ELEC3042 Embedded Systems

Practical 3

Timers and Timer Interrupts

Laboratory Equipment
Personal Equipment Kit

Software
MPLAB X

Aims
The aims of this practical are:

- To utilise the timer and interrupt hardware units on the Atmel ATmega328P.

Pre-Work

- Complete the self-assessment quiz

Outcomes
After completing this practical, you will have the following knowledge and skills:

- Knowledge of the architecture and functionality of the Atmel ATmega328
- Beginning understanding of timers

There are three parts to this practical. All three parts should be completed.

	Completed
Part 1	Done
Part 2	
Part 3	

Part 1: Timers and Interrupts

Introduction

The ATmega328 has three internal timers. Timers 0 and 2 are 8-bit timers, while timer 1 is a 16-bit timer. Each timer has various modes of operation and three interrupt generators. Each timer has some special features; Timer 1 is 16-bits, and can thus handle larger numbers, it can also run as a counter to count external events. Timer 2 can run asynchronously to the I/O clock, allowing it to run even while the rest of the system is asleep.

In general, the timer that is used depends on several factors:

- Whether an external output is required, hence which pins are available
- The size of the divisor
- Whether an external clock source is required
- If a PWM output is required, what accuracy is needed

The answers to these choices determine which timer should be used.

In general, the timer/counter works by counting clock pulses. In its simplest form it counts up starting from 0 and when it reaches all 1s the next clock pulse overflows the counter and it changes to all 0s again. On the overflow, an overflow interrupt is generated. In this mode of operation, the clock is sourced from either the I/O clock, or an external clock. If the I/O clock is utilised, there is a five position prescaler that can select either the I/O clock or the I/O clock divided by 8, 64, 256 or 1024. If this is all that is required from the timer then the rest of the timer configuration can be ignored.

Sometimes, we may want to count for a shorter period. In this case, we can use the CTC mode where the timer counts to the value set in the OCRxA register and then resets the count to zero.

As mentioned in the lectorials and practical 2, internal hardware can also create interrupts. We will use a timer to create an interrupt, which we will use to emulate a button being pressed.

In this case we need to do three things, firstly: configure the timer, secondly: change the Interrupt routine to watch the timer's interrupt vector and thirdly: use the timer interrupts to "press" or "release" the button; to toggle the button.

A framework has been provided for this in the file `ELEC3042_L3P1_timer_interrupt.c`. The framework uses timer 1 for the lab. We will use the timer to generate an internal interrupt which we will use as a simulated button press/release.

To configure the timer, we read the datasheet (Section 16). It is recommended that you use CTC mode (Waveform Mode 4), with a clock prescaler of 64. We will use the OCR1A compare register to create the interrupt.

Setting OCR1A sets the value the counter has to count before the interrupt will occur and the pseudo button is toggled in state. The smaller OCR1A is set, the faster the pseudo button will be toggled. Initially set OCR1A to 65535, but afterwards change it to different values to verify your code is working as expected.

Part 2: Busy Loops

Busy loops are loops in the code that do nothing useful. They are placed in the code to use up CPU cycles until some event needs to occur. The delay_ms function in ELEC3042_L3P1_timer_interrupt.c. is an example of a busy loop, where the for-loop runs for 1 ms doing nothing useful. Every time the for loop finishes, the value in num is decremented until it reaches zero. Effectively, the delay_ms function does no useful work for num worth of milliseconds, which is rather inefficient.

Rather than using a busy loop, we can set up a timer to generate an interrupt every 1 ms. In the interrupt service routine, we can increment a count variable that stores the number of milliseconds that have passed. Then, in the main loop, we can check the count variable to see if the desired number of milliseconds have passed before the next event should occur.

A framework has been provided for this in the file ELEC3042_L3P2_timer_blink.c. Complete the code by setting up timer 1 in CTC mode and choose an appropriate prescaler and OCR1A value such that an interrupt is generated every 1 ms. When you run the code, you should see the LED blink every 1 second.

Make sure you understand how the 1 s blinking has been achieved without needing to use a busy loop. Can you think of another way to achieve the same effect without needing to keep checking (also called polling) the count variable?

Part 3: Exercise

Create a program that will count button presses of A1 and show the count as a binary sequence using the LEDs on the HW-262. When implementing the debouncing, make sure that there are no busy loops in your code.