Lab 11

Concurrent Event Handling

In this lab, we will learn programming concurrent events that are processed with interrupts. This lab will show us how to program multiple interrupt events that interact with one another and how to program interrupts that are enabled/disabled in various phases of the code.

11.1 HVAC Control (non-renewing delayed toggle)

In this part, we will write a program that takes user input and controls the compressor of an HVAC system. The compressor of an HVAC is damaged if it's cycled on/off at a high frequency (e.g. toggled every second). The program we'll write takes user input, represented by the button, and toggles the compressor three seconds after the button is pushed. The compressor is represented by the red LED. This mechanism is illustrated in Figure 11.1. As the figure shows, the button is not processed if it's pushed during the 3-second interval. Throughout the program, the green LED flashes every 0.5 second to indicate that the power is on.

Your program should satisfy the following requirements:

- Run the timer in the continuous mode and use the channels to time events.
- Use Channel 1 with interrupts to time the 0.5-second interval that flashes the green LED.

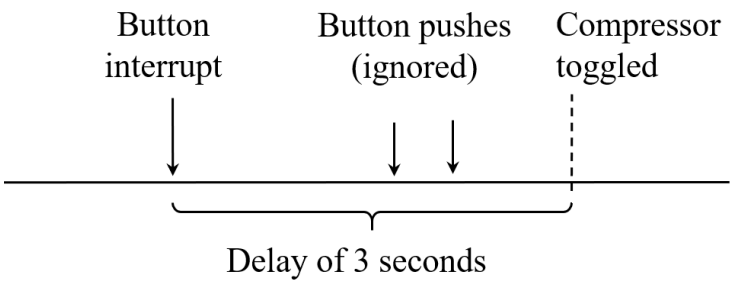

Figure 11.1: The compressor is toggled three seconds after the button is pushed

- Use Channel 2 with interrupts to time the 3-second interval when it's needed.
- When Channel 2 is not being used, its interrupt enable bit (xIE) must be set to zero.
- The button is interfaced with interrupts.
- During the 3-second interval, the button's enable bit (xIE) must be set to zero.
- Engage a low-power mode.

The requirements above are meant to practice enabling and disabling interrupt during various phases of the program.

The following observation is helpful when writing this code. In the pseudocode below, two interrupt events share an ISR and are processed with two if- statements. The first interrupt event is always enabled (xIE=1) in the program, therefore, it's sufficient to check that its flag is high before it's processed. On the other hand, the second event is enabled/disabled in various phases of the program. Therefore, it is serviced if it is currently enabled and its flag is currently high. If we don't check xIE, it's possible that the ISR is called due to the first event; if the second event is currently disabled (xIE=0) and its flag is high, it will end up being processed, which is not supposed to happen.

```
void Timer_A1_ISR() {
  // Detect Channel 1 (Channel 1 interrupt is always enabled)
  if(channel 1 xIFG =1)
    ...

  // Detect Channel 2 (Channel 2 interrupt is enabled/disabled in various
     phases)
  if(channel 2 xIE=1  AND  channel 2 xIFG =1)
    ...
}
```

It is helpful to think about this code by drawing a timeline similar to Figure 11.1 and writing under each action a list of events to be done. That is, write a list of actions that should be done when the button interrupt occurs, and another list of actions that is done when the timer interrupt occurs.

Perform the following:

- Complete the code and demo it to the TA.
- Submit the code in your report.

11.2 HVAC Control (renewing delayed toggle)

Modify the code of the previous part so that the 3-second interval renews if the button is pushed. For example, if the user pushes the button at t=0s, the toggle is supposed to happen at t=3s. However, if the button is pushed again at t=2s, the toggle is delayed until t=5s. If the user keeps pushing the button within the 3-second interval, the toggle will keep getting delayed.

Perform the following:

- Write the code and demo it to the TA.
- Submit the code in your report.

11.3 Button Debouncing

In this part, we will implement a button debouncing algorithm that makes the buttons of our board more reliable. In an earlier lab, we wrote a code that toggles the LED when a button interrupt occurs and we observed that the button doesn't work all the time because the bounces raise additional interrupts that cancel out the toggle operation.

The debouncing algorithm is illustrated in Figure 11.2. This algorithm takes two samples of the button that are separated by the maximum bounce duration. As the figure shows, when the button is pushed, the first falling edge (button is active low) raises an interrupt. At this point, we will start the timer for 20 ms, which represents the maximum bounce duration, so that we wait out all the bounces. During this interval, the button interrupt is disabled to avoid causing further interrupts. At the end of the 20 ms interval, if the button is still pushed, we'll interpret a button push and take the necessary action, e.g. toggle the LED.

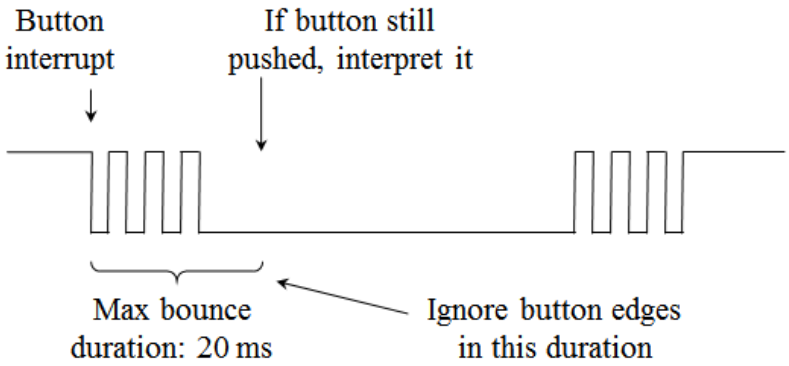

Figure 11.2: Button debouncing algorithm

A relevant question is what happens when the button is released? At release, the first falling edge triggers the same procedure. After waiting for the maximum bounce duration, however, the button will be found to be released and a button push is not interpreted.

We will start by assuming that the maximum bounce duration is 20 ms, however, we will find this value experimentally. High quality buttons have a short bounce duration and buttons with built-in debouncers exist although at a higher price point. Simple buttons like the ones on our board may not have a data sheet, therefore, we'll experiment with the maximum bounce duration in order to find the smallest such delay that makes the button reliable. If the button is not reliable when we set the maximum bounce duration to 20 ms, we should increase it. Otherwise, we can try decreasing this duration as long as the button remains reliable. It's possible that the two buttons on our board may have differing maximum bounce durations. Note that our algorithm effectively introduces a 20 ms response delay which should not be noticeable to the user.

Implement this algorithm so that the LED is toggled when the button is pushed. Use interrupts for the button and the timer. The interrupts of the button and the timer must be disabled (xIE=0) when they are not needed. Engage a low-power mode. Experiment with the maximum bounce duration to find the smallest duration with which the code works reliably.

It is helpful to think about this code by drawing a timeline similar to Figure 11.2 and writing under each action a list of events to be done. That is, write a list of actions that should be done when the button interrupt occurs, and another list of actions that is done when the timer interrupt occurs.

Perform the following and submit the answers in your report:

- Write the program and test it. Push the button 30 or 40 times and make sure it works every time.
- Demo your program to the TA
- What is the maximum bounce duration that is set in your code?
- Submit the code in your report.

Student Q&A

1. An interrupt uses a shared ISR and is always enabled in the program. What does the if-statement in the ISR check for before servicing this interrupt?
2. An interrupt uses a shared ISR and is enabled/disabled in various phases of the program. What does the if-statement in the ISR check for before servicing this interrupt?
3. For the debouncing algorithm that we implemented, is it possible that the LED will be toggled when the button is released? Explain.
4. If two random pulses occur on the push button line due to noise and these pulses are separated by the maximum bounce duration, will our debouncing algorithm fail? Explain.