Human Response Timer

Matthew Beckler
beck0778@umn.edu
EE2361 Lab Section 007

March 29, 2006

Abstract

In this lab, we create a very useful application, a human response timer. The user's reaction time is measured as the duration between observing a lit LED, and pressing a push-button switch. This duration is accurately measured using Timer0, and the computed value is displayed on two, 7-segment LED displays. When user error is encountered, customized error codes help guide the user to productive operation.

Introduction

In an attempt to create a useful project, this human response timer has been created. While working with timers, interrupts, and parallel I/O is not particularly new, the complexity of this project, and the way in which is combines all three has some inherent challenges. We set up a finite state machine to control the operation of the circuit. There are five states, with multiple paths between some of the states. A global integer variable controls the current state. Timer 0 is used to provide a one millisecond clock in conjunction with the interrupt handlers. In this lab, we will only be using the high-priority interrupt handler. The random number generating routine has been recycled, but slightly modified to produce integers between 1000 and 10000, which is used as the random wait time (in milliseconds), before lighting the LED. The 7-segment display routine has also been recycled, but modified to include a digit of **E**, for displaying coded error messages.

Observations

For the scope of the main program, we have created a number of helper functions. The `void display(void)` function has returned from previous labs. The initial values of the digits are set to off. A digit for the letter **E** has been added, to provide a method for displaying useful error codes. There are two error possibilities in this device, and they are enumerated **E0** and **E1**. 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.

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`.

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 very accurate timer period of 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. The LED is connected to RB0, and the switch is connected to RB7;

On initial power-on, the device's displays should be blank. When the switch is first pressed, the device will wait a random amount of time, between 1 and 10 seconds. If the user presses the switch before the LED is turned on, the error code **E0** is displayed. When the device has waited the random amount, the LED will be turned on. At this point, the device will start the timer on the response time. If the users does not press the switch within the next 999 miliseconds, the error code **E1** is displayed. Assuming the user pushes the switch withing 999 miliseconds, the processor waits for 1 second before displaying the user's reaction time. The number displayed is only two digits, and the digits are simply the most significant digits of the milisecond count.

A software-controlled finite state machine has been implimented for this device. A global integer value is used to control the finite state machine. Here is a summary of the states:

- **STATE 0:** Display digits, wait for button press, then proceed to state 1.
- **STATE 1:** Wait for button release, generate random wait time, then

proceed to state 2.

- **STATE 2:** Enable Timer0, wait for random time, or until premature button press, which leads to state 5. If the random time expires, set the LED and reset the timer to upward counting mode, then proceed to state 3.
- **STATE 3:** Wait for button press (go to state 5), or timer reaches 999 miliseconds (go to state 4); turn off LED.
- **STATE 4:** Wait one second to display results, update display digits, then proceed to state 5;
- **STATE 5:** Display digits, wait for button release, then return to state 0;

Conclusion

Altogether, this is the first truly useful project we have created in this course. It has summarized the implimentations of timers, counters, parallel I/O, and multiplexed displays. The use of a finite state machine has helped to streamline and compartmentalize the program's code into easily tested and understood sections. Program control flows logically from one state to the next depending on the inputs to the processor.

Circuit Schematic

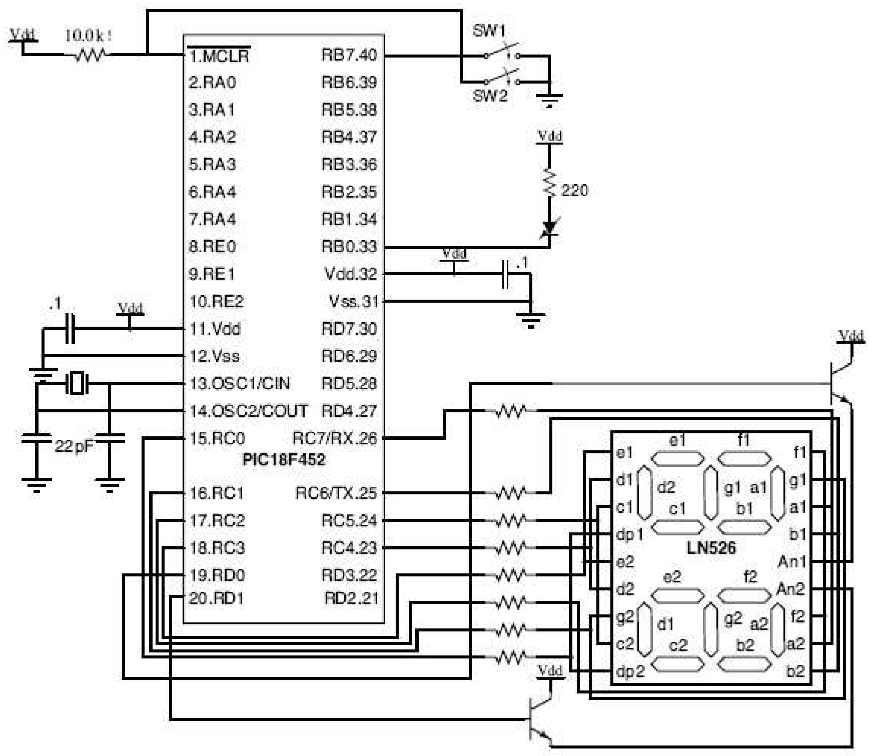

Figure 1: Circuit Schematic - Courtesy Professor Rennolet

FSM Flowchart

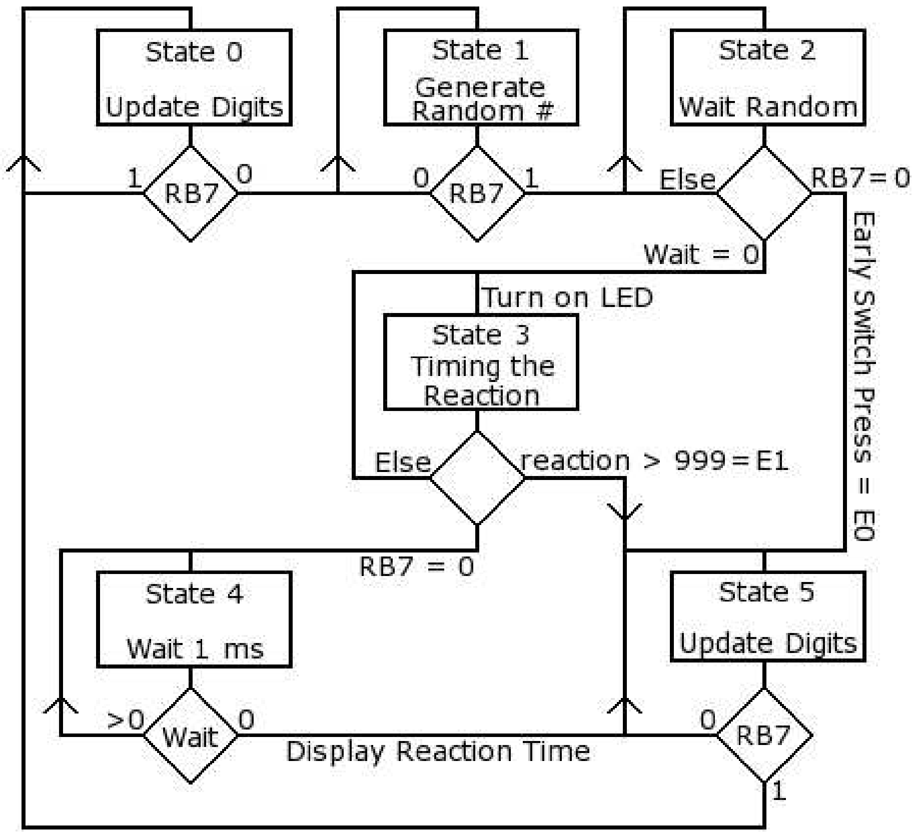

Figure 2: Flowchart for Finite State Machine

C Language Program

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

static unsigned volatile int state = 0;
static unsigned int wait = 0;
static unsigned int reaction = 0;

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

#pragma code high_isr_entry=0x8
void high_isr_entry(void)
{
    _asm GOTO high_isr _endasm
}

#pragma code low_isr_entry=0x18
void low_isr_entry(void)
{
    _asm GOTO low_isr _endasm
}

#pragma code

/* pragma for generating interrupt code */
#pragma interrupt high_isr
void high_isr(void)
{
    //reset to one millisecond
    TMR0H = TVAL >> 8;
    TMR0L = TVAL & 255;
    INTCONbits.T0IF = 0;

    switch(state)
    {
        case 3:
            reaction++;
        break;

        case 2:
        case 4:
            wait--;
        break;
    }
}

#pragma interrupt low_isr
```

```
void low_isr(void){}

static unsigned long int SEED_X = 521288629L;
static unsigned long int SEED_Y = 362436069L;
unsigned int random(void)
{
    volatile unsigned float tempf = 0;
    unsigned int tempi = 0;
    static unsigned int a = 18000, b = 30903;

    SEED_X = a*(SEED_X&65535) + (SEED_X>>16);
    // requires 16X16 multiply, 32bit add
    SEED_Y = b*(SEED_Y&65535) + (SEED_Y>>16);
    // requires 16X16 multiply, 32bit add

    tempi = ((SEED_X&65535) + (SEED_Y&65535))/2;
    tempf = (unsigned float) tempi / (unsigned float) 65535; // 0 < temp < 1
    tempf = tempf * 9000;  // 0 < temp < 9000
    tempf = tempf + 1000;  // 1000 < temp < 10000
    return (int) tempf;
}

static unsigned char values[] = {10, 10}; //start out 'off'

void display(void)
{
    static unsigned char numbers[] =
    {
        0b00000011,
        0b10011111,
        0b00100101,
        0b00001101,
        0b10011001,
        0b01001001,
        0b01000001,
        0b00011111,
        0b00000001,
        0b00001001,
        0b11111111,
        0b01100001
    };

    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 debounce(void)
{
    unsigned int count = 0;
    while (count < 0x2FFF)
    {
        display();
        count++;
    }
}
void main(void)
{
    TRISC = 0;
    TRISD = 0;
    PORTC = 0;
    PORTD = 0;

    INTCON2bits.RBPU = 0;
    TRISB = 0x80;
    PORTBbits.RB0 = 1;
    PORTBbits.RB1 = 1;
    PORTBbits.RB2 = 1;

    while (1)
    {
        switch (state)
        {
            case 0:
                /* case 0 = wait for button press, debounce*/
                while( PORTBbits.RB7 == 1 )
                {
                    display();
                }
                //turn off display:
                values[0] = 10;
                values[1] = 10;
                debounce();
                state = 1;
            break;
```

```
case 1:
    while( PORTBbits.RB7 == 0 );
    debounce();
    wait = random(); // random generates integer from 1000 to 10000 ms
    state = 2;
break;

case 2:
    T0CON = 0x80; /* enable timer 0 16 bit op w prescaler = 1 */
    INTCONbits.T0IF = 0;
    INTCONbits.T0IE = 1;
    INTCONbits.GIE = 1;
    TMR0H = TVAL >> 8;
    TMR0L = TVAL & 255;

    while( PORTBbits.RB7 == 1 && wait > 0 );
    if( PORTBbits.RB7 == 0 )
    {
        // Pushed the button prematurely = E0
        T0CON = 0;
        values[0] = 0;
        values[1] = 11;
        state = 5;
        debounce();
    }
    else
    {
        //timer finished
        PORTBbits.RB0 = 0;
        TMR0H = TVAL >> 8;
        TMR0L = TVAL & 255;
        reaction = 0;
        state = 3;
    }
break;

case 3:
    while ( PORTBbits.RB7 == 1 && reaction < 999 );
    PORTBbits.RB0 = 1;

    if( PORTBbits.RB7 == 0 )
    {
        T0CON = 0;
        debounce();
        state = 4;
    }
    else
    {
        T0CON = 0;
```

```
                //too slow = E1
                values[0] = 1;
                values[1] = 11;
               state = 5;
            }

        break;

        case 4:
            //wait one second to display results
            wait = 1000;
            TMR0H = TVAL >> 8;
            TMR0L = TVAL & 255;
            T0CON = 0x80;
            while ( wait > 0 );
            values[0] = (reaction \% 100) / 10;
            values[1] = reaction / 100;
            state = 5;
        break;

        case 5:
            //wait for button up
            while( PORTBbits.RB7 == 0 )
            {
                display();
            }
            debounce();
            state = 0;

        break;
    }
  }
}
```