CAB202 Tutorial 9
Switch debouncing, timers, and interrupts

Overview

Execution on computing systems naturally runs on a single linear thread (*i.e.* one instruction after another). Throughout this unit you have most likely encountered a number of situations where you have had to construct solutions to essentially emulate running two processes in parallel. These convoluted solutions are difficult to manage, add unnecessary complication to coded solutions, and in some cases aren't even feasible. Interrupts confront these challenges by facilitating event driven programming on microcontrollers. This allows parallel processes to run (or the illusion of such) on a linear thread by essentially checking and obeying execution priorities and conditions. This tutorial will also cover button debouncing techniques and interfacing with hardware based timers. This tutorial is worth 4% of your final mark for this subject.

Switch Debouncing

A switch is a mechanical component, and as a result there is often mechanical noise caused by contact bouncing. Contact bouncing is a physical problem that arises when the switch's contacts come together. This connection is not instantaneous and consequently noise is generated. This noise causes bouncing, which means that the output from the switch bounces between logical high (i.e. the high voltage) and logical low (i.e. the low voltage which is often ground). As a result of this bouncing, the pin we are interested in reading goes rapidly and repeatedly between a circuit's high voltage rail and its low voltage rail. This can cause problems when reading switches as a single press can be registered as multiple presses by the microcontroller. Fortunately this can be overcome through switch debouncing, which is typically performed in software implementations (can also be performed at the hardware level). In both software and hardware there are multiple ways of dealing with the problem, as is explained on the second page of this link: http://www.ganssle.com/debouncing.htm. The hardware provided for this unit does not have hardware bouncing in the physical circuitry and consequently debouncing will have to be implemented with software. Switch debouncing is extremely important in producing microcontroller programs that are responsive to input.

Timers

Each of the AVR microcontrollers typically has two or more timers. The Atmega32u4 chip has five timers. Generally, each of the timers comprises of slightly different hardware. As a result they often have different resolutions and consequently different applications. On the Atmega32u4 there is one 8-bit timer and two 16-bit timers. The predominant difference between an 8-bit timer and a 16-bit timer is the resolution (consider how many different values you can represent with 8 binary bits vs 16 binary bits). Another difference is that the 16-bit timers have more possible modes of operation than the 8-bit timers. For most AVR timers there are four modes of operation: 1) **normal**, 2) **clear timer on compare**, 3) **fast PWM**, and 4) **phase correct PWM**.

The difference between Polling and Interrupts

In all of the solutions to the tutorials throughout this semester, the methods that you have made use of have been what is known as **polling**. A polling method loops over the same code, checking if a particular state has changed (*i.e.* performing a desired action by continuously checking if a button has been pressed). In other words, when we are polling we are waiting for a specific event. This action is considered blocking due to the fact that no other code can be run until this condition is met. Consequently, this blocking code can cause problems in larger embedded systems where you have multiple inputs and outputs. It is in scenarios like these where the distinct advantage of interrupts is apparent. Interrupts can be thought of as event handlers (*i.e.* they will only execute when a certain event has occurred). Obviously, there are a number of different types of events that can occur on a microcontroller (e.g. UART data transmission, timing, overflow, ADC, etc). As a result, there are many different types of interrupts in the AVR microcontroller architecture that can be used to drive the execution of specific blocks of code.

Interrupt Operation

Interrupts are a crucial part of writing code for embedded systems. As the name suggests, interrupts halt the execution of the main program and jumps to a prioritised section of code. This allows the immediate execution of functions when a specific hardware event occurs. In performing the immediate execution, the microcontroller goes through the following three steps:

1. Halts the current process (noting where it is in this process)
2. Runs the interrupt code until it completes
3. Returns back to the original process and continue from where it was before the interrupt

In AVR microcontroller programming, the interrupt code is called the interrupt service routine (ISR). The syntax for code that creates an ISR is similar to that of a function (except it does not need a declaration – only an implementation). An example of an ISR is shown below for the USART receive complete interrupt event (you must include **`avr/interrupt.h`** to run any interrupt related code):

```
ISR(USART_RXC_vect) {
     // Code to be executed within the interrupt routine
}
```

You can think of the ISR as basically an isolated section of code which will get called anytime an event occurs. Because this code blocks the execution of the main process, you must carefully consider what code to run within the interrupt. If you place code with a long execution time in an interrupt, you are essentially just reversing things so that your interrupt is blocking the execution of your main process (whereas before you had the main process blocking your extra process). **This defeats the purpose of using an interrupt!**

As already discussed, there are many different types of interrupts. Each type has as its own interrupt vector. In the above example, the interrupt vector is the **`USART_RXC_vect`** part. It is this part of the ISR declaration code that would need to be changed for a different interrupt event.

There are three important conditions that must be met for the ISR to be called and executed correctly:

1. The enable bit for global interrupts must be set. This allows the microcontroller to process interrupts via ISRs when set, and prevents them from running when cleared. It defaults to being cleared on power up, so we need to set it by using the **`sei()`** utility function. Conversely, you can clear it by using the **`cli()`** utility function.
2. The individual interrupt source's enable bit must explicitly be set. Each hardware interrupt source has its own separate interrupt enable bit (this resides in the related peripheral's control registers).
3. The condition for the interrupt must be met. For example, a character must have been received through USART for the USART receive complete (USART_RXC) interrupt to be executed.

Assessable Exercises

NOTE: All questions are to be marked in your allocated tutorial session only!

1. Complete the code to detect debounced button presses (1 mark)

Download the **`question_1.c`** source file from Blackboard. The code currently displays a press count that doesn't change with any button presses. When the function **`check_presses_debounced()`** is called, the global variable `press_count` should be incremented if the right button was pressed, and decremented if the left button was pressed. This must only happen on the release of the button (i.e. on the falling edge), and only once per physical button press. Implement a software debouncing solution in the **`check_presses_debounced()`** function.

2. Use a timer to print the system clock to the screen (1 mark)

Download the **`question_2.c`** source file from Blackboard. The code currently provides a verbose suggestion that it displays a system time, but only displays a default value. Complete the sections in **`init_hardware()`** to configure `TIMER1` in normal mode, and prescaled such that its maximum value (i.e. overflow) equates to approximately 8.3 seconds. Implement **`get_system_time()`** such that it returns the real time based on the supplied value of `TIMER1`, and update the main loop so that this value is correctly printed to the screen. When the right button is pressed, `pause_while_pressed()` should pause your program until released. When released, the time display still must be correct.

3. Use a time-based interrupt to flash an LED at a constant rate (1 mark)

Download the **`question_3.c`** source file from Blackboard. The code currently does exactly the same as the previous question, but will show the timer overflowing approximately every 2.1 seconds. When this overflow occurs, the state of the LED must be toggled. Configure `TIMER1` in **`init_hardware()`** so that toggling the LED in the overflow interrupt service routine (**`ISR()`**) for `TIMER1` will produce the required behaviour. As in question 2, holding a button down must stop the count on the screen, but **must not** stop the LED from toggling!

Note: demonstrating this exercise to your tutor will award you the marks for question 2

4. Using interrupts for robust, non-blocking button press detection (1 mark)

Another way button presses can be detected is to take a more pragmatic approach. In this question consider that a button can only have two possible states (**UP** and **DOWN**) and must be in one of these states at all times. We then use strict conditions to define when a button should transition between states. This question uses bit-packed histories (a sequence of bits representing the button's value over time) as robust conditions for the change between states. A button cannot change its state unless every bit in the history is **opposite** to its current state (for example, if a button's current state is **UP** it can't change to **DOWN** until the history contains all 0's). By also making use of interrupts, we now have a system that is robust against signal bouncing, detects presses without blocking your code, and is prioritised above normal code execution.

Download the **`question_4.c`** source file from Blackboard. You must use an interrupt, and bit-packed state histories, to implement robust state-based button press detection. The code displays the current press count in the top left, value of a timer in the top right, and a visual representation of the button states on the bottom of the screen. Complete the following:

- Use your code from question 2 (in **`init_hardware()`** and **`get_system_time()`**), to display a system time in the top right corner (overflowing approximately every 8.3 seconds).
- Correctly initialise all of the buttons as inputs, and set up `TIMER0` to overflow approximately every 8 milliseconds.
- In the interrupt for `TIMER0`, do the following:
    - Shift each of the bytes in **`btn_hists`** left one place
    - Put the current value of each button pin into the LSB (least significant bit) of the corresponding byte in **`btn_hists`**. Make use of the `#define` directives given to you.
    - Update the state of each button in `btn_states` if necessary. A button's state should change to **`BTN_STATE_DOWN`** if its current state is BTN_STATE_UP, and if every bit in the history byte is high. Likewise, if a button's current state is BTN_STATE_DOWN, and every bit in the history byte is low, the state should change to **`BTN_STATE_UP`**.
    - If the state has changed from BTN_STATE_DOWN to BTN_STATE_UP a press should be registered (increment `press_count`).

If you've completed this correctly, the following must happen:

- The time display is never interrupted by a button press under any circumstances
- The press count only ever increases once per button press
- The visualisation should never flicker – the rectangles fill in when a button is pressed down and empty when it is released
- Any number of simultaneous button presses can be handled without any interference

Note: demonstrating this exercise to your tutor will award you the marks for questions 1, 2, and 3