Accurate Time and Interrupts

Matthew Beckler
beck0778@umn.edu
EE2361 Lab Section 007

March 7, 2006

Abstract

In this lab, we create a very accurate digital clock using one of the microcontroller's timers. We will use interrupts to provide a structured code enviroment, and utilize the dual digit, seven-segment display from the previous experiment.

Introduction

It can be very difficult to accurately measure a duration of time using a microcontroller, especially when you are trying to accomplish other work between ticks. Without using the hardware timers, making an accurate loop requires careful counting of the instructions inside the loop. If timers are used, you are guaranteed that the proper flag will be set at the proper time. Normally, this is accomplished through the use of a while loop, waiting for the proper bit to change. Using interrupts simplifies this process greatly. To utilize the interrupts, you first need to define the interrup handlers, `low_isr` and `high_isr`. In this lab, we will only be using the high-priority interrupt handler. While waiting for the interrupt, we will be updating the dual-digit seven-segment displays, using the same alternating, multiplexed display as the previous lab.

For recording the digits to display, we store both digits in a single byte. We stored the least-significant digit in the lower nybble, and the most-significant digit in the upper nybble.[1] We will be using the binary coded digit encoding for storing both digits in the register.

We will also be investigating the accuracy of the timer, both from an analysis of the software in the simulator, and from real-world investigations and measurements. For the actual measurements, we will be measuring how much real time must elapse to observe one second of error. These results will be displayed later in this document.

Observations

For the scope of the main program, we have created a number of helper functions. The `void display(void)` function has returned from the previous lab.

[1] For a discussion of *nibble* vs. *nybble*, see The New Hacker's Dictionary, Second Edition. 1993. MIT Press, Cambridge, Massachusetts. p. 306

The initial values of the digits have been changed from all segments off, to displaying 0. This is how the clock will start, and is a better choice than to start off blank, then jump right to 01. We have removed the hexadecimal digits from our tabular data, because the characters for `A B C D E F` will not be used. Other than these small changes, the `display()` function is the same as before.

Since we are using interrupts in our program's source code, a brief explanation of the PIC microcontroller's implimentation of interrupts is is necessary. The microcontroller has the ability to use two classes of interrupts, known as *low-priority* and *high-priority* interrupts. For most applications and devices, only one class of interrupts is necessary. Under normal operation, the microcontroller starts execution at program memory location 0. When interrupts are in use, two additional addresses in program memory have special meaning. When an interrupt occurs, the processor will jump the program counter to one of two *interrupt handlers*, which are specific locations in program memory. The high-priority interrupt handler starts at program memory location 0x8, and the low-priority interrupt handler starts at program memory location 0x18. Normally, there is not enough space to store your complete interrupt handling code in these locations, so a `goto` instruction is often used to branch somewhere else in the program memory. In this experiment, we will only be using the high-priority interrupts.

A new class of statements for this experiment are the `#pragma` directives, which are instructions to the compiler, suggesting how to compile the program. We are using the `#pragma` directives to specify where the compiler places our program code. As explained in the previous section, the interrupt handling code must be placed in specific places in the PIC's program memory. We use the `#pragma` directive to place the interrupt handlers in the correct place. Here is a code exerpt from the main program, demonstrating the embedded assembly that is used to branch the execution to a location in program memory with more room for the actual interrupt handlers:

```
//Instruct the compiler to place the following code at 0x8
#pragma code high_isr_entry=0x8
void high_isr_entry(void)
{
        //embedded assembly code
        _asm GOTO high_isr _endasm
}

//Instruct the compiler to place the following code at 0x18
#pragma code low_isr_entry=0x18
void low_isr_entry(void)
{
        //embedded assembly code
        _asm GOTO low_isr _endasm
}

//Give code placement back to the compiler
#pragma code
```

Nearly every microcontroller includes timing devices on-chip. The PIC18F452 is no exception, with a total of four timers available, with many features and

options to customize their operation. For this experiment, we will be using the `TIMER0` device. This is a 16-bit counter/timer, with internal/external clock inputs, variable edge-detection settings, and an 8-bit programmable prescaler. In 16-bit mode, Timer 0 has two registers that store the current count, `TMR0L` and `TMR0H`. The high byte is latched, so reading/writing from/to `TMR0L` will do the same operation between `TMR0H`(the latched register) and the actual register. Another important feature of Timer 0 is the ability to pre-set the count to customize the time to interrupt. This is done by loading the uppper byte of the desired value into `TMR0H`, and writing the lower byte of the desired value into `TMR0L`.

At the start of the `void main(void)` function, we initialize the `TIMER0` device. There are a number of options available for Timer 0; our timer is setup for 16-bit mode with a pre-scaler of 1. This is done by writing the value `0x80` into the `T0CON` register. We also must enable global interrupts and the Timer 0 interrupt. We will be loading a starting count of `60543 = 0xEC7F` into the `TMR0` registers. We must load this value at the start of the program, as well as every time an interrupt occurs. This value corresponds to a timer period of approximately 1 millisecond.

The output to the LED display is handled with `PORTC` for the 8 data lines, and `PORTD pins 0 & 1` for the digit select lines. We enable output for these ports and clear their values at the start of the program.

When the Timer 0 interrupt occurs, the high-priority interrupt handler is called. The first statements in this function reset the counter's value. This is very important in regards to the accuracy of the timer. We want to reset the counter's value as soon as possible immediately after it rolls-over, because any statements executed between roll-over and counter reset are not accounted for in the counter's value, and will affect the total time, albeit very slightly. For the rest of the interrupt handler, we are going to keep a variable to hold the number of milliseconds until the next whole second has elapsed. For 999/1000 interrupts, all we do is decrement the `milliseconds` variable from 1000 to 0. When it reaches 0, we first reset `milliseconds` to 1000, and increment our `count` variable, which is holding the number of elapsed seconds, modulo 100, in BCD packed-digit format. Since the `count` variable is really holding two values, we need to check when the lower nybble has reached a value of 10. When this happens, we need to produce a carry into the upper nybble, which we accomplish by adding 6 to `count`. We then check to see if `count` has just been incremented to the value of `100 = 0xA0`(using BDC packed-digit format). If we have reached 100, we reset `count` to 0x00. After adjusting `count`, we use its value to set the digits to be displayed.

When analyzing the accuracy of the clock, we have looked at two ways of measurement. When analyzing the accuracy of the clock using the MPLAB simulator's stopwatch feature, we found that the clock would be $\frac{159}{10{,}000} = 0.0159$ seconds fast per true second. This translates to one second fast in 62 minutes, 53.5849 seconds. This seems a bit large, as the clock would be 5.807 days fast after one year. To measure the real-world accuracy, we have chosen to run the timer against a (fairly) accurate digital wristwatch, and observe how much time must elapse until the wristwatch and the microcontroller clock differ by one second. This will be somewhat inaccurate, as we are depending on a mere mortal to identify when the clocks are exactaly one second off, which is quite difficult to do, and has a large margin of error. The value that we both agreed

upon was approximately 90 minutes, but the microcontroller clock was running slower than the reference clock. This equates to an error of 0.00185 seconds slow per true second, which is 16 hours, 13 minutes, 58.75 seconds slow per true year, which is more accurate than the simulator-based estimation.

In the interrupt handling function, we are using a total of 34 instruction clock cycles. Operating our processor at 40 MHz, this translates to 3,400ns, or $3.4\mu s$. Since we are resetting the count of Timer 0 as the first statements in the handler, the $3.4\mu s$ delay is not a factor in the accuracy of the timer.

Conclusion

Altogether, the timer project accomplished the stated goals. The clock keeps fairly accurate time, only loosing 1 second in 90 minutes. We have learned a great deal about the workings of the PIC's timers, as well as interrupt setup and handling. We have successfully used the `#pragma` directives to influence the compiler's operation.

Circuit Schematic

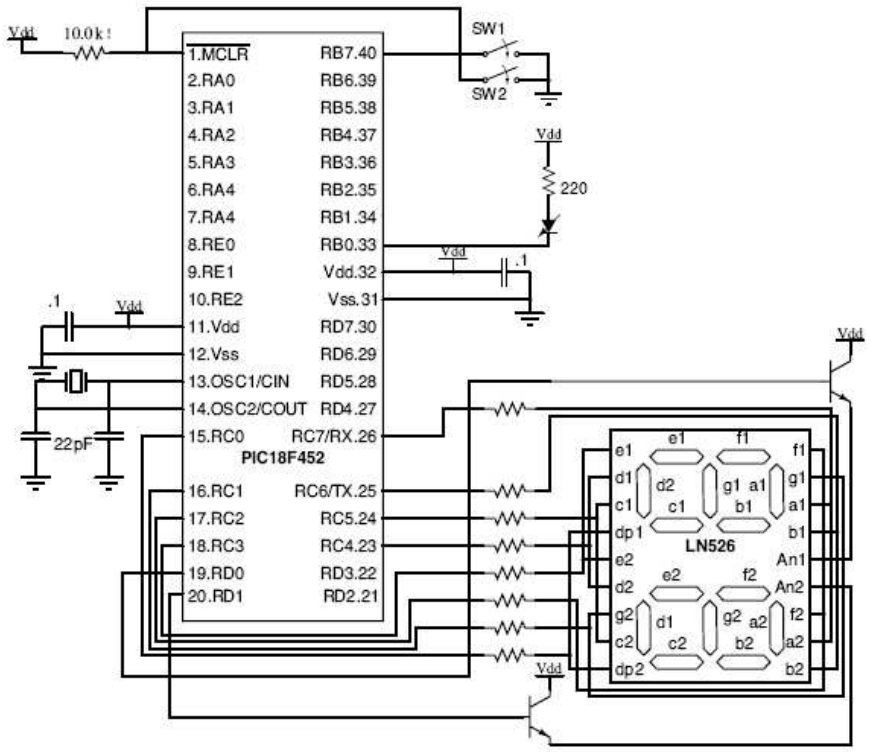

Figure 1: Circuit Schematic - Courtesy Professor Rennolet

C Language Program

```
#include <p18f452.h>
#pragma config OSC=HSPLL, WDT=OFF, BOR=OFF, PWRT=ON
#define TVAL 60543

void low_isr(void);
void high_isr(void);

static unsigned char values[] = {0, 0};
static unsigned char count = 0;
static unsigned int milliseconds = 1000;

//Instruct the compiler to place the
// following code at 0x8
#pragma code high_isr_entry=0x8
void high_isr_entry(void)
{
        //embedded assembly code
        _asm GOTO high_isr _endasm
}

//Instruct the compiler to place the
// following code at 0x18
#pragma code low_isr_entry=0x18
void low_isr_entry(void)
{
        //embedded assembly code
        _asm GOTO low_isr _endasm
}

//Give code placement back to the compiler
#pragma code

/* pragma for generating interrupt code */
#pragma interrupt high_isr
void high_isr(void) /* definition of high_isr */
{
        TMR0H = TVAL >> 8;
        TMR0L = TVAL & 255;
        milliseconds--;
        if (!milliseconds)
        {
                count++;
                milliseconds = 1000;
                if ((count & 0x0F) == 0x0A)
                {
                        count = count + 6;
                        //increments upper nybble
                        // & clears lower nybble
```

```
                }

                if (count == 0xA0)
                {
                        count = 0x00;
                        //reset the count
                }
                values[0] = (count & 0x0F);
                values[1] = (count >> 4);
        }
        INTCONbits.T0IF = 0;
}

#pragma interrupt low_isr
void low_isr(void){}

void display(void)
{
        static unsigned char numbers[] =
        {
                0b00000011,
                0b10011111,
                0b00100101,
                0b00001101,
                0b10011001,
                0b01001001,
                0b01000001,
                0b00011111,
                0b00000001,
                0b00001001
        };

        static unsigned char which = 0;

        if (!which)
        {
                PORTDbits.RD0 = 0;
                PORTC = numbers[values[0]];
                PORTDbits.RD1 = 1;
        }
        else
        {
                PORTDbits.RD1 = 0;
                PORTC = numbers[values[1]];
                PORTDbits.RD0 = 1;
        }

        which = !which;
        return;
}
```

```
void main(void)
{
/* enable timer 0 16 bit op w prescaler = 1 */
        T0CON = 0x80;
        INTCONbits.T0IF = 0;
        INTCONbits.T0IE = 1;
        INTCONbits.GIE = 1;
        TMR0H = TVAL >> 8;
        TMR0L = TVAL & 255;

        TRISC = 0; //7-segment outputs
        TRISD = 0; //segment-selector bits
        PORTC = 0;
        PORTD = 0;

        while(1)
        {
                display();
        }
}
```