Lab 3: Interrupts and Debouncers

Objective

Demonstrate knowledge of interrupts, ISRs, and debouncers.

Background

B1: Interrupts

In previous labs, the program written for the CPU has polled the hardware for updates. While it has worked in those simple applications, polling becomes less viable in more complicated systems where the CPU needs to process many forms of input. Checking each input continuously wastes time and power, and there will likely be a delay between an input change and the CPU noticing it, which may be unacceptable depending on the application. The alternative to polling is using interrupts.

An interrupt is a signal sent to the CPU that causes it to stop what it is executing, store away its current state, and jump to an interrupt service routine (ISR). The ISR is a function that defines what the CPU does to respond to an interrupt. After an ISR completes, the CPU restores its state and continues normal execution. ISRs should be made to execute as quickly as possible, as returning to normal execution as soon as possible is usually desirable, especially in an application where interrupts occur frequently.

Interrupts operate differently between different CPUs, but the general purpose is the same. There are also different limitations that may be present. For example, not all architectures support interrupt preemption, also known as nested interrupts, and some programming features may be unavailable. This section will focus on how interrupts work in the PSoC 5LP.

Note that there may be concurrency issues when using interrupts, similar to when using threads in applications. Take note of warnings in component datasheets and try to avoid calling functions of components that are used in the main execution thread.

To create an interrupt, add an Interrupt component to the schematic and connect the input to a signal that the CPU should interrupt on.

This component has only one configurable option: InterruptType. These are the choices for InterruptType:

- RISING_EDGE: the interrupt is triggered to go off only once when the input digital signal goes from low to high. A sustained high signal does not trigger further interrupts; the signal must go low first. Note that if the signal goes low and high again before the ISR begins execution (for example, if the interrupt is waiting for a higher priority one to finish), the interrupt will only occur once.
- LEVEL: the interrupt is triggered to go off as long as the input digital signal is high. If the ISR returns and the input is still high, the interrupt is triggered again.

- DERIVED (default): PSoC Creator determines whether to use RISING_EDGE or LEVEL depending on the input to the Interrupt component.

Components may also have interrupts built-in to them called IRQs. For example, the SAR ADC component used in lab 1 has an IRQ that is triggered when the ADC completes a conversion.

All interrupts, excluding a few special cases (such as reset), are disabled on startup and must be enabled individually. Additionally, all of these interrupts remain disabled until calling the macro CyGlobalIntEnable and can be disabled again with CyGlobalIntDisable. Note that CyGlobalIntDisable simply stops the CPU from responding to interrupts. Any incoming interrupts are set to pending to be handled when interrupts are enabled again. The ClearPending function may be used to clear out an interrupt's pending state if desired.

Many interrupts that are connected to the interrupt output of a component or are IRQs require notifying the component that the interrupt has been handled. Otherwise, the interrupt output will remain high and either cause continuous interrupts if the type is level or no more interrupts if the type is rising edge. In many cases, reading the status register of a component clears the interrupt and drops its interrupt output low.

Interrupts can be further configured in the design-wide resources editor under the Interrupts tab. This view lists all of the interrupts, their priority, and their vector.

When an interrupt occurs, there are a few things that can happen. If an ISR is already executing and the new interrupt has a higher priority, the first ISR is preempted and the new interrupt runs. Execution returns to the first ISR after the new interrupt finishes. If the new interrupt has the same or lower priority, it waits until the first one finishes. If two interrupts occur simultaneously, the one with higher priority, or the one with the lower vector if both have the same priority, goes first.

When configuring the priority, the largest value (7) is the lowest priority and the smallest (0) is the highest. The vector cannot be configured. ISR entries (or pointers) are listed in a dedicated array, and the vector is an index in that array. The array holds one entry for each of 32 interrupt lines, so the number of unique interrupts is limited to 32.

After generating the project, the Interrupt component creates a .c file with the same name given to the component. Along with the functions created to configure the component, the ISR is one of these functions, with the signature CY_ISR(ISR_Name_Interrupt) (where ISR_Name is the name given to the component). This .c file may be modified between the #START and #END comments within that function to customize what the ISR does, and this code won't be touched if the project is generated again later. There is another #START and #END comment pair at the top of the file where you can add custom includes, defines, and code.

Components with IRQs create a folder with the same type of .c file for the IRQ. For example, the ADC SAR component creates a folder named ADC_IRQ, where ADC is the name given to the component.

Interrupts may be triggered by the software if desired using the ISR_SetPending function.

Here are the most important functions in the programming interface. "ISR" is replaced with the name given to the component in the schematic view.

void ISR_Start()	Sets up the interrupt and enables it
void ISR_StartEx (cyisraddress address)	Same as ISR_Start, except the vector address is set to a different ISR function. Useful if it is preferable to put the ISR in main.c or a different C file.
void ISR_Stop()	Disables the interrupt and resets its configuration
void ISR_Interrupt()	The default ISR used by ISR_Start. This may be edited in the .c file.

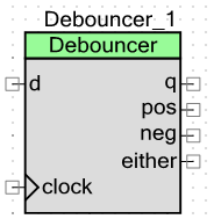
void ISR_SetPriority (uint8 priority)	Sets the priority of the interrupt to something different from default. Lowest priority is 7 and highest priority is 0. Overridden by ISR_Start
void ISR_Enable()	Enables the interrupt. Ensure vector and priority is set (Start does this)
void ISR_Disable()	Disables the interrupt
void ISR_SetPending()	Triggers the interrupt (if it's enabled) by setting it to pending execution
void ISR_ClearPending()	If the interrupt is pending execution, clear its pending state

For further information, see chapter 7: Interrupt Controller of the PSoC 5LP Architecture Technical Reference Manual (document 001-78426; can be found using Help>Documentation>PSoC Technical Reference Manuals in PSoC Creator).

You may now start Procedure Part A. Return here for background for Part B.

B2: Debouncer Component

Physical switches do not always cleanly transition between the pressed and released states. There can be a brief period of time where the output of a button quickly oscillates between states before it settles, which could cause parts of the system to detect multiple state changes. Since it could cause erratic or unexpected behavior, it is best to connect the switch to a debouncer, which filters out oscillations by sampling the state of the pushbutton at the rising edge of a slow clock. If the time it takes a pushbutton to transition between states is less than the speed of the clock, then that is sufficient most of the time. Note that a debouncer by design will delay the detection of a state change press, so debouncers should not be used in applications where reaction time matters, such as the stopwatch.



Here are the inputs and outputs for the component:

- Inputs
  - d: the input that is sampled
  - clock: the clock used to sample the input. The frequency of this clock should be set to the anticipated transition time of d. Typically, it should be between 10 and 200 Hz.
- Outputs
  - q: the filtered output value, which is the value of d at the latest rising edge of the clock
  - pos: indicates that a positive edge was detected by going high one clock cycle after the transition and staying high for one clock cycle
  - neg: same as pos, except for the negative edge
  - either: same as pos, except for either edge

There are only two configuration options:

- Signal width (bits): if multiple signals need to be debounced, use this to increase how many inputs and outputs there are. This utilizes buses (multiple signals per wire in the schematic) for the inputs and outputs.

- Edge type: unchecking boxes will hide any unused outputs from this component, except q

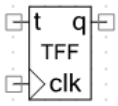
This component has no programming interface; it exists purely in hardware. Other components, such as interrupts or registers, should be used if the debouncer needs to be used in the software.

Important note: because pushbutton inputs are resistive pull-up and a released button is read as logic high, it is best to run a pushbutton input pin through an inverter first. Otherwise, since the debouncer component is initialized to 0 on reset, a released button on startup will generate a false positive transition event.

B3: Flip-Flops

A flip-flop is a component that is used to store one bit of information. In PSoC Creator, a flip-flop component can be configured to hold multiple bits using the ArrayWidth parameter, but behind the scenes it creates one flip-flop per bit. The way a flip-flop stores a bit depends on the type of flip-flop used. All flip-flops do not have software interfaces.

Toggle Flip Flop



A toggle flip-flop stores a bit that toggles between low and high states depending on the input t, synchronized to the clock. When t is high at the rising edge of the clock, the output q is toggled. When the input is low, q stays the same. There are no configuration options other than ArrayWidth.

Other flip-flops are covered in Lab 6.

Procedure Part A

Outcome

In this lab, the PSoC will be programmed in two parts. This first part will use an interrupt to count how many times SW2 is pressed.

Part 1: Designing the System

1. Create a new project in your workspace named Lab3 and open its TopDesign.cysch.
2. Add a digital input pin named SW2 configured to be resistive pull-up.
3. Add a System>Interrupt to the schematic area, name it SW2ISR, and configure it to be RISING_EDGE type.
4. Connect SW2 to a NOT gate, then connect the NOT gate to SW2ISR.
5. Add a Character LCD and name it Display.
6. Open Lab3.cydwr and configure SW2 to use port P6[1] and configure Display to use P2[6:0].
7. Generate the application.

Part 2: Programming the Firmware

Outcome

The hardware generates an interrupt on every rising edge caused by (the inverted) SW2 being depressed. In the ISR, the software increments a counter that keeps track of how many button presses there are. In the main function, the software updates the LCD display with the count whenever it changes.

SW2ISR

Open the following file in Lab3: Generated_Source/PSoC5/SW2ISR/SW2ISR.c
ISRs are one of the few generated source files that are designed to be edited. Only add code between the #START and #END comments in this file; any code added outside of these comments will be removed after the next code generation.
Define two global variables between the #START and #END comments at the top of the file:

```
uint8 sw2_pressed
uint16 sw2_count
```

Inside the function SW2ISR_Interrupt (the signature is CY_ISR(SW2ISR_Interrupt)) and between the #START and #END comments, set sw2_pressed to 1 and increment sw2_count.

Initialization

In main.c, add references to the global variables to the top of the file.

As a reminder, to reference global variables that are defined in a separate C file, use the same declaration and add the keyword extern. For example:

extern uint8 sw2_pressed;

In the main function, set sw2_count to 0 and sw2_pressed to 1 (allows for the main loop to print the initial count).
Call the Start functions of Display and SW2ISR. Print "SW2 Count" to the first line of the display. Since the initial state of the SW2 pin might be low and the NOT gate makes it high, the SW2ISR interrupt is probably already pending. Clear it using the SW2ISR_ClearPending function. Enable interrupts using CyGlobalIntEnable (this is a macro, not a function, so don't use parentheses).

Main Loop

Check to see if sw2_pressed is non-zero. If it is:

- Disable interrupts using the CyGlobalIntDisable macro. This should be done because the contents of this if statement is a critical section (stuff modified in the ISR is being modified here). Incoming interrupts will be set to pending and will be handled when interrupts are enabled again. Note that each interrupt can only be marked pending once, so an interrupt marked pending will only be handled once when interrupts are re-enabled even if it was triggered multiple times. In this application, there shouldn't be multiple interrupts within the time the interrupts will be disabled.

- Set sw2_pressed to false (zero).
- Print sw2_count to the display on the second line using Display_PrintDecUint16.
- Enable interrupts using CyGlobalIntEnable.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Test the program to make sure it works:
    - Starting out, the display should say “SW2 Count” and 0
    - Press SW2. The number on the display should increment.
3. Note that the number sometimes increments more than once. This happens because the pushbutton doesn’t always cleanly transition between states. There tends to be a small period of time where the button oscillates between being pressed and not pressed. The next part of the lab will fix this issue.

Procedure Part B

Outcome

In this part of the lab, the SW2 will be debounced so every press is counted only once. Additionally, a toggle flip-flop will be used to toggle an LED on and off with every press of SW2.

Part 1: Designing the System

1. Reopen TopDesign.cysch
2. Delete the wire connecting the NOT gate to SW2ISR.
3. Add a Digital>Utility>Debouncer to the schematic area, name it SW2Debounce, and uncheck Negative and Either edges in the configuration to hide their respective outputs.
4. Connect the output of the NOT gate to the d input of SW2Debounce.
5. Add a new clock to the schematic, name it SW2Debounce_clock, and configure it to be 60 Hz.
6. Connect SW2Debounce_clock to the clock input of SW2Debounce.
7. Connect the pos output of SW2Debounce to SW2ISR.
8. Add a Digital>Logic>Toggle Flip Flop (TFF) to the schematic.
9. Connect the pos output of SW2Debounce to the t input of the TFF and connect SW2Debounce_clock to the clk input of the TFF.
10. Create a digital output pin named LED4 and connect it to the q output of the TFF.
11. Open Lab3.cydwr and configure LED4 to use port P6[3].
12. Generate the application.

Part 2: Programming the Firmware

There is nothing to change in the firmware, as all the components added either have no software interface or start working on their own. There is no longer a need to clear the pending flag of the interrupt during initialization now that SW2 is debounced, but removing that code is not necessary.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Test the program to make sure it works:
    - Starting out, the display should say “SW2 Count” and 0 and LED4 should be off.
    - Press SW2. The number on the display should become 1 and LED4 should turn on.
    - Press SW2. The number on the display should become 2 and LED4 should turn off.
    - Pressing SW2 should always increment the number on the display by 1 and toggle the LED. The LED should only be on when the number is odd and off when the number is even. The display should only increment on the depress and only once per press.