Simon $^{\text{embedded}}$

Ece331-Embedded system design
Final Project
May 22, 2002

Jason Wentland
Jeremy Williams

Table of Contents

Introduction

For our final project for Embedded System Design, we chose to develop a game, similar to the electronic Simon game. This game is a sequence memory game. In a nutshell, the computer makes one of four lights light up in a random sequence. The player then repeats the sequence back to the computer via each light's corresponding button. The sequence starts with a length of one and with each round, increases by one. The player tries to see how many rounds they can go before they can no longer repeat the sequence successfully back to the computer.

Objectives and Specifications

Our main objective was to complete a simple version of this memory game, which includes random number generation to develop the sequence, four lights to show the sequence, and four buttons for the player to repeat the sequence back to the computer.

Time permitting, we also wanted to add a few extra features, including simple sound to the game that will play tones along with each light that is lit, as well as short sound sequences when the player repeats each sequence successfully, or gets the sequence wrong. Another additional feature that we envisioned was multi player support.

In our final production of our project, we have included the basic features, as well as simple sounds. A different tone is played for each button the player presses, and an end-of-game sound is played when the game is over.

Strategy and Implementation

Our project is centralized around the Microchip PIC16F84 microprocessor. This processor allows us several I/O ports, interrupts, and timing. Most of the project development was in the software, which we programmed using C.

We began the project by researching interrupts and I/O on the PIC microprocessor. Interrupts are used by our project to generate random numbers, detect button presses, and provide timing for the lighting and sounds.

The random number generation uses a background timer and the PortB change interrupt. When the player pushes a button, the PortB change interrupt occurs. In the interrupt service routine for this interrupt, the last two bits of the current value of the timer are stored, thus providing a random number (0-3) for the next light to flash in the sequence. This provides completely unique game play every time the user starts a game.

The game is controlled by a simple state machine. The game starts in a wait-state, waiting for the user to push a button to start the game. Once the user starts the game the display state is entered, displaying the current sequence of LED's stored in RAM. After this, the user wait state is entered, waiting for the user to start repeating the sequence displayed on a set of buttons, one button corresponding to one LED. If, in this state, the user does not push a button within a specified time-out, the timer interrupt occurs and the game will end. Once the user pushes a button, the game moves into the verification state, checking to see if the button pushed matches the next number in the sequence. If yes, the game moves back to the user wait state, checking for the next number in the sequence. If the user has successfully reached the end of the sequence, a new random number is added to the end of the sequence currently stored in RAM and returns to the display state. If the user gets the next number incorrect, the game moves to the game over state, flashing the LED's and playing the game over sound.

The code was developed in a modular fashion, allowing easy modification. The modules include the main state machine, which was described above, the addToSequence algorithm, the getFromSequence algorithm, and the sound method.

The addToSequence and getFromSequence algorithms are a key part of the game's operation. They provide the method of adding a random number to the currently stored sequence and retrieving previously stored numbers. The number stored in our sequence can be a 0, 1, 2, or 3, each requiring 2 bits to store. Since there are 8-bits in a byte we can store 4 numbers per byte, providing we have a way to access our memory at the bit-level. However, because of the lack of bit-indexed addressing in the PIC, we had to develop an algorithm that modifies byte-indexed addressing to the bit level. The algorithms work by using an 8-bit address. The 6 most significant bits address at the byte level, specifying which byte the current number we are trying to retrieve is located in. The last 2 bits address the actual number in the byte. Thus we use 2 bits to index each number stored in the byte. These algorithms save considerable amounts of memory, allowing a much longer number sequence to be stored in RAM. If we only used byte-indexed addressing, one number per byte, we could only store about 40-60 numbers. These algorithms provide 4-times that amount.

The sound method allows for the playing of sounds of varying frequencies throughout our project. It is a simple method, producing a square-wave with a user-defined period, for a set amount of time. This method uses for-loops to cycle between high and low.

For a more detailed breakdown of the project, please see the Appendix for a software flowchart, commented c-code, and a hardware schematic.

Testing Procedure and Results

Following our modular design strategy, we tested each module as it was developed. For the first test we developed the state machine and tested its operation, using sequences already pre-defined in memory. Next we added in our add and get algorithms, testing random sequence generation and retrieval. Finally, we added in the sound methods, testing sounds at various frequencies.

We chose to do all the testing on the actual circuit. We chose this because of the simplicity of our circuit and the difficulty of the software debugger in handing multiple triggers, interrupts, and delays.

In our testing we were able to successfully verify each module of our code as working.

Bill of Materials

The following is the approximate costs of the materials that were used to make this project. Many of the items were either already owned by us, or were provided by the RHIT ECE lab free of charge.

Cost	Cost to us	Provide by	Part
$7.00	$7.00	RHIT ECE Stockroom	PIC Microcontroller
$1.00	$0.00	Already owned by us	4 colored LEDs
$0.50	$0.00	Already owned by us	Various resistors
$2.00	$0.00	Discarded part we found	4 connected pushbuttons from old calculator
$5.00	$0.00	Already owned by us	Breadboard
$0.50	$0.00	RHIT ECE Lab	Wire
$4.00	$0.00	Already owned by us	4 AA batteries
$20.00	***$7.00***	***Total***	

Users Manual

Game play is simple and fun!

First flip the power switch on. All four LED’s will flash together, indicating the game is ready to play. Push any one of the four buttons and the game will begin. At the beginning of the game, one of the LED’s will light, showing you the first number in the sequence. After this LED lights, push the button corresponding to this LED. The game will then repeat this sequence, displaying one more number each time. All you have to do is remember the order of the LED flashes and repeat them back to the computer. Be quick about it though! You will lose if you take too long or if you get any number in the sequence incorrect. If you lose, the game will flash all the LED's and a sound will indicate the game is over. To start a new game, simply turn the power switch off and back on. The game is now ready to start again.

Conclusions and Recommendations

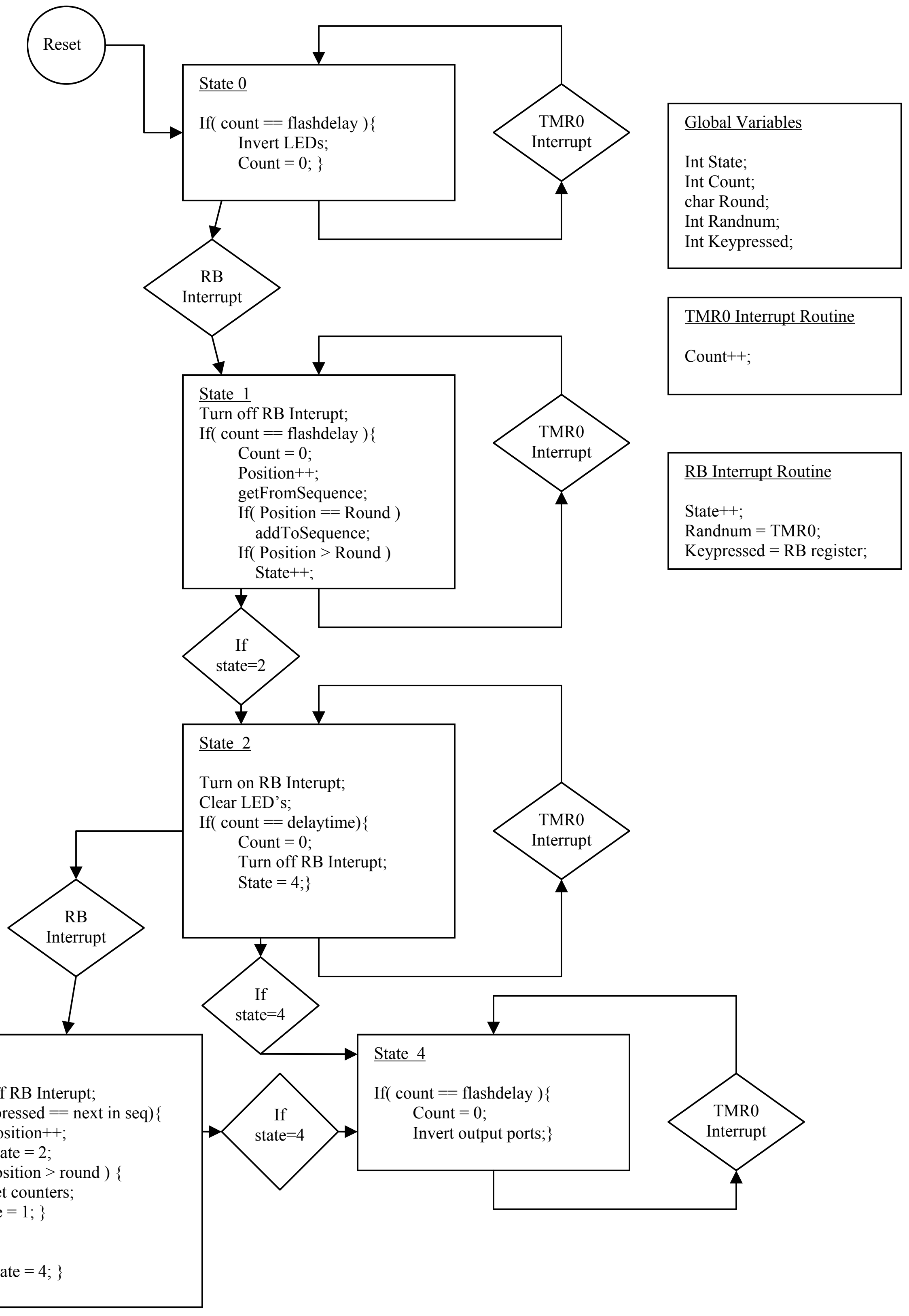
We really enjoyed working on this project. Because of the simplicity of this project, we recommend it be used as an introduction to microcontrollers. The hardware is easy to assemble and the code is straightforward enough for a simple mind to understand.

To expand on the project we would like to see a multi-player option, more sounds, and a difficulty setting option.

References and Acknowledgements

Our source of information on programming the PIC came from the "Reference Materials on PIC16F84 Microcontroller" reference guide provided to us in ECE331 by our professor, Jianjian Song.

Appendix A – Software Flowchart



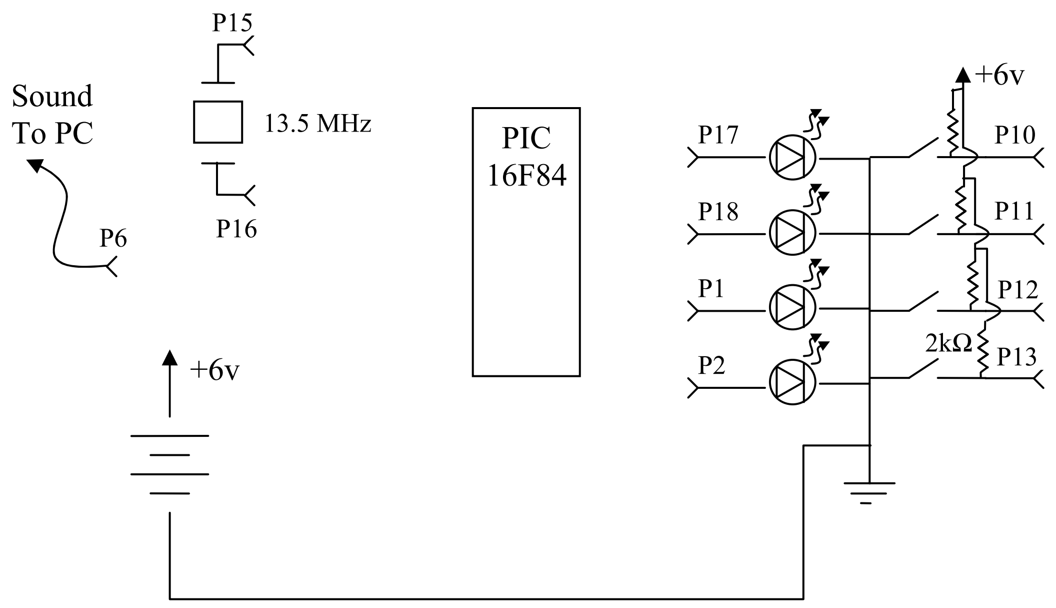
Appendix B – Documented Source Code

```
/******************************************************************************
 * Simon-Electronic Sequence Memory Game
 * Developed by:
 * Jason Wentland and Jeremy Williams
 * Rose-Hulman Institute of Technology
 * ECE331-Embedded System Design
 * May 22, 2002
 ******************************************************************************/



#include <pic.h>

__CONFIG(FOSC1|CP);

#define WAITTIME 255            //the allotted time to press a button
#define FLASHDELAY 80                   //the time used for producing flashing LEDs

char randnum = 0;               //global variable to hold the current random number
char state = 0;                 //global variable that denotes the state of the prog
char count = 0;                 //global variable used to count Timer0 interupts for delays
char keypressed = 0;            //global variable used to save the state of input ports
char sequence[20];              //global variable used to save the sequence

void interrupt interruptRoutine(void);
void addToSequence(char newnumber, char location);
char getFromSequence(char location);
void sound(unsigned int duration, int pitch);
void sound1(unsigned int duration, int pitch);

/******************************************************************************
 * Main program routine
 ******************************************************************************/
void main(void)
{
        char position = 0;
        char round = 0;
        //int i,j;

        RP0 = 1;        //switch to bank 1
        TRISA = 0b11110000;     //set pins on PORT A
        TRISB = 0b11110000;     //set pins on PORT B
        T0CS = 0;               //set clock interupt to use internal clock
        PSA = 0;                //assign prescaler to Timer0
        PS2 = 1;        //Set prescaler bits
        PS1 = 1;        //...
        PS0 = 1;        //...

        RP0 = 0;        //switch back to bank 0
        RBIF = 0;               //Clear RB port change interrupt flag
        RBIE = 0;               //RB port change interrupt
        T0IF = 0;               //Clear Timer0 interupt flag
        T0IE = 1;               //Timer0 interupt
```

```
GIE = 1;            //turn on global interrupt flag



PORTA = 0b00001111;    //turn on all LEDs initially to signal new game

//state machine
while( 1 )
{
        while( state == 0 )                         //wait to start state
        {
                RBIF = 0;                           //clear RB interrupt flag
                RBIE = 1;                           //allow RB interrupt
                if( count == FLASHDELAY/2 )
                {
                        RA0 = RA0 + 1;              //toggle the four output LEDs
                        RA1 = RA1 + 1;              //...
                        RA2 = RA2 + 1;              //...
                        RA3 = RA3 + 1;              //...
                        count = 0;
                        TMR0 = 0;
                }
                RBIE = 0;                           //disable RB interrupt
        }

        while( state == 1 )             //sequence replay state
        {
                //PORTB = state;            //for debugging purposes
                //RB0 = RB0 + 1;

                RBIE = 0;                           //turn off RB interrupt

                if( position == round)
                {
                        addToSequence(randnum, round);//set new number
                }

                if( count < FLASHDELAY-20 )     //display next number
                        PORTA = 0b1 << getFromSequence(position);
                else
                        PORTA = 0;

                //check if waited enough, move to next number
                if( count == FLASHDELAY )
                {
                        count = 0;                  //clear count
                        TMR0 = 0;                   //reset timer0

                        position++;

                        if( position > round )
                        {
                                keypressed = 0;     //clear keypress
                                state++; //increment to move to next state
                                position = 0;       //reset position index
                        }
                }
        }

        while( state == 2 )          //player repeat sequence state
        {
                RBIF = 0;                           //clear RB interrupt flag
                RBIE = 1;                           //enable RB interrupt

                PORTA = keypressed;                 //display pressed key on LEDs

                if( count == WAITTIME )             //if player takes too long
                {                                   //move to game over state
                        count = 0;
                        TMR0 = 0;
                        RBIE = 0;
                        state = 4;
                }
                RBIE = 0;
        }

        while( state == 3 )          //key press verification state
        {
                RBIE = 0;                           //disable RB interrupt

                //check if keypressed = nextnumber
                if( keypressed == (0b1 << getFromSequence(position)) )
                {
                        count = 0;
                        position++;                 //move position to next
                        state = 2;                  //back to player repeat
                        if( position > round )      //back to sequence replay
                        {                           // if finished sequence
                                round++;
                                state = 1;
                                position = 0;
                                keypressed = 0;
                                PORTA = 0;
                        }
                }
                else                                //pushed wrong button
                {
                        state = 4;                  //go to game over state
                }
        }

        while( state == 4 )          //game over state
        {
                if(RA0)          //toggle between sounds in game over state
                        sound(64000, 400);
                else
                        sound(64000, 600);

                //if( count == FLASHDELAY/2 )
                //{
                        count = 0;
                        TMR0 = 0;
```

```
                        RA0 = RA0 + 1;          //toggle the four output LEDs
                        RA1 = RA1 + 1;          //...
                        RA2 = RA2 + 1;          //...
                        RA3 = RA3 + 1;          //...
                //}
            }
        }


}

/*****************************************************************************
 *Interrupt service routine function for handling interrupts
 *****************************************************************************/
void interrupt interruptRoutine(void)
{
        char tempkey;

        tempkey = PORTB;        //grab portb right away and save it in case of rb interrupt

        if( T0IF == 1 )         //handle timer interrupt
        {
                count++;
        }
        else if( RBIF == 1 )    //handle RB input interrupt
        {
                RBIE = 0;       //turn off RB interrupt to prevent interrupt between states
                if( (tempkey & 0b11110000) != 0b11110000 )
                {
                        keypressed = (0b11111111-tempkey) >> 4;

                        sound1(24000, 100+50*keypressed);

                        randnum = TMR0 & 0b00000011;
                        state++;
                }

        }

        RP0 = 0;
        T0IF = 0;       //clear timer0 interrupt flag
        RBIF = 0;       //clear RB port change interrupt flag
}

/*****************************************************************************
 *Function to add number to sequence.
 *****************************************************************************/
void addToSequence(char newnumber, char location)
{
        char bitnumber;
        char bytenumber;
        char temp1;
        char temp2;

        bitnumber = (location & 0b11) << 1;
	bytenumber = (location >> 2);		//location>>2 == location/4 but faster, unless the compiler is
smart, but i dont know
	bytenumber = bytenumber & 0b00111111;

	//get the byte at bytenumber,
	//mask out the two bits we want to overwrite/where we want to place newnumber,
	//add our newnumber shifted to the right place, and put the byte back.
	//2*bitnumber because newnumber is 2 bits, so there are 4 locations per byte

	temp1 = 0b11111111 - (0b11 << bitnumber);
	temp2 = (sequence[bytenumber]) & temp1;
	temp1 = newnumber << bitnumber;
	sequence[bytenumber] = temp2 + temp1;
}

/*****************************************************************************
 *Function to get a number from the sequence
 ****************************************************************************/
char getFromSequence(char location)
{
	char bitnumber;
	char bytenumber;
	char temp1;
	char temp2;

	bitnumber = (location & 0b11) << 1;
	bytenumber = (location >> 2);
	bytenumber = bytenumber & 0b00111111;

	//get the byte at bytnumber,
	//mask out everything but the 2 bits we want,
	//shift the bits right to make them the LSBs, and return it.

	temp1 = 0b11 << bitnumber;
	temp2 = sequence[bytenumber] & temp1;
	temp1 = temp2 >> bitnumber;

	return temp1;
}

/*****************************************************************************
 *Routine to generate sound (used by interrupt routine)
 ****************************************************************************/
void sound1(unsigned int duration, int pitch)
{
	int i,j;
	for(i = 0; i < (duration/pitch); i++)
	{
		for(j = 0; j < pitch; j++ )
			RB0 = RB0+1;
	}
}

/*****************************************************************************
 *Routine to generate sound (same as previous, but used by all other routines)
 * (needed because you cant use a function in both interrupt routine and elsewhere)
 ******************************************************************************/
void sound(unsigned int duration, int pitch)
{
        int i,j;
        for(i = 0; i < (duration/pitch); i++)
        {
                for(j = 0; j < pitch; j++ )
                        RB0 = RB0+1;
        }
}
```

Appendix C – Hardware Schematic


P15
Sound
To PC
13.5 MHz
PIC
16F84
+6v
P17
P10
P18
P11
P1
P12
P2
2kΩ
P13
P6
P16
+6v