr>
<td>Consolidated Interrupt for SDHC</td>
<td>Consolidated interrupt (reserved)</td>
<td>-</td>
<td>167</td>
</tr>
</tbody>
</table>
Document History

Document Title: AN217666 – PSoC 6 MCU Interrupts
Document Number: 002-17666

<table>
<thead>
<tr>
<th>Revision</th>
<th>ECN</th>
<th>Orig. of Change</th>
<th>Submission Date</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>5839225</td>
<td>ARVI/JSLN</td>
<td>09/15/2017</td>
<td>New application note</td>
</tr>
<tr>
<td>*A</td>
<td>6378913</td>
<td>ARVI/JSLN</td>
<td>11/08/2018</td>
<td>Updated for CY8C62x8/A and ModusToolbox</td>
</tr>
<tr>
<td>*B</td>
<td>6493502</td>
<td>ARVI</td>
<td>02/25/2019</td>
<td>Update section “Configuring Interrupts Using ModusToolbox”</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Products</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm® Cortex® Microcontrollers</td>
<td>cypress.com/arm</td>
</tr>
<tr>
<td>Automotive</td>
<td>cypress.com/automotive</td>
</tr>
<tr>
<td>Clocks &amp; Buffers</td>
<td>cypress.com/clocks</td>
</tr>
<tr>
<td>Interface</td>
<td>cypress.com/interface</td>
</tr>
<tr>
<td>Internet of Things</td>
<td>cypress.com/iot</td>
</tr>
<tr>
<td>Memory</td>
<td>cypress.com/memory</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>cypress.com/mcu</td>
</tr>
<tr>
<td>PSoC</td>
<td>cypress.com/psoc</td>
</tr>
<tr>
<td>Power Management ICs</td>
<td>cypress.com/pmic</td>
</tr>
<tr>
<td>Touch Sensing</td>
<td>cypress.com/touch</td>
</tr>
<tr>
<td>USB Controllers</td>
<td>cypress.com/usb</td>
</tr>
<tr>
<td>Wireless Connectivity</td>
<td>cypress.com/wireless</td>
</tr>
</tbody>
</table>

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

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

© Cypress Semiconductor Corporation, 2017-2019. This document is the property of Cypress Semiconductor Corporation and its subsidiaries (“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, non-transferable 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, CYRESS 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. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress shall have no liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. CYPRESS DOES NOT REPRESENT, WARRANT, OR GUARANTEE THAT CYPRESS PRODUCTS, OR SYSTEMS CREATED USING CYRESS PRODUCTS, WILL BE FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (collectively, “Security Breach”). Cypress disclaims any liability relating to any Security Breach, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any Security Breach. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. 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. “High-Risk Device” means any device or system whose failure to cause personal injury, death, or property damage. Examples of High-Risk Devices are weapons, nuclear installations, surgical implants, and other medical devices. “Critical Component” means any component of a High-Risk Device whose failure to perform can be reasonably expected to cause, directly or indirectly, the failure of the High-Risk Device, 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 any use of a Cypress product as a Critical Component in a High-Risk Device. You shall indemnify and hold Cypress, its directors, officers, employees, agents, affiliates, distributors, and assigns harmless from and against all claims, costs, damages, and expenses, arising out of any claim, including claims for product liability, personal injury or death, or property damage arising from any use of a Cypress product as a Critical Component in a High-Risk Device. Cypress products are not intended or authorized for use as a Critical Component in any High-Risk Device except to the limited extent that (i) Cypress’s published data sheet for the product explicitly states Cypress has qualified the product for use in a specific High-Risk Device, or (ii) Cypress has given you advance written authorization to use the product as a Critical Component in the specific High-Risk Device and you have signed a separate indemnification agreement.

Cypress, the Cypress logo, Spanion, the Spanion 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.