ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO : 7

EXPERIMENT DATE : 8.01.2021

LAB SESSION : FRIDAY - 15.30

GROUP NO : G11

GROUP MEMBERS:

150170063 : BURAK ŞEN

150180080 : BAŞAR DEMİR

150190719 : OBEN ÖZGÜR

FALL 2020

Contents

1 INTRODUCTION

In this week's experiment, we have learned working principles of communication protocols such as UART, I^2C, SPI and CAN. We implemented UART communication protocol with our Arduino circuit boards. We also used I^2C protocol to communicate each of the Arduino circuits. Also, we have experienced usage of flex sensors and servo-motor components.

2 MATERIALS AND METHODS

- Tinkercad
- 3 Arduino Uno R3
- 4 Flex sensors
- 4 Micro Servos
- Jumper Cables

2.1 PART 1

In the first part of the experiment we were supposed to read the analog values from flex sensors and print analog value, resistor and angle of each sensor on serial monitor. For angle calculation, we have assumed that the flex sensors work linearly which means the angle is directly proportional to resistance of the sensor. But in real life, flex sensors don't work linearly. That is the reason why calculated and observed result will differ. The comparison will be shown in results section.

At first, we have calculated analog at the port of the flex sensor. Then, we have converted it to voltage by mapping it between 0-1023.
After voltage calculation, since we know that the other port of the 100k resistor is fed with 5V, we were able to calculate the current going through the 100k resistor and the flex sensor. We have calculated it using our basic circuitry analysis knowledge.
With the current's value we were able to calculate the corresponding resistor value of the flex sensor.
To get 5Hz sampling we set our delay to 200ms using waitMillis function

Linear Working Assumption

Finally, assuming the flex sensor works linearly. We have scaled the minimum and maximum values of the resistor to angle values and calculated the angle of the flex sensor.

Our implementation is given below.

```
int flexs[4] = {A0,A1,A2,A3}; //set pins

void setup()
{
  Serial.begin(9600); //for using serial monitor
  pinMode(flexs[0], INPUT);
  pinMode(flexs[1], INPUT);
  pinMode(flexs[2], INPUT);
  pinMode(flexs[3], INPUT);
}

//function for delaying
void waitMillis(int time)
{
  long start_time = millis();
  while(millis() - start_time >= time);
}


void loop()
{
  for(int i=0; i<4;i++){ //loop through pins
  double data = analogRead(flexs[i]);   //read pins analog value
    double readVoltage = (data/1023.0)*5.0;     //convert it to voltage
    double current = (5.0 - readVoltage)/100.0; //calculate current
    double resistance = readVoltage/current; //calculate flex resistance
    //calculate angle assuming its linear
    double angle = abs(((resistance-29.99)/132.99)*180.0);
    Serial.print('!');  //print accordingly
    Serial.print(i);
    Serial.print(';');
```

```
      Serial.print(data);
      Serial.print(';');
      Serial.print(resistance);
      Serial.print(';');
      Serial.print(angle);
      Serial.println('#');
    }
    waitMillis(200);
}
```

2.2 PART 2

In this part of experiment, we are expecting to implement a circuit that manipulates servo motors according to input that is given to Serial Input (**UART Protocol**). Also, we have to check input format to prevent wrong user inputs. Corresponding to error type, we have to write error message and prevent execution of the input.

Firstly, we have defined necessary variables and initialized them as global. These are buffer for input streams, size for counting input size, temp for parsing input strings. Also, in setup phase, we have defined servo motors and their pins. They are initialized with 0 angle value.

Arduino allows us to read values from Serial Monitor. For this part, we have use this functionality of the micro controller. With Serial.available() function, we can check input size that is pending to read. If there exist a input we read it character by character and store them in buffer. After reading phase, we have to check all input to determine it is proper to format or not. Many inputs can be given in one line and if one of the input does not given in right format, also other inputs must not be executed. For this situation, we have implemented a function that is responsible for checking all inputs. If it returns valid, we execute input.

For the validation phase, we have considered all possible input combinations. We have performed input character validation while reading serial input. For other errors, we called checker function. The checker function, based on divide and conquer principle. It parses given input from # and ! characters and calls another function to validate input that is located between these characters. In checker function, we validate start and end characters in the given input. In check command function, we validate characters between # and !. There might be insufficient number of parameters, not filled

parameters, extra parameters etc. To determine these type of error, we iterate over the command and check position of the semicolons and consecutive distribution of them. If input format is proper, then we check values of given parameters. Angle values must be between 0-180 degrees. By casting characters to integers, we determined that parameters is between these limits.

If the given input is in proper format, we call execute function. This function is based on checker function without validation statements. We parse input again and we reach parameter values. Then, we write these values into servo motors. Our implementation is given below.

```
#include <Servo.h>

//Defined servo motors as servo1,2,3,4
Servo servo1;
Servo servo2;
Servo servo3;
Servo servo4;

//Defines corresponding pins
int servoPin1 = 3;
int servoPin2 = 5;
int servoPin3 = 6;
int servoPin4 = 9;

char buffer[100]; //buffer for input stream
char temp[100]; //helper for string checking operations

int size = 0; //input size variable

void setup() {
  Serial.begin(9600); // opens serial port
  //associates pins and servos
  servo1.attach(servoPin1);
  servo2.attach(servoPin2);
  servo3.attach(servoPin3);
  servo4.attach(servoPin4);
  //initialized servos with 0
```

```
  servo1.write(0);
  servo2.write(0);
  servo3.write(0);
  servo4.write(0);
}

//function that parses commands
void execute(){
  int i = 1; //iterator starts with 1
  int temp_index=0; //keeps tamp array index
  bool expecting_hash = true; //flag for #
  //iterates over the buffer
  while(buffer[i]!='\0'){
    //if it comes to end of first input
    if(buffer[i]=='#' && expecting_hash){
      //adds end chat to temp
      temp[temp_index] = '\0';
      //it does not expecte #
      expecting_hash = false;
      //calls print command
      printCommand();
      temp_index=0; //updates temp index
      i++; //increments iterator
      continue;
    }
    //if it is start character
    if(buffer[i]=='!' && !expecting_hash){
      //it expects #
      expecting_hash = true;
      temp_index = 0;//updates temp index
      i++; //increments iterator
      continue;
    }
    //adds char from buffer to temp //increment
        temp[temp_index]=buffer[i];
    i++; //increment iterator
    temp_index++;//increment temp index
```

```
  }
}

//function that executes string between ! and #
void printCommand(){
  int i=0; //initialized iterator
  int semicolon_counter =0; //keeps number of semicolons
  int semicolon_index[3]; //keeps positions of semicolons
  //it iterates over the command
  while(temp[i]!='\0'){
    //if element is semicolon
    if(temp[i] ==';'){
      //keeps index of semicolon
      semicolon_index[semicolon_counter] = i;
      semicolon_counter+=1; //increments counter
    }
    i++; //increments iterator
  }
  Serial.print("S0:");
  int item = 0;
  int pow = 1;
  //it iterates over the first input
  for(int j=semicolon_index[0]-1;j>=0;j--){
    //transforms character to integer
    item+= (int)(temp[j]-48)*pow;
    pow*=10;
  }
  //send value to servo3
  servo1.write(item);
  Serial.print(item);
  Serial.print("; ");
  Serial.print("S1:");

  item = 0;
  pow = 1;
  //it iterates over the second input
  for(int j=semicolon_index[1]-1;j>semicolon_index[0];j--){
```

```
    //transforms character to integer
    item+= (temp[j]-48)*pow;
    pow*=10;
  }
  //send value to servo2
  servo2.write(item);
  Serial.print(item);
  Serial.print("; ");
  Serial.print("S2:");
  item = 0;
  pow = 1;
  //it iterates over the third input
  for(int j=semicolon_index[2]-1;j>semicolon_index[1];j--){
    //transforms character to integer
    item+= (temp[j]-48)*pow;
    pow*=10;
  }
  //send value to servo3
  servo3.write(item);
  Serial.print(item);
  Serial.print("; ");
  Serial.print("S3:");
  item = 0;
  pow = 1;
  //it iterates over the fourth input
  for(int j=i-1;j>semicolon_index[2];j--){
    //transforms character to integer
    item+= (temp[j]-48)*pow;
    pow*=10;
  }
  //send value to servo4
  servo4.write(item);
  Serial.println(item);
}

//function that checks input structure
bool checker(){
```

```
//if its first character is not !
if(buffer[0] != '!'){
  //prints error message
  Serial.println("Error! There is no start char.");
  return false;
}
//if its first character is not #
if(buffer[size-1] != '#'){
  //prints error message
  Serial.println("Error! There is no end char.");
  return false;
}
int i = 1; //iterator starts with 1
int temp_index=0; //temp array index
bool expecting_hash = true; //hash is needed or not
//starts iteration until the end of input
while(buffer[i]!='\0'){
  //if # comes, when we do not expect
  if(buffer[i]=='#' && !expecting_hash){
    //prints error message
      Serial.println("Error! There is no start char.");
      return false;
  }
  //if ! comes, when we expect #
  if(buffer[i]=='!' && expecting_hash){
    //prints error message
      Serial.println("Error! There is no end char.");
      return false;
  }
  //if end char arrives
  if(buffer[i]=='#' && expecting_hash){
    //adds end char to temp
    temp[temp_index] = '\0';
    //updates expect bool
    expecting_hash = false;
    //calls check command
    bool valid = checkCommand();
```

```
      //updates temp index
      temp_index=0;
      //if command is not valid
      if(!valid){
        //directly returns false
        return false;
      }
      i++; //increments iterator
      continue;
    }
    //when start char arrives
    if(buffer[i]=='!' && !expecting_hash){
      //it expects for #
      expecting_hash = true;
      //initialized tamp index
      temp_index = 0;
      i++;//increments iterator
      continue;
    }
    //adds char from buffer to temp
        temp[temp_index]=buffer[i];
    i++;//increments iterator
    temp_index++;//increments temp iterator
  }
  return true;
}

//function that checks string between ! and  #
bool checkCommand(){
  int i=0; //initialized iterator
  int semicolon_counter =0; //keeps number of semicolons
  int semicolon_index[3]; //keeps positions of semicolons
  //if directly starts with semicolon
  if (temp[0] == ';'){
    //prints error message
     Serial.println("Error! All parameters must be filled.");
     return false;
```

```
}
//it starts iteration through the temp
while(temp[i]!='\0'){
  //if last element or two consecutive elements are semicolon
  if((temp[i] ==';' && temp[i+1] =='\0') || (temp[i]==';' && temp[i+1]==
      //prints error message
      Serial.println("Error! All parameters must be filled.");
      return false;
  }
  //if element is semicolon
  if(temp[i] ==';'){
      //increments counter
              semicolon_counter++;
      //if semicolon numbers are higher than 3
      if(semicolon_counter>3){
              //prints error message
              Serial.println("Error! There is extra input.");
              return false;
              }
      //takes index of semicolon to array
      semicolon_index[semicolon_counter-1] = i;
  }
  i++; //increments iterator
}
//if number of semicolons is less than 3
if(semicolon_counter<3){
  //prints error message
  Serial.println("Error! There is not enough input.");
  return false;
}

int item = 0;
int pow = 1;
//it iterates over the first input
for(int j=semicolon_index[0]-1;j>=0;j--){
  //transforms character to integer
  item+= (temp[j]-48)*pow;
```

```
  pow*=10;
}
//if it is bigger than 180
if(item>180){
  //prints error message
  Serial.println("Error! Angle value must be less than 180.");
  return false;
}
item = 0;
pow = 1;
//it iterates over the second input
for(int j=semicolon_index[1]-1;j>semicolon_index[0];j--){
  //transforms character to integer
  item+= (temp[j]-48)*pow;
  pow*=10;
}
//if it is bigger than 180
if(item>180){
  //prints error message
  Serial.println("Error! Angle value must be less than 180.");
  return false;
}
item = 0;
pow = 1;
//it iterates over the third input
for(int j=semicolon_index[2]-1;j>semicolon_index[1];j--){
  //transforms character to integer
  item+= (temp[j]-48)*pow;
  pow*=10;
}
//if it is bigger than 180
if(item>180){
  //prints error message
  Serial.println("Error! Angle value must be less than 180.");
  return false;
}
item = 0;
```

```
  pow = 1;
  //it iterates over the fourth input
  for(int j=i-1;j>semicolon_index[2];j--){
    //transforms character to integer
    item+= (temp[j]-48)*pow;
    pow*=10;
  }
  //if it is bigger than 180
  if(item>180){
    //prints error message
    Serial.println("Error! Angle value must be less than 180.");
    return false;
  }
  return true;
}

void waitMillis(int time)
{
  long start_time = millis();
  while(millis() - start_time <= time);
}



void loop() {
  size = 0; //keeps size of input
  bool isRead = false; //input reading finish or not flag
  bool invalid = false; //input validation flag
  //until input stream finishes
  while (Serial.available()) {
    isRead =true; //read flag changes
    waitMillis(2); //2ms delay
    char value = Serial.read(); //reads first char
    if(!invalid){ //if previous part was not invalid
        //checks character types
        if(!((value<=57 && value>=48) || value=='#' || value==';'|| value=
                //if it is not valid
                Serial.println("Error! Invalid Input."); //prints error mes
```

```
                invalid =true; //assigns true to invalid flag
                }
        buffer[size] = value; //takes character into buffer
    }
    size++; //increments size
}
  buffer[size]='\0'; //end character
  //if input is read and characters are valid
  if(isRead && !invalid){
    //calls structure checker
    bool valid = checker();
        if(valid){
        //if it is valid, prints and runs command
                execute();
        }
  }



}
```

2.3 PART 3

In this part of the experiment, we used other codes that we wrote in the previous parts. We implemented a circuit that can take the angle values from the first part's Arduino circuit and send those values to the second part's Arduino circuit accordingly.

In the first part's code, we had to change some part's of it. First of all, we had to communicate with our upper slave Arduino circuit with the master Arduino. To achieve this we user wire library, with this library we can communicate with our Arduino wire begin address number we set the first Arduino' wire address to 8. When master Arduino requests data from the upper slave Arduino, a function must be executed. We attached this function with Wire.onRequest method. We created a function called requestEvent and when master requests data from upper Arduino we trigger requestEvent function with Wire.onrequest method. In the requestEvent function, we use the same angle calculation approach from part 1. However, instead of printing these values, we send these values to an array of characters. Because we have a constant template for servo part, we just set some characters and our calculated values according to this template

("!000;000;000;000" in this template we set each value to 000 because the biggest angle is 180 there must be at least three columns for each angle value. And after calculations we set the values according to their places). After completing our template we just print the template and send our char array to the master Arduino with wire.write method.

In the master Arduino, we just had to communicate with the upper slave Arduino and take the senditems char array from it. To make that happen, we had to use wire library we did not have to set any wire address for the master Arduino. Firstly we have to specify where does our data come from and how many bytes are there. Thus, we have to calculate our bytes and set requestFrom function in the wire library, because we send an array of characters with size 17 our data is 17 bytes (each char is 1 byte). And we set our request address to 8 because our data comes from the upper Arduino which has address 8. We created a local array again for the received data and we created a received boolean variable. To receive the data we have to create a loop that takes each character one by one. Because upper Arduino could send less data than the requested amount we have to check if upper Arduino is available or not for each loop, in this loop we get our characters one by one with wire.read method and assign each character to our local array. We also set our received flag to control if our data is the correct size or not. If our data is received we begin the transmission with wire.beginTransmittion to address 4 (address 4 is our lower Arduino) method after this we send our data with wire.write and send our local array to lower Arduino, after this, we end our transmission with wire.endTransmission method.

In the lower slave Arduino, we used almost all of the functions we just moved the loop function's inner part to receive event function. In this function we did the same thing at the master Arduino we just loop through each character with wire.read and send each character to an array. And if we received our values we call the printvalues function and control our servo motors.

Our implementation is given below.

```
//
//        Upper Arduino
//

#include <Wire.h>
int flexs[4] = {A0,A1,A2,A3}; // flex pins
```

```
void setup() {
  Wire.begin(8);                 // set address #8
  Wire.onRequest(requestEvent); // register event
  pinMode(flexs[0], INPUT); // set flex pinmode 1
  pinMode(flexs[1], INPUT);
  pinMode(flexs[2], INPUT);
  pinMode(flexs[3], INPUT);
}


void loop() {

}

void requestEvent() {   //requestEvent function
  char send_item[17];   // local char array
  send_item[0] = '!';   // first character
  for(int i=0; i<4;i++){
  // for four numbers there must be 4 iteration
        double data = analogRead(flexs[i]); // take the value from flex
    double readVoltage = (data/1023.0)*5.0; //calculate the voltage
    double current = (5.0 - readVoltage)/100.0; //calculate the current
    double resistance = readVoltage/current;    //calculate the resistance
    //calculate angle from the resistance
    int angle = (int)abs(((resistance-29.99)/132.99)*180.0);
    int val = angle%10;//angle's one's place
    send_item[4*(i+1)-1]= 48+val;//set ones place
    angle/=10;
    val = angle%10; // angle's ten's place
    send_item[4*(i+1)-2]= 48+val;   // set tens place
    angle/=10;
    val = angle%10; //angle's hundred's place
    send_item[4*(i+1)-3]= 48+val;   //set hundreds place
  }
  send_item[4]= ';'; //place divider chars
  send_item[8]= ';';
```

```
  send_item[12]= ';';
  send_item[16] ='#'; // place end char
  Wire.write(send_item);     // send data to master
}

//
//       Master Arduino
//

#include <Wire.h>

void setup() {
  Wire.begin();         // join i2c bus (address optional for master)
}

void waitMillis(int time) // custom delay
{
  long start_time = millis();
  while(millis() - start_time <= time);
}



void loop() {
  waitMillis(200);
  Wire.requestFrom(8, 17);    // request 17 bytes from slave device #8
  char send_item[17];
  int i=0;
  bool received = false;
  while (Wire.available()) { // slave may send less than requested
    received = true;
    send_item[i] = Wire.read(); // receive a byte as character
    i++;
  }

  if(received){
    Wire.beginTransmission(4); // transmit to device #4
        Wire.write(send_item);
```

```
        Wire.endTransmission();     // stop transmitting
  }

}

//
//      Lower Arduino
//

#include <Servo.h>
#include <Wire.h>

Servo servo1; //create servos
Servo servo2;
Servo servo3;
Servo servo4;

int servoPin1 = 3;  // set pins
int servoPin2 = 5;
int servoPin3 = 6;
int servoPin4 = 9;

int incomingByte = 0; // for incoming serial data

char buffer[17];    // buffer array for characters
char temp[100];

int size = 0;

void printValues(){
  int i = 1; //iterator starts with 1
  int temp_index=0; //keeps tamp array index
  bool expecting_hash = true; //flag for #
  //iterates over the buffer
  while(buffer[i]!='\0'){
    //if it comes to end of first input
    if(buffer[i]=='#' && expecting_hash){
```

```
      //adds end chat to temp
      temp[temp_index] = '\0';
      //it does not expecte #
      expecting_hash = false;
      //calls print command
      printCommand();
      temp_index=0; //updates temp index
      i++; //increments iterator
      continue;
    }
    //if it is start character
    if(buffer[i]=='!' && !expecting_hash){
      //it expects #
      expecting_hash = true;
      temp_index = 0;//updates temp index
      i++; //increments iterator
      continue;
    }
    //adds char from buffer to temp //increment
        temp[temp_index]=buffer[i];
    i++; //increment iterator
    temp_index++;//increment temp index
  }
}

//function that executes string between ! and #
void printCommand(){
  int i=0; //initialized iterator
  int semicolon_counter =0; //keeps number of semicolons
  int semicolon_index[3]; //keeps positions of semicolons
  //it iterates over the command
  while(temp[i]!='\0'){
    //if element is semicolon
    if(temp[i] ==';'){
      //keeps index of semicolon
      semicolon_index[semicolon_counter] = i;
      semicolon_counter+=1; //increments counter
```

```
  }
  i++; //increments iterator
}
int item = 0;
int pow = 1;
//it iterates over the first input
for(int j=semicolon_index[0]-1;j>=0;j--){
  //transforms character to integer
  item+= (int)(temp[j]-48)*pow;
  pow*=10;
}
//send value to servo3
servo1.write(item);


item = 0;
pow = 1;
//it iterates over the second input
for(int j=semicolon_index[1]-1;j>semicolon_index[0];j--){
  //transforms character to integer
  item+= (temp[j]-48)*pow;
  pow*=10;
}
//send value to servo2
servo2.write(item);

item = 0;
pow = 1;
//it iterates over the third input
for(int j=semicolon_index[2]-1;j>semicolon_index[1];j--){
  //transforms character to integer
  item+= (temp[j]-48)*pow;
  pow*=10;
}
//send value to servo3
servo3.write(item);
```

```
  item = 0;
  pow = 1;
  //it iterates over the fourth input
  for(int j=i-1;j>semicolon_index[2];j--){
    //transforms character to integer
    item+= (temp[j]-48)*pow;
    pow*=10;
  }
  //send value to servo4
  servo4.write(item);

}

void setup()
{
  Wire.begin(4);                // set address 4
  Wire.onReceive(receiveEvent); // register event
  servo1.attach(servoPin1);           // attach servos
  servo2.attach(servoPin2);
  servo3.attach(servoPin3);
  servo4.attach(servoPin4);
  servo1.write(0);                              //write servo 0
  servo2.write(0);
  servo3.write(0);
  servo4.write(0);
}

void loop()
{

}

// when data receives this function executes
void receiveEvent(int howMany)
{
  int i=0;
  bool received = false;
```

```
  while(Wire.available()) // loop through all but the last
  {
    received = true;
    buffer[i] = Wire.read(); // receive a byte as character
    i++;
  }
  if(received){// if data received
        printValues();  // execute the servo operations
  }
}
```

3 COMMUNICATION PROTOCOLS

In the micro-controller world sometimes we have to communicate different devices. Therefore for this communication some communication protocols developed. Some of these are UART(Universal Asynchronous Reception and Transmission),I^2C (Inter-Integrated Circuit), SPI(Serial Peripheral Interface) and CAN(Controller Area Network).

3.1 Universal Asynchronous Reception and Transmission-UART

UART is an asynchronous (works without clock) communication protocol that is widely used in micro-controller communication. The serial monitor of the Tinkercad also uses this protocol. UART is a simple serial communication protocol that allows the host communicates with an external device. UART supports serial, asynchronous and bi-directional data transmission. UART has two data lines, one of them is for transmitting data and other one is for receiving data.

3.1.1 UART's Advantages

UART is a simple to operate communication protocol, it is a widely used protocol therefore it has lots of documentation on the internet. And there are simple error checking features on the UART communication protocol.

3.1.2 UART's Disadvantages

Because UART has limited sized data frame it can not send big sized data. Also it can not communicate more than one slave or master. It's also a slow communication protocol.

3.2 Inter-Integrated Circuit-I^2C

I^2C is a serial communications protocol similarly to UART, but in this communication protocol we don't deal with the PC-device communication. This communication protocol's actual usage is on the external sensors and modules. This is useful for projects that requires many different parts. Also I^2C uses address system and a shared bus to communicate devices.

3.2.1 I²C's Advantages

I²C is a flexible communication protocol, as it supports multi-master and multi slave communication.

3.2.2 I²C's Disadvantages

I²C may become complex as the number of devices increases. It is also a slow communication protocol type.

3.3 Serial Peripheral Interface-SPI

SPI is is similar to I²C and it is a different form of serial-communication protocol. It is designed for connecting micro-controllers to each other. It can simultaneously transmit the data and receive the data.

3.3.1 SPI's Advantages

SPI is a simple communication system because there is no master slave relation in this communication protocol. It is a faster communication protocol then UART and I²C. Data can transferred without interrupt unlike UART protocol because there is no start stop functionality. It can also simultaneously transmit the data and receive the data.

3.3.2 SPI's Disadvantages

SPI uses more pin ports therefore it has limited number of device communication capability at the same time. There is no mechanism that controls whether data is received or not unlike I²C. And there is no form of error check unlike in UART. It can also has just one master device.

3.4 Controller Area Network-CAN

CAN is used to allow micro-controllers and devices to communicate with each other without a host computer. It was firstly designed for vehicles. Compared to other communication protocols like UART, SPI and I²C, using CAN BUS communication protocol are much more reliable because if a vehicle's vital data communication fails, it can cause vital damages to us and to their system.

3.4.1 CAN's Advantages

CAN supports multi master and multi slave features. It has single serial bi-directional line to achieve half duplex communication and it has automatic retransmission for

message that lost.

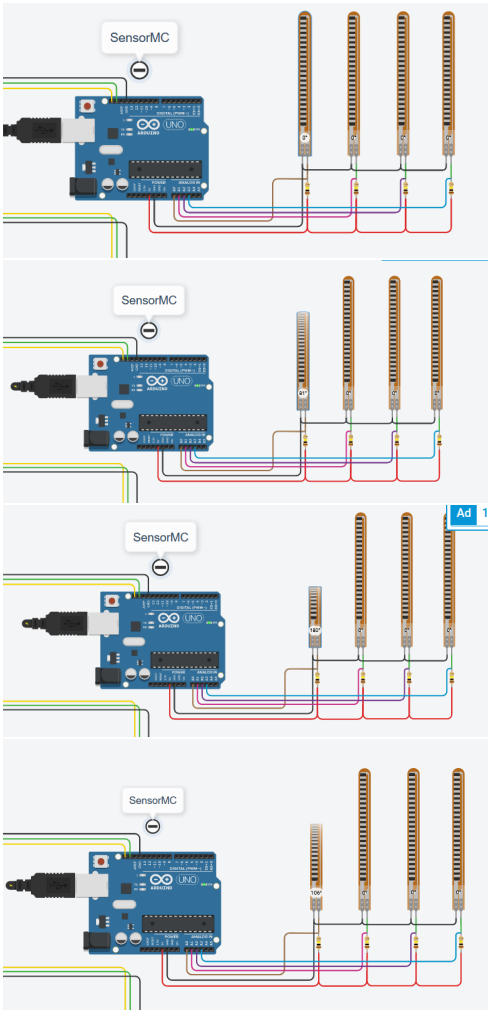
3.4.2 CAN's Disadvantages

CAN is likely to have undesirable interactions between devices.

4 RESULTS

4.1 PART 1

After our code implementation, we checked the observed values and calculated values with real angles of 0,81,106 and 180. For simplicity we have only changed the first flex sensor. At the values of 0 and 180 calculated values are equal with the observed values but for values of 81 and 106 observed values and calculated values differ since we have assumed the flex sensors work linearly but actually they don't. Our calculated values coincide with the given data laboratory document.




```
!0;236.00;29.99;0.00#
!1;236.00;29.99;0.00#
!2;236.00;29.99;0.00#
!3;236.00;29.99;0.00#

!0;400.00;64.21;46.31#
!1;236.00;29.99;0.00#
!2;236.00;29.99;0.00#
!3;236.00;29.99;0.00#

!0;634.00;162.98;180.00#
!1;236.00;29.99;0.00#
!2;236.00;29.99;0.00#
!3;236.00;29.99;0.00#

!0;459.00;81.38;69.56#
!1;236.00;29.99;0.00#
!2;236.00;29.99;0.00#
!3;236.00;29.99;0.00#
```

Figure 1: First Pattern

4.2 PART 2

After our code implementation, We can use our stopwatch easily as shown below.

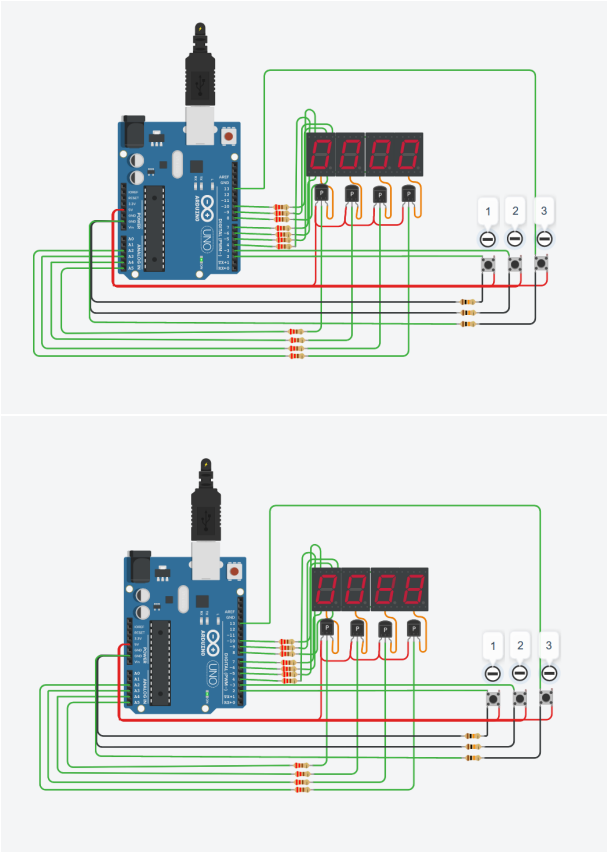


Figure 2: Stopwatch with starting and counting and finally reset

Figure 3: Stopwatch lap information on serial monitor

4.3 PART 3

After our code implementation, we can see that when we change the angle of the flex sensor our servos change their angle according to their assigned flex sensors.

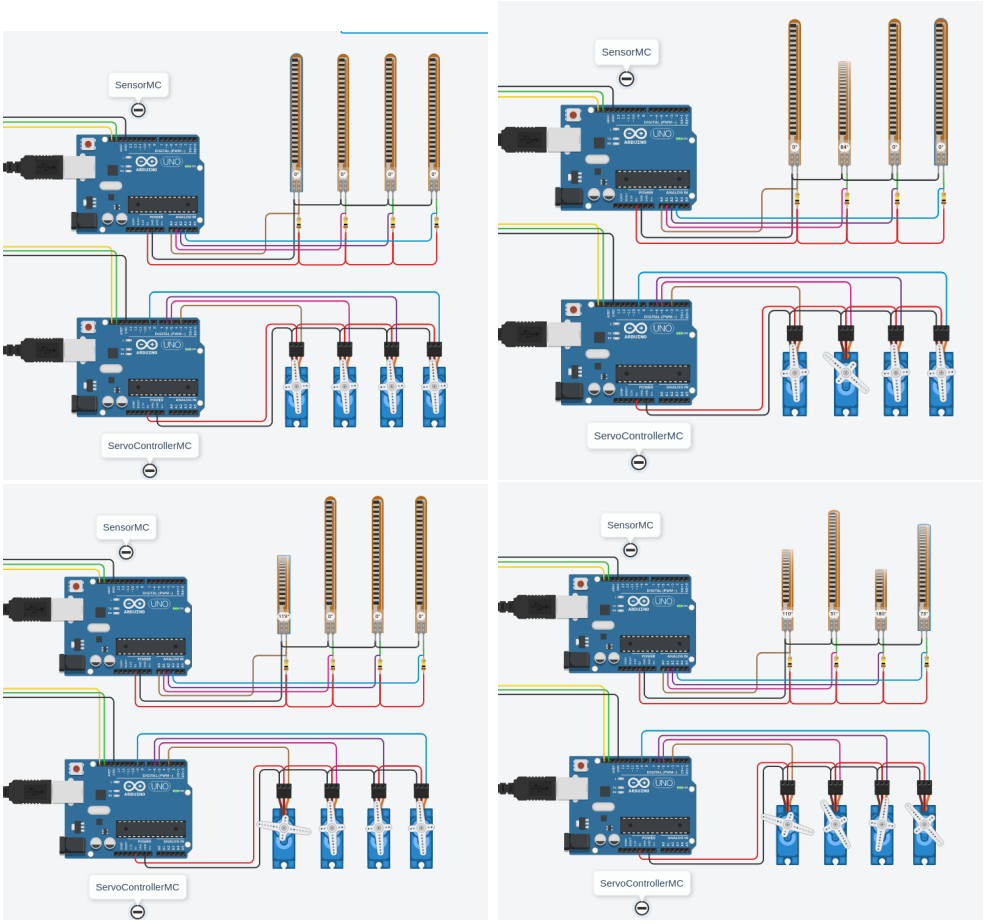


Figure 4: Flex angles and servo angles

5 DISCUSSION

5.1 PART 1

In the first part of this week's experiment, we have observed flex sensor data using our analog value reading capability of our Arduino circuitry. At first, figuring out how to calculate the resistor and angle values was a little complex to think. After some research and using our basic circuitry knowledge from our electronics courses we did not have much of a hard time with first part of the experiment and easily moved with the second part.

5.2 PART 2

In this part of experiment, we have expected to implement circuit that reads input from serial monitor and according to input values, it manipulates angles of servomotors. Actually, the fundamental principle of this part is simple but in practical aspect, it is hard to validate all input combinations completely. While implementing this circuit, we have changed our approach many times. As a final decision, we have performed divide and conquer approach. It needs high concentration while writing code. Therefore, we have faced many errors and we debugged our code many times. It was the longest part of the whole experiment.

The input reading part was new for us and we have learned that without using delay, it could not read all given characters. String validation part was usual programming problem. The manipulation of servo motors was easy but it was new knowledge that we learned about micro controllers.

We thought that this part of experiment was the most challenging part of homework but we are happy with our result and new knowledge.

5.3 PART 3

In this part of experiment, we have expected to combine previous parts of experiments and implement a three micro-controllers that are communicating with each other. Actually, learning and implementing communication principles of two micro-controllers excited us. Providing functionality of the circuit was easy, because we have already implemented codes that reads flex sensors and manipulates servomotors. We have only changed some part of codes to obtain standardized circuit.

To implement communication principles of circuit, we have read documentation of the Arduino and we have learned master-slave organization of circuit. Documentation is describes all information about I2C protocols and required functions. Then, we have applied our knowledge to the circuit. Implementation phase was hassle-free and fast.

At the end of the experiment, observing a working and communication circuit was really satisfied us. We highly like networking of the micro-controllers and this experiment allowed us to discover and learn its principles. Thanks to our teaching assistant for giving this homework.

6 CONCLUSION

In conclusion, we learned communication protocols of Arduino in this experiment we implemented a basic message template to send our message with communication protocols and with our new knowledge on this subject we send our messages through our Arduino circuits and controlled our servos. With this experiment we learned some fundamental skills about Arduino communication protocols that we can use in the real world applications.

References

[1] Kadir Ozlem. *BLG 351E – Microcomputer Laboratory Slides*. 2020.

[2] Arduino. Arduino documentation. https://www.arduino.cc/reference/en/, 2018.

[3] Autodesk. Tinkercad. https://www.tinkercad.com/, 2011.