Embedded Control Game Report

Michael Stark & David Melecio-Vázquez

Section 4, Side A (TA: Xuemei Gao)

Due: 8 October 2010

Table of Contents

[I. Introduction and Statement of Purpose 3](#_Toc274144879)

[II. System Development 4](#_Toc274144880)

[a. Hardware Details 4](#_Toc274144881)

[74F365A Buffer Chip 4](#_Toc274144882)

[Light Emitting Diode (LED) 5](#_Toc274144883)

[Bi LED 5](#_Toc274144884)

[Resistor 6](#_Toc274144885)

[Potentiometer 6](#_Toc274144886)

[Push Button 7](#_Toc274144887)

[Slide Switch 7](#_Toc274144888)

[Buzzer 8](#_Toc274144889)

[b. Software Details 9](#_Toc274144890)

[III. Results and Conclusions 10](#_Toc274144891)

[IV. References 11](#_Toc274144892)

[V. Appendices 12](#_Toc274144893)

[a. C Code 12](#_Toc274144894)

[b. Hardware Schematic 19](#_Toc274144895)

[c. Participation Summary 20](#_Toc274144896)

1. Introduction and Statement of Purpose

Computer gaming has reached new heights in the world. Today players interact with expansive and complex 3D environments that react to player inputs. The start of what we know today as the culture of computer gaming can be traced back to the unveiling of the first Simon game, introduced by Milton and Bradley in May 15, 1978(^2). Since then, with the advancement of computer hardware, graphics processors, sound cards, and the development of more advanced types of integrated systems and embedded controls, user-friendly electronics are rapidly becoming a part of everyday life in work and play.

In this project a simplified version of the Simon game was built to help introduce concepts of Embedded Control. Using the guidelines provided, a game is made where the player can interact with the microcontroller using a series of pushbuttons. The microcontroller then gives the player feedback in the form of visual and/or auditory outputs.

The purpose of the design of this game is to gain an understanding of (1) how different hardware components can be placed in a circuit to interact with both the microcontroller and the player   (2) how different software building techniques can be used to implement the hardware to help a player understand their interaction with the electronic device, and (3) how different troubleshooting techniques can be used, in both the hardware and the software, to correct and debug the embedded control system so that it performs the necessary tasks.

**NOTE**: Figures are numbered in the following format

Figure Xy-Z

where “Xy” corresponds to the section and subsection, respectively, of the Figure, and “Z” corresponds to the number of the figure being observed.

1. System Development

The software was written first, according to the description given in the assignment.  Once the software was written and syntactic errors were removed, the hardware was rewired according to the schematic given (see Appendix B, Hardware Schematic).  Several hardware components were utilized, and will be described in the Hardware Details section.  In addition, various software constructs were used, as explained in the Software Details section.

* 1. Hardware Details

Note: please refer to Appendix B, Hardware Schematic, to see where each component was included in the wiring.

### 74F365A Buffer Chip

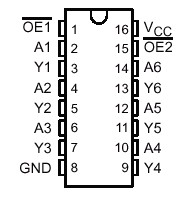


Figure 2a-1 - 74F365A Buffer Chip

The Buffer chip is composed of a series of 6 inputs and 6 outputs, each of which is connected with a logical buffer gate.  Outputs such as LEDs and the Buzzer were connected to the microcontroller through the buffer in order to isolate the outputs and protect the components from having current flow across them in the incorrect direction.

### Light Emitting Diode (LED)

Description: https://lh3.googleusercontent.com/8sGi0q_2c7M5F0q6mjeJS9VCJHh_c6PZaBSsFbvxZiuAmyR36XtULayn9cz60s1cvHS9NAUasVjhnp3g35FUJk8-MlReP2FwFKlRaMrhzLKNd2rGKQ

Figure 2a-2 - LED

A diode is similar to a buffer, in that it only allows current to flow one way through the device.  The LED, however, gives off light when current is flowing in the correct direction.  These LEDs have various shadings on their glass casing, resulting in different colored lights being emitted.  LEDs were used in the hardware as outputs for the User.

### Bi LED

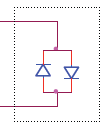


Figure 2a-3 - Bi LED

The Bi LED component is a simple circuit with two LEDs wired perpendicularly to each other.  When current flows in one direction across the Bi LED, one of the colored LEDs is lit - when current flows in the opposite direction, the other colored LED is lit.  The BiLED was used as an output to the User to indicate various states of the game (e.g. ready to accept input, incorrect input given, etc.).

### Resistor

Description: https://lh6.googleusercontent.com/URtQ7hrRH-ZcndER7vW8Bg1UtzWXmmTWRVF6JTG4t5PmAkg8hkyfYISBv448RU99gJfMatEyJU-2dAurCi_NvEB4nsu4MTO5ECiyf_Bh5A5a0Nzx6Q

Figure 2a-4 - Resistor

Resistors provide electrical resistance on a circuit.  They are used to absorb some voltage, so that various other components are not overloaded/short circuited by large power surges.  Resistors were used between power and the LEDs, between power and the push buttons, and between power and the potentiometer.

### Potentiometer

Description: https://lh6.googleusercontent.com/KIlaIPhyH8zuqgYOeh1hU3N8-qY3UhMwowsc-6Ckgx1thYzOkTYWGB2qFKh2TzRgSs_R7GYqrucKzGMVE3cuVRU6DinmB3UW1E_W5MNpxLIIpxFrGA

Figure 2a-5 - Potentiometer

Potentiometers are analog input devices.  By sliding a contact along a resistant surface, a variable amount of voltage is ready along the input wire.  This was used in the game to vary the speed at which LEDs were lit (a larger voltage input resulted in a larger wait time).

### Push Button

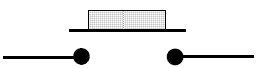


Figure 2a-6 - Push Button

Push Buttons are simple input devices.  When not pressed, an open circuit exists between power and the microcontroller, and a logical high (1) is read.  When pressed a closed circuit is created from power to ground, resulting in a logical low (0) being read in the microcontroller.  Push Buttons were used by the User to indicate which LED was lit, as well as restarting the game.

### Slide Switch

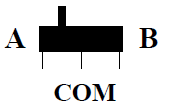
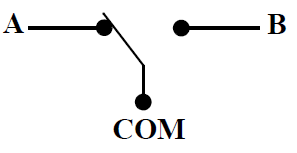


Figure 2a-7 - Slide Switch Schematic (left) and physical appearance (right)

A slide switch operates similarly to a Push Button, except that the Slide Switch stays in one of two states without the User having to continually press.  Just like the Push Button, in the On position, a closed circuit is formed between power and ground, and a logical low (0) is read on the microcontroller.  In the Off position, an open circuit is formed, and a logical high (1) is read on the microcontroller.  The Slide Switch was used in initial versions of the game to control when the game was played, and to restart the game.

### Buzzer

Description: https://lh3.googleusercontent.com/sRkbBWqq_aR5E84IBBQ4OkRVQuk4d5-vXGwaaeKQ4DrBp45Yn7PV6ZEZxtn7qirrqAz1bkOnI0KnHovyRH8_0M9EO2qlz7pKdTxBZJH7iB9dCvtobA

Figure 2a-8 - Buzzer

The Buzzer makes a loud buzzing sound when current flows across it.  The Buzzer was used as an output to indicate to the User that incorrect input was given.

* 1. Software Details

**Overview** Once all the code is downloaded to the microcontroller the initialization are run. The player at this point sets the game speed for the Simon game. The game speed is an analog input and thus is converted using the Analog-to-Digital converter in Port 1. The timer is also started at the beginning by the code, before any player interaction (with the exception of the game speed) so that time is being measured in the amount of overflows that take place.

**Description of Game Flow** The Simon game begins when Pushbutton number 4 is pressed. At this, and at any point in the code, it is possible to reset the game and have a new sequence of LEDs lit. At the start, the code calculates an array of values that correspond to the LED that is to be lit, and the order that that they are to be lit as well. When this routine takes place, there is a mechanism in place to prevent identical numbers (to light the LEDs) from appearing in the sequence so that all five answers are susceptible to the user and easier to spot. The LED’s are then lit corresponding the number and order in the array.

When the LEDs are finished blinking, the BILED turns green to indicate to the player to that the microcontroller is ready to receive their input. The player input is logged in an array of five spaces. If at any point the player presses an incorrect pushbutton, a buzzer will go off and the BILED is shut off. If the player manages to match the exact array programmed, the three LED’’s blink three times to signal victory to the player. This concludes the game and the program waits for pushbutton 4 to be pressed again to re-start the game.

**Masking (means, purpose)** In the code there is an area where the ports and the timer are initialized using masks. These masks allow specific bits to be assigned without changing bits which may be being used for a different task other than than the Simon game. To perform these maskings the logical “AND” and “OR” operators are used. The “OR” operator is represented as “ | ” and the “AND” operator uses an “&” symbol.

**Initialization (port, timer, adc)/Reading input/output pins** In Port\_Init, Ports 1, 2, and 3 are initialized. The Port 1 masks set Pin 1 output bit to open-drain and to configure it for analog input. For an open-drain pin, the impedance has to be set high so Port1 Pin 1 is set to a high impedance. Port 1 is where all the A/D conversion takes place. Port 1 is only performing the A/D conversion and is thus providing an input to the microcontroller.

The Port 2 masks set Pins 0-3 to open-drain and to a high impedance. Port 2 is where all the push buttons are connected. These push buttons are regarded as “inputs” with respect to the microcontroller. They send information to the Part about their current state. If the pushbutton is pressed then a voltage difference of 0V is detected, and 5V is detected when the push buttons are not being pressed.

The Port 3 masks sets Pins 2-7 to push pull. Any feedback the player receives is coming from this port. Port 3 contains the BILED, the three LED’s, and the buzzer. For all these three outputs a voltage difference of 0V activates, while a voltage difference of 5V indicates that all these outputs are off.

In order to use the timer 0 interrupt, Interrupt\_Init configures the Interrupt Enable Register (IE) to enable all interrupts and select Timer 0 for the source of the interrupts.

In Timer\_Init, the timer is i nitialized. CKCON is set so that SYSCLK is used. TMOD’s least significant bits are cleared to use the internal clock and mode 1 is selected to use a 16 bit mode for the timer. This inititialization also functions to reset the clock by first stopping the timer (in case it was already running) and clearing the low and high bytes of the 16 bit counter/timer.

In ADC\_init, the Analog to Digital Conversion routine is initialized for Port 1. REF0CN is set so that the ADC1 voltage reference is 2.4V. ADC1CF is then used to set the gain of the system to 1. ADC1CN is then set to enable the analog to digital conversion.

For this simple version of the Simon game, the player presses Push Button 4 to start or reset the game. Once the game is started the microcontroller reads the voltage from the potentiometer and calculates the game speed. The potentiometer can only give off an analog input to our microcontroller, which only reads digital inputs. To perform an Analog-to-Digital Conversion (A/D Conversions) the potentiometer is connected to Port 1 where the conversion event occurs. For this Simon game an internal reference voltage of 2.4V is used, i.e. the maximum value of any analog input for this case is 2.4V. The A/D result at the end of the conversion, is now in the form of a digital signal, of which the microcontroller can easily read.

With the result from Port 1, the “on time” and “off time” can be found for the LED’s. It is important to note that in order to measure time, Timer 0 is used in the interrupt service routine. Timer 0 only counts overflows. When a byte overflows it updates the number of overflows and increments the “Counts” variable. This “Counts” variable is used to measure the passing of time while performing some function, such as lighting an LED or sounding the Buzzer. Things that   The “game speed” is set by calculating the amount of overflows for which an LED is to be lit and then calculate how many overflows for which the LEDs are to be off.

Results and Conclusions

When the code is run on the microcontroller, the system simulates a simple version of the Simon game, explained above (Section I). For 5 turns, a randomly selected LED is lit for a duration determined by the potentiometer input, and then all LEDs are turned off for half of the on time. Once the sequence of 5 lights is complete, the User tries to reproduce the sequence by pressing the corresponding push buttons in the correct order. If all 5 inputs are correct, the LEDs flash three times, and the game is over. However, if the User records an incorrect input (by pressing the wrong pushbutton), the buzzer sounds for one and a half seconds, and the game is over. The game starts a new round if the fourth push button is ever pressed.

A unique approach of the software is failing as soon as the user presses a wrong button, rather than waiting for 5 button presses to indicate incorrect input. This more closely represented the Simon game, which would buzz as soon as the player pressed the wrong color button.

During development, the wiring of the hardware became especially messy, due to the format of the labs, in which components were added one at a time to existing circuitry. While this lowered the amount of circuitry errors encountered in each new lab, it also meant that not enough space was given for future expansion in the first lab. An entire lab section was spent ripping out all components and starting over in order to neatly arrange the wires and components.

Overall, the game was designed, written, and implemented without much difficulty.

1. References
2. Appendices
   1. C Code

Following is a copy of our file lab2.c.

/\*  
\* Names: Michael Stark + David Melecio-Vázquez  
\* Section: 4 A  
\* Date: 14 September 2010  
\* File name: lab2.c  
\* Description: Play a game of LITEC Memory with the user.  
\*/  
  
#include <c8051\_SDCC.h>// include files. This file is available online  
#include <stdio.h>  
#include <stdlib.h>  
  
//-----------------------------------------------------------------------------  
// Function Prototypes  
//-----------------------------------------------------------------------------  
  
// Initialization Functions  
void Port\_Init(void);      // Initialize ports for input and output  
void Timer\_Init(void);     // Initialize Timer 0   
void ADC\_Init(void);       // Initialize A/D Conversion  
unsigned char Read\_Port\_1(void);       // Performs A/D Conversion  
void Interrupt\_Init(void); // Initialize interrupts  
void Timer0\_ISR(void) interrupt 1; // Called at Timer0 overflow  
  
// Game Functions  
void play\_game(void); // Plays the LITEC Memory game.  
int CalculateMaxCounts(unsigned char x); // Converts the port 1 result into a number of  
                                         // overflows to wait for.  
unsigned char light\_LED(unsigned char LED\_to\_light, short on\_time, short off\_time);

// Light the designated LED for on\_time, then wait for off\_time

unsigned char ReadPushbuttons(); // Wait for a pushbutton event, and return which one was  
                                 // pressed  
unsigned char unique\_random(unsigned char last\_state); // Generates a random number  
                                                       // different from the last one  
unsigned char random(unsigned char N); // Generates a random number between 0 and N-1   
void light\_green(void); // Light BiLED green  
void light\_red(void); // Light BiLED red  
int CheckPushButton1(void); // function which checks push button 1  
int CheckPushButton2(void); // function which checks push button 2  
int CheckPushButton3(void); // function which checks push button 3  
int CheckPushButton4(void); // function which checks push button 4  
char\* newline(); // Helper function, used when printing (returns "\r\n")  
  
//-----------------------------------------------------------------------------  
// Global Variables  
//-----------------------------------------------------------------------------  
  
// Pushbuttons (Inputs)  
sbit at 0xA0 PB3; // Push button 3  
                  // Associated with Port 2, Pin 0 (second from left, red PB)  
sbit at 0xA1 PB4; // Push button 4  
                  // Associated with Port 2, Pin 1 (far left, black PB)  
sbit at 0xA2 PB1; // Push button 1  
                  // Associated with Port 2, Pin 2 (far right, red PB)  
sbit at 0xA3 PB2; // Push button 2  
                  // Associated with Port 2, Pin 3 (second from right, black PB)  
  
// LEDs and Buzzer (Outputs)  
sbit at 0xB2 LED2; // LED2, associated with Port 3, Pin 2  
sbit at 0xB3 BILED0; // BILED0, associated with Port 3 Pin 3  
sbit at 0xB4 BILED1; // BILED1, associated with Port 3 Pin 4  
sbit at 0xB5 LED1; // LED1, associated with Port 3 Pin 5  
sbit at 0xB6 LED0; // LED0, associated with Port 3 Pin 6  
sbit at 0xB7 BUZZER; // Buzzer, associated with Port 3 Pin 7  
  
unsigned int Counts = 0;  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
void main(void) {  
 Sys\_Init();      // System Initialization  
 Port\_Init();     // Initialize ports 2 and 3   
 Interrupt\_Init();  
 Timer\_Init();    // Initialize Timer 0   
 ADC\_Init();      // Initialize A/D Conversion  
 putchar(' ');    // The quote fonts may not copy correctly into SiLabs IDE  
 putchar('\r');  
  
 // Enable Timer 0  
 TR0 = 1;  
  
 while (!CheckPushButton4()); // Wait for the Start Game button to be pushed  
  
 while (1) {  
   play\_game();  
 }  
}  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
void Port\_Init(void) {  
 // Port 1 Constant Masks  
 unsigned char P1MDIN\_LO = 0xFD; // 1111 1101, Set P1.1 as an analog input   
 unsigned char P1MDOUT\_LO = 0xFD; // 1111 1101, Set P1.1 as a input port bit  
 unsigned char P1\_HI = 0x02; // 0000 0010 Set P1.1 to a high impedance state  
  
 // Port 2 Constant Masks  
 unsigned char P2MDOUT\_LO = 0xF0; // 1111 0000  
 unsigned char P2\_HI = 0x0F; // 0000 1111  
   
 // Port 3 Constant Masks  
 unsigned char P3MDOUT\_HI = 0xFC; // 1111 1100  
  
 // Set Port 1 analog input bits  
 P1MDIN &= P1MDIN\_LO;  
 // Set Port 1 output (low) bits  
 P1MDOUT &= P1MDOUT\_LO;  
 // Set Port 1 impedence (high) bits  
 P1 |= P1\_HI;  
  
 // Set Port 2 MDOUT low bits  
 P2MDOUT &= P2MDOUT\_LO;  
 // Set Port 2 impedence (high) bits  
 P2 |= P2\_HI;  
   
 // Set Port 3 MDOUT high bits  
 P3MDOUT |= P3MDOUT\_HI;  
}  
  
void Interrupt\_Init(void) {  
 // Enable Timer 0 Overflow interrupts (bit 1), and enable all interrupts (bit 7)  
 IE |= 0x82; // 1000 0010  
}  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
void Timer\_Init(void) {  
 CKCON |= 0x08; // 0000 1000 (use SYSCLK)  
 TMOD &= 0xF0; // clear the 4 least significant bits (1111 0000)  
 TMOD |= 0x01; // 0000 0001 (Timer0 in mode 1, for 16-bit)  
 TR0 = 0; // Stop Timer0  
 TL0 = 0; // Clear low byte of register T0  
 TH0 = 0; // Clear high byte of register T0  
}  
  
void ADC\_Init(void) {  
 REF0CN &= 0xF7; // 1111 0111 Configure ADC1 to use VREF  
 REF0CN |= 0x03; // 0000 0011  
 ADC1CF = 0x01; // 0000 0001 Set a gain of 1  
 ADC1CN |= 0x80; // 1000 0000 Enable ADC1  
}  
  
unsigned char Read\_Port\_1(void) {  
 AMX1SL = 0x01; // 0000 0001 Set the Port pin number  
 ADC1CN &= 0xDF; // 1101 1111 Clear the flag from the previous ADC1 conversion  
 ADC1CN |= 0x10; // 0001 0000 Start A/D Conversion  
 while ((ADC1CN & 0x20) == 0x00); // Wait for conversion to be complete  
 return ADC1; //Assign the A/D conversion result  
}  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
void Timer0\_ISR(void) interrupt 1 {  
 TF0 = 0; // clear interrupt request  
 Counts++;  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*  
\* Play the LITEC Memory game.  Returns if PB4 is ever pressed.  
\*/  
void play\_game(void) {  
 unsigned char turns[5], inputs[5], port1\_result, i, correct;  
 unsigned short on\_max\_counts, off\_max\_counts;  
  
 // Turn off all outputs  
 LED0 = 1;  
 LED1 = 1;  
 LED2 = 1;  
 BUZZER = 1;  
 BILED0 = 1;  
 BILED1 = 1;  
  
 // Calculate the on\_time and off\_time by reading Port 1’s A/D conversion  
 port1\_result = Read\_Port\_1();  
 on\_max\_counts = CalculateMaxCounts(port1\_result);  
 off\_max\_counts = (on\_max\_counts / 2);  
  
 // Generate the sequence of LEDs to light  
 turns[0] = unique\_random(3);  
 for (i = 1; i < 5; i++) {  
   // Generate a value different from the previous turn’s value.  
   turns[i] = unique\_random(turns[i - 1]);  
 }  
  
 // Light LED sequence  
 for (i = 0; i < 5; i++) {  
   // Restart the game if the player presses PushButton4.  
   if (light\_LED(turns[i], on\_max\_counts, off\_max\_counts) == 1) {  
     return;  
   }  
 }  
  
 // Light the green BiLED to indicate the ready-for-input state.  
 light\_green();  
  
 correct = 1;  
  
 for (i = 0; i < 5; i++) {  
   // Read a pushbutton, and record which one was pressed.  
   inputs[i] = ReadPushbuttons();  
   if (inputs[i] == 3) { // Pushbutton 4 pressed, restart game  
     return;  
   } else if (inputs[i] != turns[i]) { // Incorrect response  
     correct = 0;  
     printf("Input %d was incorrect!%s", i + 1, newline());  
     break;  
   } else { // Correct response  
     printf("Input %d was correct!%s", i + 1, newline());  
   }  
 }  
  
 if (correct) {  
   // flash LEDs 3 times  
   for (i = 0; i < 3; i++) {  
     // Turn LEDs on  
     LED0 = 0;  
     LED1 = 0;  
     LED2 = 0;  
     // Wait about a quarter of a second  
     Counts = 0;  
     while (Counts < 75);  
     // Turn LEDs off  
     LED0 = 1;  
     LED1 = 1;  
     LED2 = 1;  
     // Wait about a quarter of a second  
     Counts = 0;  
     while (Counts < 75);  
   }  
 } else {  
   // sounds buzzer for 1.5 seconds  
   BUZZER = 0;  
   light\_red();  
   Counts = 0;  
   while (Counts < 506);  
   BUZZER = 1; // Turn buzzer back off (thank God)  
 }  
  
 // Turn off BILED  
 BILED0 = 1;  
 BILED1 = 1;  
  
 // Wait for the Start Game button to be pressed  
 while (!CheckPushButton4());  
}  
  
/\*  
\* Returns the number of overflows to wait for in on\_time.  
\*  
\* NOTE: off\_time is simply (on\_time / 2).  
\*/  
int CalculateMaxCounts(unsigned char x) {  
 int on\_time\_millis = (x \* 5) + 200;  
 // This gives us the milliseconds to wait for.  In 16 bit counting mode,  
 // based on SYSCLK, there are exactly 337.5 overflows per second, or  
 // 0.3375 overflows per millisecond.  We multiply by this constant  
 // to convert from milliseconds to overflows.  
 return (on\_time\_millis \* 0.3375);  
}  
  
/\*  
\* Light the LED for on\_time, then wait for off\_time.  Returns 0 if successful, or 1  
\* if PushButton4 was pressed.  
\*/  
unsigned char light\_LED(unsigned char LED\_to\_light, short on\_time, short off\_time) {  
 switch(LED\_to\_light) {  
   case 0:  
     // Light LED0  
     LED0 = 0;  
     break;  
   case 1:  
     // Light LED1  
     LED1 = 0;  
     break;  
   case 2:  
     // Light LED2  
     LED2 = 0;  
     break;  
   default:  
     printf("Invalid LED input: %d%s", LED\_to\_light, newline());  
 }  
  
 // Wait for on\_time  
 Counts = 0;  
 while (Counts < on\_time) {  
   if (CheckPushButton4()) {  
     return 1;  
   }  
 }  
  
 // Turn LEDs off again  
 LED0 = 1;  
 LED1 = 1;  
 LED2 = 1;  
  
 // Wait for off\_time  
 Counts = 0;  
 while (Counts < off\_time) {  
   if (CheckPushButton4()) {  
     return 1;  
   }  
 }  
  
 return 0; // Pushbutton 4 not pressed  
}  
  
/\*  
\* Waits for a pushbutton to be pressed, then returns which button was pressed  
\* (as an integer between 0 and 3).  
\*/  
unsigned char ReadPushbuttons() {  
 while (1) {  
   if (CheckPushButton1()) {  
     Counts = 0;  
     while (Counts < 25); // Wait for PushButton to be completely pressed.  
     if (CheckPushButton1()) { // Check again, to make sure it was no accident  
       while (CheckPushButton1()); // Wait until the button is released.  
       Counts = 0;  
       while (Counts < 25); // Wait for PushButton to be completely released.  
       return 0;  
     }  
   }  
   if (CheckPushButton2()) {  
     Counts = 0;  
     while (Counts < 25); // Wait for PushButton to be completely pressed.  
     if (CheckPushButton2()) { // Check again, to make sure it was no accident  
       while (CheckPushButton2()); // Wait until the button is released.  
       Counts = 0;  
       while (Counts < 25); // Wait for PushButton to be completely released.  
       return 1;  
     }  
   }  
   if (CheckPushButton3()) {  
     Counts = 0;  
     while (Counts < 25); // Wait for PushButton to be completely pressed.  
     if (CheckPushButton3()) { // Check again, to make sure it was no accident  
       while (CheckPushButton3()); // Wait until the button is released.  
       Counts = 0;  
       while (Counts < 25); // Wait for PushButton to be completely released.  
       return 2;  
     }  
   }  
   if (CheckPushButton4()) {  
     Counts = 0;  
     while (Counts < 25); // Wait for PushButton to be completely pressed.  
     if (CheckPushButton4()) { // Check again, to make sure it was no accident  
       while (CheckPushButton4()); // Wait until the button is released.  
       Counts = 0;  
       while (Counts < 25); // Wait for PushButton to be completely released.  
       return 3;  
     }  
   }  
 }  
}  
  
/\*  
\* Returns a random number different from last\_state.  
\*/  
unsigned char unique\_random(unsigned char last\_state) {  
 unsigned char return\_value;  
 do {  
   return\_value = random(3);  
 } while(return\_value == last\_state);  
 return return\_value;  
}  
  
/\*  
\* Returns a random integer between 0 and N-1 (a range of N numbers).  
\*/  
unsigned char random(unsigned char N) {  
 return (rand() % N);  
}  
  
/\*  
\* Light BiLED green.  
\*/  
void light\_green(void) {  
 BILED0 = 0;  
 BILED1 = 1;  
}  
  
/\*  
\* :Light BiLED red.  
\*/  
void light\_red(void) {  
 BILED0 = 1;  
 BILED1 = 0;  
}  
  
/\*  
\* Returns a 0 if CheckPushButton1 not activated or a 1 if CheckPushButton1 is activated.  
\* This code reads a single input only, associated with PB1  
\*/  
int CheckPushButton1(void) {  
 // If PB1 is pressed, the value of PB1 is 0 (logic low).  If PB1 is not pressed, the  
 // value of PB1 is 1 (logic high).  This is the opposite of what makes sense in the  
 // code (with 0 usually meaning off, and 1 meaning on), so we negate the value in  
 // PB1 before returning.  
 return !PB1;  
}  
  
/\*  
\* Returns a 0 if push button 2 not activated or a 1 if push button 2 is activated.  
\* This code reads a single input only, associated with PB2  
\*/  
int CheckPushButton2(void) {  
 // If PB2 is pressed, the value of PB2 is 0 (logic low).  If PB2 is not pressed, the  
 // value of PB2 is 1 (logic high).  This is the opposite of what makes sense in the  
 // code (with 0 usually meaning off, and 1 meaning on), so we negate the value in  
 // PB2 before returning.  
 return !PB2;  
}  
  
/\*  
\* Returns a 0 if push button 3 not activated or a 1 if push button 3 is activated.  
\* This code reads a single input only, associated with PB3  
\*/  
int CheckPushButton3(void) {  
 // If PB3 is pressed, the value of PB3 is 0 (logic low).  If PB3 is not pressed, the  
 // value of PB3 is 1 (logic high).  This is the opposite of what makes sense in the  
 // code (with 0 usually meaning off, and 1 meaning on), so we negate the value in  
 // PB3 before returning.  
 return !PB3;  
}  
  
/\*  
\* Returns a 0 if push button 4 not activated or a 1 if push button 4 is activated.  
\* This code reads a single input only, associated with PB4  
\*/  
int CheckPushButton4(void) {  
 // If PB4 is pressed, the value of PB4 is 0 (logic low).  If PB4 is not pressed, the  
 // value of PB4 is 1 (logic high).  This is the opposite of what makes sense in the  
 // code (with 0 usually meaning off, and 1 meaning on), so we negate the value in  
 // PB4 before returning.  
 return !PB4;  
}  
  
/\*  
\* Helper function to return a string containing a newline character and return  
\* character.  
\*/  
char\* newline() {  
 char\* retval = "\r\n";  
 return retval;  
}

* 1. Function Glossary

**Port\_Init**

Prototype

void Port\_Init(void);

Description

Port\_Init initializes ports 1 and 2 for input, and port 3 for output.

Example:

void Port\_Init(void);

void main() {

Port\_Init();

}

**Timer\_Init**

Prototype

void Timer\_Init(void);

Description

Timer\_Init initializes Timer 0 to use a 16-bit counting mode, based on SYSCLK.  It also resets TH0 and TL0, the timer SFRs, to 0.

Example:

void Timer\_Init(void);

void main() {

Timer\_Init();

}

**ADC\_Init**

Prototype

void ADC\_Init(void);

Description

ADC\_Init initializes the Analog to Digital Converter.  It sets the gain to 1 and uses Vref as its internal voltage reference.

Example:

void ADC\_Init(void);

void main() {

ADC\_Init();

}

**Read\_Port\_1**

Prototype

unsigned char Read\_Port\_1(void);

Description

Read\_Port\_1 performs the A/D conversion, reading Port 1, Pin 1, and returns the result.

Example:

unsigned char Read\_Port\_1(void);

void main() {

unsigned char result = Read\_Port\_1();

printf(“Result of A/D conversion: %d\r\n”, result);

}

**Interrupt\_Init**

Prototype

void Interrupt\_Init(void);

Description

Interrupt\_Init initializes the Interrupt Enable Register to allow all interrupts, and enables Timer 0 overflow interrupts.

Example:

void Interrupt\_Init(void);

void main() {

Interrupt\_Init();

}

**Timer0\_ISR**

Prototype

void Timer0\_ISR(void) interrupt 1;

Description

Timer0\_ISR is called whenever Timer 0 overflows.  The sole function of Timer0\_ISR is to increment the global variable Counts.

Example:

void Timer0\_ISR(void) interrupt 1 {

Counts++;

}

**play\_game**

Prototype

void play\_game(void);

Description

play\_game is the main function that implements the LITEC Memory game.  It determines the on\_time variable based on the potentiometer input, generates a sequence of 5 numbers corresponding the LEDs, lights those LEDs, and reads user input to determine whether the user wins or loses the game.

Example:

void play\_game(void);

void main() {

while (1) {

play\_game();

}

}

**CalculateMaxCounts**

Prototype

int CalculateMaxCounts(unsigned char x);

Description

CalculateMaxCounts accepts the A/D conversion result and converts it to a number of Timer 0 overflows to wait.  The range of overflows returned is approximately 67 to 506, corresponding to an approximate range of 0.2s to 1.5s.

Example:

void Timer\_Init(void);

unsigned char Read\_Port\_1(void);

int CalculateMaxCounts(unsigned char x);

void main() {

unsigned char ADC\_result;

int maxCounts;

Timer\_Init();

ADC\_result = Read\_Port\_1();

maxCounts = CalculateMaxCounts(ADC\_result);

Counts = 0;

TR0 = 1;

while (Counts < maxCounts); // Wait

printf(“Waited for %d overflows\r\n”, maxCounts);

}

**light\_LED**

Prototype

unsigned char light\_LED(unsigned char LED\_to\_light, short on\_time, short off\_time);

Description

light\_LED lights the LED corresponding to LED\_to\_light, then waits for on\_time counts to pass.  Following this, all LEDs are turned off, and the function waits for off\_time counts to pass before returning.  A value of 0 for light\_LED makes the function light LED0, a value of 1 lights LED1, and a value of 2 lights LED2.

If Push Button 4, the game reset button, was pressed during this time, a value of 1 is returned.  Otherwise, a value of 0 is returned.

Example:

unsigned char light\_LED(unsigned char LED\_to\_light, short on\_time, short off\_time);

void main() {

unsigned char result = light\_LED(0, 300, 150);

if (result) {

printf(“Push Button 4 pressed\r\n”);

} else {

printf(“Push Button 4 NOT pressed\r\n”);

}

}

**ReadPushbuttons**

Prototype

unsigned char ReadPushbuttons(void);

Description

ReadPushButtons waits for a pushbutton to be pressed, then returns a value indicating the push button pressed.  If Push Button X is pressed, a value of X - 1 is returned (e.g. if Push Button 3 is pressed, a value of 2 is returned).  This value mapping was chosen so that the expected return value of ReadPushbuttons matches the randomly generated values determining the LEDs to light.

This function accounts for the physical reality of the circuitry and hardware by waiting about a quarter of a second before registering a downpress, and another quarter of a second before registering a release of the button.  This keeps the code from registering multiple button presses when the user may have only pressed the button once.

Example:

void ReadPushButtons(void);

void main() {

unsigned char button = ReadPushbuttons();

printf(“Push button %d pressed\r\n”, button);

}

**unique\_random**

Prototype

unsigned char unique\_random(unsigned\_char last\_state);

Description

unique\_random accepts a value, and returns a randomly generated value between 0 and 2, inclusive, guaranteed to not match last\_state.

When generating the first turn, there is no last\_state to avoid, so a dummy value of 3 is given.

Example:

unsigned char unique\_random(unsigned char last\_state);

void main() {

unsigned char random1, random2;

random1 = unique\_random(3);

random2 = unique\_random(random1);

printf(“These two values do not match: %d, %d\r\n”, random1, random2);

}

**random**

Prototype

unsigned char random(unsigned char N);

Description

random generates a random number between 0 and N-1 (a range of N numbers).

Example:

unsigned char random(unsigned char N);

void main() {

unsigned char random\_num = random(10);

printf(“Your random number is %d\r\n”, random\_num);

}

**light\_green**

Prototype

void light\_green(void);

Description

light\_green lights the proper BILED output so that the green BiLED is lit.

Example:

void light\_green(void);

void main() {

light\_green();

}

**light\_red**

Prototype

void light\_red(void);

Description

light\_red lights the proper BILED output so that the red BiLED is lit.

Example:

void light\_red(void);

void main() {

light\_red();

}

**CheckPushButton1**

Prototype

unsigned char CheckPushButton1(void);

Description

CheckPushButton1 returns a logical true value if Push Button 1 is pressed, or a logical false value if Push Button 1 is not pressed.

Example:

unsigned char CheckPushButton1(void);

void main() {

if (CheckPushButton1()) {

printf(“Push Button 1 pressed.\r\n”);

} else {

printf(“Push Button 1 not pressed.\r\n”);

}

}

**CheckPushButton2**

Prototype

unsigned char CheckPushButton2(void);

Description

CheckPushButton2 returns a logical true value if Push Button 2 is pressed, or a logical false value if Push Button 2 is not pressed.

Example:

unsigned char CheckPushButton2(void);

void main() {

if (CheckPushButton2()) {

printf(“Push Button 2 pressed.\r\n”);

} else {

printf(“Push Button 2 not pressed.\r\n”);

}

}

**CheckPushButton3**

Prototype

unsigned char CheckPushButton3(void);

Description

CheckPushButton3 returns a logical true value if Push Button 3 is pressed, or a logical false value if Push Button 3 is not pressed.

Example:

unsigned char CheckPushButton3(void);

void main() {

if (CheckPushButton3()) {

printf(“Push Button 3 pressed.\r\n”);

} else {

printf(“Push Button 3 not pressed.\r\n”);

}

}

**CheckPushButton4**

Prototype

unsigned char CheckPushButton4(void);

Description

CheckPushButton4 returns a logical true value if Push Button 4 is pressed, or a logical false value if Push Button 4 is not pressed.

Example:

unsigned char CheckPushButton4(void);

void main() {

if (CheckPushButton4()) {

printf(“Push Button 4 pressed.\r\n”);

} else {

printf(“Push Button 4 not pressed.\r\n”);

}

}

**newline**

Prototype

char\* newline(void);

Description

newline returns the sequence of characters “\r\n” corresponding to a new line in terminal output.

Example:

char\* newline(void);

void main() {

printf(“Sample text here%s”, newline());

}

* 1. Hardware Schematic

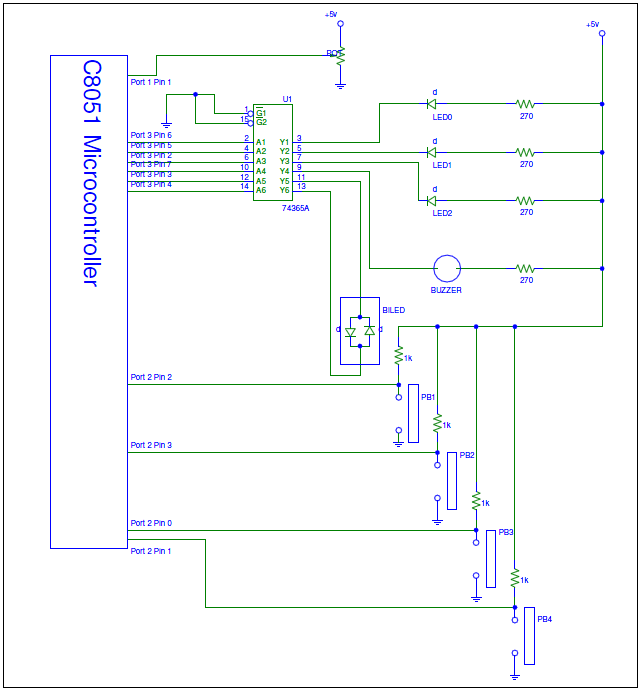


Figure 5b-1 – Lab 2 Hardware Schematic

## Participation Summary

MDS – Michael Stark

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Signature)

DMV – David Melecio-Vázquez

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Signature)

* Cover Sheet
  + Written: MDS
* Table of Contents
  + Written: MDS
* Introduction and Statement of Purpose
  + Written: DMV
* System Development
  + Written: MDS
  + Hardware Details
    - Written: MDS
    - Images: DMV
  + Software Details
    - Written:
    - Flowchart:
* Results and Conclusions
  + Written: MDS
* Appendices
  + C Code
    - Included: MDS
  + Hardware Schematic
    - Designed in PSpice: MDS
  + Participation Summary
    - Written: MDS