Pivot: A Motion Based User Interface

6.111 Final Project Report

Nestor Franco and Andrew Kurtz

1 Overview (NF + AK)

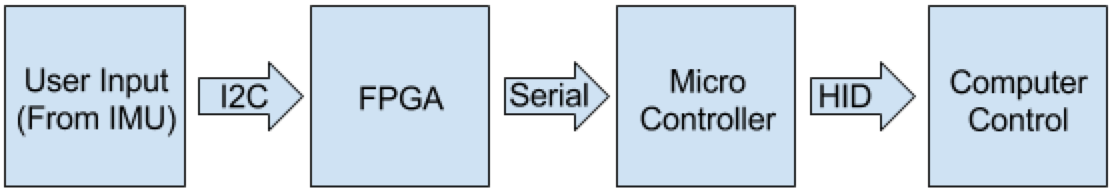
With the advent of online video streaming sites such as YouTube and Netflix, traditional cable and satellite TV subscriptions are declining. As such, more and more people are connecting a computer to a TV screen not just to stream video, but also to play games and browse websites. It can be inconvenient to use a full sized wireless keyboard and mouse while sitting on the couch - a better alternative would be to have some kind of compact, handheld controller that contains both keyboard key input and mouse control.

Our final project focused on developing the mouse portion of this controller using a combination of angular rate, and acceleration sensors. We used an Inertial Measurement Unit (IMU) to collect these measurements, and fed these signals into a small FPGA board. The FPGA translated the signals into mouse movement and function (e.g. mouse clicks and scrolling). The mouse signals were then communicated to a computer through USB utilizing the HID protocol. Additionally, a housing, with built in wiring, mounting points, buttons, and power was constructed.

2 Design Approach (AK)

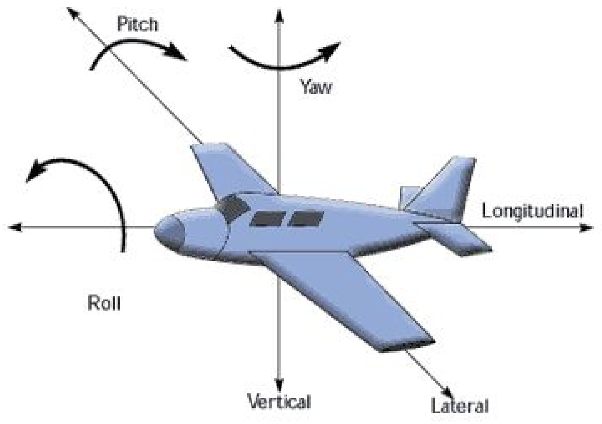
2.1 Software

2.1.1 High level Block Diagram



One item of note from this high level diagram is the inclusion of a microcontroller. To make our lives easier, we will be using a standard off the shelf microcontroller to produce the USB HID protocol, but more on that later.

2.1.2 Definition of User Input



User inputs can fall in 5 categories: translations, rotations, taps, button presses, and gestures.

- Translations are defined as planar motion in either the lateral/longitudinal plane or the lateral/vertical plane.
- Rotations are defined as either pitch, roll, or yaw motions.
- Taps are defined by an event of large acceleration (as recorded by the IMU) happening over a short time period (on the order of 10ms).
- Button presses are defined as an action by the user to actuate a physical momentary push button on the device.
- Gestures are defined as a planar motion which traces out a distinct path. For example if the user were to quickly trace a circle with the controller this would be considered a gesture.

Our device is able to recognize rotations and button presses. We hoped to add gestures and taps but this proved infeasible in the time allotted.

2.2 Hardware

2.2.1 Product Requirements

At the beginning of this endeavor, to ensure our device would be as unobtrusive, and user friendly as possible, we set the following requirements:

- The device shall be graspable within the hand of 95% of adults.
- The device shall interface with any modern operating system (e.g. Mac, Ubuntu, Windows)
- The device shall interface using a single USB 2.0 cable, and require no additional, external, cables or connections
- All device sub components shall be attached to each other using standard connectors and fasteners to allow for easy assembly, disassembly, and maintenance.
- All wires between sub components shall be soldered in place.
- The device shall have at least 2 user programmable, external, buttons.
- The device should have a clean, outward design.
- The device should be simple enough to operate that new users do not need instructions.

2.2.2 Component Selection

After developing our requirements and discussing the available components with our advisor we made the following component selections:

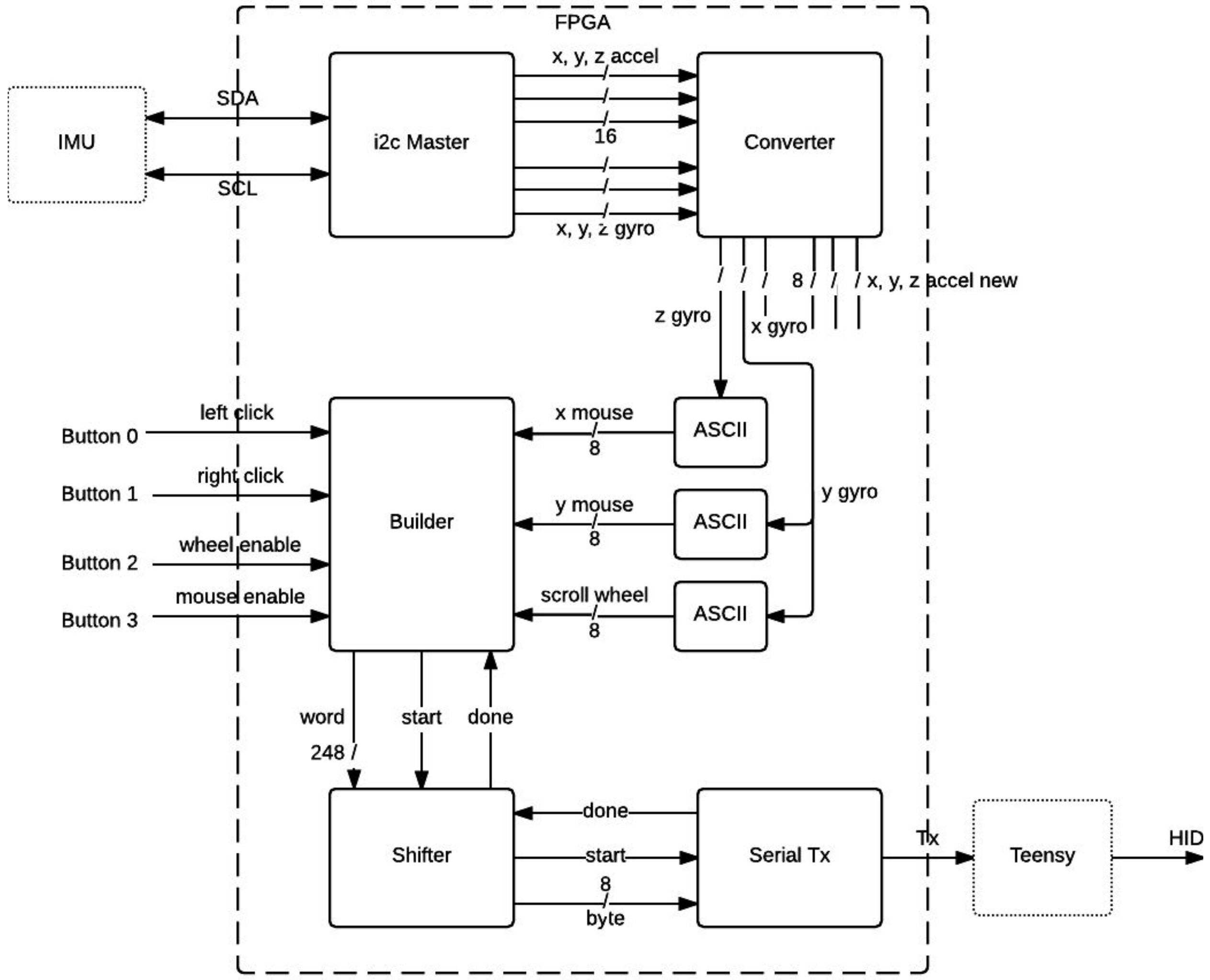
IMU: For the IMU we went with the Invensense MPU-9250. This chip contains a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer. This chip is extremely inexpensive and commonplace in a wide variety of consumer grade applications. Additionally, the chip comes with extensive documentation and a wide knowledge base available on the internet.

FPGA: For our FPGA we will be using the Mojo V3, which is a small development board from EmbeddedMicro featuring a Spartan-6 chip. This board is just small enough (less than 1/4th the footprint of the nexys-4 dev board) to fit comfortably in a user's hand, so it made the obvious choice for our project. An added bonus for selecting the Mojo was the large number of in depth tutorials offered online by EmbeddedMicro.

Microcontroller: In order to stay small, we chose to go with a Teensy 3.2 microcontroller. This micro is capable of extreme speeds, is easy to program, and features a large amount of available code bases and tutorials available online.

3 Implementation

3.1 Block Diagram (NF)



3.2 FPGA Modules

3.2.1 I2C Interpreter (NF)

The purpose of the i2c module is to communicate with the IMU using the i2c protocol in order to retrieve gyroscopic and accelerometer data. It has a clock input from the 50 MHz clock from the Mojo, a reset input, bidirectional input/outputs for SDA and SCL, and 16 bit outputs for each IMU reading.The module consists of an 82 state FSM, which changes state with a frequency of 200 kHz using an internal 200 kHz clock module. What the states do is that they drive the SDA pin to send the start signal, send the device address of the IMU (0x68), send a write bit (0), wait for acknowledgement, send the register address for the MSB of the x acceleration (0x3b), wait for acknowledgement, send a restart signal, send the device address again, send a read bit (1), wait for acknowledgement, and then read the bits on the SDA line being driven by the IMU. Whenever the IMU finishes sending a byte of data, it automatically increments the register address that it is looking at by 1, and so by acknowledging the IMU after every byte received, the i2c master can receive the data in each register sequentially without having to specify each register address. We do run into the problem of reading some unwanted data (temperature sensor data) since it is in between the accelerometer and gyroscope registers, but rather than having to go through the process of specifying a new register address for the IMU to look at, we just collect the unwanted data and leave it unused. After the LSB of the z gyro register is collected, the i2c master does not acknowledge the IMU, and then the i2c master sends a stop bit and updates all of its data outputs in one chunk.

Creating 82 states for this module may not have been the most efficient way to collect data, but the benefit of creating so many states was that it was bound to be relatively error free. And indeed it was, as we were able to assign the readings to the 8 LEDs on the Mojo board and verify that they changed they we expected it to when we manipulated the IMU. In the future though, if we had to read data from many more than 12 registers we would probably design it to work in a way similar to how our shifter and serial modules work together.

3.2.2 Converter Module (AK)

The information found in the IMU registers are in a fractional 2's complement form. Meaning that if the value of the z gyro register was 11796 in binary, then we would have to multiply it by the full scale range of the register (250 degrees per second), and divide by 2^{15}, giving us a value of 90 degrees per second, in binary. So in this module, the original values are converted and passed onto the ASCII module.

This module also offers to some degree control over how fast the mouse will move because it depends on this converted number, so by using a different multiplier we can make its speed faster or slower.

3.2.3 Binary to BCD and Ascii Modules (AK)

An ASCII module will take a signed 8 bit number and convert it to a 3 byte ASCII representation of the value, to be used by the builder module. It makes use of a binary to BCD converter to determine what the values for byte 2 and 3 should be, and there is a combinational block that checks whether the number is positive or negative, and sets the first byte to either the ASCII code for "-" or "0".

3.2.4 Builder Module (NF)

The builder module constructs a 30 byte message for the serial module to send out, where each byte is an ASCII code. The format of the message is "X000_Y000_W000_L0_R0_M0_K000_Z", where the numbers following X tell the Teensy how many pixels to move the mouse in the x direction, and so on. The Y corresponds to y movement, W for scrolling movement, L for left click, R for right click, M for middle click, K for key press, and Z is just a "stop" signal. So for example, the 5 bytes "X-10_" would look like "0x58, 0x2D, 0x31, 0x30, 0x5f". The module takes in external inputs in the form of buttons, which allow for clicks and the enabling of scrolling and mouse movement. Furthermore, it has an internal timing system such that it will signal the shifter module to start, wait for the "done" signal from the shifter, and then trigger a timer, which causes the builder to only send out a message every hundredth of a second.

An issue with the shifter module caused the last byte of the message to never be transmitted, and after many hours of trying to debug the shifter module we decided to just add an extra byte to the end of the message that the builder builds, so that the shifter will transmit the second to last byte.

3.2.5 Shifter Module (NF + AK)

The Serial module requires that only one byte of data be transmitted at one time. The shifter module was made to take the 30 byte message produced by the builder module, then send it to the serial module one byte at a time, while waiting to receive confirmation from the serial module that the previous byte had been sent before providing the next byte to send.

This module was designed as a small finite state machine, with states WAIT, SHIFT, and TRANSMITTING. The module normally stays in the WAIT state until it receives a signal from the builder module telling it to start, after which it will bounce between SHIFT and TRANSMITTING. When it is transitioning between these two states, builder keeps an index of the byte that it is providing for the serial module to send, and shifts its position whenever the serial module says it's done sending the current byte. When builder reaches the last byte of the message, it returns to the WAIT state and signals the builder module that it is done shifting.

This module also gave us the most trouble while trying to integrate, mainly due to timing issues and a few logic errors in our verilog code. We probably re-wrote this module from the ground up 3 or 4 different times. It just goes to show that you should *always* start with a block diagram before trying to code an FSM. We tested this module by providing it with the message "abcdefg...0123" to send (but in ASCII), and we ran into errors such as the shifter only sending every other byte ("aceg…"), sending every other byte and then sending the missing bytes ("aceg...02bdfh...13aceg…"), or sending everything but the last byte ("abcde...012").

3.2.6 Serial Module (AK)

Unlike usual serial systems, this serial module is a purely transmitting one since there is no need to receive any data from the Teensy microcontroller. This module is a FSM with states IDLE, START_BIT, DATA, and STOP_BIT. The state is normally in IDLE where the tx output is held high until a signal is sent to the module that says new data is available and ready to be sent. Then it moves to the START_BIT state where it pulls tx low for 434 clock cycles, which is the necessary number of cycles for the agreed upon baud, which is 115200 in this case. After the start bit is sent, then the state transitions to DATA, where it sends every bit in the byte LSB first, and then transitions to STOP_BIT, where it transmits a 1 on the tx line, and then returns to IDLE. While the module is in the START_BIT, DATA, or STOP_BIT states it outputs a busy signal, which can tell the shifter module its current status.

3.3 Teensy Code (AK)

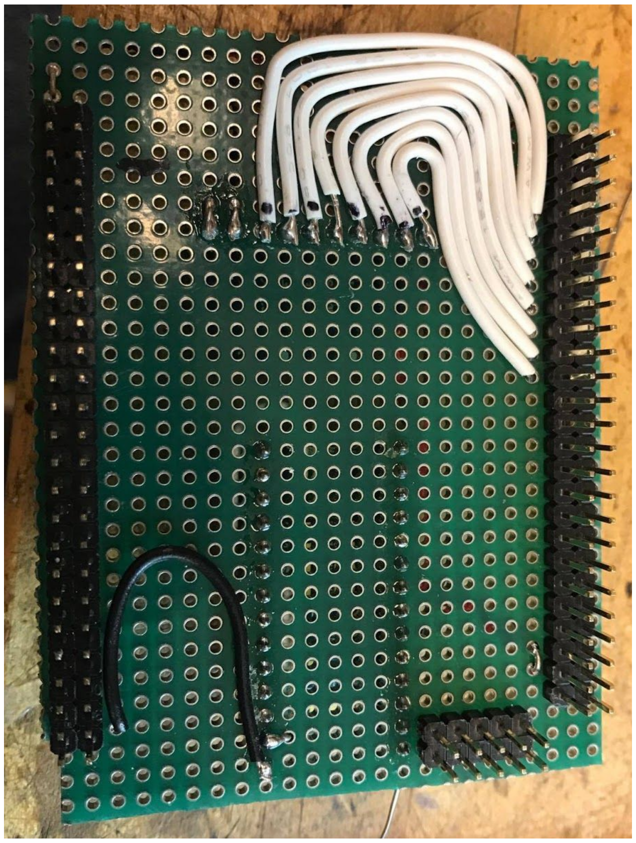
The Teensy code (written in C++) interprets the serial commands from the Mojo, and spits out commands to the computer over USB following the HID protocol. We used open source libraries for the HID protocol and serial communications.

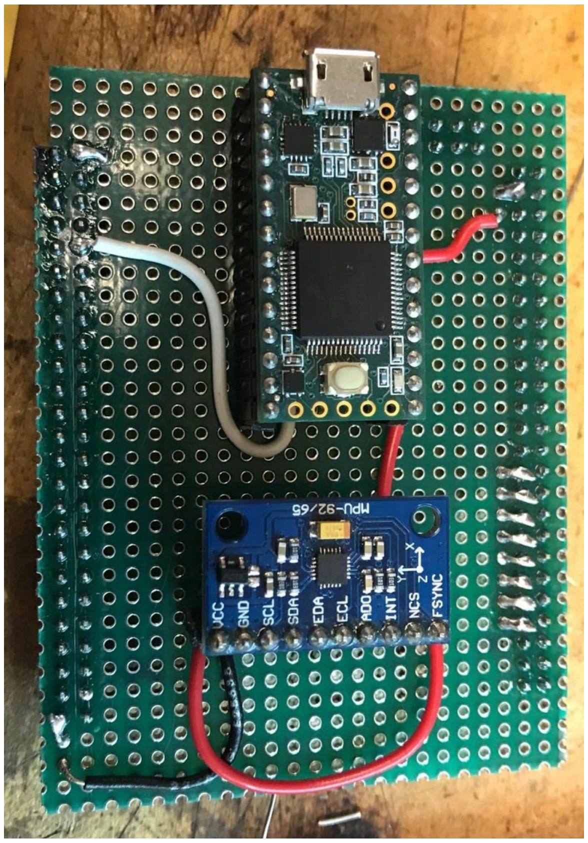
All that the code needed to do was read each individual ASCII character from the onboard serial buffer sequentially and concatenate it to the “message” if the message had not yet reached its full length, or if it had not yet received the “stop” character.

The code then split the message up and interpreted the individual parts as their commands for functions such as mouse movement, scroll wheel movement, mouse clicks, or keypresses.

3.4 Hardware Construction (AK)

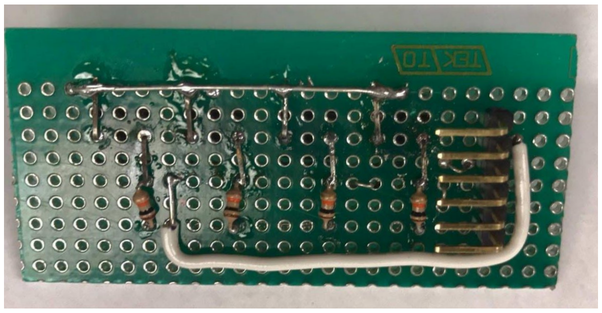
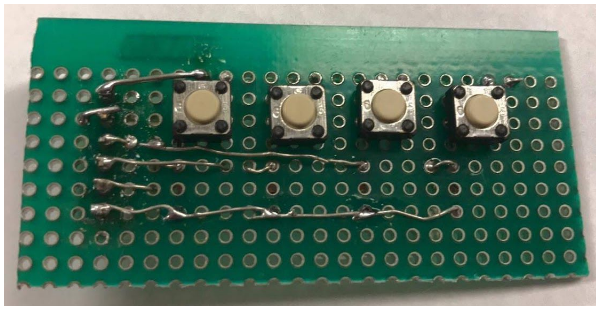
3.4.1 Mojo Shield





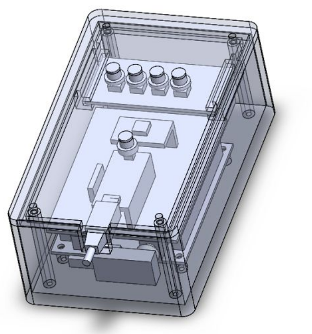
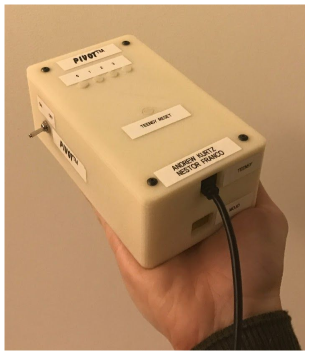
Since our entire project revolves around our ability to easily manipulate our device, it was important that we create a semi-permanent mounting for our electronics, so that we wouldn't have to deal with a mess of breadboards and wires. We thought about potentially etching a custom PCB but settled on using some standard perf board instead.

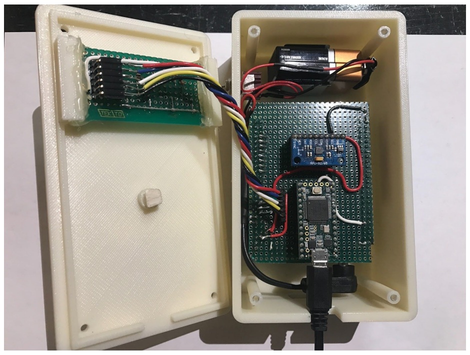
3.4.2 Button Daughter Board

At the time of creation of the Mojo shield and button daughter board, it had not yet been determined where exactly the buttons would be placed in the enclosure. Therefore to ensure that we would not limit ourselves later on, we decided to put the buttons onto a separate small daughter board.

3.4.3 Custom Enclosure



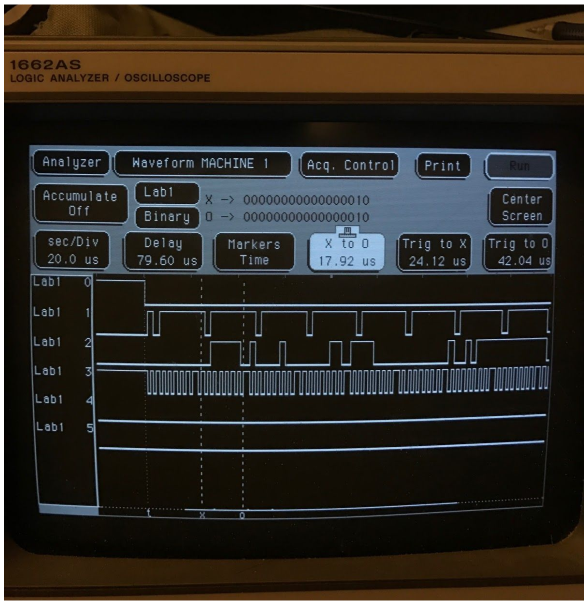
A custom enclosure for Pivot was designed and 3D printed. It features ample room for all the electronics plus a power source, as well as a power switch, pass throughs for reprogramming the FPGA and the Teensy, as well as external, user programmable buttons.

4 Challenges

4.1 Hardware (NF)

The Mojo V3 board uses an FPGA in the Spartan 6 family, and choosing to go with an FPGA that is not typically used in the course curriculum posed a few constraints. The versions of ISE and Vivado installed on the lab computers did not have the option to program FPGAs in the Spartan 6 family, so I had to install ISE and a custom Mojo IDE on my personal desktop computer, restricting the possible places I could work on this project. Luckily I managed to obtain one of the logic analyzers that was being given away making it easier to see and debug signals at home, although that came with a learning curve as well.

For an unknown reason, the Mojo board we received at the start of the project had a few non functional IO pins. Our original plan was to use the SPI protocol to communicate with the IMU since it is a simpler communication scheme, and around the time that I had first started to view some working signals on the logic analyzer, Andrew had created the Mojo shield. Unfortunately, some of the Mojo pins that breadboard wired the IMU pins to turned out to be bad, which I did not realize until later. When I failed to see the same signals on the logic analyzer that I did previously, I got frustrated and stopped developing the SPI module. At that point, I decided to use the i2c module that Joe Steinmeyer had developed, since it was proven to work. When I replicated Joe's setup, I unknowingly reverted back to using the good pins that I was testing with before, and surely enough I got readings from the IMU that made sense. It was only when I plugged the IMU into the breadboard shield and saw that nothing worked did I realize that the Verilog module was not the problem, but rather the pins.



rest in peace SPI

4.2 Teensy Serial Issues (AK)

Originally, it had been assumed that writing the few dozen lines of code which would be needed to get the teensy functioning in a stable state would be trivial, even having gotten a few demos of the HID protocol working with some analog potentiometers. While testing, however, There was just nothing we could do which was working properly. We could not get the teensy to properly read in serial commands, at any baud rate. Eventually, after many hours of testing, debugging, and trying different things we found that we could get the code to work fine if we added a 7ms delay at the end of each code loop, before the start of the next.

Even after figuring out how to circumvent the weird bug one it took a few additional hours, with the help of a Computer Engineer who was more experienced with C++ firmware to find the source of the error: the teensy code ended up calling the serial.available function too quickly, which revealed a race condition between the serial.available() method and cal;calculating how many bytes were in the serial buffer and the serial interrupt that handles putting bytes into the serial buffer. If it was called too quickly there was a higher chance of the update interrupt going off between the serial.available() method copying the head and tail values of the buffer locally before calculating their difference and that would cause an incorrect number of bytes to be calculated inside the buffer.

4.3 Debugging Communication Issues (NF)

The shifter module also proved to be a problem even though it appeared to be a simple concept: provide a byte out of the entire 30 byte message for the serial module to send, keep track of which byte it is currently providing, and loop back when it reaches the end of the message. The first error that we encountered was the first one that was mentioned previously, where the output was just every other byte in the message. First, we tried to work alongside the bug, so we doubled the length of the message to 60 bytes by adding a junk ASCII character after every valid byte in hopes that the bugged shifter would simply send the good bytes and skip the junk bytes. Instead, it would send out our desired message “abcde….xyz0123” followed by “a999999….”, with “9” being our junk character. After going through a few iterations of our shifter and other undesirable outputs, we ended up with a bug that we could work with, which was where the last byte of our message was never sent. So hoping that the bug would be consistent, we added a junk character to the end of our message making it 31 bytes long instead of 30, and luckily that worked out.

5 Future Work (NF)

One thing that remained underutilized from the IMU were the accelerometers. Our intentions were to develop some kind of tap detection so that we would not need to use a button for mouse clicking, but our top priority was to at least get click functionality working. Another thing that the accelerometers could have been used for is gesture recognition. Gestures like drawing a circle could trigger a command such as reload page, or go back a page.

Another thing we tried to implement after base mouse functionality was achieved was keyboard shortcuts, as referenced by the “K000” portion in the builder module. Mouse users might have found it convenient to have a button that acts as the Control/Command + W keys to close tabs and windows, or perhaps an Alt/Command + Tab button to rapidly switch between programs.

Ideally, this project would have been completely contained within the FPGA (apart from the IMU of course). We believe with a bit more research and work into how USB devices communicate, we could have gotten rid of the the Teensy, making this a wholly hardware-based controller.

6 Conclusion (NF + AK)

Our motivation was simple: The ways in which humans interact with computers has not changed much in the last 20+ years. We knew there had to be a better way.

At the start of this project we had a simple goal: implement a motion based user input device designed on an FPGA which could recognized by any computer as a standard keyboard/mouse interface. We achieved, and in a lot of ways, exceeded this goal.

Our final product may not have all of the functionality which we had originally desired, but it has something even more valuable: robust functionality in a user friendly package. The effort we put into the hardware (soldered circuit boards and 3D printed enclosure) and software (robust management of HID/USB interfacing and modular FPGA code) design created a stable development platform which can easily be used by future students to test and implement additional features, such as more complex motion tracking or gesture recognition, without having to worry about messy wiring, I2C clocks, or serial communications. Further, effective development on any user interface requires the designers be able to easily and rapidly test changes, our “no worries” enclosure allows for significantly more effective testing then a few precariously stacked breadboards could ever allow.

Appendix A: Verilog Sources

```
//this is the top module in the hierarchy, that puts together all the other
modules

module mojo_top(
    // 50MHz clock input
    input clk,
    // Input from reset button (active low)
    input rst_n,
    // cclk input from AVR, high when AVR is ready
    input cclk,
    // Outputs to the 8 onboard LEDs
    output[7:0]led,
    // AVR SPI connections
    output spi_miso,
    input spi_ss,
    input spi_mosi,
    input spi_sck,
    // AVR ADC channel select
    output [3:0] spi_channel,
    // Serial connections
    input avr_tx, // AVR Tx => FPGA Rx
    output avr_rx, // AVR Rx => FPGA Tx
    input avr_rx_busy, // AVR Rx buffer full
    inout p33, // SCL
    inout p35, // SDA
    input p29, //button 0
    input p26, //button 1
    input p23, //button 2
    input p21, //button 3
    output p67 // Tx to Teensy
    );

wire db_p29;
debounce db1(.reset(1'b0), .clock(clk), .noisy(p29), .clean(db_p29));

wire db_p26;
debounce db2(.reset(1'b0), .clock(clk), .noisy(p26), .clean(db_p26));

wire db_p23;
debounce db3(.reset(1'b0), .clock(clk), .noisy(p23), .clean(db_p23));

wire db_p21;
debounce db4(.reset(1'b0), .clock(clk), .noisy(p21), .clean(db_p21));
```

```
wire rst = ~rst_n; // make reset active high
wire [6:0] state_display;
wire [15:0] x_accel, y_accel, z_accel, x_gyro, y_gyro, z_gyro;
wire sys_clock;
i2c_master i2c(.clock(clk), .reset(rst), .scl(p33), .sda(p35),
.x_accel(x_accel), .y_accel(y_accel), .z_accel(z_accel),
                      .x_gyro(x_gyro), .y_gyro(y_gyro),
.z_gyro(z_gyro),.state_out(state_display), .sys_clock(sys_clock));

wire [7:0] x_accel_new, y_accel_new, z_accel_new, x_gyro_new, y_gyro_new,
z_gyro_new;
converter poop_convert(.clk(clk), .x_accel(x_accel), .y_accel(y_accel),
.z_accel(z_accel), .x_gyro(x_gyro), .y_gyro(y_gyro), .z_gyro(z_gyro),
                        .x_accel_new(x_accel_new), .y_accel_new(y_accel_new),
.z_accel_new(z_accel_new), .x_gyro_new(x_gyro_new), .y_gyro_new(y_gyro_new),
.z_gyro_new(z_gyro_new));

wire [23:0] x_mouse;
ascii mouse_x(.clk(clk), .toConvert(z_gyro_new), .message(x_mouse));
wire [23:0] y_mouse;
ascii mouse_y(.clk(clk), .toConvert(y_gyro_new), .message(y_mouse));
wire [23:0] wheel;
ascii mousewheel(.clk(clk), .toConvert(y_gyro_new), .message(wheel));

wire shifter_done, shifter_start;
wire [247:0] word;
builder bob(.clock(clk), .mouse_x(x_mouse), .wheel(wheel), .mouse_y(y_mouse),
.enable(db_p21), .left_click(~db_p29), .right_click(~db_p26),
.middle_click(~db_p23), .shifter_done(shifter_done),
.shifter_start(shifter_start), .word(word));

wire serial_done, serial_start;
wire [7:0] serial_to_send;
shifter shifty(.clock(clk), .word(word), .serial_done(serial_done),
.start(shifter_start), .serial_start(serial_start), .done(shifter_done),
.serial(serial_to_send));

serial_tx #(.CLK_PER_BIT(434))
          toTeensy(.clk(clk), .rst(1'b0), .tx(p67), .block(1'b0),
.busy(serial_done), .data(serial_to_send), .new_data(serial_start));

////////////////this stuff is in every skeleton module of mojo_top, so just
ignore//////////////////////////////
// these signals should be high-z when not used
assign spi_miso = 1'bz;
assign avr_rx = 1'bz;
assign spi_channel = 4'bzzzz;
```

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////

assign led = z_gyro[15:8]; //use LEDs for easy debugging of signals

endmodule

module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 100000) clean <= new;
     else count <= count+1;

endmodule

//the i2c_master module is a modified version of Joe's module. I added like 50
more states to the original module,
//which basically just keeps acknowledging after every byte that the IMU sends,
until this module has received
//the information in all the accelerometer and gyroscope registers, after which
it sends a "not acknowledge"

module i2c_master(input clock,
    input reset,
    output reg [15:0] x_accel,
    output reg [15:0] y_accel,
    output reg [15:0] z_accel,
    output reg [15:0] x_gyro,
    output reg [15:0] y_gyro,
    output reg [15:0] z_gyro,
    inout sda,
    inout scl,
    output [6:0] state_out,
    output  sys_clock);

    localparam IDLE = 7'd0; //Idle/initial state (SDA= 1, SCL=1)
    localparam START1 = 7'd1; //FPGA claims bus by pulling SDA LOW while SCL is
HI
    localparam ADDRESS1A = 7'd2; //send 7 bits of device address (7'h68)
    localparam ADDRESS1B = 7'd3; //send 7 bits of device address
    localparam READWRITE1A = 7'd4; //set read/write bit (write here) (a 0)
    localparam READWRITE1B = 7'd5; //set read/write bit (write here)
    localparam ACKNACK1A = 7'd6; //pull SDA HI while SCL ->LOW
    localparam ACKNACK1B = 7'd7; //pull SCL back HI
    localparam ACKNACK1C = 7'd8; //Is SDA LOW (slave Acknowledge)? if so, move
on, else go back to IDLE
    localparam REGISTER1A = 7'd9; //write MPU9250 register we want to read from
(8'h3b)
    localparam REGISTER1B = 7'd10; //write MPU9250 register we want to read
from
    localparam ACKNACK2A = 7'd11; //pull SDA HI while SCL -> LOW
    localparam ACKNACK2B = 7'd12; //pull SCL back HI
    localparam ACKNACK2C = 7'd13; //Is SDA LOW (slave Ack?) If so move one,
else go to idle
    localparam START2A = 7'd14; //SCL -> HI
    localparam START2B = 7'd15; //SDA -> HI
    localparam START2C = 7'd16; //SDA -> LOW (restarts)
    localparam ADDRESS2A = 7'd17; //Address again (7'h68)
    localparam ADDRESS2B = 7'd18; //Address again
    localparam READWRITE2A = 7'd19; //readwrite bit...this time read (1)
    localparam READWRITE2B = 7'd20; //readwrite bit...this time read (1)
    localparam ACKNACK3A = 7'd21; //like other acknacks...wait for MPU to
respond
    localparam ACKNACK3B = 7'd22; //else go back to IDLE
    localparam ACKNACK3C = 7'd23; //""""""
    localparam READ1A = 7'd24; //start reading in data from device
    localparam READ1B = 7'd25; //this data is 8MSB of x accelerometer reading
    localparam ACKNACK4A = 7'd26; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK4B = 7'd27; //Effectively asking for more data
    localparam READ2A = 7'd28; //start reading next 8 bits (8LSB)
    localparam READ2B = 7'd29; //assign to lower half of 16 bit register
    localparam ACKNACK5A = 7'd30; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK5B = 7'd31;
    localparam READ3A = 7'd32; //ACCEL_YOUT_H
    localparam READ3B = 7'd33;
    localparam ACKNACK6A = 7'd34; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK6B = 7'd35;
    localparam READ4A = 7'd36; //ACCEL_YOUT_L
    localparam READ4B = 7'd37;
    localparam ACKNACK7A = 7'd38; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK7B = 7'd39;
    localparam READ5A = 7'd40; //ACCEL_ZOUT_H
    localparam READ5B = 7'd41;
    localparam ACKNACK8A = 7'd42; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK8B = 7'd43;
    localparam READ6A = 7'd44; //ACCEL_ZOUT_L
    localparam READ6B = 7'd45;
    localparam ACKNACK9A = 7'd46; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK9B = 7'd47;
    localparam READ7A = 7'd48; //TEMP_OUT_H
    localparam READ7B = 7'd49;
    localparam ACKNACK10A = 7'd50; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK10B = 7'd51;
    localparam READ8A = 7'd52; //TEMP_OUT_L
    localparam READ8B = 7'd53;
    localparam ACKNACK11A = 7'd54; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK11B = 7'd55;
    localparam READ9A = 7'd56; //GYRO_XOUT_H
    localparam READ9B = 7'd57;
    localparam ACKNACK12A = 7'd58; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK12B = 7'd59;
    localparam READ10A = 7'd60; //GYRO_XOUT_L
    localparam READ10B = 7'd61;
    localparam ACKNACK13A = 7'd62; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK13B = 7'd63;
    localparam READ11A = 7'd64; //GYRO_YOUT_H
    localparam READ11B = 7'd65;
    localparam ACKNACK14A = 7'd66; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK14B = 7'd67;
    localparam READ12A = 7'd68; //GYRO_YOUT_L
    localparam READ12B = 7'd69;
    localparam ACKNACK15A = 7'd70; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK15B = 7'd71;
    localparam READ13A = 7'd72; //GYRO_ZOUT_H
    localparam READ13B = 7'd73;
    localparam ACKNACK16A = 7'd74; //Master (FPGA) assets acknowledgement to
Slave
    localparam ACKNACK16B = 7'd75;
    localparam READ14A = 7'd76; //GYRO_ZOUT_L
    localparam READ14B = 7'd77;
    localparam NACK = 7'd78; //Fail to acknowledge Slave this time (way to say
"I'm done so slave doesn't send more data)
    localparam STOP1A = 7'd79; //Stop/Release line
    localparam STOP1B = 7'd80; //FPGA master does this by pulling SCL HI while
SDA LOW
    localparam STOP1C = 7'd81; //Then pulling SDA HI while SCL remains HI
    reg [6:0] device_address = 7'h68;
    reg [7:0] register_address = 8'h3b;
    reg [7:0] count = 0;

    reg [6:0] state = IDLE;
    assign state_out = state;

    reg [15:0] accel_x = 16'h0000;
    reg [15:0] accel_y = 16'h0000;
    reg [15:0] accel_z = 16'h0000;
    reg [15:0] temp = 16'h0000;
    reg [15:0] gyro_x = 16'h0000;
    reg [15:0] gyro_y = 16'h0000;
    reg [15:0] gyro_z = 16'h0000;

    reg sda_val=1; //from the fsm perspective, where SDA output data is placed.
    assign sda =  sda_val ? 1'bz: 1'b0;  //if sda_data  = 1, make hiZ, else
0...rely on external pullup resistors

    reg scl_val=1;
    assign scl = scl_val ? 1'bz : 1'b0; //if scl_val = 1, make hiZ, else 0...do
this for clock stretching.

    reg read_write =1;

    assign sys_clock = state==IDLE?1'b1:1'b0;

    reg clock_reset = 0;
    wire clock_for_sys;
    //assign sys_clock = clock_for_sys?  1'bz : 0;
    clock_200khz local_clock(.reset(clock_reset), .clock(clock),
.slow_clock(clock_for_sys));


    always @(posedge clock_for_sys)begin //update only on rising/fall edges of
i2c clock
        if (reset &&(state !=IDLE))begin
            state <= IDLE;
            count <=0;
        end else begin
            case (state)
                IDLE: begin
                    if (reset) state <= IDLE;
                    else if (count == 60)begin
                        state <= START1;
                        count <=0;
                    end
                    count <= count +1;
                    sda_val <=1;
                    scl_val <=1;

                end
                START1: begin
                    sda_val <= 0; //pull SDA low
                    scl_val <=1;
                    state <=ADDRESS1A;
                    count <= 6;
                end
                ADDRESS1A: begin
                    scl_val<=0;
                    sda_val <= device_address[count];
                    state <= ADDRESS1B;
                end
                ADDRESS1B: begin
                    scl_val <=1;
                    if (count >= 1) begin
                        count <= count -1;
                        state <= ADDRESS1A;
                    end else begin
                        state <= READWRITE1A;
                    end
                end
                READWRITE1A: begin
                    scl_val <=0;
                    sda_val <=0;//write address
                    state <= READWRITE1B;
                end
                READWRITE1B: begin
                    scl_val <=1;
                    state <= ACKNACK1A;
                end
                ACKNACK1A: begin
                    scl_val <=0;
                    sda_val <=1; //float sda for listening next time
                    state <= ACKNACK1B;
                end
                ACKNACK1B: begin
                    scl_val <=1;
                    state <=ACKNACK1C;
                    count <=7;
                end
                ACKNACK1C: begin
                    scl_val <=0;
                    //acknowledge <= sda;  //what do we have?
                    if (sda ==1'b1)begin //no acknowledgement
                        count <=0;
                        state <= IDLE;
                    end else begin
                        state <= REGISTER1B;
                        sda_val <= register_address[count];
                    end
                end
                REGISTER1A: begin
                    scl_val <=0;
                    sda_val <= register_address[count];
                    state <= REGISTER1B;
                end
                REGISTER1B: begin
                    scl_val <=1;
                    if (count>0) begin
                        count <= count -1;
                        state <= REGISTER1A;
                    end else begin
                        state <= ACKNACK2A;
                    end
                end
                ACKNACK2A: begin
                    scl_val <=0;
                    sda_val <=1; //float sda for listening next time
                    state <= ACKNACK2B;
                end
                ACKNACK2B: begin
                    scl_val <=1;
                    state <=ACKNACK2C;
                end
                ACKNACK2C: begin
                    scl_val <=0;
                    //acknowledge <= sda;  //what do we have?
                    if (sda ==1'b1)begin //no acknowledgement
                        state <= IDLE;
                        count <=0;
                    end else begin
                        state <= START2A;
                        sda_val<=0;
                        count <=15;
                    end
                end
                START2A: begin
                    scl_val <=1;
                    state <= START2B;
                end
                START2B: begin
```

```
    sda_val <= 1;
    state <= START2C;
end
START2C: begin
    sda_val <= 0; //pull down while SCL is high
    state <= ADDRESS2A;
    count <=6;
end
ADDRESS2A: begin
    scl_val<=0;
    sda_val <= device_address[count];
    state <= ADDRESS2B;
end
ADDRESS2B: begin
    scl_val <=1;
    if (count >= 1) begin
        count <= count -1;
        state <= ADDRESS2A;
    end else begin
        state <= READWRITE2A;
    end
end
READWRITE2A: begin
    scl_val <=0;
    sda_val <=1;//read address
    state <= READWRITE2B;
end
READWRITE2B: begin
    scl_val <=1;
    state <= ACKNACK3A;
end
ACKNACK3A: begin
    scl_val <=0;
    sda_val <=1; //float sda for listening next time
    state <= ACKNACK3B;
end
ACKNACK3B: begin
    scl_val <=1;
    state <=ACKNACK3C;
    count <=7;
end
ACKNACK3C: begin
    scl_val <=0;
    //acknowledge <= sda;  //what do we have?
    if (sda ==1'b1)begin //no acknowledgement
        count <=0;
        state <= IDLE;
    end else begin
```

```
        state <= READ1A;
        sda_val <= 1;
    end
end
READ1A: begin
    scl_val <=1;
    state <= READ1B;
end
READ1B: begin //ACCEL_XOUT_H
    scl_val <=0;
    accel_x[count+8] <= sda;
    if (count >=1)begin
        count <= count -1;
        state<=READ1A;
    end else begin
        state<=ACKNACK4A;
        sda_val <=0;
    end
end
ACKNACK4A: begin
    scl_val <=1;
    state <=ACKNACK4B;
    count <=7;
end
ACKNACK4B: begin
    scl_val <=0;
    state <=READ2A;
    count <=7;
    sda_val <=1;
end
READ2A: begin
    scl_val <=1;
    state <= READ2B;
end
READ2B: begin //ACCEL_XOUT_L
    scl_val <=0;
    accel_x[count] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ2A;
    end else begin
        state<=ACKNACK5A;
        sda_val<=0;
    end
end
ACKNACK5A: begin
    scl_val <=1;
    state <=ACKNACK5B;
    count <=7;
end
ACKNACK5B: begin
    scl_val <=0;
    state <=READ3A;
    count <=7;
    sda_val <=1;
end
READ3A: begin
    scl_val <=1;
    state <= READ3B;
end
READ3B: begin //ACCEL_YOUT_H
    scl_val <=0;
    accel_y[count+8] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ3A;
    end else begin
        state<=ACKNACK6A;
        sda_val<=0;
    end
end
ACKNACK6A: begin
    scl_val <=1;
    state <=ACKNACK6B;
    count <=7;
end
ACKNACK6B: begin
    scl_val <=0;
    state <=READ4A;
    count <=7;
    sda_val <=1;
end
READ4A: begin
    scl_val <=1;
    state <= READ4B;
end
READ4B: begin //ACCEL_YOUT_L
    scl_val <=0;
    accel_y[count] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ4A;
    end else begin
        state<=ACKNACK7A;
        sda_val<=0;
    end
```

```
end
ACKNACK7A: begin
    scl_val <=1;
    state <=ACKNACK7B;
    count <=7;
end
ACKNACK7B: begin
    scl_val <=0;
    state <=READ5A;
    count <=7;
    sda_val <=1;
end
READ5A: begin
    scl_val <=1;
    state <= READ5B;
end
READ5B: begin //ACCEL_ZOUT_H
    scl_val <=0;
    accel_z[count+8] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ5A;
    end else begin
        state<=ACKNACK8A;
        sda_val<=0;
    end
end
ACKNACK8A: begin
    scl_val <=1;
    state <=ACKNACK8B;
    count <=7;
end
ACKNACK8B: begin
    scl_val <=0;
    state <=READ6A;
    count <=7;
    sda_val <=1;
end
READ6A: begin
    scl_val <=1;
    state <= READ6B;
end
READ6B: begin //ACCEL_ZOUT_L
    scl_val <=0;
    accel_z[count] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ6A;
    end else begin
        state<=ACKNACK9A;
        sda_val<=0;
    end
end
ACKNACK9A: begin
    scl_val <=1;
    state <=ACKNACK9B;
    count <=7;
end
ACKNACK9B: begin
    scl_val <=0;
    state <=READ7A;
    count <=7;
    sda_val <=1;
end
READ7A: begin
    scl_val <=1;
    state <= READ7B;
end
READ7B: begin //TEMP_OUT_H
    scl_val <=0;
    temp[count+8] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ7A;
    end else begin
        state<=ACKNACK10A;
        sda_val<=0;
    end
end
ACKNACK10A: begin
    scl_val <=1;
    state <=ACKNACK10B;
    count <=7;
end
ACKNACK10B: begin
    scl_val <=0;
    state <=READ8A;
    count <=7;
    sda_val <=1;
end
READ8A: begin
    scl_val <=1;
    state <= READ8B;
end
READ8B: begin //TEMP_OUT_L
    scl_val <=0;
    temp[count] <= sda;
    if (count >= 1)begin
        count <= count -1;
        state<=READ8A;
    end else begin
        state<=ACKNACK11A;
        sda_val<=0;
    end
end
ACKNACK11A: begin
    scl_val <=1;
    state <=ACKNACK11B;
    count <=7;
end
ACKNACK11B: begin
    scl_val <=0;
    state <=READ9A;
    count <=7;
    sda_val <=1;
end
READ9A: begin
    scl_val <=1;
    state <= READ9B;
end
READ9B: begin //GYRO_XOUT_H
    scl_val <=0;
    gyro_x[count+8] <= sda;
    if (count >=1)begin
        count <= count -1;
        state<=READ9A;
    end else begin
        state<=ACKNACK12A;
        sda_val <=0;
    end
end
ACKNACK12A: begin
    scl_val <=1;
    state <=ACKNACK12B;
    count <=7;
end
ACKNACK12B: begin
    scl_val <=0;
    state <=READ10A;
    count <=7;
    sda_val <=1;
end
READ10A: begin
    scl_val <=1;
```

```
        state <= READ10B;
    end
    READ10B: begin //GYRO_XOUT_L
        scl_val <=0;
        gyro_x[count] <= sda;
        if (count >= 1)begin
            count <= count -1;
            state<=READ10A;
        end else begin
            state<=ACKNACK13A;
            sda_val<=0;
        end
    end
    ACKNACK13A: begin
        scl_val <=1;
        state <=ACKNACK13B;
        count <=7;
    end
    ACKNACK13B: begin
        scl_val <=0;
        state <=READ11A;
        count <=7;
        sda_val <=1;
    end
    READ11A: begin
        scl_val <=1;
        state <= READ11B;
    end
    READ11B: begin //GYRO_YOUT_H
        scl_val <=0;
        gyro_y[count+8] <= sda;
        if (count >= 1)begin
            count <= count -1;
            state<=READ11A;
        end else begin
            state<=ACKNACK14A;
            sda_val<=0;
        end
    end
    ACKNACK14A: begin
        scl_val <=1;
        state <=ACKNACK14B;
        count <=7;
    end
    ACKNACK14B: begin
        scl_val <=0;
        state <=READ12A;
        count <=7;
        sda_val <=1;
    end
    READ12A: begin
        scl_val <=1;
        state <= READ12B;
    end
    READ12B: begin //GYRO_YOUT_L
        scl_val <=0;
        gyro_y[count] <= sda;
        if (count >= 1)begin
            count <= count -1;
            state<=READ12A;
        end else begin
            state<=ACKNACK15A;
            sda_val<=0;
        end
    end
    ACKNACK15A: begin
        scl_val <=1;
        state <=ACKNACK15B;
        count <=7;
    end
    ACKNACK15B: begin
        scl_val <=0;
        state <=READ13A;
        count <=7;
        sda_val <=1;
    end
    READ13A: begin
        scl_val <=1;
        state <= READ13B;
    end
    READ13B: begin //GYRO_ZOUT_H
        scl_val <=0;
        gyro_z[count+8] <= sda;
        if (count >= 1)begin
            count <= count -1;
            state<=READ13A;
        end else begin
            state<=ACKNACK16A;
            sda_val<=0;
        end
    end
    ACKNACK16A: begin
        scl_val <=1;
        state <=ACKNACK16B;
        count <=7;
    end
```

```
    ACKNACK16B: begin
        scl_val <=0;
        state <=READ14A;
        count <=7;
        sda_val <=1;
    end
    READ14A: begin
        scl_val <=1;
        state <= READ14B;
    end
    READ14B: begin //GYRO_ZOUT_L
        scl_val <=0;
        gyro_z[count] <= sda;
        if (count >= 1)begin
            count <= count -1;
            state<=READ14A;
        end else begin
            state<=NACK;
            sda_val<=1;
        end
    end
    NACK: begin
        scl_val <=1;
        count <=0;
        x_accel[15:0] <= accel_x[15:0];
        y_accel[15:0] <= accel_y[15:0];
        z_accel[15:0] <= accel_z[15:0];
        x_gyro[15:0] <= gyro_x[15:0];
        y_gyro[15:0] <= gyro_y[15:0];
        z_gyro[15:0] <= gyro_z[15:0];
        state <= STOP1A;
    end
    STOP1A: begin
        scl_val <=0;
        sda_val <=0;
        state <= STOP1B;
    end
    STOP1B: begin
        scl_val <= 1;
        sda_val <=0;
        state <=STOP1C;
    end
    STOP1C: begin
        sda_val <=1;
        state <= IDLE;
    end

endcase
```

```
        end
    end


endmodule

//clock for the i2c module
module clock_200khz   (input reset, input clock, output reg slow_clock);
    reg [7:0] count=0;
    always @(posedge clock)begin
        if (reset) begin
            count <=0;
            slow_clock <=1;
        end else begin
            if (count ==125)begin
                count <=0;
                slow_clock <= !slow_clock;
            end else begin
                count <= count +1;
            end
        end
    end
endmodule

//converts the raw data into reasonable numbers

module converter(
      input signed [15:0] x_accel,
      input signed [15:0] y_accel,
      input signed [15:0] z_accel,
      input signed [15:0] x_gyro,
      input signed [15:0] y_gyro,
      input signed [15:0] z_gyro,
      output reg signed [7:0] x_accel_new,
      output reg signed [7:0] y_accel_new,
      output reg signed [7:0] z_accel_new,
      output reg signed [7:0] x_gyro_new,
      output reg signed [7:0] y_gyro_new,
      output reg signed [7:0] z_gyro_new
      );
      reg signed [15:0] gyro_mult = 250;
      reg signed [15:0] accel_mult = 2;
      always@ (*)
      begin
            x_accel_new = ((x_accel*accel_mult) >>> 14)*5;
            y_accel_new = ((y_accel*accel_mult) >>> 14)*5;
            z_accel_new = ((z_accel*accel_mult) >>>14)*5;
            x_gyro_new = ((x_gyro*gyro_mult) >>> 14)*2;
            y_gyro_new = (~(((y_gyro*gyro_mult) >>> 14))+1)/2;
            z_gyro_new = (~(~((z_gyro*gyro_mult) >>> 14)+1)+1)/2; //I found
that for some reason inverting the value twice made it work right
      end

endmodule

//this module will take a signed 8-bit binary number
//and output a 3 character long ascii representation
//for example it would take "-30" and output "45" (ascii code for '-')
//"51" (ascii code for '3') "48" (ascii code for '0')

module ascii (
      input signed [7:0] toConvert,
      output [23:0] message
      );

      reg sign;
  reg [7:0] reg_message = 8'd48;
      reg [7:0] absToConvert = 8'd0;
      wire [1:0] Hundreds;
      wire [7:0] Tens;
      wire [7:0] Ones;

      always@(toConvert)
      begin
            sign = toConvert[7];
            if (sign) begin
                  reg_message = 8'd45;
                  absToConvert = (~toConvert + 1);
            end
            else begin
                  reg_message = 8'd48;
                  absToConvert = toConvert;
            end
      end

  binary_to_BCD b1(.A(absToConvert), .ONES(Ones), .TENS(Tens),
.HUNDREDS(Hundreds));
  assign message[23:16] = reg_message;
  assign message[15:8] = (Tens + 8'd48);
  assign message[7:0] = (Ones + 8'd48);
endmodule

module binary_to_BCD(A,ONES,TENS,HUNDREDS);
input [7:0] A;
output [7:0] ONES, TENS;
output [1:0] HUNDREDS;
wire [3:0] c1,c2,c3,c4,c5,c6,c7;
wire [3:0] d1,d2,d3,d4,d5,d6,d7;

assign d1 = {1'b0,A[7:5]};
assign d2 = {c1[2:0],A[4]};
assign d3 = {c2[2:0],A[3]};
assign d4 = {c3[2:0],A[2]};
assign d5 = {c4[2:0],A[1]};
assign d6 = {1'b0,c1[3],c2[3],c3[3]};
assign d7 = {c6[2:0],c4[3]};
add3 m1(d1,c1);
add3 m2(d2,c2);
add3 m3(d3,c3);
add3 m4(d4,c4);
add3 m5(d5,c5);
add3 m6(d6,c6);
add3 m7(d7,c7);
assign ONES = {4'b0,c5[2:0],A[0]};
assign TENS = {4'b0,c7[2:0],c5[3]};
assign HUNDREDS = {c6[3],c7[3]};

endmodule

module add3(in,out);
input [3:0] in;
output [3:0] out;
reg [3:0] out;

always @ (in)
      case (in)
      4'b0000: out <= 4'b0000;
      4'b0001: out <= 4'b0001;
      4'b0010: out <= 4'b0010;
      4'b0011: out <= 4'b0011;
      4'b0100: out <= 4'b0100;
      4'b0101: out <= 4'b1000;
      4'b0110: out <= 4'b1001;
      4'b0111: out <= 4'b1010;
      4'b1000: out <= 4'b1011;
      4'b1001: out <= 4'b1100;
      default: out <= 4'b0000;
      endcase
endmodule

//this module constructs a 31 byte message of ASCII characters to send via
serial
```

```
module builder (input clock,
                    input [23:0] mouse_x,
                    input [23:0] mouse_y,
                    input [23:0] wheel,
                    input left_click,
                    input right_click,
                    input middle_click,
                    input enable,
                    input shifter_done,
                    output reg shifter_start,
                    output [247:0] word);

    reg state = 0, next_state = 0, reg_shifter_start = 1;
    parameter [1:0] SEND = 0, SEND2 = 1, WAIT = 2, WAIT2 = 3;

    wire clock_start;
    wire expired;
    clock_100hz timer(.reset(~clock_start), .clock(clock),
.slow_clock(expired));

    always @(*) begin
        case (state)
            SEND:   begin
                        reg_shifter_start = 1;
                        next_state = SEND2;
                    end

            SEND2:   begin
                        reg_shifter_start = 1;
                        next_state = WAIT;
                    end

            WAIT:   begin
                        reg_shifter_start = 0;
                        next_state = WAIT2;
                    end

            WAIT2:   begin
                        reg_shifter_start = 0;
                        next_state = expired ? SEND : WAIT2;
                    end
        endcase
    end

    always @(posedge clock) begin
        state <= next_state;
        shifter_start <= reg_shifter_start;
    end
    assign clock_start = shifter_done; //when shifter is done, start the timer

    wire [39:0] xvel = (enable) ? {8'h58, 8'h30, 8'h30, 8'h30, 8'h5f} : {8'h58,
mouse_x, 8'h5f};
    wire [39:0] yvel = (enable) ? {8'h59, 8'h30, 8'h30, 8'h30, 8'h5f} : {8'h59,
mouse_y, 8'h5f};
    wire [39:0] wvel = (middle_click) ? {8'h57, wheel, 8'h5f} : {8'h57, 8'h30,
8'h30, 8'h30, 8'h5f};
    wire [39:0] kpress = {8'h4b, 8'h30, 8'h30, 8'h30, 8'h5f};

    wire [23:0] lclick = (left_click) ? {8'h4c, 8'h32, 8'h5f} : {8'h4c, 8'h30,
8'h5f};
    wire [23:0] rclick = (right_click) ? {8'h52, 8'h31, 8'h5f} : {8'h52, 8'h30,
8'h5f};
    wire [23:0] mclick = {8'h4d, 8'h30, 8'h5f};

    assign word = {xvel,yvel,wvel,lclick,rclick,mclick,kpress,8'h5a,8'h5a};
endmodule

//this module isn't really a clock, I'm just using it like a timer for the
builder module
module clock_100hz   (input reset, input clock, output reg slow_clock);
    reg [7:0] count=0;
    always @(posedge clock)begin
        if (reset) begin
            count <=0;
            slow_clock <= 0;
        end else begin
            if (count == 369792) begin
                count <=0;
                slow_clock <= !slow_clock;
            end else begin
                count <= count +1;
            end
        end
    end
endmodule


//this module splits the 31 byte message into byte sized chunks for the serial
module to send

module shifter (input clock,
                    input [247:0] word,
                    input serial_done,
                    input start,
                    output reg serial_start,
                    output reg done,
                    output reg [7:0] serial); //baud = 115200

    reg [7:0] count = 8'd247, next_count = 8'd247;
    reg [1:0] state = 2'b00, next_state = 2'b00;
    reg next_serial_start = 0;

    parameter WAIT = 2'b00;
    parameter SHIFT = 2'b01;
    parameter TRANSMITTING = 2'b10;

    always @(posedge clock ) begin
        case (state)
            WAIT:   begin
                        next_state <= start ? SHIFT : WAIT;
                        next_serial_start <= 0;
                        done <= 1;
                    end

            SHIFT:  begin
                        if (count > 7) begin
                            next_serial_start <= 1;
                            next_count <= count-8;
                            next_state <= TRANSMITTING;
                            done <= 0;
                        end
                        else if (count == 8'd7) begin
                            next_serial_start <= 1;
                            next_count <= count;
                            next_state <= TRANSMITTING;
                            done <= 0;
                        end
                    end
            TRANSMITTING: begin
                            if ((count == 8'd7) && (~serial_done)) begin
                                next_state <= WAIT;
                                next_count <= 8'd247;
                                next_serial_start <= 0;
                                done <= 0;
                            end
                            else if (~serial_done) begin
                                next_state <= SHIFT;
                                next_count <= count;
                                next_serial_start <= 0;
                                done <= 0;
                            end

                            else begin
```

```
                                next_state <= TRANSMITTING;
                                next_count <= count;
                                next_serial_start <= 0;
                                done <= 0;
                            end
                end
            endcase

            count <= next_count;
            state <= next_state;
            serial_start <= next_serial_start;
            serial[7] <= word[count];
            serial[6] <= word[count-1];
            serial[5] <= word[count-2];
            serial[4] <= word[count-3];
            serial[3] <= word[count-4];
            serial[2] <= word[count-5];
            serial[1] <= word[count-6];
            serial[0] <= word[count-7];
        end

endmodule

//serializes byte data

module serial_tx #(
    parameter CLK_PER_BIT = 434
  )(
    input clk,
    input rst,
    output tx,
    input block,
    output busy,
    input [7:0] data,
    input new_data
  );

  // clog2 is 'ceiling of log base 2' which gives you the number of bits needed
to store a value
  parameter CTR_SIZE = $clog2(CLK_PER_BIT);

  localparam STATE_SIZE = 2;
  localparam IDLE = 2'd0,
    START_BIT = 2'd1,
    DATA = 2'd2,
    STOP_BIT = 2'd3;

  reg [CTR_SIZE-1:0] ctr_d, ctr_q;
  reg [2:0] bit_ctr_d, bit_ctr_q;
  reg [7:0] data_d, data_q;
  reg [STATE_SIZE-1:0] state_d, state_q = IDLE;
  reg tx_d, tx_q;
  reg busy_d, busy_q;
  reg block_d, block_q;

  assign tx = tx_q;
  assign busy = busy_q;

  always @(*) begin
    block_d = block;
    ctr_d = ctr_q;
    bit_ctr_d = bit_ctr_q;
    data_d = data_q;
    state_d = state_q;
    busy_d = busy_q;

    case (state_q)
      IDLE: begin
        if (block_q) begin
          busy_d = 1'b1;
          tx_d = 1'b1;
        end else begin
          busy_d = 1'b0;
          tx_d = 1'b1;
          bit_ctr_d = 3'b0;
          ctr_d = 1'b0;
          if (new_data) begin
            data_d = data;
            state_d = START_BIT;
            busy_d = 1'b1;
          end
        end
      end
      START_BIT: begin
        busy_d = 1'b1;
        ctr_d = ctr_q + 1'b1;
        tx_d = 1'b0;
        if (ctr_q == CLK_PER_BIT - 1) begin
          ctr_d = 1'b0;
          state_d = DATA;
        end
      end
      DATA: begin
        busy_d = 1'b1;
        tx_d = data_q[bit_ctr_q];
        ctr_d = ctr_q + 1'b1;
```

```
        if (ctr_q == CLK_PER_BIT - 1) begin
          ctr_d = 1'b0;
          bit_ctr_d = bit_ctr_q + 1'b1;
          if (bit_ctr_q == 7) begin
            state_d = STOP_BIT;
          end
        end
      end
      STOP_BIT: begin
        busy_d = 1'b1;
        tx_d = 1'b1;
        ctr_d = ctr_q + 1'b1;
        if (ctr_q == CLK_PER_BIT - 1) begin
          state_d = IDLE;
        end
      end
      default: begin
        state_d = IDLE;
      end
    endcase
  end

  always @(posedge clk) begin
    if (rst) begin
      state_q <= IDLE;
      tx_q <= 1'b1;
    end else begin
      state_q <= state_d;
      tx_q <= tx_d;
    end

    block_q <= block_d;
    data_q <= data_d;
    bit_ctr_q <= bit_ctr_d;
    ctr_q <= ctr_d;
    busy_q <= busy_d;
  end

endmodule
```

Appendix B: Mojo Constraints

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 50 MHz HIGH 50%;

NET "clk" LOC = P56 | IOSTANDARD = LVTTL;
NET "rst_n" LOC = P38 | IOSTANDARD = LVTTL;

NET "cclk" LOC = P70 | IOSTANDARD = LVTTL;

NET "led<0>" LOC = P134 | IOSTANDARD = LVTTL;
NET "led<1>" LOC = P133 | IOSTANDARD = LVTTL;
NET "led<2>" LOC = P132 | IOSTANDARD = LVTTL;
NET "led<3>" LOC = P131 | IOSTANDARD = LVTTL;
NET "led<4>" LOC = P127 | IOSTANDARD = LVTTL;
NET "led<5>" LOC = P126 | IOSTANDARD = LVTTL;
NET "led<6>" LOC = P124 | IOSTANDARD = LVTTL;
NET "led<7>" LOC = P123 | IOSTANDARD = LVTTL;

NET "spi_mosi" LOC = P44 | IOSTANDARD = LVTTL;
NET "spi_miso" LOC = P45 | IOSTANDARD = LVTTL;
NET "spi_ss" LOC = P48 | IOSTANDARD = LVTTL;
NET "spi_sck" LOC = P43 | IOSTANDARD = LVTTL;
NET "spi_channel<0>" LOC = P46 | IOSTANDARD = LVTTL;
NET "spi_channel<1>" LOC = P61 | IOSTANDARD = LVTTL;
NET "spi_channel<2>" LOC = P62 | IOSTANDARD = LVTTL;
NET "spi_channel<3>" LOC = P65 | IOSTANDARD = LVTTL;

NET "avr_tx" LOC = P55 | IOSTANDARD = LVTTL;
NET "avr_rx" LOC = P59 | IOSTANDARD = LVTTL;
NET "avr_rx_busy" LOC = P39 | IOSTANDARD = LVTTL;

NET "p33" LOC = P33 | IOSTANDARD = LVTTL;
NET "p35" LOC = P35 | IOSTANDARD = LVTTL;
NET "p67" LOC = P67 | IOSTANDARD = LVTTL;

NET "p29" LOC = P29 | IOSTANDARD = LVTTL;
NET "p26" LOC = P26 | IOSTANDARD = LVTTL;
NET "p23" LOC = P23 | IOSTANDARD = LVTTL;
NET "p21" LOC = P21 | IOSTANDARD = LVTTL;
```

Appendix C: Teensy Code

```
void setup() {
  // put your setup code here, to run once:

Serial1.begin(115200);
//Serial.begin(115200);
pinMode(13,OUTPUT);
Mouse.begin();
Keyboard.begin();
}

void loop() {
//int messageLength = 30;
int xmove = 0;
int ymove = 0;
int wmove = 0;
//int wmove_new = 0;
int lclick = 0;
int rclick = 0;
int mclick = 0;
int Kpress = 0;
char currentChar;
String Message = String("");
//Serial2.println('q');

while (Serial1.available() ) {
  currentChar = Serial1.read();
  //Serial.print(currentChar);
  digitalWrite(13,HIGH);

  if (currentChar == 'Z' ) {
    //Serial1.println("here");
    break;
  }

  //else if (currentChar == '9'){
    //break;
  //}
  else if (Message.length() < 29  && currentChar != 'X'){
    Message += currentChar;
    //Serial2.println(Message);
    //break;
 }
  //else {
   // Message += currentChar;
```

```
    // Serial1.println(Message);
    // break;
  //}
  //Message += currentChar;
  //Serial1.println(Message);
 }
 //Serial2.println("made");
 digitalWrite(13,LOW);

 //if (Message.length() != 0){
 //Serial2.println(Message.length());
 //}
 //message should be of the form "x###_y###_w###_l#_r#_m#_k###_z"
 if (Message.length() == 28){
 xmove = ((Message.substring(0,3).toInt())/-2);
 ymove = ((Message.substring(5,8).toInt())/2);
 //wmove = ((Message.substring(10,13).toInt())/-20);
 wmove = ((Message.substring(10,13).toInt())/-20);
 //wmove = map(wmove, 0, 100, 0, 10);
 lclick = Message.substring(15,16).toInt();
 rclick = Message.substring(18,19).toInt();
 mclick = Message.substring(21,22).toInt();
 Kpress = Message.substring(24,27).toInt();
 Serial1.println("hi");
 //digitalWrite(13,HIGH);
 //delay(500);
 }
 if ((xmove != 0) || (ymove != 0) || (wmove != 0)){
   Mouse.move(xmove,ymove,wmove);
   //digitalWrite(13,HIGH);
   //delay(250);
 }
 if (Kpress != 0){
   Keyboard.write(Kpress);
 }
  if (lclick > 0){
   Mouse.press(MOUSE_LEFT);
 }
   if (lclick == 0){
     Mouse.release(MOUSE_LEFT);
   }
 if (rclick > 0){
   Mouse.click(MOUSE_RIGHT);
 }
 if (mclick > 0){
   Mouse.click(MOUSE_MIDDLE);
 }
 delay(6);
```

```
}
```