Project 2.7.3 Medium ISP: Autobox

Purpose

In addition to displaying the current gear or cadence, the purpose the Autobox, which is an automatic bike gear shifter, is to automatically put the rider into the optimal gear, based on how quickly they are pedaling.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI3M/2324/images/RohanMedium.png

Theory

A servo motor is a rotary actuator, who's angular position can be manipulated by the duty cycle of a PWM signal (see Figure 1). As a result of the PWM control, the servo needs only one pin in addition to power and ground to be controlled. An MCU sends out a PWM signal, allowing the angle to be precisely modulated. In this configuration, a feedback device, usually a potentiometer, is used to detect the position of the servo. A motor with a high gear ratio spins the head, giving it a high amount of torque. The motor spins, until the desired position is detected by the potentiometer.

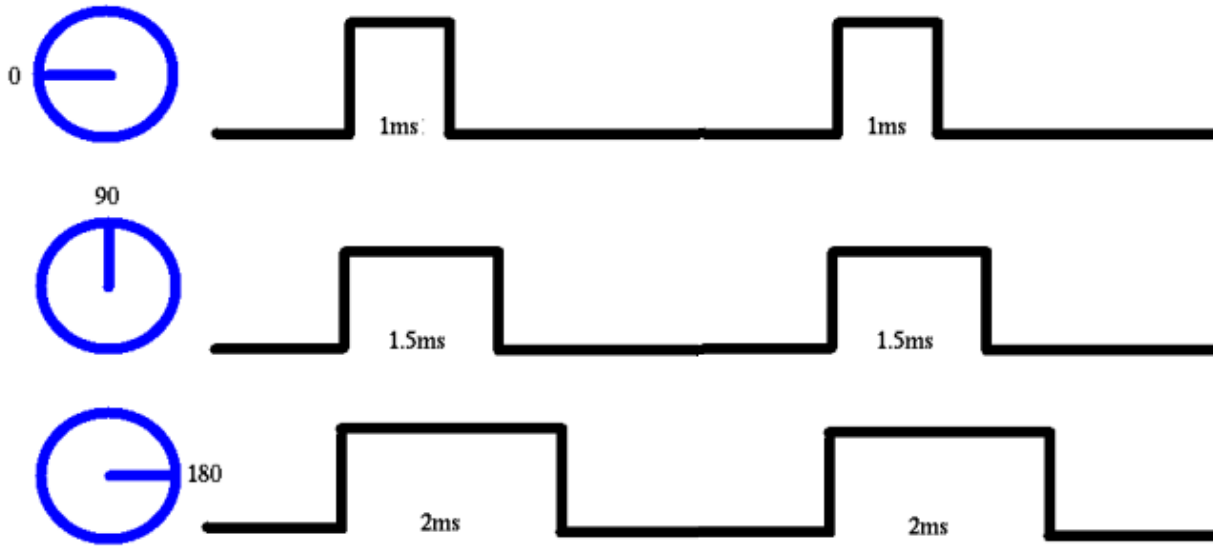


Figure 1. Servo control using PWM duty cycle

I2C is a synchronous protocol that allows different devices to communicate with each other, one byte at a time (see Project 2.6.1). Normally, it is used to communicate between a controller, and a non-processing unit, such as an MCU and RTC (see Project 2.6.2). In some cases, however, it can be used to communicate between multiple MCUs. In this configuration, there is no "master." While there is still one device that is designated as the master, each MCU is primary. This setup has many advantages; it allows multiple processes to run at the same time, and it can take stress off the main MCU to free up processing power for other tasks. It also allows for modular designs; each major section of a machine can have its own MCU that performs a simple task, reporting back to the main MCU to perform a larger, more complex task. Instead of one MCU taking input, processing it, and actuating an output with the data, one MCU can monitor each group of inputs, and send streamlined data to another MCU. Then, the data can be compiled, and sent to the appropriate output. In this configuration, the machine operates as a network, greatly increasing its efficiency and precision.

Procedure

The Autobox has three main sections: the buttons input unit, the main board, and the display. The main board is housed with the servo motor in a box mounted on the bike frame. It is setup as the master; however, all three devices are considered "primary" (see Theory section). Each device is capable of keeping track of metrics, and performing a complex action.

The buttons unit keeps track of the count, bound at 0 and 5, from the up and down buttons. It sends the current count using the 3 least significant bits, and the state of the switch on bit 7 (see Figure 1).

The main board receives the byte from the buttons device, and chooses manual or automatic mode based on bit 7 (see Figure 1). In manual mode, the gear is adjusted to the gear stored in the count from the buttons device. In automatic mode, the gear is dynamically adjusted based on the rider's cadence.

Parts Table	
Quantity	**Description**
1	ATtiny84 DIP MCU
2	ATtiny85 DIP MCU
1	SN74HC595N Shift Register
1	TPIC6C595 Shift Register
1	4 Digit Seven Segment Display
1	16 MHz Crystal Oscillator
2	22 pF Ceramic Capacitor
6	2 × 1 Terminal Block
3	2 × 1 ISP Header
8	10 KΩ Fixed Resistor
8	330 Ω Fixed Resistor
2	Push Button Normally Open
1	SPDT Slide Switch
1	I2C Display PCB
1	I2C Buttons PCB
1	Main PCB
1	5V Power Supply
~	Solder
~	Wires

Figure 1. Buttons Device Bit Assignment							
BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
Switch	N/A	N/A	N/A	N/A	Count (A)	Count (B)	Count (C)

The main board uses a reed switch and magnet system to find cadence (see Project 2.5.3). The onboard ATtiny84 waits for two revolutions of the pedals to go around, and measures the time between to calculate RPMs. If the RPMs are too high, it shifts up a gear. Conversely, if the RPMs are too low, it shifts down a gear. Using this mechanism, the rider is always put in the optimal gear. The Autobox keeps the cadence between 70 and 80 RPMs.

Figure 2. Display Device Bit Assignment								
Byte	**BIT 7**	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
1	Val (G)	Val (H)	Val (I)	Val (J)	Val (K)	Val (L)	Val (M)	Val (N)
2	N/A	N/A	Val (A)	Val (B)	Val (C)	Val (D)	Val (E)	Val (F)
3	N/A	N/A	N/A	N/A	N/A	N/A	Dec (A)	Dec (B)

On the display device, when in automatic mode, the cadence is displayed. In manual mode, the current gear is displayed. Similar to the buttons device, the display is its own device. It uses an ATtiny85, a standard 8 output shift register for the anodes, and a TPIC shift register for the cathodes. There are 3 bytes that must be sent for the display to be updated: 2 for the value to be displayed, and 1 for the decimal position. While there are two extra bits for the decimal in the second bit (see Figure 2), due to the manipulation to use them, it is more efficient to send an extra byte.

When the display board receives the bytes in Figure 2, it puts the first two together, to recreate the 16-bit unsigned integer that will be displayed. Then, it splits the transmitted number into its 4 digits and stores it to a buffer. The display MCU then constantly shifts out the correct values from an array-based segment map, and uses POV to display all 4 digits until the next I2C transmission (see Project 2.6.2). This system allows the main MCU to perform other tasks, while the POV effect continues. Additionally, it eliminates the need for the main MCU to perform resource-heavy division operations to split the number into its digits. In this configuration, the number is simply sent over I2C, and the rest is taken care of by an external processor.

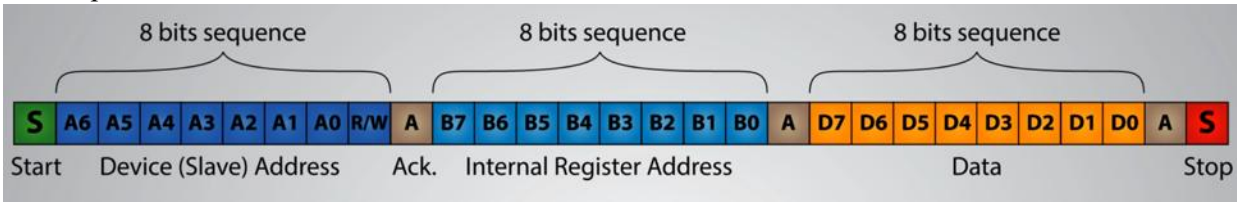


Figure 3. I2C Bus Bit Sequences

When using I2C, there is a "stop condition" that signals the receiving device that the transmission has ended. When creating I2C devices in "slave" mode using an ATtiny85, the stop condition it not automatically detected.; instead, the device must detect it. This is accomplished by running a short loop function, with the line `TinyWireS_stop_check();` executed as many times per second as possible.

As a result of using I2C for the display and buttons, only two pins are utilized, for a total of 4 pins with the cadence sensor and servo motor. While an ATtiny85 has sufficient pins for this application, an ATtiny84 (see Figure 4) makes more sense. The extra pins can be used to accommodate a 16 MHz crystal oscillator, which is necessary to keep precise time. Without the oscillator, the time kept is not as accurate, and the calculated cadence would be inaccurate. The ATtiny84 can accommodate the crystal oscillator which makes it more suitable.

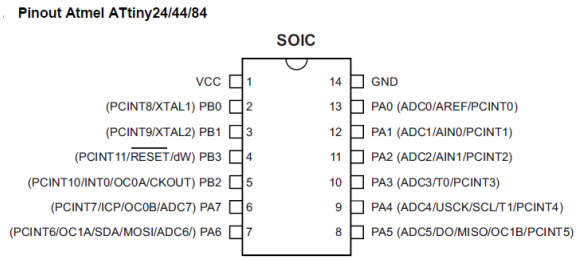


Figure 4. ATtiny84 Pinout

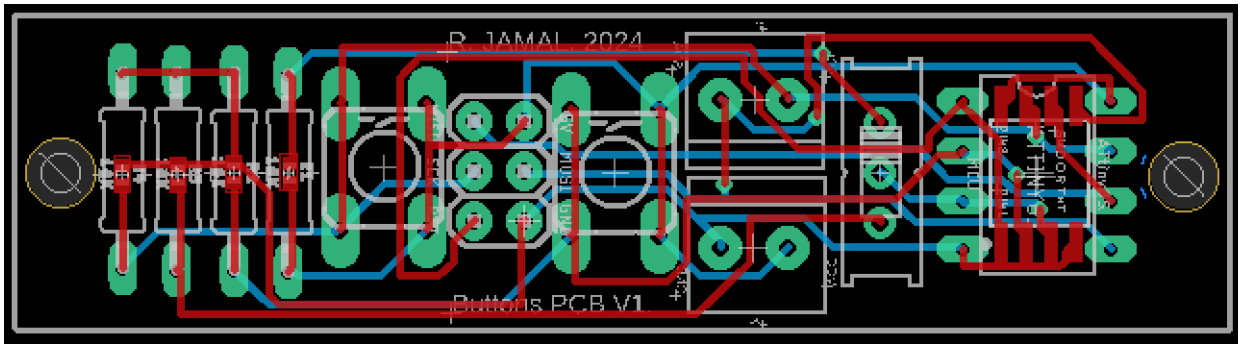


Figure 5. Buttons Device Fusion 360 Board View

Each of the previously mentioned devices feature a PCB, custom-made for the application. The buttons unit features an ISP header, an ATtiny85, 4 resistors, two PBNOs, two terminal blocks, and a SPDT slide switch (see Figure 5). Where applicable, both surface mount technology (SMT) and through hole technology (THT) is included. Since SMT parts are so small, this feature does not increase the footprint.

The main board contains most of the Autobox's components; however, it is still relatively small, with a total size of 25 mm by 16 mm. On the main board are 4 terminal blocks, 1 for I2C communication, 1 for power, and 2 that provide access to I/O pins (see Figure 6). One of these is used to detect the cadence, and the other can be used with a second reed switch system. This would allow the system to have access to the speed, which could be displayed on the seven-segment display. In this configuration, the footprint of the actual display would be relatively small as it would not require a battery onboard.

The board is designed to be a plug and play system; it includes everything necessary to run. All pullup and pulldown resistors are on-board. This includes two 10 KΩ pullup resistors for I2C communication.

Many of the components pictured in Figure 5 are not used in the final design of the Autobox; some components were not necessary. For example, the 7805-voltage regulator is unable to provide enough current for the servo motor, and was replaced with a boost converter and lithium battery. As a result, the two supporting capacitors are also not needed.

The utilization of a PCB allows the footprint of the circuit to be greatly decreased. If it were done on a point-to-point board, the space requirement of the project would be vastly larger.

The final PCB used in the Autobox is the display PCB (see Figure 7). Similar to the buttons PCB, this board contains space for SMT parts in addition to THT parts. This provides flexibility in the application; if the bottom of the board must be relatively flat, SMT parts can be utilized; if the flatness of the board does not matter, THT can be used. The display PCB is essentially a custom, programmable I2C "backpack," that can perform calculations. This is an important feature, as it frees up resources on the other MCUs.

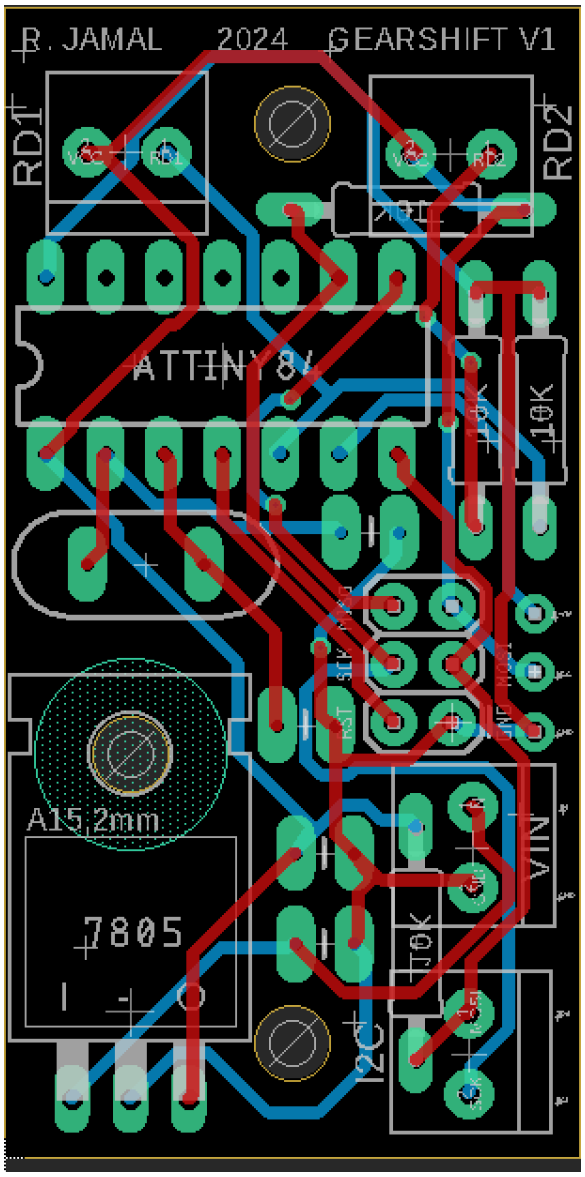


Figure 6. Main PCB Fusion 360 View

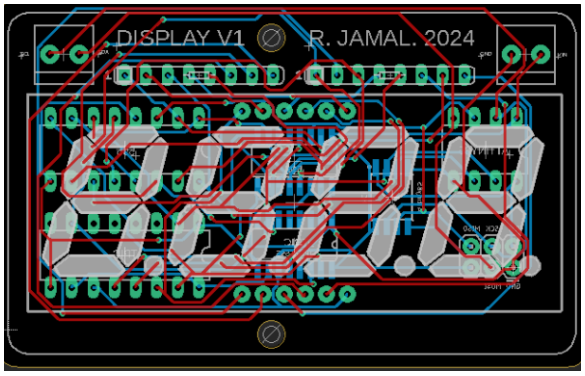


Figure 7. Display PCB Fusion 360 View

Code

Main (ATtiny84)

```
// PROJECT    : Bike gear shifter
// AUTHOR     : R. Jamal
// PURPOSE    : To automatically put bike in correct gear using servo
// COURSE     : ICS3U-E
// DATE       : 03 02 2024
// MCU        : TINY84
// STATUS     : [Semi] Working
// NOTES      : Software works, hardware overheated. Written in IDE 1.8.19
#include <Servo_ATTinyCore.h> //Servo library
#include <TinyWireM.h>        //I2C library
#define CAD 8                 //Cadence pin
#define MAX 80                //Max cadence
#define MIN 70                //Min cadence
#define CLDN 20               //Display cooldown time
#define SERVO 7               //Servo pin
#define TOUT 2500             //Timeout (cadence=0)
#define CVRSN 60000           //RPM conversion constant
#define BADDRESS 0x22         //Buttons address
#define DISPADDR 0x30         //Display address
uint8_t buttonState, preCog;  //8-bit unsigned variables
uint8_t cog = 0;              //Current gear variable
uint8_t gearsUp[] = {0, 60, 80, 105, 130, 165}; //Index up
uint8_t gearsDown[] = {0, 25, 42, 60, 80, 165}; //Index down
uint16_t bufferRPM = 1234;    //Buffer to store cadence to
uint64_t lastTrans;           //Keeps track of I2C transmissions
Servo myservo;                //Servo object
void setup() {                //Setup, runs once
  TinyWireM.begin();          //Start I2C communication
  myservo.attach(SERVO);      //Attach servo to servo pin
}
void loop() {                                   //Loop function, runs continously
  buttonState = getButtons();                   //Store number in buttons
  if (buttonState >> 7) {                       //MSB stores mode switch value
    bufferRPM = getReedRPM(CAD);                //Read pedals speed (cadence)
    lightDisplay(bufferRPM, 0);                 //Displays the cadence
    if (cog != 5 && bufferRPM > MAX) cog++;     //Statement to increase gear
    if (cog && bufferRPM && bufferRPM < MIN)    //Statement to decrease gear
      cog--;                                    //Statement to decrease gear
  }
  else {                                        //Sets stored gear when manual
    cog = buttonState;                          //Sets cog to stored gear
    lightDisplay(cog, 0);                       //Display the current gear
  }
  if (cog < preCog)                             //Decide which index to use
    myservo.write(gearsDown[cog]);              //Shift down a gear
  else if (cog > preCog)                        //Decide to use up index
    myservo.write(gearsUp[cog]);                //Shift up a gear
  preCog = cog;                                 //Update previous cog, to monitor
}
uint16_t getReedRPM(uint8_t pin) {                          //Function to get cadence
  uint64_t timeOld = millis();                              //Start timeout timer
  while (!digitalRead(pin) && millis() - timeOld < TOUT); //Wait for reed to go high
  while (digitalRead(pin));                                 //Debounce
  timeOld = millis();                                       //Start timer (for RPMs)
  while (!digitalRead(pin) && millis() - timeOld < TOUT); //Wait for full revolution
  while (digitalRead(pin));                                 //Debounce
  if ((millis() - timeOld) < TOUT)                          //Return RPMs; no timeout
    return CVRSN / (millis() - timeOld);                    //Return RPMs
  else return 0;                                            //Returns 0 if timed out
}
```

```
void lightDisplay(uint16_t val, uint8_t dec) {  //Function to light display
  if ((millis() - lastTrans) > CLDN) {          //Display has 20 ms "cooldown"
    lastTrans = millis();                       //Keeps track of last transmission
    TinyWireM.beginTransmission(DISPADDR);      //Begins transmission to display
    TinyWireM.write(val);                       //Sends first byte
    TinyWireM.write(val >> 8);                  //Sends second byte
    TinyWireM.write(dec);                       //Sends third (decimal) byte
    TinyWireM.endTransmission();                //Ends transmission
  }
}
uint8_t getButtons() {                  //Receives state of buttons from slave
  TinyWireM.requestFrom(BADDRESS, 1); //Requests 1 byte from buttons device
  while (!TinyWireM.available());     //Waits for data to be placed into buffer
  return TinyWireM.read();            //Returns the buffer-stored value
}
```

Buttons Device (ATtiny85)

```
// PROJECT   : Button device slave (I2C)
// AUTHOR    : R. Jamal
// PURPOSE   : To keep track of buttons and communicate states over I2C
// COURSE    : ICS3U-E
// DATE      : 03 02 2024
// MCU       : TINY85
// STATUS    : Working
// NOTES     : Written in IDE 1.8.19
#include <TinyWireS.h>                          //I2C slave library
#define UP 1                                    //Up button pin
#define DN 3                                    //Down button pin
#define SWTCH 4                                 //Switch pin
#define ADDRESS 0x22                            //I2C address
uint8_t val = 0;                                //Value for +/-
void setup() {                                  //Setup, runs once
  TinyWireS.begin(ADDRESS);                     //Define address
  TinyWireS.onRequest(request_ISR);             //Define request ISR
}
void loop() {                                   //Loop, runs continously
  TinyWireS_stop_check();                       //Detect stop condition
  if (digitalRead(UP) && val != 5) val++;       //Wait for up button
  while(digitalRead(UP));                       //Debounce
  if (digitalRead(DN) && val != 0) val--;       //Wait for down button
  while(digitalRead(DN));                       //Debounce
}
void request_ISR() {                            //ISR for requests
  TinyWireS.send(val | digitalRead(SWTCH) << 7);  //Send values over I2C
}
```

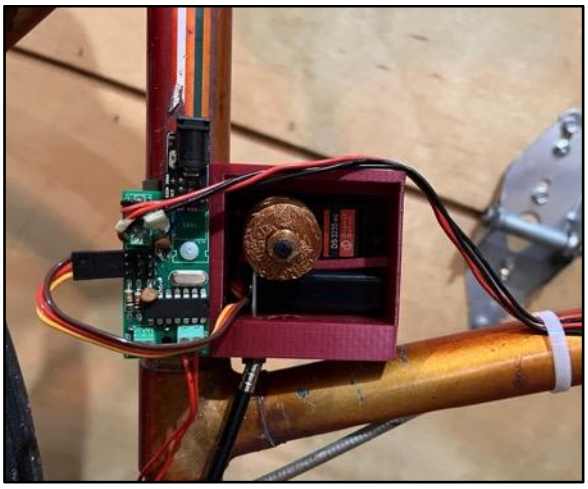
Display Device (ATtiny85)

```
// PROJECT   : Display device slave (I2C)
// AUTHOR    : R. Jamal
// PURPOSE   : To display value received over I2C on display
// COURSE    : ICS3U-E
// DATE      : 03 02 2024
// MCU       : TINY85
// STATUS    : Working
// NOTES     : Written in IDE 1.8.19
#include <TinyWireS.h>              //I2C slave library
#define clockPin 4                  //Clock pin shift register
#define latchPin 3                  //Latch pin shift register
#define dataPin 1                   //Data pin shift register
#define ADDRESS 0x30                //I2C address
uint16_t rec;                       //Receieved data buffer
uint8_t dec;                        //Decimal data buffer
bool de;                            //Decimal enabled bool
uint8_t numbers[] = {               //Start segment map
  B11111100,                        //0 (a,b,c,d,e,f)
  B01100000,                        //1 (b,c)
  B11011010,                        //2 (a,b,d,e,g)
  B11110010,                        //3 (a,b,c,d,g)
  B01100110,                        //4 (b,c,f,g)
  B10110110,                        //5 (a,c,d,f,g)
  B10111110,                        //6 (a,c,d,e,f,g)
  B11100000,                        //7 (a,b,c)
  B11111110,                        //8 (a,b,c,d,e,f,g)
  B11110110,                        //9 (a,b,c,d,f,g)
};                                  //End segment map
uint8_t buffer[] = {4, 3, 2, 1};    //Intermediate data holder
void setup() {                      //Setup function, runs once
  TinyWireS.begin(ADDRESS);         //Define I2C address
  pinMode(dataPin, OUTPUT);         //Set dataPin as output (DDR)
  pinMode(latchPin, OUTPUT);        //Set latchPin as output (DDR)
  pinMode(clockPin, OUTPUT);        //Set clockPin as output (DDR)
}
void loop() {                               //Loop function, runs continuously
  TinyWireS_stop_check();                   //Detect stop condition
  writeBuffer();                            //Store received number to buffer
  for (uint8_t dig = 4; dig < 8; dig++) {   //Light digits, set decimal
    if ((dig - dec) == 3) de = 1;           //Decide when to enable decimal
    else de = 0;                            //Logic for decimal point
    lightDisp(1 << dig, numbers[buffer[dig - 4]] | de); //Shiftout segments
  }
}
void lightDisp(uint8_t anode, uint8_t cathode) {  //Function to shiftout segments
  PORTB &= ~(1 << latchPin);                      //Pull latch low
  shiftOut(dataPin, clockPin, LSBFIRST, anode);   //Shift out anodes
  shiftOut(dataPin, clockPin, LSBFIRST, cathode); //Shift out cathodes
  PORTB |= 1 << latchPin;                         //Set latch high
}
void writeBuffer() {                  //Function, stores received value to buffer
  if (TinyWireS.available()) {        //Waits until there is data being sent
    rec = TinyWireS.receive();        //Receives first byte
    rec |= TinyWireS.receive() << 8;  //Receives second byte
    dec = TinyWireS.receive();        //Receives decimal byte
  }
  buffer[3] = rec / 1000;             //Gets thousands digit
  buffer[2] = (rec / 100) % 10;       //Gets hundreds digit
  buffer[1] = (rec / 10) % 10;        //Gets tens digit
  buffer[0] = rec % 10;               //Gets onces digit
}
```

Media

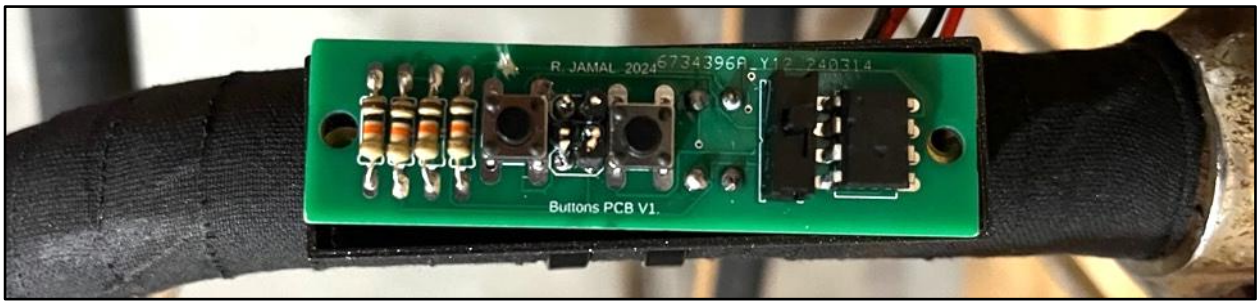
Project video: https://youtu.be/KcbHWWFukyQ

Autobox Mounted on Bike, Side Profile



Autobox, Closeup

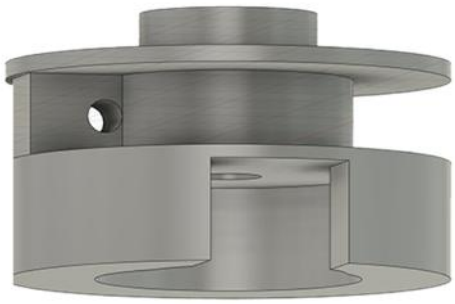
Buttons



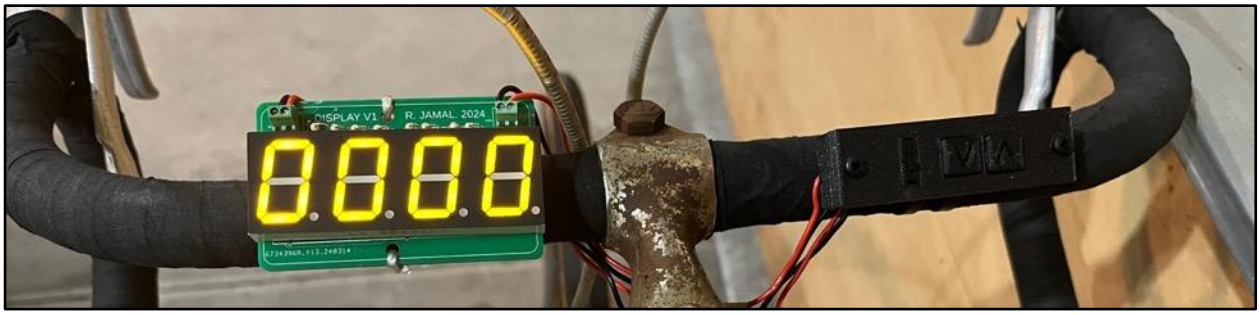
Buttons PCB Mounted on Handlebars

Buttons PCB Mounted on Handlebars with Cover

Display PCB Closeup



Servo Horn Attachment



Handlebars View

Reflection

This ISP has been a complete rollercoaster for me. Certain parts of the circuit worked really well right away, but others took a lot of fiddling with to get working. For example, when I was trying to get my devices to talk to each other over I2C, I spent many hours debugging the code, hardware, and connections. Eventually, I found out that the problem was not with my setup, but the library I was using. That version turned out to be corrupt, so I had to downgrade it. I guess what Max said in his presentation really resonated with me: libraries should be avoided due to their mystery. They're great when they're working, but when something goes wrong, you have no idea where to start to debug it. This is exactly what happened to me. My hardware was right, and the code I had written seemed to be okay. As a result, I spent much time and frustration trying to fix my setup. Had I not stumbled upon a forum post explaining that the most recent version of the TinyWireS library was corrupt, I don't know that I would ever have found out. The biggest problem here with libraries is that you generally have little to no idea what is really happening. Even after I found out that the library was corrupt, I still have no idea what that means. Was the file not readable? Was there a mistake in the library? Something else entirely?

Another lesson learned this ISP is that even if your idea is theoretically a good idea, even if you do manage to get it to work, it may not have a super long lifespan. I had my project mostly working on the Tuesday before the Friday I was presenting. Unfortunately, as I learned, 3d-printed plastic was not the best choice for my project; the servo motor got really hot and started bending the plastic, which made the horn rub against itself. As a result, the servo did not have enough torque to move the cable anymore. The 2 days that the project worked were pretty good, but I definitely wish I would have used a stronger material that didn't insulate heat so well. I'm pretty sure the servo also overheated, because its behaviour was also unpredictable after it got hot.

This part is not really related to the project, more of a reflection on the course, but I'm already seeing the dividends of my work into the course outside of school. So, backstory: almost all the lights in my house are controlled by a centralized computer system/control panel. Unfortunately, the computer stopped working for some reason on Friday, so we couldn't use any of the lights. Eventually, based on testing it in certain scenarios, I figured out that the problem was the power supply. Before this course, I would have just looked at all the wiring and figured it was impossible for me to figure out. But then, I realized if this thing had a datasheet, it would probably have a pinout. Sure enough, I found the pinout and I was able to solve the problem. As a bonus, the control panels and buttons throughout the house communicate using I2C! I think that this experience does, in a way, tie back to ISPs. The reason that we can choose whatever we want for the project with almost no restrictions, is because it does not matter what we choose. It's not what we learn, but the habits we build along the way. Moral of the story: read the datasheet.