**LAB 2 REPORT**

LITEC

Jordan Cabahug-Almonte

Jacob Montenieri

Kristen Weatherbee

Section 4B

Teng Liu

March 2, 2018

**Table of Contents**

[**Introduction and Statement of Purpose**](#_9vock6vuodns) **3**

[**System Description**](#_n95xdqfpqqms) **4**

[Description of Hardware](#_slr86gy1ey1q) 4

[Table of Inputs and Outputs](#_9a6hsrwncbc3) 5

[Schematic](#_9fgnvog1c6z2) 6

[Software Description](#_b27txienz2lv) 7

[Pseudocode](#_yjhe3kji7fw2) 10

[**Results and Conclusions**](#_6mbt06t1fpmn) **16**

[**List of References**](#_hnexg7ssc4np) **18**

[**Appendix A - Mode 1 Terminal Printout**](#_7aqwzgjji2z4) **19**

[**Appendix B - Mode 2 Terminal Printout**](#_a0kngyc8irlg) **20**

[**Appendix C - Results of Game**](#_muck9c5wq8dx) **21**

[**Appendix D - Academic Integrity and Participation**](#_3xuminl9552) **22**

[Academic Integrity Certification](#_ocq8qzpbq5o5) 22

[Participation](#_147n2zr) 22

# 

# **Introduction and Statement of Purpose**

This was the second lab we completed through this course. This lab implemented timers, interrupts, debouncing, analog to digital conversions, and port initializations to create outputs and inputs. The objective of this lab was to introduce analog-to-digital conversion with a potentiometer and to develop an interactive game using a microcontroller. We programmed delays in the inputs and outputs because the microcontroller runs faster than a human’s interactions. This made the game more user-friendly.

We created the game “LITEC Morse Code”. It played one of two modes. Each mode required the player to identify the Morse code from a random letter from A-J. Before the player began the game, they chose the mode through a slide switch. They also had the option of adjusting the speed of the Morse code patterns by turning the potentiometer.

The first mode generated the Morse code from a random letter through sounding a buzzer. The player had to type the letter that matched the Morse code that they heard from the buzzer. The second mode printed out a random letter and instructs the player to enter that letter’s Morse code through pushbuttons that sound dots and dashes on the buzzer.

To indicate if the player’s answer was correct in either mode, the BILED turned green. Otherwise, the BILED turned red and ten penalty points are added to the player’s score. More penalty points were added when the player took a longer time to input their answer. The points for each round and the total points allocated so far in the game were displayed right after the round was completed. After four rounds of a single mode, the final score was displayed, the BILED blinked from red to green, and the game restarted.

The length of time the buzzer sounded depended on the analog value read from the potentiometer. This value was converted to a delay time. This time was how long the buzzer sounded for a “dot” in Morse code. The buzzer’s delays for “dash”, “spaces”, and separation between dots and dashes also depended on this time.

For mode 2 and the beginning of the game, we considered a debounce for the “enter” pushbutton so that the player’s answer was read properly.

The points added for time penalty were calculated through a conversion of interrupts to seconds.

The learning objective for this lab was to write, wire, and debug a program that implemented all of the skills we have learned so far. The overall goal was to make an enjoyable guessing game that helped its players learn Morse code.

# **System Description**

## **Description of Hardware**

The circuit used for this game consisted of three pushbuttons, one BILED, one buzzer, one slide switch, one 74365 hex buffer chip and one potentiometer along with resistors and connections to power, ground, and the EVB. These components can be broken down into three smaller circuits within the entire circuit.

The first smaller circuit includes the buzzer, buffer chip, two resistors and the BILED with connections to port 3 on the EVB. The C8051 microcontroller does not have enough current to drive these components by itself so the buffer chip connects to the BILED and buzzer to drive these outputs. This happens because of the architecture of the chip. Each pin of the chip has the same output voltage as the input voltage for those particular matching pins. There was also one resistor in series with the buzzer and one in series with the BILED to limit the amount of current going through them.

The second smaller circuit includes the three pushbuttons and the slide switch along with resistors connected to each component. This “sub-circuit” was also mostly connected to port 3 on the EVB aside from the slide switch. The slide switch was connected to port 2. These components did not require a buffer chip because they are digital inputs. However, they did require resistors because the resistors set the default state to logic high. This was needed because the pushbuttons and slide switch read either a logic high value while they were off or read a logic low value while they were on. The inputs read a logic high when they were off because the default state was high and this caused no voltage difference, thus no current. When these inputs were on and they read logic low so current would be flowing due to the voltage difference from high to low.

The third smaller circuit includes the potentiometer connected to the EVB through port 1 and a resistor. This is where the analog value was read from the potentiometer as the analog input. A resistor was also put in series with the potentiometer to limit the maximum analog value to 2.4V, which was our reference voltage.

The combination of these three smaller circuits make up the entire circuitry for the game. The buzzer is used to sound the dots and dashes in Morse code, while the BILED is used to blink red for incorrect and green for correct answers. The three pushbuttons allow the player to submit an answer in Morse code: one pushbutton is to sound a dash, another is to sound a dot, and the last one is to enter the answer. The slide switch is to switch between game modes and the potentiometer is to change the game’s speed.

### **Table of Inputs and Outputs**

**Port 3:**

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Description | Code Variable | Input or Output |
| 3.7 | Buzzer | BUZZER | Output |
| 3.3 | Biled 0 | BILED0 | Output |
| 3.4 | Biled 1 | BILED1 | Output |
| 3.0 | Dash Pushbutton | PBDash | Input |
| 3.1 | Dot Pushbutton | PBDot | Input |
| 3.2 | Enter Pushbutton | PBEnter | Input |

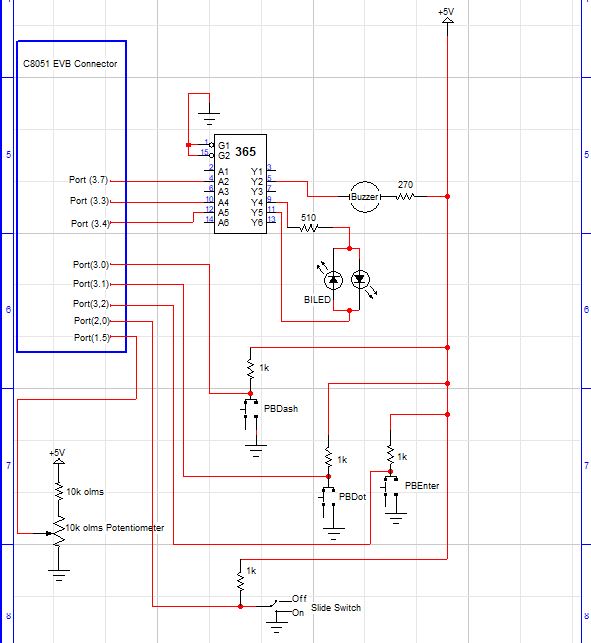
**Port 2:**

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Description | Code Variable | Input or Output |
| 2.0 | Slide Switch | SS | Input |

**Port 1:**

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Description | Code Variable | Input or Output |
| 1.5 | Potentiometer | N/A(uses read\_AD\_input function) | Input (Analog) |

### **Schematic**

****

## **Software Description**

Looking at the program we created, we have several main parts: the A/D value conversion based on the voltage input to calculate dot time, the mode selection, the random Morse code value, a mode that asks the user to input the value (Mode 1), a mode that asks the user to sound the value (Mode 2), checking the answer the user gives, giving the user points for their choice, and adding points to the final score based off of their trials. We also had to implement other functions, such as the initialization functions.

There are several initialization functions that we used. We used a port initialization method. This function also calls the analog to digital initialization function. With this function, our C8051 device configures a port pin to create an analog value from 0 to 255 based on a reference voltage and then an input voltage is sent through the pin. We also used a timer initialization to enable Timer 0 in 16-bit mode in the C8051 to keep track of overflows and interrupts. In this case, one overflow is 35.5 milliseconds. With this, we also need an interrupt initialization and an interrupt service routine to keep track of the amount of overflows in the program.

In the main function, we, first, initialize the system and use the “putchar(‘ ’)” function. We call these two functions so that the C8051 can properly read the code. The buzzer and the BILED were set to 1 to turn them off. Then the instructions are printed out at the start the game instructing the player to turn the potentiometer to vary the speed of the game and switch the slide switch to change the mode that the user plays. After doing this, the program waits for the user to hit the “enter” pushbutton so that they can keep changing the values until they are ready. After the user hits this pushbbutton, the analog to digital method reads the A/D value off of a certain pin. For our program, the value is read off of pin 5. The A/D conversion reads the value of the voltage from the potentiometer and will generate a number from 0 to 255. This value is then passed through the dot time calculator (“dot\_time\_calculator(z);”), where ‘z’ is the A/D value. It is passed so that the time for a dot can vary every time the potentiometer is changed. In this method, we return the A/D value multiplied by a constant plus 0.1. This constant is 0.9 divided by 255 because we equated the A/D value over 255 to the time we wanted over 0.9 seconds. We set it to a max of 0.9 because the dot time had to range from 0.1 to 1 second and we could not have a value of 0 seconds. We used 0.9 seconds and added 0.1 seconds to change it to a range of 0.1 to 1 second. After getting this value, we store it in a variable so that it can be used in other functions that will need it later. Before entering either mode, we seeded the random function to get different Morse code values every time we run trials. To do this we use the “Rand\_Seed();” function which has the user enter a random character to seed the random function with “srand()”. After doing this, we used the slide switch variable to change from Mode 1 to Mode 2.

For Mode 1, the program outputs the noise by the buzzer of a given letter (A through J) in Morse code and the user has to enter what letter they think is being projected. The first step in this mode is to get the random letter. This is done by calling the “Get\_Rand();” function which creates a random number from 0 to 9 using modulus 10. Different letters corresponded to the different numbers generated, where 0 was ‘a’ and 9 was ‘j’. The random letter generated is sent to the Morse code method (“morsecode(‘x’);”), where x is the random letter. This function uses a case method to sound the buzzer for dots and dashes based on the parameter sent to it. To make the sound of dots and dashes, we call “buzz(y);” while inside “morsecode(‘x’);”. The y value that is sent to buzz is one of three things: 0, 1, or 2 which represent, respectively, space, dot, and dash. In the buzz method, the buzzer turns on for a duration of the dot time calculated for dots, off for the duration of that dot time for space, on for three times the duration of that dot time for dashes, and off for the duration of three times the dot time to separate the dots and dashes. After doing this the buzzer turns off and the “buzz(y);” method ends. Since every letter has a combination of 4 of these choices, every time “morsecode(‘x’);” is called, the buzz method is called four times. After finishing the last buzz method, the program stores the letter we just sounded into a variable and compares it to what the user enters. Next, the timer starts and we keep track of how long it takes the user to enter an answer through count variables that increment by overflows in the interrupt service routine function. Once they enter an answer the timer is stopped and the amount overflows allocated right before they put in their answer is stored to figure out the time penalty to add to their score. Every half second (169 overflows) they take to input their answer adds one point to their score. If their answer matches the randomly generated letter, the BILED will turn green, but if their answer is wrong the BILED will turn red and ten penalty points are added to their score. Once the total score for the round is determined by adding these values, the mode runs three more times adding all the runs to a final score that prints at the end. Before closing Mode 1, the BILED blinks green and red for one second to signify that it is over. After closing, it goes back to the beginning of the main function.

For Mode 2, the program will tell the user how to use pushbuttons to represent dots and dashes and how to enter the answer. After doing this, the program will generate a random answer using the random function “Get\_Rand();” similar to mode 1 and store it in a variable to compare the user answer later. Then “Mode2print(x);” is called where x is the random number we just called. This method creates print statements of what letter the user should enter in Morse code from A to J with the buttons for dots and dashes. After doing this, we set a sentinel flag to false and the user presses the pushbuttons for dots and dashes as needed in a while loop. If the user pushes the dot or dash button, an array is updated with four locations. Initially, each index in that array is filled with a default 0 value. An index in the array is updated with either 1 or a 2 in the pattern the pushbuttons are pressed until the user hits the enter pushbutton to leave the while loop. To help the user understand what they pressed, we used another type of buzzer method which is called “buzz2(x)” as the dot and dash pushbutton are pressed. It is similar to the buzz method we used before, but this function will only accept 1 and 2 as answers for dots and dashes respectively since the user does not have to enter spaces. This is because the spaces (0’s) are in the array initially. When the user enters these values, the buzz2 method will also sound as soon as the corresponding pushbuttons are pushed. This allows the user to have a more accurate game experience. We also deducted the amount of overflows collected when the dot or dash button was hit from the total overflow so that the time that the user takes to enter their answer was not affected by waiting for the dots and dashes to sound. We decided to make the sentinel flag so that when the user hits the enter pushbutton, the flag will be activated causing the while loop to end. We decided to initialize the array to zero so that if a letter does not use all 4 spots it can be exited early and still work in our comparison method. After exiting the while loop for the input, the next method called is the “check(x,y);” method, where ‘x’ is the user’s input array from the previous while loop and the ‘y’ variable is the generated random value from the beginning of mode 2. We initially assigned the random value to a letter from 0 to 9. The check function checks each index in the array to see if it matches the generated random letter and its corresponding value with a sequence of ten if statements. If check returns true, the system prints correct and the BILED flashes green, but if it is false, the system prints incorrect and the BILED flashes red and adds 10 points to the user’s score. Then program adds a point for every half second the user took to input their answer and adds it to the round score before adding this total to the final. When the final score has been printed the BILED flashes for a second to let the player know the round is over. At the beginning of the new loop, gamescore is set to zero again, but final score is kept the same until the end of the fourth loop. After completing all four trials, the game ends and goes back to the beginning of the program like it did at the end of mode 1 and will ask the user to set the switch and potentiometer to start again.

### **Pseudocode**

compiler directives

#include <c8051\_SDCC.h>

#include <stdio.h>

declare global variables

Sbit BUZZER, BILED1, BILED0, PBEnter, PBDot, PBDash, LED1, LED0, SS

Int i, j, k, generatedAnswer, flag, AD\_value

Unsigned int counts2, input[4], overflow\_count, randumnum = 0, game\_score = 0,

final\_score = 0, saved\_overflow = 0, overflowCount = 0

Char randchar

Unsigned char inputrand,

function prototypes

Void ADC\_Init(void)

Void Port\_Init(void)

Void Timer\_Init(void)

Void Interrupt\_Init(void)

Unsigned char read\_AD\_input (unsigned char pin\_number)

float dot\_time\_calculator (int value)

Void Rand\_Seed(void)

Void Mode1(void)

Void Mode2 (void)

Int Get\_Rand(void)

Void buzz(int type)

Void Mode2print(int random)

Unsigned int check(unsigned int input[4], unsigned int rand)

void morsecode(char letter)

Void buzz2(int buzz\_type)

main function

declare local variables

(NONE)

Initialization functions

Sys\_Init();

putchar(‘ ‘);

Port\_Init();

Interrupt\_Init();

Timer\_Init();

ADC\_Init();

Print start and turn off buzzer

Begin infinite loop

Turn off BiLED and BUZZER

Print out instructions on how to turn on different modes and how to

adjust the speed with the potentiometer

Print to ask user to press the enter pushbutton (PBEnter)

While PBEnter is pressed and released

Read AD\_value off of port 1 pin

Print out AD\_value

Set dotTime to dot\_time\_calculator(AD\_value)

Print out dotTime

Rand\_Seed()

If SS

Print to tell user they are playing mode 1

Mode1()

else

Print to tell user that they are playing mode 2

Mode2()

Functions

Void Port\_Init

Configures Port 1 pin 5 to be set to an analog input (potentiometer)

Configures Port 2 with SS

Configures Port 3 with outputs of BUZZER, LED1, LED0, BILED0, and BILED1,

and inputs of PBEnter, PBDash, and PBDot

End Port\_Init

Void ADC\_Init(Void)

Initializes analog to digital converter with a gain of 1

End ADC\_Init

Unsigned char read\_AD\_input(unsigned char pin\_number)

Reads of specific pin number

Starts and ends analog conversion

Returns ADC1 value

End read\_AD\_input

float dot\_time\_calculator (int value)

Calculates how long the buzzer will buzz for a single dot, based on

potentiometer value

Set new variable (dot\_time) to incoming potentiometer value

dot\_time = [(A/D conversion result from the potentiometer) \*

0.0035294118 + 0.1]

return dot\_time

End dot\_time\_calculator

Void Rand\_Seed(void)

Print instructions to press any button on the keyboard

Seeds the number generator

End Rand\_Seed

Int Get\_Rand(void)

Returns a random number from 0-9

End Get\_Rand

Void Mode1(void)

Set game\_score and final\_score to 0

For a loop that repeats 4 times

Print “The board will generate morse code for a letter from a to j

Type the letter that is chosen when prompted.”

Turn off BiLED

Set game\_score to 0

randomnum will get a random number from the get\_rand

For a loop that repeats four times

Generate a random number from 0-9 and get the corresponding letter

If (random number == 0)

Letter is A

morsecode(A)

…

If (random number == 9)

Letter is J

moresecode(J)

Stop timer0

Clear the timer overflow variable

Start timer0

Wait for keyboard press with the user’s potential answer (input)

Save overflow variable value at this point

If key is correct

Green BiLED for correct

Else

Red BiLED for wrong

Add 10 points to game\_score

For every 500 ms on overflow variable

Add 1 point to game\_score

Display total points for the try and total score with previous

trials

Set overflowCount to 0

Start timer0

Delay for half a second before turning BILED off

End for loop

Display final\_score

Set overflowCount and overflow\_count to 0

For 1 second flash BILED from red to green

End Mode1

void morsecode(char letter)

Use C switch/case statement with a set of commands for each letter

If letter == ‘a’

Buzzer sounds dots and dashes for letter ‘a’ with buzz function

Break

…

If letter = ‘j’

Buzzer sounds dots and dashes for letter ‘j’ with buzz function

Break

End morsecode

void buzz(int buzz\_type)

Set overflow\_count to 0

if buzz\_type == 1

Sound buzzer for dot for as long as dotTime

if buzz\_type == 2

Sound buzzer for dash for as long as 3 times the dotTime

if buzz\_type = 0

Don’t sound buzzer for as long as dotTime

wait 1 dot duration in between dots and dashes

End buzz

Void Mode2(void)

Print instructions for mode2 indicating what each push button does

(PBEnter, PBDot, PBDash)

For 4 loops

Clear input array with all 0’s and set index to 0 with a for loop

Stop timer0, set counts2 to 0, then start timer0 again

Generate random number from generatedAnswer and Get\_Rand()

Print out statement depending on what generatedAnswer is with

Mode2print

Set index (i) to 0

Set flag to 0 (False)

Set game\_score to 0

Turn off BILED

Clear timer overflow\_count value

While flag is 0 (False)

If !PBDot

sound dot symbol with buzz2(1), doesn’t account for

spaces

input[index] = 1

increment index

Take time of dot sound out of counts2

If !PBDash

Sound dash symbol with buzz2(2), doesn’t account for spaces

Input[index] = 2

Increment index

Take time of dash sound out of counts2

While PBEnter is not pressed

Set flag to 1 to break while loop

Stop Timer0

set overflow\_count 0

Start Timer0 again

Debounce PBEnter

End while loop when PBEnter is released

End while loop when flag is 1 (True)

If generatedAnswer is the same as input array (use check function)

Print Answer is Correct

BILED is green

Don’t add any points to game\_score

Else

Print Answer is Incorrect

BILED is red

Increment game\_score by 10

Add overflow\_count/169 to game\_score

Add game\_score to final\_score

Display game\_score for the round and the final\_score

Set counts2 to 0

End for loop with 4 loops

Display the final score after 4 rounds

Set overflowCount to 0

Set overflow\_count to 0

Blink the BILED from red to green for one second

End Mode2

Void Mode2print(int random)

If the randomnum is 0

Print statement to tell user to input morsecode for A

…

Else if the random num is 9

Print statement to tell user to input morsecode for J

End Mode2print

unsigned int check(unsigned int input[4], unsigned int rand)

If input array matches the morse code for A (rand ==0) {1,2,0,0}

Return 1 (True)

…

Else if input array matches the morse code for J (rand==9) {1,2,2,2}

Return 1 (True)

Else

Return 0 (False)

End check

Void Interrupt\_Init(void)

Enables timer0 and global interrupts

End Interrupt\_Init

Void Timer\_Init(void)

Activates SYSCLK, puts timer in 16-bit mode, stops timer

End Timer\_Init

Void Timer0\_ISR(void)\_\_interrupt 1

Increments overflowCount, overflow\_count, counts2

End Timer0\_ISR

Void buzz2(int buzz\_type)

Set overflow\_count to 0

if buzz\_type == 1

Sound buzzer for dot for as long as dotTime

if buzz\_type == 2

Sound buzzer for dash for as long as 3 times the dotTime

End buzz2

# **Results and Conclusions**

We successfully developed and debugged Lab 2 within a week. With the completion of lab 2, we created the “LITEC Morse Code” game. This was a 2 mode game that required the players to know the Morse code from a randomly generated letter from A-J. The speed of the Morse code pattern was adjusted by a potentiometer. This parameter adjusted the difficulty of the game. The first mode sounded the Morse code from the random letter and allowed the player to enter the corresponding letter. The second mode printed the random letter and allowed the player to sound the buzzer to emulate the corresponding Morse code pattern. Each mode played through four rounds before displaying the total penalty points the player allocated throughout the game.

Each of us was responsible for a different subsystem of the game. One team member was responsible for controlling the A-to-D conversion with the potentiometer and other miscellaneous functions (initialization functions, printing functions, and a Timer Interrupt function). The other two team members were responsible for writing functions for Mode 1 and Mode 2. To debug the game, we separately tested each subsystem of the game. We first tested the speed of the game by seeing how accurately the time was adjusted by the potentiometer. Then, we tested and debugged each mode of the game.

Throughout this lab, we learned and practiced important embedded control concepts. We learned how to initialize ports, how to use the microcontroller’s timer, how to debounce certain inputs, how to use overflows to increment values, and how to convert analog values to digital values. Initializing the ports was a continuation of what we did in the worksheets and Lab 1. We had to use timers and A-to-D conversion to adjust the speed of the game. We calculated a certain time for how long the buzzer had to sound for a “dot” in Morse code. We also used interrupts and timers to debounce certain pushbuttons.

We came across simple hardware problems early on when testing Mode 1. There were several times where we did not realize that a ground or power wire was not connected properly through the protoboard. This realization helped us debug the Mode 2 hardware faster. As far as the coding aspect of the lab, Mode 2 was harder to implement than Mode 1 because it incorporated more pushbuttons. In this mode, we had to use debouncing for the “enter” push button. The other pushbuttons activated functions associated with the buzzer that took a longer time for the microcontroller to process so no debouncing was needed. Debouncing the “enter” pushbutton was essential. This allowed the microcontroller to read the player’s full answer first and then check it with the “enter” pushbutton. Adding time penalty points to the player’s score in mode 2 also brought about minor difficulties. We did not want to penalize the player for the time the buzzer sounded. Otherwise the penalty score for Mode 2 would be significantly higher than Mode 1. Taking this into consideration, we subtracted the amount of overflows that were calculated during a “dot” and a “dash” sound from the total amount of overflows and then used that parameter to calculate the time penalty.

Once the lab was completed, we tested the entire game by playing the game ourselves. Each of us played each mode twice at different speeds. We kept track of how long the buzzer sounded by mentally counting the seconds the buzzer sounded. Penalty points for incorrect answers and delay of the answers were kept track by hand. The results at the end of the game matched our desired results.

There are several things we know now that we would have done before finishing this lab. To debug the simple wiring issues, we would use a logic probe more meticulously to see the different voltage highs and lows throughout the protoboard. This would have helped us clearly see the problems with our LED’s, buzzer, and pushbuttons sooner. This would have also showed us what was connected and not connected to the protoboard. If we had a newer protoboard, we would also be able to avoid this issue.

We also found that Mode 2 was slightly more difficult than Mode 1 after playing the game repeatedly. Overall, we collected more penalty points in this mode. This is most likely due to the issues we had from differentiating between the three buttons. If we had access to different colored or shaped pushbuttons, the player could figure out what each button did before the game in the instructions displayed. We could have also placed the buttons in a more organized manner on the protoboard to solve this issue. Our overall wiring should have been neater, but we will take this into consideration for future labs. To make this game even more challenging, we would have implemented the full alphabet into the game, instead of only the first 10 letters.

Despite these improvements that could have been made, we are still satisfied with how the game turned out in the end. The game was enjoyable and user friendly. We were able to get through several rounds of the game without any difficulty.

# 

# **List of References**

International Morse Code. (2017). <<https://morsecode.scphillips.com/morse2.html>>.

RPI-ECSE. (2017). Laboratory Introduction to Embedded Control Lab Manual v14.8. Troy, NY.

# 

# **Appendix A - Mode 1 Terminal Printout**

## 

# **Appendix B - Mode 2 Terminal Printout**

# 

# 

# **Appendix C - Results of Game**

# 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dot\_time | Final Score (out of 4 tries) |  |  |
|  |  | Player 1: Kristen Weatherbee | Player 2: Jacob Montenieri | Player 3: Jordan Cabahug-Almonte |
| Mode 1 | 0.975 | 33 | 20 | 33 |
| Mode 1 | 0.707 | 22 | 5 | 17 |
| Mode 2 | 0.566 | 27 | 20 | 54 |
| Mode 2 | 0.873 | 24 | 21 | 69 |
| Total |  | 106 | 66 | 173 |

# 

# **Appendix D - Academic Integrity and Participation**

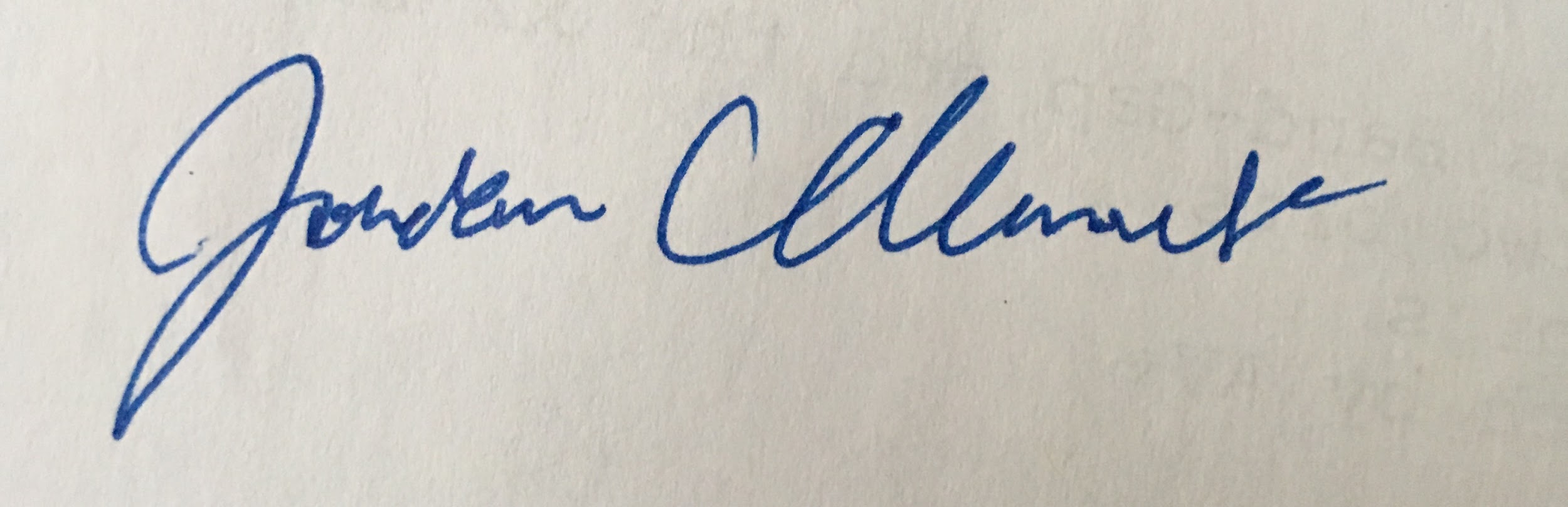
## **Academic Integrity Certification**

All the undersigned hereby acknowledge that all parts of this laboratory exercise and report, other than what was supplied by the course through handouts, code templates and web-based media, have been developed, written, drawn, etc. by the team. The guidelines in the Embedded Control Lab Manual regarding plagiarism and academic integrity have been read, understood, and followed. This applies to all pseudo-code, actual C code, data acquired by the software submitted as part of this report, all plots and tables generated from the data, and any descriptions documenting the work required by the lab procedure. It is understood that any misrepresentations of this policy will result in a failing grade for the course.

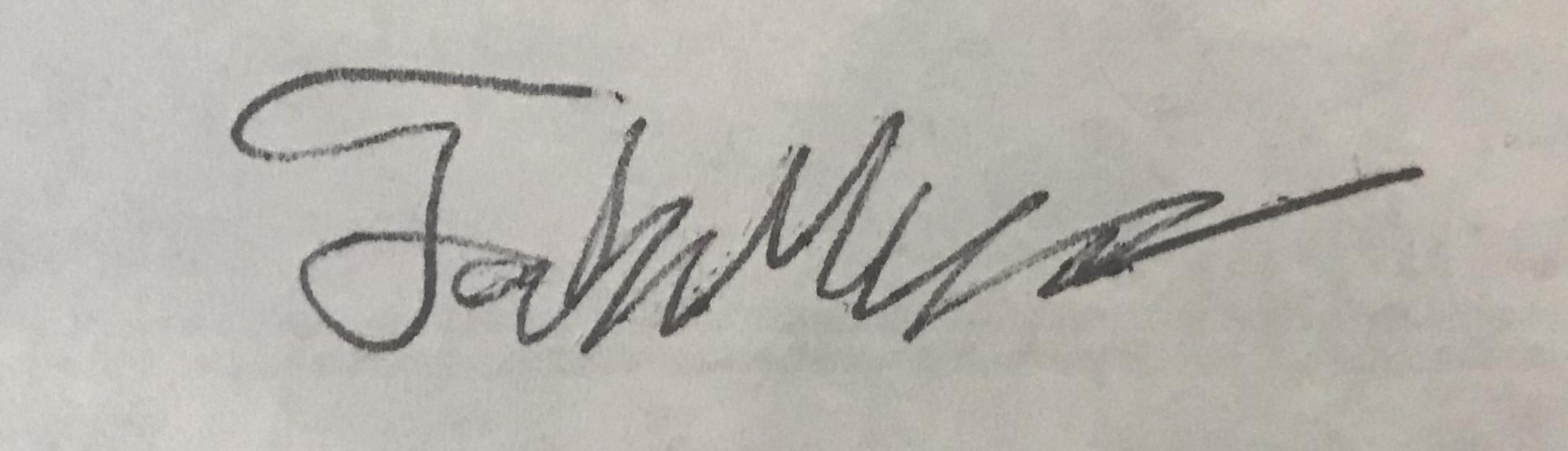
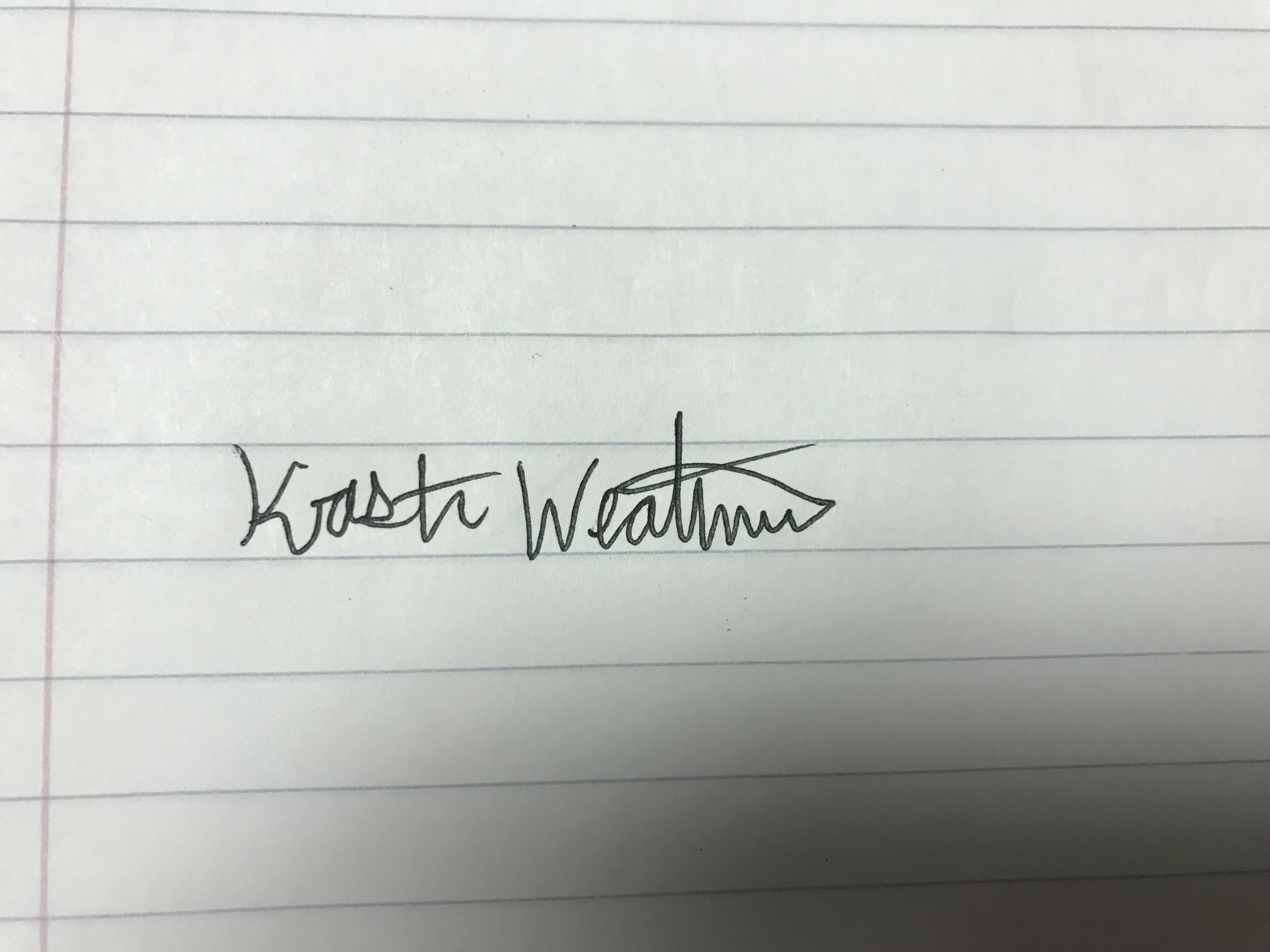
## **Participation**

|  |  |
| --- | --- |
|  | **Lab 2** |
| Lab Description | Jordan (100%) |
| Worksheets | Jordan (100%) |
| Pseudo code | Jordan (60%), Kristen (20%), Jacob (20%) |
| Actual code | Jordan (20%), Kristen (40%), Jacob (40%) |
| Debugging | Jacob (70%), Kristen (30%) |
| Schematic | Kristen (100%) |
| Pinout | Jacob (100%) |
| Notebook organization | Jordan (50%), Jacob (50%) |
| Circuit Assembly | Jacob (100%) |
| Report Editing and Organization | Jordan (100%) |

The following signatures indicate awareness that the above statements are understood and accurate.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_3/2/18\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_3/2/18\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Date: \_3/2/18\_\_\_\_\_\_\_\_\_\_