AN90833

PSoC® 1 Interrupts

Author: Rajiv Badiger, Anshul Gulati
Associated Project: Yes
Associated Part Family: All PSoC® 1 families
Software Version: PSoC Designer™ 5.4

AN90833 introduces you to the PSoC® 1 interrupt architecture and interrupt sources. This document also includes sections on interrupt priority, interrupt latency, and several recommendations on writing efficient and defect-free interrupt routines.

Contents

Introduction 1
PSoC 1 Interrupt Architecture 1
  PSoC 1 Interrupt Sources 1
  Interrupt Controller 2
  PSoC 1 Interrupt Priority 3
Interrupt Support in PSoC Designer 10
Interrupt Latency 12
Project - Timer_Interrupt 14
Tips and Tricks 18
  Optimizing the Interrupt Code 18
  Multi-byte Variable Usage 18
  Nested Interrupts 18
  Conditional Loop in ISR 19
Summary 19
Related Application Notes 19

Introduction

Interrupts are an important part of any embedded application because they free the CPU from continuously polling the occurrence of a specific event. Instead, interrupts notify the CPU only when that event occurs. In a system-on-chip (SoC) architecture, such as PSoC® 1, interrupts are frequently used to communicate the status of on-chip peripherals to the CPU.

AN90833 introduces you to the PSoC 1 interrupt architecture and explains how interrupt service routines (ISRs) are implemented in PSoC Designer™, the integrated design environment (IDE) for PSoC 1. An example project is also provided with this application note.

PSoC 1 Interrupt Architecture

This section gives an overview of the PSoC 1 interrupt architecture.

Figure 1. Basic PSoC 1 Interrupt Architecture

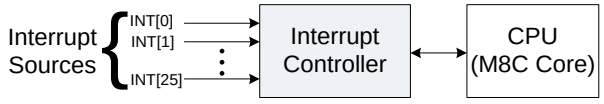


Figure 1 shows a simplified view of the PSoC 1 interrupt architecture. PSoC 1 can have up to 26 interrupt sources. Each one is assigned a fixed priority and fixed interrupt vector address. The interrupt controller acts as the interface between the interrupt lines and the CPU. The controller sends the interrupt vector address of an interrupt line to the CPU along with the interrupt request signal.

PSoC 1 Interrupt Sources

Almost every functional block in PSoC 1 has an interrupt associated with it. Interrupts are available for the following:

Reset

This is the highest-priority interrupt, and is caused by the following events:

- A logic HIGH signal on the XRES pin
- A Watchdog timer overflow event – refer to the application note AN32200 – PSoC® 1 Clocks and Global Resources for details on watch dog timer.
- A drop in the V_{DD} below the power-on-reset (POR) threshold. POR levels are configured based on the V_{DD} setting. See the DC POR Specifications section of PSoC 1 Device Datasheet for more details.

Supply Voltage (LVD)

The low-voltage detect (LVD) circuitry in PSoC 1 continuously monitors the V_{DD} of the device. When it drops below the threshold, it causes an interrupt to the CPU. The LVD interrupt is disabled by default. When enabled, the default instruction executed on an LVD interrupt is the “halt” instruction that halts the CPU. You can change this default “halt” instruction to execute your own interrupt handler. You can configure the LVD thresholds in PSoC Designer with the options in the Global Resources section. For details on LVD, refer to the LVD Specifications section of PSoC 1 Device Datasheet and the application note AN32200 – PSoC® 1 Clocks and Global Resources.

Analog Column

PSoC 1 has many analog blocks organized in columns. A column consists of one to three analog blocks. At a time, one block in an analog column can generate an interrupt through its comparator output.

Digital Block

Each digital block of PSoC 1 can generate an interrupt. Depending on the type of function, the interrupt type may vary. For example, a counter can generate an interrupt either on compare true or terminal count; a timer on capture or terminal count; a UART on events such as the TX buffer empty, TX complete, or RX buffer full. See the respective User Module Data Sheets for more information on interrupts associated with specific functions.

VC3 Clock

VC3 is a variable clock that can take its input from VC1 or VC2, SysClk or SysClk*2, and can have a divider of 1 to 256. This clock can trigger an interrupt on every period, which can be used for implementing a timer when all the digital blocks have been used.

GPIO

Each I/O of PSoC 1 can generate an interrupt. However, all GPIOs share a common interrupt vector. You can configure each pin to interrupt on the rising edge, falling edge, or a change from the last read. For more information on GPIOs, refer the application note AN2094 – PSoC® 1 Getting Started with GPIO.

I^2C

PSoC 1 has a maximum of two hardware I^2C blocks. Each block can generate interrupts on the following events:

- Start or Address byte received
- Byte complete
- Stop event
- Bus Error

For more information on I^2C, see the application note AN50987 - Getting Started with I^2C in PSoC® 1.

Sleep Timer

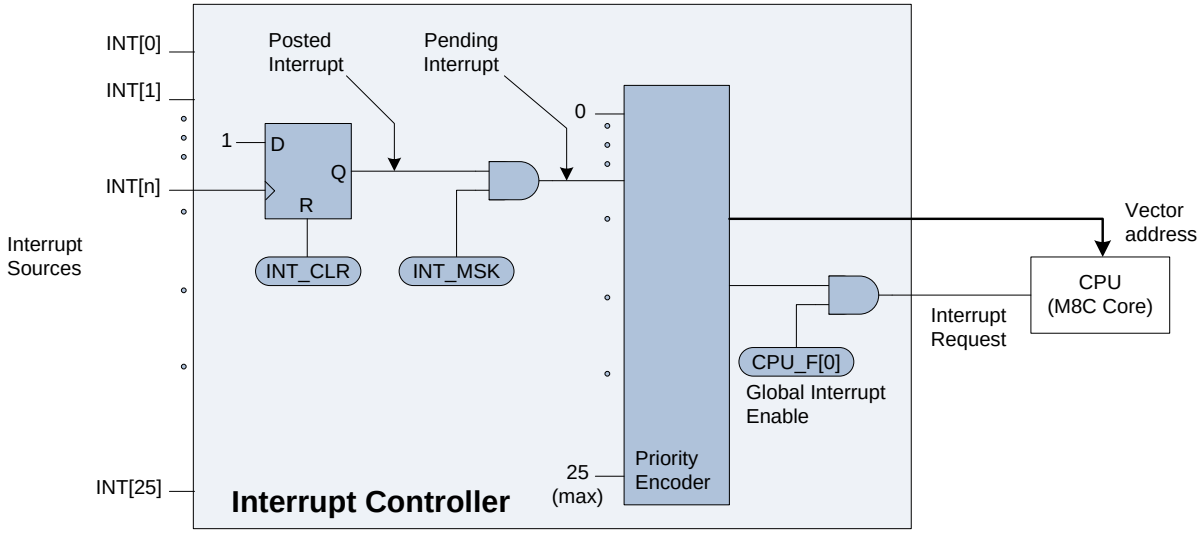
This is a 15-bit timer with the clock input set to a 32-kHz ILO or an external crystal oscillator (ECO). When enabled, the sleep timer generates periodic interrupts with a frequency configurable to 1, 8, 64, or 512 Hz. For details on how to use sleep timer, see the application note, AN47310 - PSoC® 1 Power Savings Using Sleep Mode.

Interrupt Controller

The interrupt controller takes the interrupt signals as inputs and triggers the CPU with a request signal and a corresponding vector address when any enabled interrupt becomes active. Figure 2 shows the block diagram of the interrupt controller. Here’s how the interrupt mechanism works:

1. A rising-edge signal at the interrupt line “posts” an interrupt.
2. If this interrupt is enabled, it is tagged as a “pending”.
3. A priority encoder scans the pending interrupts and selects the one with the highest priority.
4. If the global interrupt is enabled, the priority encoder forwards the vector address of the selected interrupt with a request signal to the CPU.
5. The CPU finishes the instruction currently in execution, and then pushes the program counter (PC) and flag register (CPU_F) to the stack.
6. CPU_F is then cleared by CPU which disables the global interrupt, thereby blocking any other interrupt request.
7. The vector address from the interrupt controller is loaded onto the PC. The CPU jumps to execute the interrupt service routine (ISR) written at the vector address. The posted and pending states of this interrupt are cleared.
8. At the end of the ISR, the return from interrupt (RETI) instruction restores the PC and CPU_F register. This re-enables the global interrupt.
9. If there are any other pending interrupts, the priority encoder again sends the vector address of the highest-priority interrupt that is pending with a request signal to the CPU, and the process repeats.

Figure 2. Interrupt Controller



A particular interrupt can be disabled by writing into the Interrupt Mask register, INT_MSKx. There are four mask registers: INT_MSK0, INT_MSK1, INT_MSK2, and INT_MSK3. Each bit in these registers enables or disables a particular interrupt. For example, as shown in Figure 3, in INT_MSK1, bit 0 corresponds to the digital block DBB00. Writing a logic 1 to this bit enables the DBB00 interrupt. However, when disabled by writing 0 to this bit, an interrupt signal from DBB00 can still post the interrupt, but it will not be executed. A posted interrupt can be cleared by writing into the Interrupt Clear register, INT_CLRx. Like the INT_MSK mask registers, there are four INT_CLR registers. Each bit in these registers clears a particular posted interrupt when written with a logic 0. Figure 4 shows the INT_CLR1 register. Notice the 1-to-1 correspondence with the INT_MSK1 register. Reading the INT_CLR register returns the status of the posted interrupts – a logic 1 indicates that the interrupt is posted. When a logic 0 is written to a particular bit and if the bit has a posted interrupt, then the posted interrupt is cleared. When a 1 is written to a particular bit, and if the ENSWINT (Enable Software Interrupt) in INT_MSK3 is enabled, this will result in the interrupt getting posted. If the ENSWINT bit is not set, then writing a 1 to a bit does not have any effect.

Figure 3. INT_MSK1 Interrupt Mask Register

4, 2 COLUMN	7	6	5	4	3	2	1	0
Access : POR	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0
Bit Name	DCB13	DCB12	DBB11	DBB10	DCB03	DCB02	DBB01	DBB00

Figure 4. INT_CLR1 Interrupt Clear Register

4, 2 Rows	7	6	5	4	3	2	1	0
Access : POR	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0	RW : 0
Bit Name	DCB13	DCB12	DBB11	DBB10	DCB03	DCB02	DBB01	DBB00

To clear all pending interrupts, use the INT_VC register. Writing any value to this register clears all the posted and pending interrupts. Reading this register returns the address of the next highest-priority interrupt that is pending. This helps to know the other pending interrupts while executing a specific ISR.

All interrupts can be controlled by a Global Interrupt Enable (GIE) bit in the CPU_F register. Setting this bit to 1, enables all the interrupts. However, individual interrupts can still be controlled using the INT_MSKx register.

For details on these registers, refer to the Technical Reference Manual of the PSoC 1 device.

PSoC 1 Interrupt Priority

PSoC 1 has interrupt sources with fixed vector addresses and priorities. As Table 1 and Table 2 shows, Reset (watchdog timer reset or external reset) has the highest priority, followed by LVD, configurable analog blocks, VC3 clock, GPIOs, configurable digital blocks, I^2C, and Sleep Timer. Many times, it becomes essential to keep some interrupts with the priority higher than the others. Even with fixed priority interrupts in PSoC 1, you can get the required priority among the digital blocks and analog blocks. The interrupt priority in the analog section depends on the column being used; interrupt from analog column 0 has the highest priority. In the digital section, a user module placed in the left-most and highest block has the highest priority, as Figure 5 shows. Place the user modules so that it occupies blocks of a particular priority.

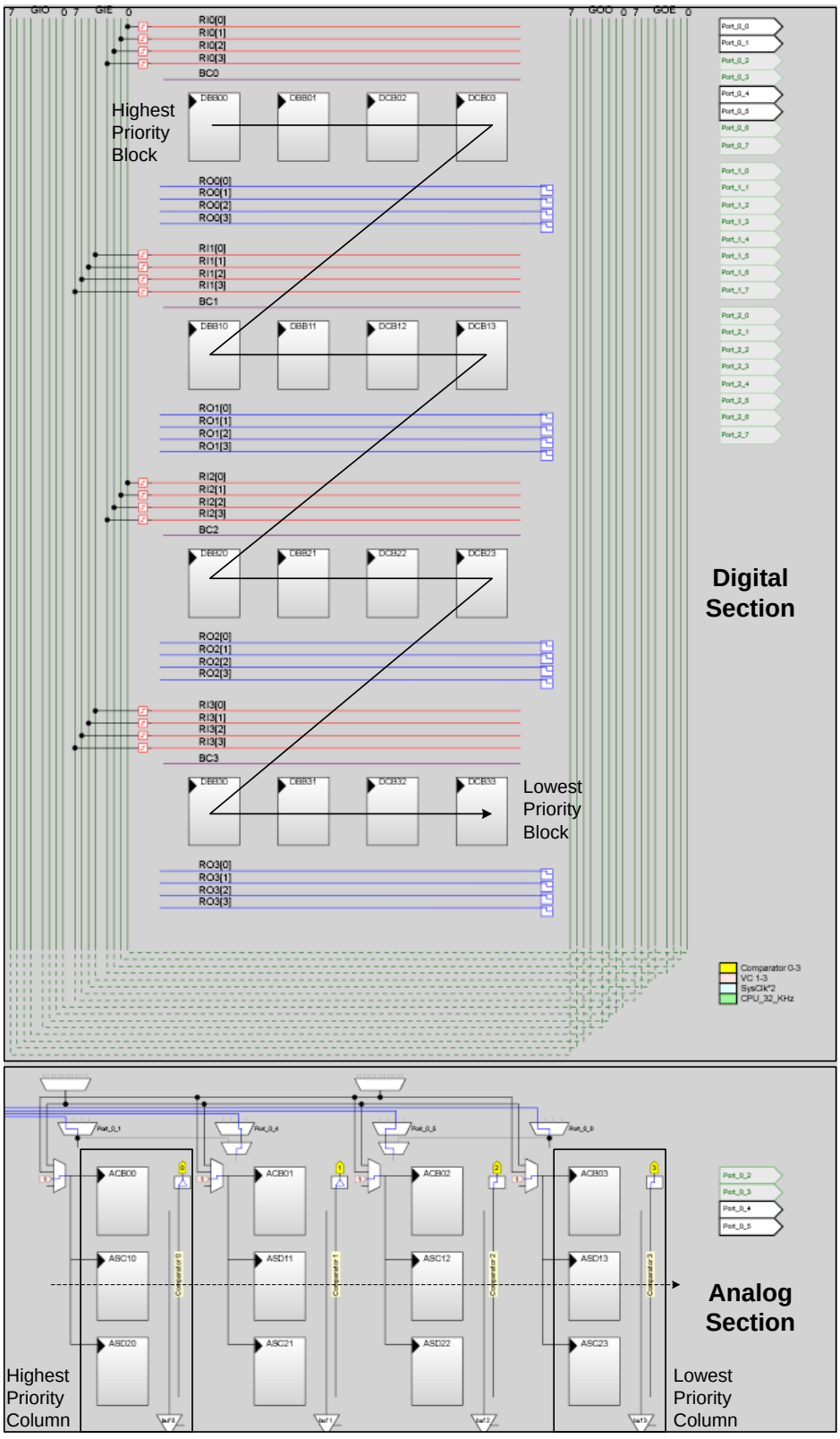
Table 1. Device Interrupts for PSoC 1 Devices Except CY8C28xxx

Interrupt Vector #	Interrupt Address	CY8C29X66	CY8C27X43	CY8C24x94	CY8C24x23	CY8C24x23A	CY8C22x13	CY8C21x34	CY8C21x23	CY8C22x45	CY8C21345	
0 (Highest Priority)	0000h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Reset
1	0004h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Supply Voltage Monitor (LVD)
2	0008h	Y	Y	Y	Y	Y		Y	Y	Y	Y	Analog Column 0
3	000Ch	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Analog Column 1
4	0010h	Y	Y							Y	Y	Analog Column 2
5	0014h	Y	Y							Y	Y	Analog Column 3
6	0018h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	VC3
7	001Ch	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	GPIO
8	0020h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBB00
9	0024h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBB01
10	0028h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCB02
11	002Ch	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCB03
12	0030h	Y	Y							Y		PSoC Block DBB10
13	0034h	Y	Y							Y		PSoC Block DBB11
14	0038h	Y	Y							Y		PSoC Block DCB12
15	003Ch	Y	Y							Y		PSoC Block DCB13
16	0040h	Y		USB Bus Reset								PSoC Block DBB20
17	0044h	Y		USB Start of Frame								PSoC Block DBB21
18	00048h	Y		USB Endpoint 0								PSoC Block DCB22
19	004Ch	Y		USB Endpoint 1								PSoC Block DCB23
20	0050h	Y		USB Endpoint 2						Y		PSoC Block DBB30/ SARADC
21	0054h	Y		USB Endpoint 3						Y		PSoC Block DBB31/ CSD0
22	0058h	Y		USB Endpoint 4						Y		PSoC Block DCB32/ CSD1
23	005Ch	Y		USB Wakeup Interrupt						Y		PSoC Block DCB33/ RTC
24	0060h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	I^2C
25 (Lowest Priority)	0064h	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Sleep Timer

Table 2. Device Interrupts for CY8C28xxx

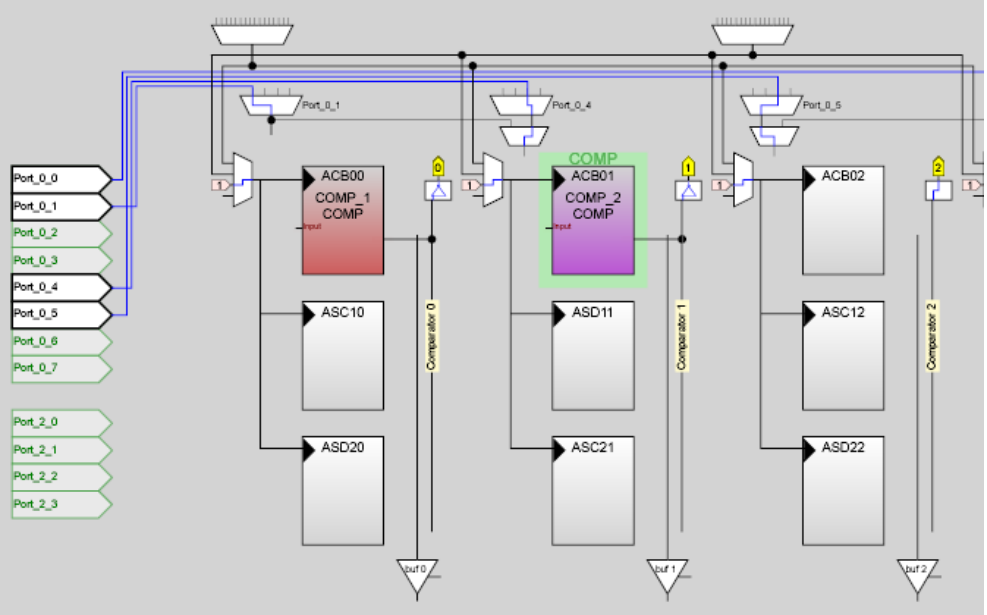
Interrupt Vector #	Interrupt Address	CY8C28x03	CY8C28x13	CY8C28x23	CY8C28x33	CY8C28x43	CY8C28x45	CY8C28x52	
0 (Highest Priority)	0000h	Y	Y	Y	Y	Y	Y	Y	Reset
1	0004h	Y	Y	Y	Y	Y	Y	Y	Supply Voltage Monitor (LVD)
2	0008h			Y	Y	Y	Y	Y	Analog Column 0/ Decimator 0
3	000Ch			Y	Y	Y	Y	Y	Analog Column 1/ Decimator 1
4	0010h					Y	Y	Y	Analog Column 2/ Decimator 2
5	0014h					Y	Y	Y	Analog Column 3/ Decimator 3
6	0018h	Y	Y	Y	Y	Y	Y	Y	VC3
7	001Ch	Y	Y	Y	Y	Y	Y	Y	GPIO
8	0020h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBC00
9	0024h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBC01
10	0028h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCC02
11	002Ch	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCC03
12	0030h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBC10
13	0034h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DBC11
14	0038h	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCC12
15	003Ch	Y	Y	Y	Y	Y	Y	Y	PSoC Block DCC13
16	0040h	Y	Y	Y	Y	Y	Y		PSoC Block DBC20
17	0044h	Y	Y	Y	Y	Y	Y		PSoC Block DBC21
18	00048h	Y	Y	Y	Y	Y	Y		PSoC Block DCC22
19	004Ch	Y	Y	Y	Y	Y	Y		PSoC Block DCC23
20	0050h								Reserved
21	0054h								Reserved
22	0058h								Reserved
23	005Ch								Reserved
24	0060h	Y	Y	Y	Y	Y	Y	Y	I^2C0
25	0064h	Y		Y		Y	Y		I^2C1
26	0068h	Y	Y		Y	Y	Y		SAR ADC
27	006Ch	Y	Y	Y	Y	Y	Y	Y	RTC
28	0070h		Y		Y		Y	Y	Analog Column 4
29	0074h		Y		Y		Y	Y	Analog Column 5
30	0078h								Reserved
31 (Lowest Priority)	007Ch	Y	Y	Y	Y	Y	Y	Y	Sleep Timer

Figure 5. Interrupt Priority for Digital and Analog Blocks



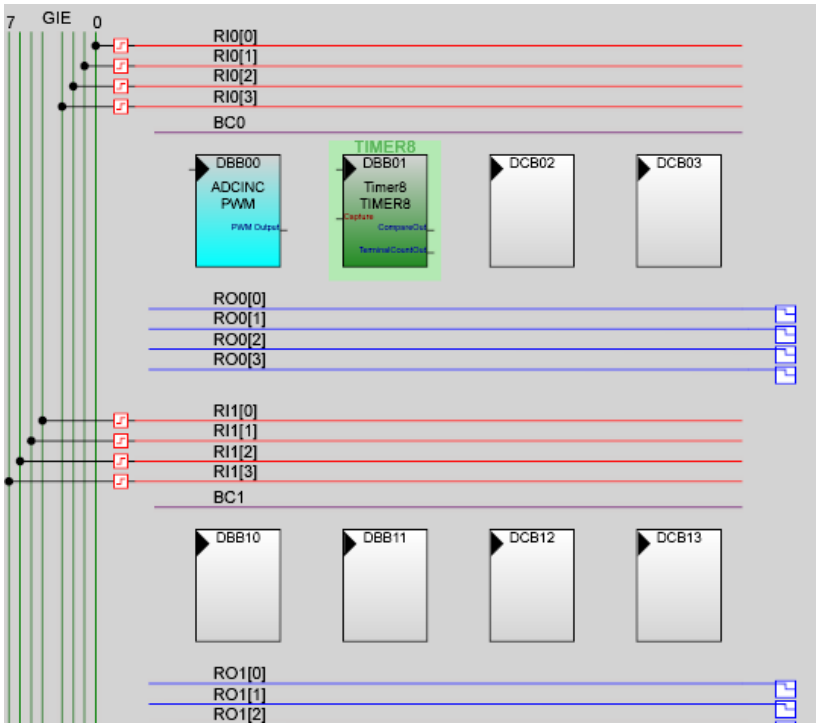
For example, if you have two comparator user modules (UM) in the analog section (COMP_1 and COMP_2), and if you want COMP_1 to be of higher priority than COMP_2, then place COMP_1 in the column to the left side of COMP_2 as Figure 6 shows.

Figure 6. PSoC Designer Project Showing the Placement of Two Comparators With the Interrupt Priority of COMP_1 Higher Than That of COMP_2



If you have an analog-to-digital converter UM (ADCINC) and timer UM (Timer8) in your design, to have the ADCINC interrupt priority higher than that of the timer interrupt, place the ADCINC UM in DBB00 and the Timer8 user module in DBB01, as Figure 7 shows.

Figure 7. PSoC Designer Project Showing the Placement of ADCINC and Timer8 with Interrupt Priority of ADCINC Higher Than That of Timer8



Vectors are listed in the *boot.asm* file, which is automatically added to the PSoC Designer project when it is generated. With the placement as shown Figure 6 and Figure 7, the interrupt priorities can be seen in Figure 8. Modules with higher priority are placed higher in the order.

Figure 8. Boot.asm File Contents Showing the Vectors Assigned for Each Interrupt

```
;@PSoC_BOOT_ISR_UserCode_START@
;-----------------------------------------------
; Insert your custom code below this banner
;-----------------------------------------------

org   04h                     ;Low Voltage Detect (LVD) Interrupt Vector
halt                          ;Stop execution if power falls too low

org   08h                     ;Analog Column 0 Interrupt Vector
ljmp    _COMP_1_ISR
reti

org   0Ch                     ;Analog Column 1 Interrupt Vector
ljmp    _COMP_2_ISR
reti

org   10h                     ;Analog Column 2 Interrupt Vector
// call void_handler
reti

org   14h                     ;Analog Column 3 Interrupt Vector
// call void_handler
reti

org   18h                     ;VC3 Interrupt Vector
// call void_handler
reti

org   1Ch                     ;GPIO Interrupt Vector
// call void_handler
reti

org   20h                     ;PSoC Block DBB00 Interrupt Vector
ljmp    _ADCINC_ADConversion_ISR
reti

org   24h                     ;PSoC Block DBB01 Interrupt Vector
ljmp    _Timer8_ISR
reti
```

Interrupt Support in PSoC Designer

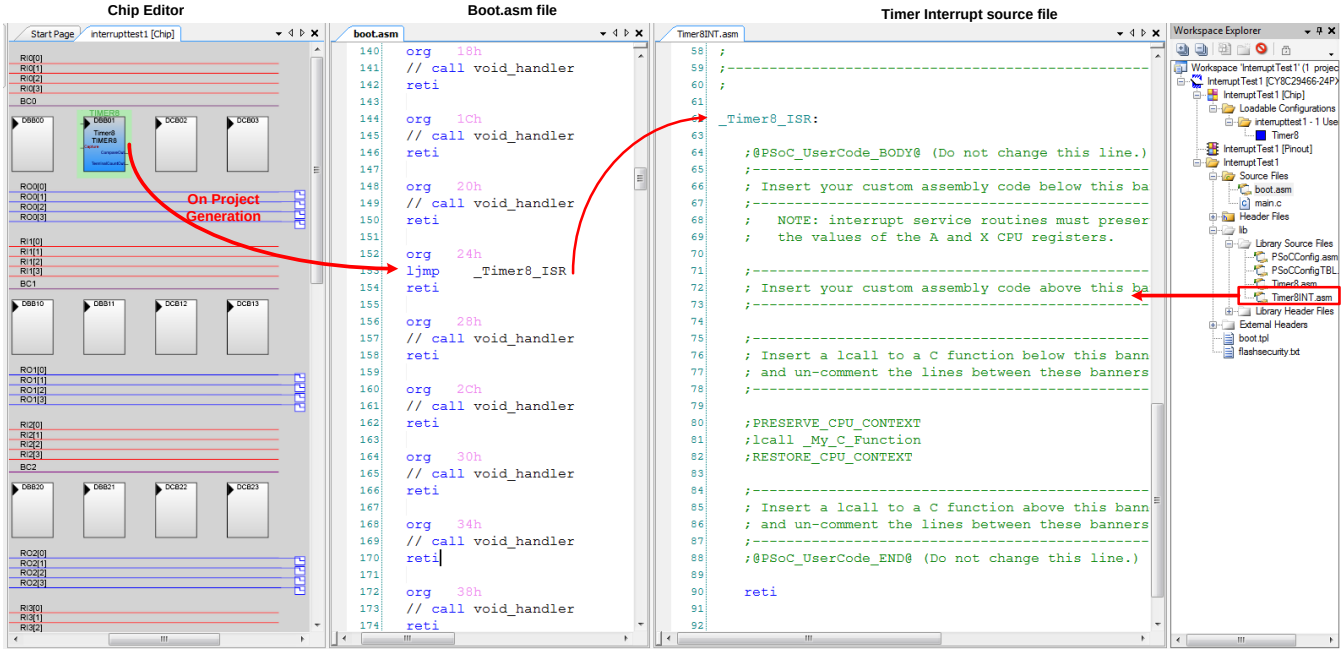
The *boot.asm* file of a project has the code for interrupt vectors. It is generated whenever a “Generate Source” operation of the project is performed in PSoC Designer. Depending on the UMs placed, PSoC Designer performs the following related to interrupts:

1. Generates the interrupt source file associated with the UM
2. Inserts an `ljmp` instruction in the vector location in the *boot.asm* file to jump to the function in the generated interrupt source file. This is done as there are only four bytes available between the vector locations and is not sufficient for any useful application. With a jump instruction consuming 3 bytes, a custom ISR code can be written somewhere else in the flash. For the UMs, the ISR is written in the associated interrupt source file.

Note that PSoC Designer does not generate source files for interrupt sources such as analog columns, LVD, and VC3. You should add the jump instruction manually to jump to a custom ISR.

The *boot.asm* code and the sample interrupt source file for a Timer8 UM (*Timer8INT.asm*) are shown in Figure 9.

Figure 9. Interrupt Source File



As Figure 9 shows, the PSoC Designer assembly instruction `ljmp _Timer8_ISR` is inserted at the vector location 24h (depending on the placement of the Timer8 module). The assembly function `_Timer8_ISR` is in the interrupt source file *Timer8INT.asm* associated with the user module. You can write the custom assembly code or call a custom ‘C’ function from `_Timer8_ISR`. Make sure that you uncomment the code `PRESERVE_CPU_CONTEXT` and `RESTORE_CPU_CONTEXT` while calling a ‘C’ function from the ISR. The `PRESERVE_CPU_CONTEXT` macro saves the accumulator register value, all virtual registers, and page pointers (current page pointer `CUR_PP`, Indexed Memory Access page pointer `IDX_PP`, MVI Read page pointer `MVR_PP`, and MVI Write page pointer `MVW_PP`) in the stack. The `RESTORE_CPU_CONTEXT` function restores these register values. Thus, a CPU state is maintained to what it was executing earlier before branching to an ISR. Note that here a Timer8 example is taken, but the same is applicable for other user modules.

The use of PRESERVE and RESTORE macros, however, results in a lot of CPU overhead and flash and stack consumption. It consumes around 190 CPU cycles, 59 bytes of flash, and 20 bytes of stack space. If you are planning to write a ‘C’ ISR, it is recommended to use the `#pragma interrupt_handler` compiler directive.

Follow the steps given below to write a ‘C’ ISR:

1. Write a C function ISR with a name such as `MyTimerInt`.
2. Add the `#pragma` directive to inform the compiler that `MyTimerInt` is an interrupt handler:

```
#pragma interrupt_handler MyTimerInt

void MyTimerInt(void)
{
        //handler code
}
```

The `#pragma interrupt_handler` directive inserts instructions at the beginning of an ISR to save only the used virtual registers, accumulator register, and page pointers in the stack. It also inserts a RETI (return from interrupt) instruction at the end of the ISR instead of a RET instruction. This restores the CPU_F register status to the state before the ISR execution. Adding the `#pragma` directive moves only the selected registers to stack, thus reducing the CPU overhead and memory usage as compared to using the `PRESERVE` and `RESTORE` macros.

3. Link this function with the interrupt vector. As Interrupt vectors are listed in the *boot.asm* file, you add the `ljmp` instruction in this file to make the CPU jump to the 'C' function on interrupt. Note that the *boot.asm* file is overwritten every time "Generate Source" operation is performed. To avoid this, modify the *boot.tpl* file of the project. This is a template file and PSoC Designer uses this file to generate the *boot.asm* file. The *boot.tpl* file is present in the project directory. Go to File > Open File menu option. Window will open as Figure 10 shows. Clear the filter to display all the files. Select the *boot.tpl* file and click the **Open** button as Figure 11 shows.
4. Add the `ljmp` instruction at the vector location in the *boot.tpl* file. To know the vector location, use the comments in the *boot.tpl* file, which mentions the block number. Figure 12 shows an example with Timer8 UM placed at DBB01 block. When a timer interrupt occurs, the CPU will first land at the location 24h and it will then be redirected to the 'C' function `MyTimerInt`. Make sure that you precede the function name with an underscore '_'. Every function or variable declared in 'C' when called from an *asm* file must begin with an underscore.

Figure 10. Boot.tpl file Directory

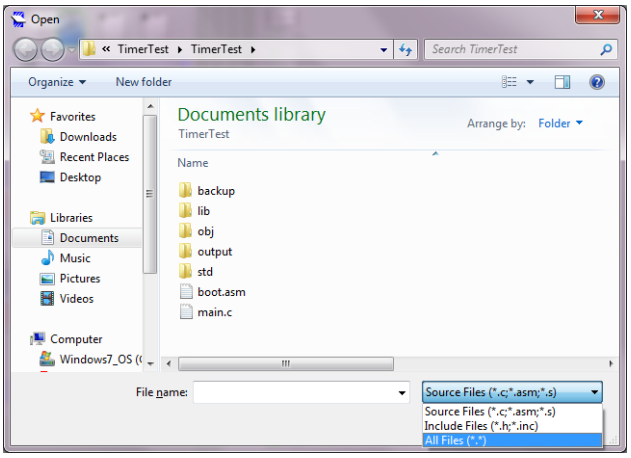


Figure 11. Boot.tpl file Selection

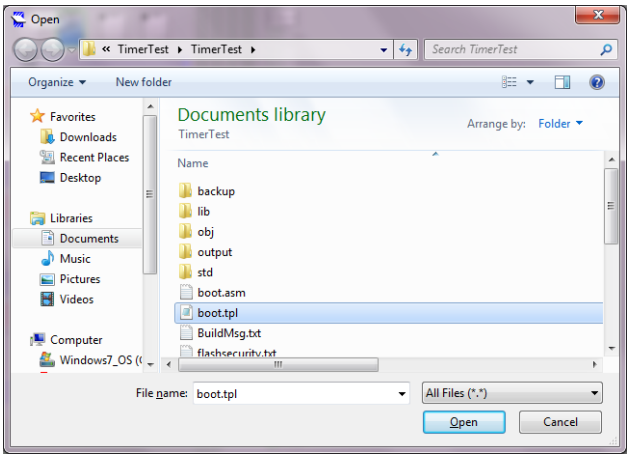
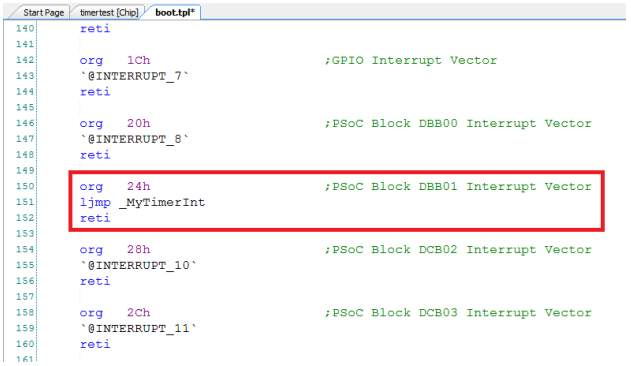


Figure 12. Adding the ljmp Instruction in the boot.tpl File



Interrupt APIs and Macros Available in PSoC Designer

The source and header files generated for the user modules provide the following APIs to enable or disable the interrupts:

- `<User Module Name>_EnableInt()`- API for enabling the interrupt.
- `<User Module Name>_DisableInt()` - API for disabling the interrupt.

There are macros defined in the *m8c.h* file that you can use to enable or disable interrupt masks, clear the interrupt flags, and so on. The following macros are available:

- `M8C_EnableGInt` – Macro for enabling the global interrupt.
- `M8C_DisableGInt` – Macro for disabling the global interrupt.
- `M8C_EnableIntMask` – Macro for enabling the interrupt mask by configuring the register `INT_MSKx`. The inputs required for this macro are registers `INT_MSKx` and `MASK`. `INT_MSKx` stands for the registers `INT_MSK0`, `INT_MSK1`, `INT_MSK2`, or `INT_MSK3`; `MASK` is the pointer to the bit field in the mask register. The name of the bit masks can be found in the *m8c.h* header file. Some examples follow:

```
/* Enable GPIO Interrupt mask */
M8C_EnableIntMask(INT_MSK0, INT_MSK0_GPIO);

/* Enable Sleep Interrupt mask */
M8C_EnableIntMask(INT_MSK0, INT_MSK0_SLEEP);

/* Enable DBB00 Interrupt mask */
M8C_EnableIntMask(INT_MSK1, INT_MSK1_DBB00);

/* Enable I2C Interrupt mask */
M8C_EnableIntMask(INT_MSK3, INT_MSK3_I2C);
```

- `M8C_DisableIntMask` – Macro for disabling the interrupt mask by configuring the register `INT_MSKx`. The inputs required are `INT_MSKx` and `MASK`. `INT_MSKx` stands for the registers `INT_MSK0`, `INT_MSK1`, `INT_MSK2`, or `INT_MSK3`; `MASK` is the bit-field in the mask register. For example:

```
/* Disable GPIO Interrupt mask */
M8C_DisableIntMask(INT_MSK0, INT_MSK0_GPIO);

/* Disable Sleep Interrupt mask */
M8C_DisableIntMask(INT_MSK0, INT_MSK0_SLEEP);

/* Disable DBB00 Interrupt mask */
M8C_DisableIntMask(INT_MSK1, INT_MSK1_DBB00);

/* Disable I2C Interrupt mask */
M8C_DisableIntMask(INT_MSK3, INT_MSK3_I2C);
```

  In most cases, you won't need the `M8C_EnableIntMask` and `M8C_DisableIntMask` macros as the user modules provide the `EnableInt()` and `DisableInt()` APIs to enable or disable the interrupt, which is easier than working with macros. However, the macros have some advantage over the APIs: APIs are executed using a call and therefore take longer time to execute than the macros.

- `M8C_ClearIntFlag` – Macro for clearing the interrupt flag by writing into the `INT_CLRx` register. The inputs required for this macro are `INT_CLRx` and `MASK`. `INT_CLRx` stands for the registers `INT_CLR0`, `INT_CLR1`, `INT_CLR2`, or `INT_CLR3`; `MASK` is the pointer to the bit field in the `INT_CLR` register. For example:

```
/* Clear GPIO Interrupt flag */
M8C_ClearIntFlag(INT_CLR0, INT_MSK0_GPIO);

/* Clear Sleep Interrupt flag */
M8C_ClearIntFlag(INT_CLR0, INT_MSK0_SLEEP);

/* Clear DBB00 Interrupt flag */
M8C_ClearIntFlag(INT_CLR1, INT_MSK1_DBB00);

/* Clear I2C Interrupt flag */
M8C_ClearIntFlag(INT_CLR3, INT_MSK3_I2C);
```

Important Note: The software interrupt should be disabled while using the macro `M8C_ClearIntFlag`. Software interrupt is controlled by the `ENSWINT` bit in the INT_MSK3 register; it is disabled by default (ENSWINT is logic 0). If software interrupt is enabled, executing the macro `M8C_ClearIntFlag` will result in seven bits of the INT_CLR register to be set to logic 1, thus triggering seven software interrupts.

Interrupt Latency

The assertion of an interrupt results in the following:

- Saving of the PC and CPU_F registers in the stack
- Clearing of the CPU_F register
- Loading the vector address of an interrupt in the PC

These three actions are completed in 13 CPU cycles. Apart from this, the CPU needs to complete the execution of the current instruction in hand (in the worst case, five cycles) and execute the `ljmp` instruction at the vector location (seven cycles) as mentioned in the section Interrupt Support in PSoC Designer. Thus, it takes 13 + 5 + 7 = 25 CPU cycles. At 24 MHz CPU frequency, it takes 1.04µs; at 12 MHz, it takes 2.08µs.

Note that there is additional overhead of preserving the virtual registers, accumulator, and page pointer registers. The time taken for these actions varies from project to

project, depending on whether the registers are in use or not. You can check the instructions added in the beginning of an ISR in list file (*<project name>.lst*) of the project. Figure 13 shows an example ISR from the .lst file.

Figure 13. CPU Cycles Overhead in ISR

```
(0075) #pragma interrupt_handler TimerInt_Track2
(0076) void TimerInt_Track2(void)
(0077) {
_TimerInt_Track2:
    1729: 71 C0     OR    F,0xC0      ---- Configures Addressing Mode
    172B: 08        PUSH  A           ---- Save Accumulator in Stack
    172C: 5D D0     MOV   A,REG[0xD0] }
    172E: 08        PUSH  A           } -- Save Current Page Pointer in Stack
    172F: 5D D4     MOV   A,REG[0xD4] }
    1731: 08        PUSH  A           } -- Save MVI Read Page Pointer in Stack
    1732: 5D D5     MOV   A,REG[0xD5] }
    1734: 08        PUSH  A           } -- Save MVI Write Page Pointer in Stack
    1735: 62 D0 00  MOV   REG[0xD0],0x0 -- Set current page pointer to page 0 of RAM
    1738: 51 B8     MOV   A,[__r0]    }
    173A: 08        PUSH  A           }
    173B: 51 B7     MOV   A,[__r1]    }
    173D: 08        PUSH  A           } -- Save Virtual Registers r0, r1, r2 and r3 in Stack
    173E: 51 B6     MOV   A,[__r2]    }
    1740: 08        PUSH  A           }
    1741: 51 B5     MOV   A,[__r3]    }
    1743: 08        PUSH  A           }
    1744: 76 DD     INC   [SampleIntCount_Track2] -- User Code
(0078)
(0079)  SampleIntCount_Track2++;
(0080)
```

As Figure 13 shows, the compiler adds instructions to save the accumulator, page pointers, and virtual registers in the stack. The execution of these instructions adds to the interrupt latency. Remember, this is the minimum latency when no other interrupts are enabled. When there are multiple interrupts enabled in a project, calculation of interrupt latency is much more complex. For example, if an interrupt is triggered when another interrupt routine is already being executed, the processor has to complete the execution of the current interrupt routine first before servicing the new interrupt. If another interrupt with a higher priority is posted meanwhile, the processor would then execute the next highest priority interrupt before servicing the lower priority one.

Project - Timer_Interrupt

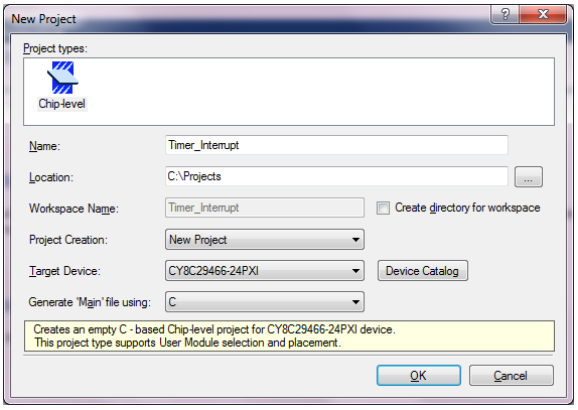
This section shows you how to create a simple interrupt-based project in PSoC Designer. With this code example, you will learn how to configure a timer interrupt and link a ‘C’ function to its vector.

In this code example, an LED connected at port P1[7] is toggled when a timer overflows. The code to toggle the LED state is written in the timer ISR. The timer interrupt frequency is configured as 1 Hz.

Use the following steps to create the project:

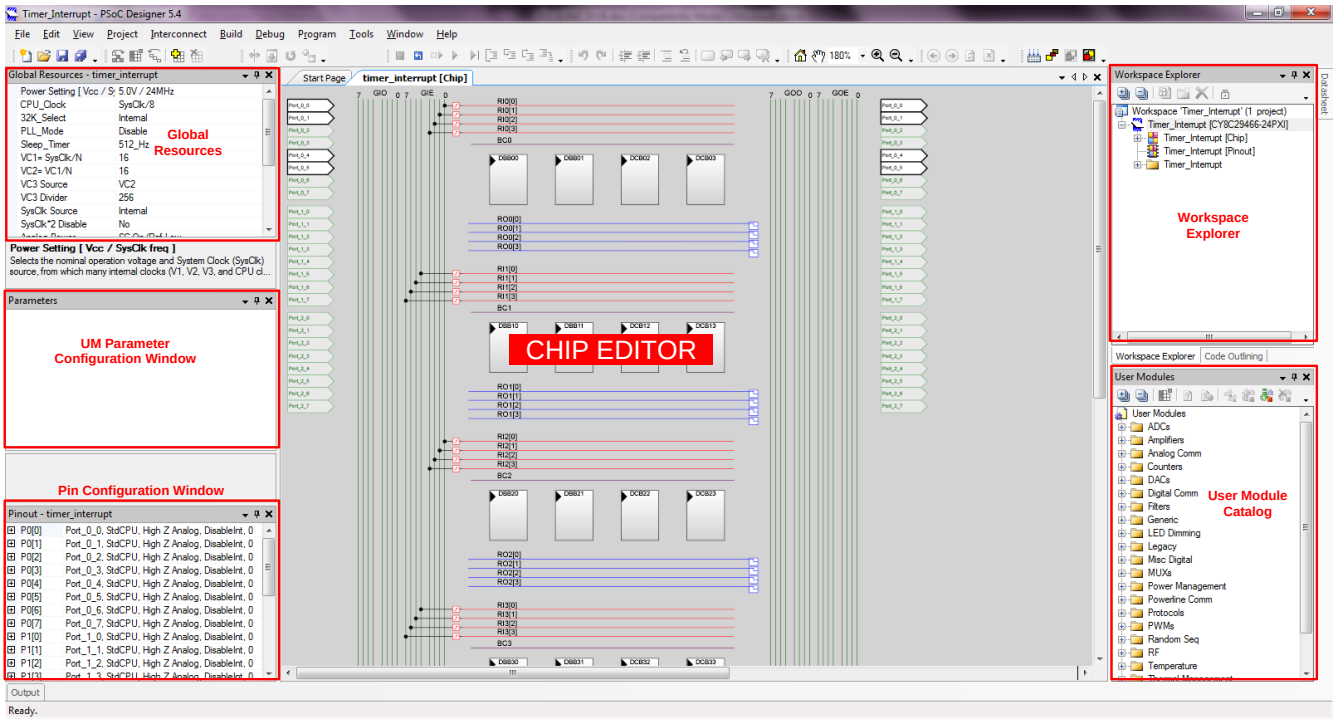
1. Create a PSoC Designer project (**File > New Project**) and name it **Timer_Interrupt**, as shown in Figure 14 .

Figure 14. Creating a PSoC Designer Project



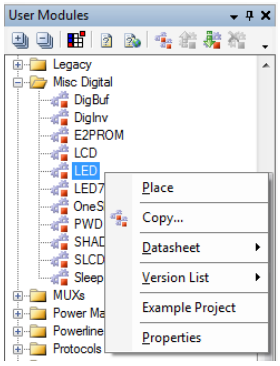
2. Select the part number for the device and the preference of coding language for the main program file. For this project, CY8C29466-24PXI is selected; thus, project can be easily tested on CY3210-PSoCEval1 kit. Select ‘C’ language for coding, as Figure 14 shows. After you have made all the changes, click **OK**. The Chip Editor view of PSoC Designer will open as Figure 15 shows.

Figure 15. PSoC Designer



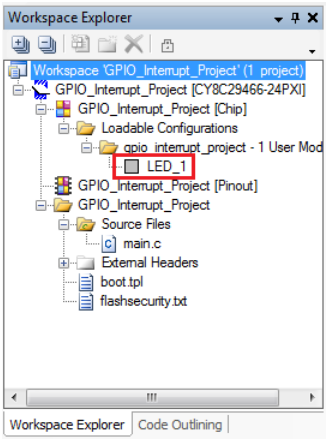
3. Select **View** > **User Module Catalog** to display the User Module Catalog, and then expand the **Misc Digital** folder. Locate the **LED** User Module, right-click on it, and select **Place** as Figure 16 shows. This UM will be used to drive the external LED.

Figure 16. LED User Module Placement



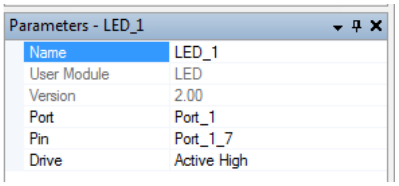
4. Expand the workspace explorer as Figure 17 shows. Click on LED_1 to configure the user module properties. After clicking on LED_1, on the left-hand side of PSoC Designer, the parameters window allows you to edit the LED's properties.

Figure 17. Selecting LED_1 UM to Configuring the Parameters



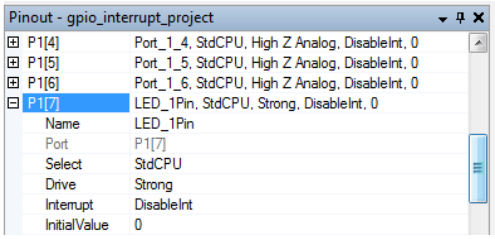
5. Configure the port and pin of LED_1 UM to "Port_1" and "Port_1_7", respectively, as Figure 18 shows.

Figure 18. LED User Module Parameters



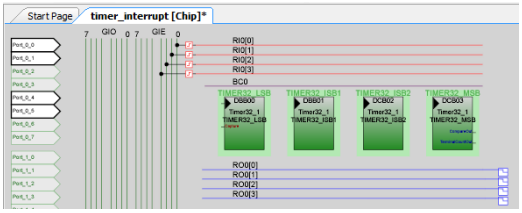
The drive mode of the pin P1_7 is automatically set to strong mode by the UM. You can verify this in the Pin Configuration window as Figure 19 shows.

Figure 19. Port_1_7 Parameters



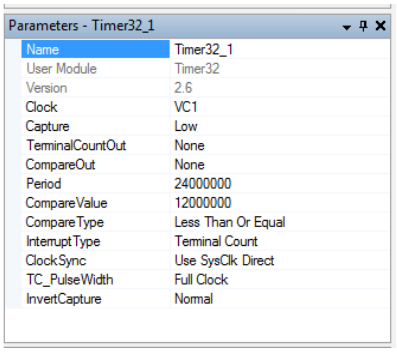
6. Select and place the 32-bit timer user module (Timer32) from the User Module Catalog as shown in Figure 20.

Figure 20. Timer32 UM Placement



7. Configure the input clock and the period of the timer to get the required interrupt frequency. Select the system clock of 24 MHz as the timer input. To get a 1-Hz timer input interrupt frequency, the required period is 24 MHz / 1 Hz = 24000000 (0x016E3600).

Figure 21. Timer32_1 Parameter Configuration

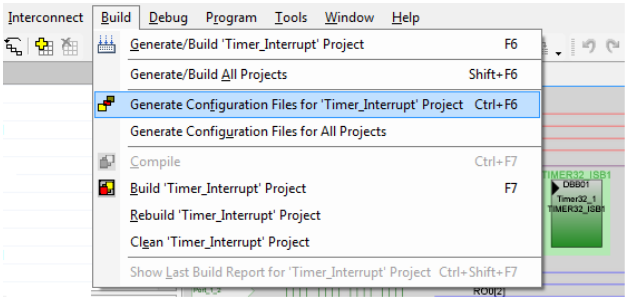


Set the **Period** parameter to 24000000, **Interrupt Type** to Terminal Count and **Clock Sync** to Use

SysClk Direct. This Clock Sync option overrides the **Clock** parameter and uses System Clock as the input to the timer. Other settings do not affect the project operation.

8. After the user modules are placed and configured, generate the configuration files for the project. Select **Build > Generate Configuration Files for 'Timer_Interrupt' Project** as shown in Figure 22 (or, press **[Ctrl] + [F6]**).

Figure 22. Generate Configuration Files



9. In the *main.c* file, start the timer and enable its interrupt and the global interrupt. Go to Workspace Explorer, locate the **Source Files** folder and open the *main.c* file. In this file, place the following source code.

```
/* Part specific constants and macros
*/
#include <m8c.h>

/* PSoC API definitions for all User
Modules */
#include "PSoCAPI.h"

void main(void)
{
    /* Enable Global Interrupt */
    M8C_EnableGInt;

    /* Start the Timer */
    Timer32_1_Start();

    /* Enable Timer Interrupt. This
    library function writes into
    INT_MSK0 register */
    Timer32_1_EnableInt();

    while(1);
}
```

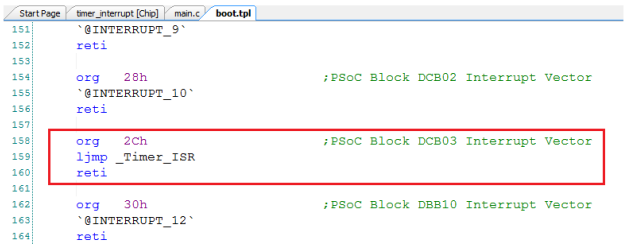
Write the code in the timer ISR to toggle the LED state. In PSoC Designer, most of the ISRs are implemented as a part of the user module library. There are two ways of writing the ISR as described in section Interrupt Support in PSoC Designer. You can write the assembly code inside the timer interrupt source file (*Timer32_1INT.asm*) or write a 'C' function and link it to the timer interrupt. This example project uses the 'C' function. Add the code below in the *main.c* file. This code is executed at the frequency of 1 Hz.

```
#pragma interrupt_handler Timer_ISR

/* Timer ISR in C where timer
interrupts are processed */
void Timer_ISR(void)
{
    /* Toggle LED */
    LED_1_Invert();

}
```

Map the `Timer_ISR` function to the Timer32_1 Interrupt vector in the *boot.tpl* file as explained in the section Interrupt Support in PSoC Designer. Figure 23 shows the *boot.tpl* file. Notice that the interrupt vector is for the "most significant byte" block out of four blocks used by the Timer32 user module.

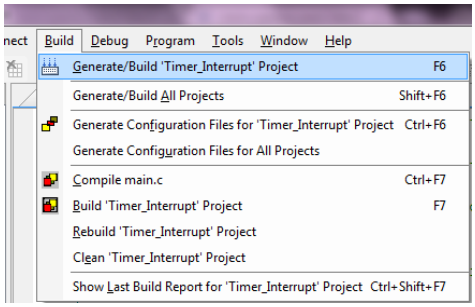
Make sure that the function name begins with an underscore ("_") because it is a 'C' function. Every function or variable declared in 'C' when called from a *.asm* file must begin with an "_".

Figure 23. `boot.tpl` File Showing the Mapping of Timer_ISR



10. Now, build and generate the project. Select **Build > Generate/Build 'Timer_Interrupt' Project** as shown in Figure 24 (or, press **[F6]**).

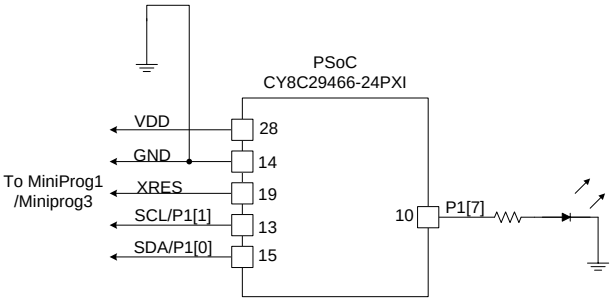
Figure 24. Build and Generate Option



Test Procedure

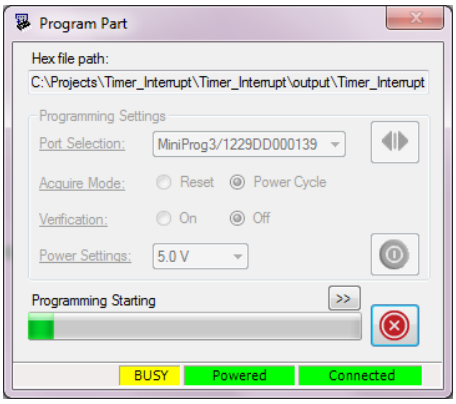
This section provides the procedure to test the project with the CY3210 – PSoCEval1 kit. To test it on any other development platform, make the connections as given in Figure 25.

Figure 25. External Connections



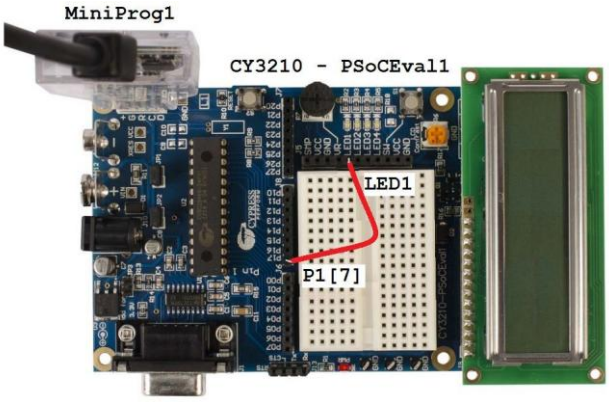
After the build process is completed without warnings or errors, the next step is to program the device. Connect the MiniProg1 or MiniProg3 programmer between your PC and CY3210-PSoCEval1. Ensure that a CY8C29466-24PXI is the device currently on the board. In PSoC Designer, locate Program in the menu bar and click on the **Program Part** button.

Figure 26. Programming Status



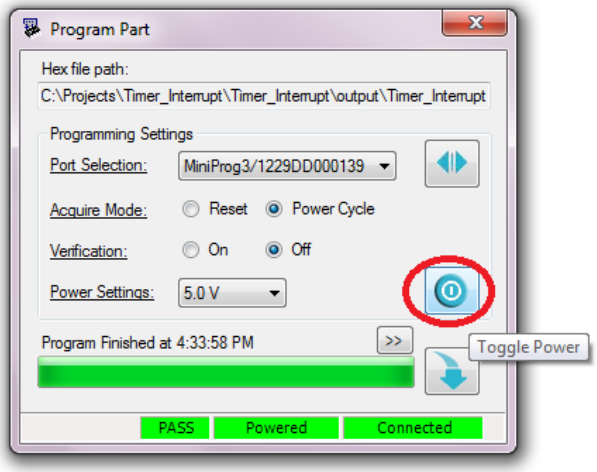
Place a wire connecting P1[7] to LED1 as Figure 27 shows.

Figure 27. CY3210-PSoCEval1 Pin Connections



Power the device from MiniProg1 or MiniProg3 by clicking on the **Toggle Power** button as Figure 28 shows.

Figure 28. Power and Program Connections



Notice that LED1 blinks at 0.5 Hz rate, that is, half the timer interrupt frequency.

Tips and Tricks

Optimizing the Interrupt Code

An important performance parameter in interrupt-based applications is the ISR code execution time. In some applications, the critical code in the ISR must be serviced within a particular time of receiving the interrupt request. In some other applications, interrupt execution should not take long because it could stall the main code execution or other interrupts. Follow the guidelines below when writing the ISR code to meet these requirements:

Avoiding function calls in the ISR

When function calls are made inside a 'C' ISR defined as `#pragma interrupt_handler`, the compiler preserves and restores the virtual registers, page pointers, and accumulator, which results in a large execution time overhead and high risk of stack overflow. Avoid making function calls in an ISR. The recommended technique is to move the non-critical function calls to the main code by setting a flag variable in the ISR. Then, periodically check the flag in the main code.

Assigning proper priority to the interrupts among digital and analog blocks

In applications that have multiple interrupts, place those interrupts that require time-critical servicing, at blocks that have a higher priority associated with it.

Multi-byte Variable Usage

Accessing multi-byte global variables in an 8-bit system requires careful attention because multi-byte variables are read byte-by-byte. Make sure that the ISR is not triggered and, therefore, modify the variable when one or more bytes of the multi-byte variable have already been read but the read has not been completed. This would lead to data corruption. The following example illustrates this scenario:

Case 1

```
void main()
{
    unsigned int localData;

    /* code */
    while (1)
    {
        localData = data;
        /* code */
    }
}

void TEST_ISR(void)
{
    data = BUF[0];
    data = (data<<8)|(BUF[1]);
}
```

An 8-bit system like PSoC 1 splits the 16-bit operation into two 8-bit operations. In PSoC 1, for a 16-bit move instruction (localData = data), first the MSB is moved, followed by the LSB. If the interrupt "TEST_ISR" occurs after moving the MSB but before the LSB and at the end of the execution of move instruction, variable localData will have old MSB and new LSB. To avoid this problem, disable the global interrupt before executing the move instruction and re-enable it after completing the move instruction. This causes the interrupt to remain in pending state until the global interrupt is re-enabled.

Case 2

```
unsigned int data;

void main()
{
    unsigned int localData;

    /* code */
    while (1)
    {
        M8C_DisableGInt;
        localData = data;
        M8C_EnableGInt;
        /* code */
    }
}

void TEST_ISR(void)
{
    data = BUF[0];
    data = (data<<8)|(BUF[1]);
}
```

Nested Interrupts

In PSoC 1, the global interrupt is disabled during the service of an interrupt, thereby disabling the CPU from jumping to another ISR. To enable execution of another interrupt while executing an ISR, enable the global interrupt. Note that priority of the new interrupt is not considered while branching.

The following code gives an example of enabling the nested interrupt with two timer interrupts. Global interrupt is enabled in Timer2_ISR. If the CPU is currently executing Timer2_ISR and if the Timer1 interrupt occurs, the CPU branches to execute Timer1_ISR. After completion, the CPU returns to complete the execution of Timer2_ISR.

```
void main(void)
{
    /* Start Timers */
    Timer8_1_Start();
    Timer8_2_Start();

    /* Enable Timer Interrupts */
    Timer8_1_EnableInt();
    Timer8_2_EnableInt();

    /* Enable Global Interrupt */
    M8C_EnableGInt;

    while (1);
}

#pragma interrupt_handler Timer1_ISR
void Timer1_ISR(void)
{
    //code
}


#pragma interrupt_handler Timer2_ISR
void Timer2_ISR(void)
{
    /* Enable Global Interrupt to
    allow nested interrupts */
    M8C_EnableGInt;

    //code
}
```

There is a risk of stack overflow when working with nested interrupts. PSoC 1 CY3215A-DK In-Circuit-Emulation (ICE) Lite Development Kit can be used to verify the risk of stack overflow. Refer to the application note AN73212 – Debugging with PSoC 1 for more details.

Conditional Loop in ISR

In some applications, a conditional loop in an ISR can cause the CPU to get stuck. Here is an example in an ADC application:

```
#pragma interrupt_handler Timer1_ISR
void Timer1_ISR(void)
{
    unsigned int Value;

    /* Check if ADC data is available. As
    this function is written in an ISR,
    if no previous ADC data is available,
    CPU will never come out of this loop
    */
    while(!ADCINC_1_fIsDataAvailable());

    /* Read ADC Data */
    Value = ADCINC_1_iClearFlagGetData();
}
```

The ADC in PSoC 1 (except SAR ADC) requires CPU in processing the reading. It is done with interrupts. As the global interrupt is disabled while servicing an ISR, the ADC interrupt is never executed during this period. If an attempt is made to check the status of ADC with a blocking statement, the CPU will remain permanently stuck. It is recommended to put an “if” statement, instead of a loop “while” statement, to avoid blocking.

Summary

Interrupts are commonly used in embedded applications. For system-on-chip architectures, such as those of PSoC 1, interrupts play the critical role of communicating the status of on-chip peripherals to the CPU. This application note has provided the information needed to quickly and easily create interrupt-based PSoC Designer projects.

About the Author

Name:	Rajiv Badiger
Title:	Applications Engineer Staff
Background:	Rajiv Badiger holds a Bachelor’s degree in Electronics and Communications Engineering from Nagpur University, India. He has 6 years of experience in embedded systems design.
Contact:	rjvb@cypress.com

Related Application Notes

AN75320 – Getting Started with PSoC 1

AN2094 – PSoC 1 Getting Started with GPIO

AN73212 – Debugging with PSoC 1

AN32200 – PSoC® 1 Clocks and Global Resources

AN47310 – PSoC ® 1 Power Savings Using Sleep Mode

Document History

Document Title: PSoC® 1 Interrupts – AN90833

Document Number: 001-90833

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4250789	RJVB	01/28/2014	New Application Note
*A	5700390	AESATP12	04/26/2017	Updated logo and copyright.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

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

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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