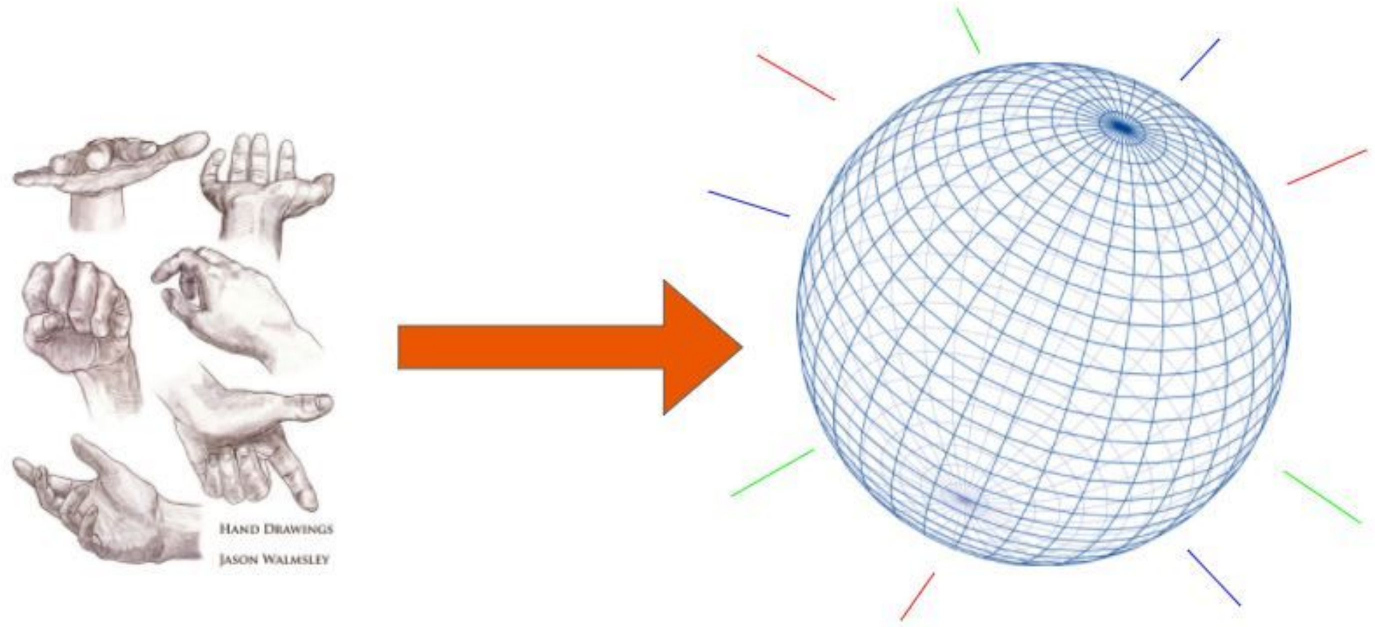
Gesture-Controlled LED Orb

Final Project Report
December 5, 2017
E155

Christopher Kotcherha and Nicholas Sakowski



Abstract

There are endless possibilities of the complex systems we can make with readily existing gadgets such as LED strips, accelerometers, and EKG sensors. We live in an opportune time where we possess a bounty of such easy-to-use digital technology alongside a lack of fun, inventive devices compared to all those that could be. This project attempts to strike a new vein in that underexploited world of fun devices and presents users the opportunity to play with light and color in new, creative ways. The "Magic Orb" prototypes an LED globe controlled by a wrist-worn device with the assistance of a microcontroller and FPGA. Motion data captured by the wristband, such as acceleration, rotation, and hand gestures, are interpreted by the microcontroller and sent off to the FPGA. This device uses a hardware-generated PWM signal to control the color of each LED in the globe, resulting in a real-time user experience consisting of multiple pre-programed animations and games.

Introduction

Our final project is an LED orb (322 individually RGB programmable LEDs) which displays motion-controlled animations based on IMU data. The user experience consists of wearing a Myo™ armband and signaling new animation states on the orb via specific hand gestures, recognizable by the Myo's 8 EKG sensors. At a high level, the user experience with the orb was organized into a top menu, navigable via gestures, and selectable mini-games, which are controllable via user motions.

The top menu functioned by having the orb emit a single color to indicate the currently selected mini-game. The user can 'swipe' through the list of games by making a 'wave-out' gesture with the hand equipped with a Myo. When the user decides on a game choice (unique by color), they can make a 'fist' gesture to begin the mini-game. However, at any time during the games, the user can make a 'wave-in' gesture to exit the game back to the original menu.

For this project, the user experience consisted of two playable mini-games. The color 'indigo' indicated the 'color spectrum' game. Once entered, the orb will display a single color at any given time, but with its various colors mapped to correspond to the user's arm orientation (pitch). Through this experience, the user can smoothly and quickly transition between the colors of our spectrum by motioning their arm up and down.

The second available game, indicated by the menu color of 'green', is the 'ring-game.' Once this game is begun, the orb would only display colors on a single axis, effectively displaying a series of concentric rings. However, each of the red rings would have a subset of 4 green LEDs indicating a subsection of the ring. These subsections would be scattered about, with the goal of this game to align the subsections together at the top of the orb. The currently 'selected' ring would display blue and green (subsection) instead of red and green. The user can change their selected ring by jerking their arm forward (shift up) or backward (shift down). They must shift the orientation of the subsection along its host ring by rolling their arm counterclockwise (left shift) or clockwise (right shift). Once the rings are all aligned, a victory animation plays and the user is returned to the menu.

New Hardware

The LED orb was constructed from a WS2812b strip arranged into 14 rings (two orthogonal groups/axis of seven arranged concentrically) inside a translucent, white acrylic lighting fixture. These LEDs were chosen as they are capable of producing a large range of colors and, despite being so popular, few people have managed to control them with FPGAs. A range of colors (red, orange, yellow, green, blue, teal, violet, magenta) can be displayed and individual LEDs can be controlled via C scripts on the Raspberry Pi (which send data to the FPGA via SPI). Any combination or pattern of these colors is possible, since each LED is individually programmable. Also various custom animations (multiple display frames in sequence) are implemented, but any scripted sequence of patterns is possible with this architecture.

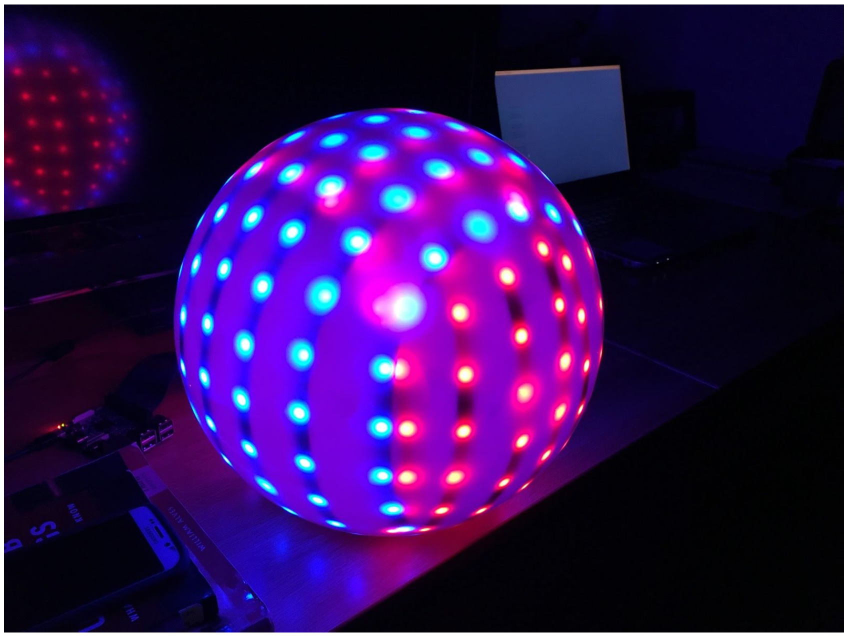
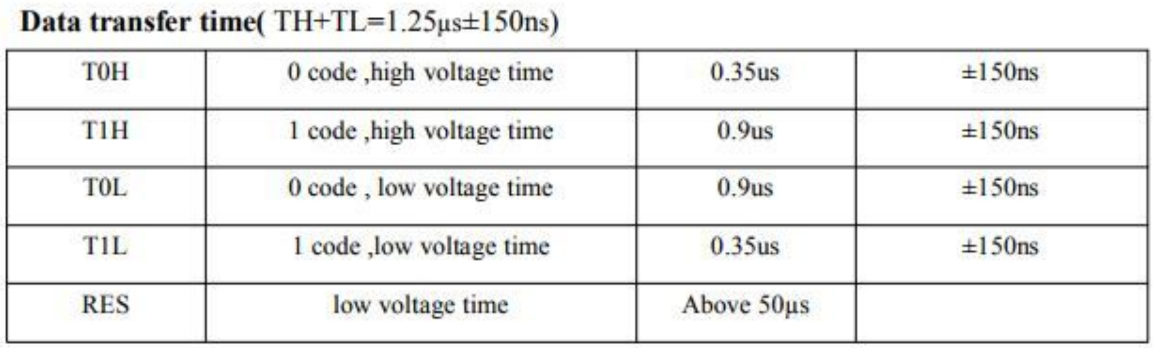


Figure 1: LED orb - 2 axis structure

The most important concern during design and development is the timing specifications presented by the LEDs. According to their datasheet, they operate as a serial register with PWM logic. The specifics of these specifications can be seen below as well as a graphic that demonstrates the three possible signals.


Data transfer time(TH+TL=1.25µs±150ns)

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.9us	±150ns
T0L	0 code , low voltage time	0.9us	±150ns
T1L	1 code ,low voltage time	0.35us	±150ns
RES	low voltage time	Above 50µs	

Table 1: LED encoding - timing specifications

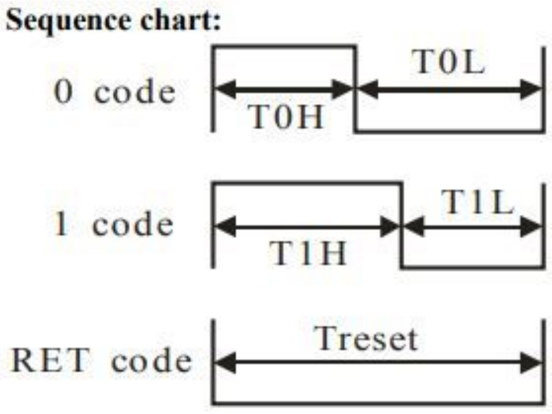


Figure 2: LED encoding - logic

Tolerances this fine cannot be achieved on a Raspberry Pi given the fact that the SOM runs a distribution of Linux that is not real-time. Thus, these signals must be produced by an FPGA's GPIO output pin.

The Myo Armband is a 3rd party product that uses a bluetooth protocol to transmit packets of its sensor data to a USB Bluetooth dongle plugged into to Raspberry Pi. Instead of using the maker company's (Thalmic Labs) official communication and scripting environment (MyoConnect), a custom python environment was created to increase creative freedom with the project. The exact myo-specific communication protocol customized for Linux (PyoConnect) was taken from the open source software community for Myo development, in order to communicate with the 3rd party device. Custom data handlers and scripting were added to process the incoming IMU and gesture data and to trigger the animations written in C that determined LED color values by communicating with the FPGA.

Schematics

The overall system can be divided into three relationships: the Myo™ armband's data processing and overall game logic done on the Raspberry Pi in Python, the determination of animation state done on the Raspberry Pi in C and communicated to the FPGA over SPI, and the control of the LED orb done by the FPGA. We will discuss each subsystem and their capabilities. Below is a high level overview of our system. Physical connections are broken down to signals behaving and inputs and outputs to each subsystem.

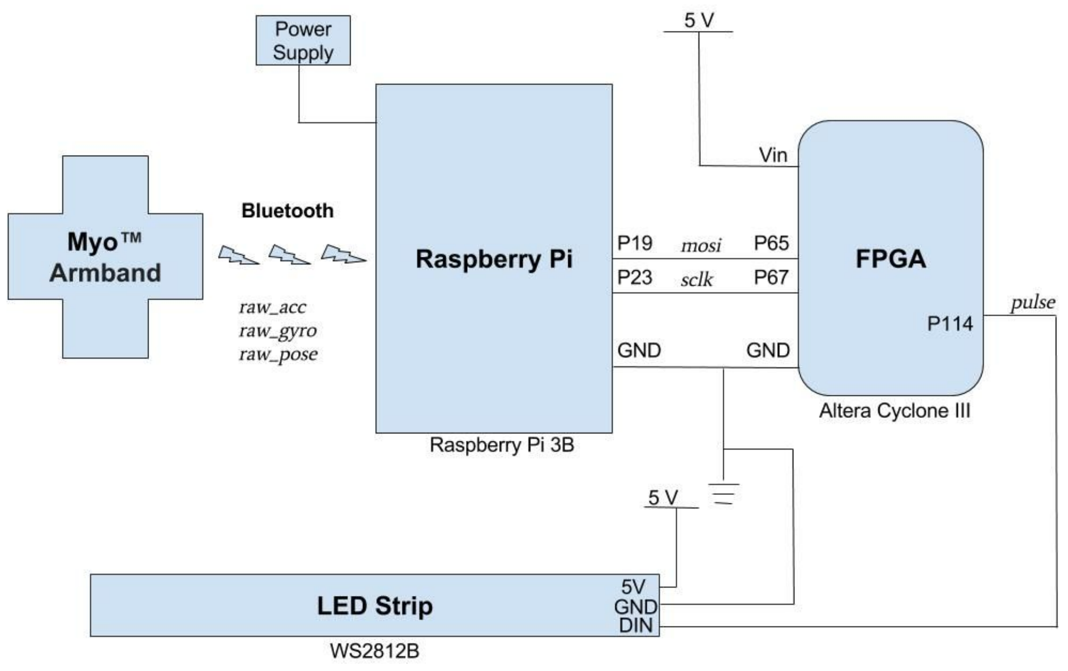


Figure 3: Full Schematic and Breadboard Diagram

Software Design

The python scripting environment on the Pi included 3 major components. The first of which is the Myo-specific bluetooth protocol. As mentioned previously, this protocol was documented in the open source development community for the Myo-armband, and was implemented in python exactly as documented. Other than these existing files, there is no official datasheet or documentation by which the bluetooth communication and package parsing can be inferred.

The second part of the python environment involves a data handler for the incoming stream of gesture-recognition data. This gesture handler takes in the already-recognized gestures, since the Myo's firmware has built in data interpretation for the EKG sensors specifically. The data comes in as strings such as 'WAVEIN', 'WAVEOUT', 'FIST', and other presets, but only when a gesture is recognized, otherwise no data enters the handler. This handler is specialized to manipulate the main menu and the current 'highlighted mode'. 'WAVEOUT' swipes through the menu options (sets the highlighted mode), and calls a C executable that corresponds to the desired highlighted mode color. 'FIST' launches the currently selected game option (indigo = color spectrum, green = ring-game) by setting the global 'mode' variable to that game's value (menu is default mode 0). 'WAVEIN' exits any game back to the main menu by setting the mode back to 0.

The third aspect of the python code is the IMU data handler. It streams in all the gyroscope and accelerometer data at 50Hz. This incoming motion data is then processed and interpreted by the mini-game scripts within the handler. Each game has an associated 'mode' so that only one game is running at any time. This variable is set by the menu interaction, as mentioned before. Once in one of the two game modes (mode 1 = color spectrum, mode 2 = ring game), the motion data is used to trigger certain events, and these events correspond to calling a C executable associated with an LED color or animation. For example, a roll-rate threshold (followed by a temporary halt on data processing to prevent multiple triggers in quick succession) causes the ring orientation in the ring-game to shift by updating a single argument in the ring_game executable call.

The link between the Raspberry Pi and the FPGA are a set of C executables which transmit the appropriate, encoded RGB data to be interpreted (the handling of this data will be discussed further in the 'FPGA' section). These files correspond to different LED animations and take in different arguments from a Python script depending on user motion data. The function of currently implemented executables include setting the orb one of eight solid colors and the 'ring game' mentioned in the Introduction. Motion data captured using the Python script is handled and a system call is made to the C file with data encoding the currently selected ring, the position of all 7 rings, and the win condition. These applications can be accessed from the orb's 'main menu', a state in which the user swipes through solid colors on the orb which correspond to different games.

Each executable file sends an array of 162 chars (322 nibbles + 1 byte) to the FPGA over SPI. To save registers on the FPGA, this array is encoded such that each nibble corresponds to the color of one LED in the strip, with the last byte acting as a ‘stop-byte’ signal ('~'). The encoding is as follows:

0x0010	Red
0x0011	Orange
0x0100	Yellow
0x0101	Green
0x0110	Blue
0x1000	Teal
0x1001	Violet
0x1010	Magenta
0x0111_1110	Stop

Note that it is impossible to send a combination of encoded colors that could be confused for the stop bit. The encodings shown were chosen as such for exactly that reason.

FPGA Design

The LED strip used in this system (WS2812b) operates in series via PWM logic and has very strict timing constraints discussed previously. To properly control the LED strip, we require the use of a real-time system such as an FPGA. With a 40 MHz clock and the ability to run complex operations in hardware, the Altera Cyclone IV is fully capable of creating pulse width modulated signals within the given 150 ns tolerances.

The purpose of the FPGA in this system is to interpret 161 bytes of encoded RGB data via SPI from the Raspberry Pi, decode said bytes to 322 24-bit RGB codes, and send those codes in sequence to the LED strip using PWM logic. We will now discuss the logic behind this operation in detail.

The hardware description can be divided into two major components: the SPI data handling, and the LED controller. Beginning with the SPI handler, we have two modules that work together to interpret 161 +1 bytes from the Raspberry Pi. The first, *spi_slave_receive_only*, contains a 1296 bit buffer that holds the last 162 bytes sent from the Pi. This module works on the Pi's serial clock, *sck*. Another faster module, *spi_data_grab*, running on the FPGA clock checks to see if the 8 least significant bits of the first module's buffer are equal to the 'stop-byte' ('~'). When this byte is encountered, the first 161 bytes are shifted to a signal named *pi_data* and the led controller is given a signal indicating that the data is ready to interpret.

A 161-byte bus was chosen over storing this data in RAM for convenience. At the time of design, this implementation made sense and made the datapath easier to visualize. This would become a hindrance, however, if data sent from the Pi was in the form of 1 byte per LED, as it would require more registers than the FPGA contains. It would be a great advantage in the future if the design was modified such that data from the Pi was stored into RAM sequentially and read out sequentially in the LED controller modules. This would allow the Pi to send more detailed information to the FPGA and could make writing the Pi-end software more easy for developers. This would not cost any additional lag to the design, but would make the SPI process twice as long (which would be a negligible difference to the user).

The second major component, the LED controller, does not operate until it sees the *dat_ready* signal go high. Comprised of two modules, *stripcontrol* and *pwm*, this major component translates each 4-bit 'nibble' from *pi_data* to a 24-bit RGB value by shifting the signal through a decoder. For each 3-byte 'chunk', *stripcontrol* indicates to *pwm* whether to send a 0 or a 1, starting with the most significant bit. The *pwm* module is the last step in the chain and acts as an FSM which alternates between sending long high signals with short low signals, and short high signals with low long signals. When a bit is sent via PWM, the *pwm* module sends a *done* signal to *stripcontrol*, and the process is repeated until 322*24 = 7728 bits have been processed over PWM. When all 322 nibbles have been processed, the *pwm* module is forced to wait for 50 μs, which indicates a 'latch' signal to the LED strip.The length of this part of the process takes 9.71 ms and is perceived as instantaneous by the user. A block diagram of the logic described is shown below.

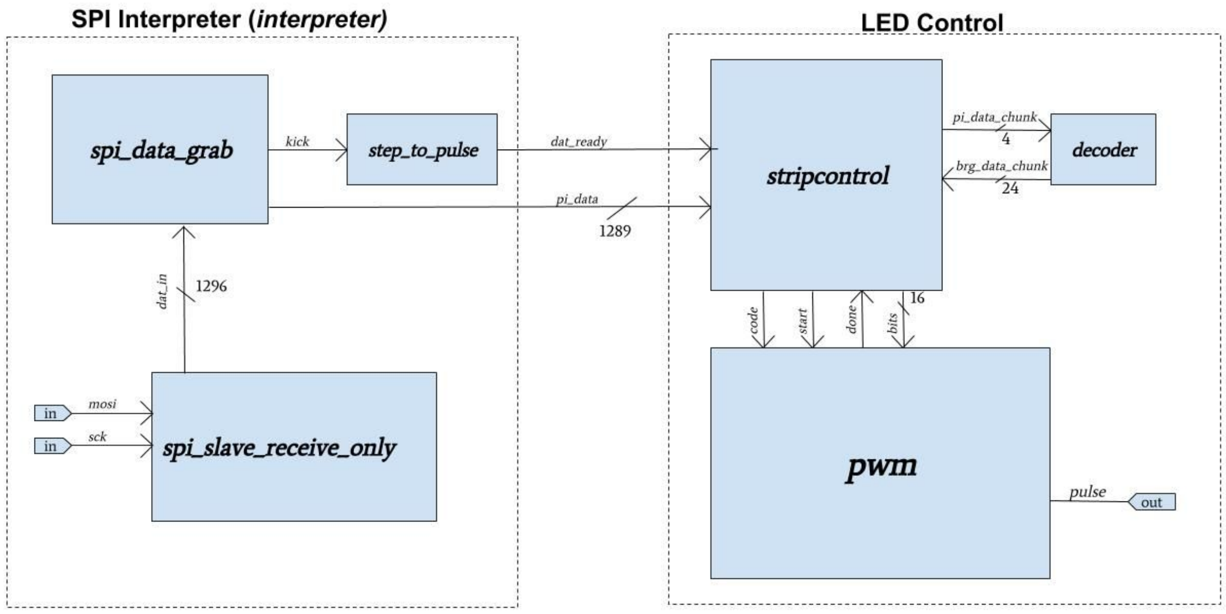


Figure 4: FPGA Logic Block Diagram

It is important to note that the data signal (called *pulse* on the FPGA) is output from a header pin on the FPGA board and not a GPIO connected to the breadboard. Due to the parasitic nature of breadboards, the levels of capacitance could cut off high frequency signals such as those being sent to the LED strip, so this is not an option.

Results

The performance of our final system is very close to the intended goals. Myo communication with the Raspberry Pi was flawless and low-latency. The python scripting executed exactly as designed, interfacing well with both incoming data and C executables. The C executables also executed flawlessly with SPI communication to the FPGA. The PWM from the FPGA is also highly reliable. Constructed hardware, including the LED display and orb also performed well with no maintenance required.

The only difference between the proposed design and the final product is the “focal point” game. This game would have allowed the user to control the position of a point of light on the orb with pitch and yaw hand motions. This design would have required the ability to easily map and translate arbitrary designs to the orb, which no previous animations had utilized. This was very close to becoming a reality with a vectorial approach on the Pi, but was not completely realized due to the lack of spatial resolution imposed by having to send 2 LEDs worth of information at a time over SPI. It was at this point that the consequences of not storing LED data in RAM on the FPGA came to fruition.

This method precalculates 322, 3-coordinate representations of vectors that point to each of the orb’s LEDs. To create designs, a ‘master vector’ is chosen and the dot product of the master and each of the other 322 vectors is taken. The value of this dot product is then sigmoid-ed to a scale of 1-7 and rounded to the nearest integer. This value then corresponds to a color in a specified color spectrum, leading to a rainbow gradient design growing in the direction of the master vector, or a single point/group of points that only exist at the tip of the master vector. This method, as mentioned previously, is only held back by the fact that 2 LEDs worth of information must be sent to the FPGA at a time, making the final resolution of the orb’s designs choppy and not pleasing to look at.

Parts List

Part Name	Purpose	Qty	Unit Cost	Cost
Raspberry Pi 3B	Master/Wifi/Motion Estimator	1	-	-
MicroMudd FPGA board	Slave/LED Driver	1	-	-
12” Acrylic Globe	Outside Cover	1	$16.97	$16.97
16.4’ RGB LED strip	LEDs	2	$19.69	$39.38
Power Wall Adapter (9 V)	Main Power	1	$11.99	$11.99
MYO wristband	Gesture/Position Tracking	2	-	-

Purchasing

Out of Pocket …………….. $28.96

To be reimbursed …………. $51.36

Total: …………………….. **$80.32**

Appendix

```
// Nicholas Sakowski and Chris Kotcherha
// Microprocessors Final Project

// This file describes the logic required to
// manipulate WS2812B LED strip, 322 elements
// long.

module final_nsck(input logic clk,
        input logic reset,
        input logic sck,
        input logic mosi,
        output logic pulse);      // Output to LED strip

logic dat_ready;         // Data is ready to be sent to LED strip
logic [1287:0] pi_data; // Last 322 nibbles + 1 byte from pi

// Hanldes data from Pi
interpreter int1(clk, sck, mosi, pi_data, dat_ready);

// Interprets data and controls LED strip
leds led1(clk, reset, dat_ready, pi_data, pulse);

endmodule

// SPI interpretation modules
module interpreter(input logic clk,
        input logic sck,
        input logic mosi,
        output logic [1287:0] pi_data,
        output logic dat_ready);

        logic kick;      // Triggers led controller
        logic [1295:0] dat_in;  // All data from pi

        // Handle SPI data
        spi_slave_receive_only spi(sck, mosi, dat_in);

        // Tell us when we're good to read SPI data and pass it on
        spi_data_grab sdg(clk, dat_in, pi_data, kick);

        // Tell LED controller when to start
        pulser p(clk, kick, dat_ready);

endmodule

// LED control modules
module leds(input logic clk,
                        input logic reset,
                        input logic dat_ready,
                        input logic [1287:0] pi_data,
                        output logic pulse);

        logic start, code, done;
        logic [11:0] bits;

        // Translate SPI data to 24 bit BRG data
        stripcontrol strip1(clk, reset, dat_ready, pi_data, done, code, start, bits);

        // LED strip driver
        pwm pwm1(clk, reset, code, bits, start, done, pulse);

endmodule
```

```
// SPI slave module to receive data from Pi
// Keeps track of last 1296 bits 322 nibbles + 1 byte
module spi_slave_receive_only(input logic sck, //From master
                              input logic mosi,
                              output logic [1295:0] dat_in);

        always_ff @(posedge sck)
                dat_in <= {dat_in[1294:0], mosi}; // shift register
endmodule

// A faster helper module that checks for
// the stop byte and alerts the leds module
// when it is time to read data
module spi_data_grab(input logic clk,
        input logic [1295:0] dat_in,
        output logic [1287:0] pi_data,
        output logic kick);

        always_ff @(posedge clk)
          begin
            if (dat_in[7:0] == 8'h7e) // '~', the stop byte
                begin
                        pi_data <= dat_in[1295:8];      // Grab the last 322 nibbles
                        kick <= 1;      // Alert the led module
                 end
            else        kick <= 0;
           end

endmodule

// Step to pulse converter
module pulser(input logic clk,
        input logic q,
        output logic d);

        logic was;

        always_ff @(posedge clk)
           begin
                was <= q;
                d = q & ~was;
           end

endmodule

// Nibble to 3byte BRG data decoder
module decoder(input logic [3:0] pi_data_chunk,
        output logic [23:0] brg_data_chunk);

        // brg_data_chunk is in G - R - B order!

        always_comb
        case(pi_data_chunk)
                4'b0010:        brg_data_chunk = 24'h004400;    // Red
                4'b0011:        brg_data_chunk = 24'h1a2c00;    // Orange
                4'b0100:        brg_data_chunk = 24'h222c00;    // Yellow
                4'b0101:        brg_data_chunk = 24'h440000;    // Green
                4'b0110:        brg_data_chunk = 24'h000044;    // Blue
                4'b1000:        brg_data_chunk = 24'h220022;    // Indigo
                4'b1001:        brg_data_chunk = 24'h00242e;    // Violet
                4'b1010:        brg_data_chunk = 24'h002420;    // Magenta
                default: brg_data_chunk = 24'h000000;   // Off
        endcase
endmodule
```

```
// This module decodes nibbles from the Pi
// and tells the PWM circuit which bits to send
module stripcontrol(input logic clk,
        input logic reset,
        input logic dat_ready,          // Data is ready to be read
        input logic [1287:0] pi_data,   // Full data from rpi
        input logic done,               // From timer
        output logic code,              // 0 or 1?
        output logic start,             // Kick timer
        output logic [15:0] bits);      // Number of bits to send to strip

        logic [1287:0] pi_data_temp;
        logic [23:0] brg_3bytes, data;
        logic [7:0] counter;

        decoder dec1(pi_data_temp[1287:1284], brg_3bytes);

        // State Register
        always_ff @(posedge clk)
           begin
                if (reset) start = 0;

                else if(dat_ready)              // Fresh data is ready to be displayed
                   begin
                        pi_data_temp = pi_data; // This needs to happen first
                        data = brg_3bytes;      // This needs to happen second
                        code = data[23];        // Read data msb to lsb
                        bits = 16'h1e30;        // # bits to send (322*24)
                        counter = 8'h00;        // # bits sent
                        start = 1;              // Kick pwm circuit
                   end

                else if (done && bits)          // If the pwm is ready and we still have data to send
                   begin
                        if (counter == 8'h18)   // if 24 bits already sent to pwm
                           begin
                                counter = 8'h0;         // reset bit-sent counter
                                // after every 3 bytes, shift pi_data by 1 nibble
                                pi_data_temp = pi_data_temp << 3'b100;
                                data = brg_3bytes;      // Read new 24 byte chunk from decoder
                           end

                        else data = data << 1'b1;

                        code = data[23];
                        bits = bits - 1'b1;             // Bits left to send overall
                        counter = counter + 1'b1;       // # bits from current chunk already sent to timer
                        start = 1;                      // Kick timer
                   end

                else start = 0;

                end

endmodule
```

```
// This module is repsonsible for
// interpreting a 1 or a 0 as a pwm
// signal
module pwm(input logic clk,
        input logic reset,
        input logic code,               // 0 or 1?
        input logic [15:0] bits,        // # of bits to send overall
        input logic start,              // start pwm
        output logic done,              // Tell stripcontrol we're ready for next bit
        output logic pulse);            // LED strip's input

        typedef enum logic [3:0] {S0,S1,S2,S3,S4,S5,S6,S7} statetype;
        statetype state, nextstate;


        logic [11:0] counter;
        logic [11:0] lastcounter;       // previous value of counter

        logic [7:0] first, second;
        logic [7:0] first0, second0, first1, second1;
        logic [11:0] res;

        assign res = 12'hfff;           // 102.375 microseconds (RES)

        assign first1 = 8'h24;          // 0.9 microseconds
        assign second1 = 8'h0e;         // 0.35 microseconds
        assign first0 = second1;
        assign second0 = first1;


        // State Register
        always_ff @(posedge clk)
                if(reset)
                   begin
                        state = S0;
                        done = 0;
                        pulse =0;
                   end

                else if(start)  // If we're ready to send pwm to LED strip
                   begin
                        state = S1;
                        pulse = 1;      // PWM always starts on a high
                        if(code)        // Bit to send  == 1
                           begin
                                first = first1;  // High duty cycle
                                second = second1;
                           end
                        else    // Bit to send == 0
                           begin
                                first = first0; // Low duty cycle
                                second = second0;
                           end
                   end

                else
                   begin
                        state = nextstate;
                        if (state == S5)        done = 1;       // Send done signal
                        else done = 0;
                        if(state == S3) lastcounter = second;
                        else if (state == S6)   lastcounter = res;
                        else lastcounter = counter;
                        if(counter == 1) pulse = 0;     // Set pulse low before waiting extra clock cycle
                   end
        // Nextstate Logic
        always_comb
        case(state)
                S0:     nextstate = S0;                          // Do nothing
                S1:     nextstate = S2;                          // Start counting
                S2:     if (counter == 1) nextstate = S3;        // Go to low pulse
                        else nextstate = S2;                     // Stay
                S3:     nextstate = S4;                                  // Start 2nd cycle
                S4:     if (counter == 1 && bits) nextstate = S5;        // Go to DONE
                        else if(counter == 1 && ~bits) nextstate = S6;   // Send RES signal
                        else nextstate = S4;                             // Stay
                S5:     nextstate = S0;         // DONE
                S6:     nextstate = S7;         // Set RES timer
                S7:     if (counter == 1) nextstate = S5;
                        else nextstate = S7;    // Stay
        endcase

        // Output Logic
        always_comb
        case(state)
                S0:             counter <= 0;
                S1:             counter <= first;               // Start counting down from first
                S2:             counter <= lastcounter - 1'b1;
                S3:             counter <= second;              // Start counting down from second
                S4:             counter <= lastcounter - 1'b1;
                S5:             counter <= 0;
                S6:             counter <= res;                 // Start counting down for reset signal
                S7:             counter <= lastcounter - 1'b1;
        endcase

endmodule
```

```
// ring_game.c
// An interactive game in which the user rotates red and green rings to
// line up the green bands. The selected ring is shown as blue (except
// for the green band)
#include "easypio.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
        char received;
        char red[1] = "\x22";
        char green[1] = "\x55";
        char halfg[1] = "\x52";
        char halfr[1] = "\x25";
        char blank[1] = "p";
        char stop[1] = "~";

        int start[7] = {75,85,95,106,117,128,138}; // LED ring start positions
        int end[7] = {84,94,105,116,127,137,161};  // LED ring end positions

        // Input arguements from interactive script
        int ring1 = (int)(argv[1][0] - '0') + start[0];
        int ring2 = (int)(argv[2][0] - '0') + start[1];
        int ring3 = (int)(argv[3][0] - '0') + start[2];
        int ring4 = (int)(argv[4][0] - '0') + start[3];
        int ring5 = (int)(argv[5][0] - '0') + start[4] + 1;
        int ring6 = (int)(argv[6][0] - '0') + start[5] + 1;
        int ring7 = (int)(argv[7][0] - '0') + start[6];
        int selected = (int)(argv[8][0] - '0');

        pioInit();
        spiInit(244000,0);

        for(int j=0;j<161;j++)
        {
                if(j < 75)
                {
                        received = spiSendReceive(blank[0]);
                }
                else if(j > ring1-2 && j < ring1+1)
                {
                        received = spiSendReceive(green[0]);
                }
                else if(j > ring2-2 && j < ring2+1)
                {
                        received = spiSendReceive(green[0]);
                }
                else if(j > ring3-2 && j < ring3+1)
                {
                        received = spiSendReceive(green[0]);
                }
                else if(j == ring4-1)
                {
                        received = spiSendReceive(halfr[0]);
                }
                else if(j > ring4-2 && j < ring4+1)
                {
                        received = spiSendReceive(green[0]);
                }
                else if(j == ring4+1)
                {
                        received = spiSendReceive(halfg[0]);
                }
                else if(j > ring5-2 && j < ring5+1)
                {
                        received = spiSendReceive(green[0]);
                }
                else if(j > ring6-2 && j < ring6+1)
```

```
            {
                    received = spiSendReceive(green[0]);
            }

            else if (j == ring7-1)
            {
                    received = spiSendReceive(halfr[0]);
            }
            else if(j > ring7-2 && j < ring7+1)
            {
                    received = spiSendReceive(green[0]);
                    if(j == ring7)
                    {
                            received = spiSendReceive(halfg[0]);
                    }
            }
            else if(j > 75)
            {
                    if(j > start[selected] && j < start[selected]){
                            received = spiSendReceive(blue[0])
                    }
                    else{
                            received = spiSendReceive(red[0]);
                    }
            }
    }
    received = spiSendReceive(stop[0]);

    return 0;
}
```

```
// menu.c
// A swirly, colorful display to represent the orb's menu.
// This program utilizes a 3D representation of vectors
// to map out the position of each LED on the orb, and calculates
// a master vector about which symmetrical displays can be made.
// Idea credits to Alex Goldstein.
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "easypio.h"

#define SPHERE_RADIUS 6.0 // inches
#define LED_INTERVAL 1.28 // inches
#define PI 3.14159265359

#define X_DIR 0
#define Y_DIR 1

#define X_NUM 7 // # rings
const float X_OFFSETS[X_NUM] = {0,0, 0, 0, 0, 0, 0};
const float X_ERRORS[X_NUM] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0}; // Compensate for hardware issues
const int X_LENGTHS[X_NUM] = {18, 22, 22 ,24 ,24, 20, 18};  // # LEDs in each ring

#define Y_NUM 7 // # rings
const float Y_OFFSETS[Y_NUM] = {0,0, 0, 0, 0, 0, 0};
const float Y_ERRORS[Y_NUM] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0};  // Compensate for hardware issues
const int Y_LENGTHS[Y_NUM] = {20, 20, 22 ,20 ,24, 20, 48};  // # LEDs in ring

char xLeds[74];
char yLeds[87];

float xVectors[148][3];
float yVectors[174][3];

float thetaR;
int numColors;

// Used to precalculate LED vectors
void setLedVec(float offset, float distance, int dir, float* retVec) {

  float x, y, z;
  float smallRadius = sqrtf(SPHERE_RADIUS*SPHERE_RADIUS - offset*offset);
  float theta;

  if (dir == X_DIR) {
    theta = PI + (distance) / (1.0*smallRadius);
  }
  else {
    theta = PI - (distance) / (1.0*smallRadius);
  }

  z = -smallRadius*cos(theta);

  if (dir == X_DIR) {
    x = offset;
    y = smallRadius*sin(theta);
  }
  else {
    y = offset;
    x = smallRadius*sin(theta);
  }
  float mag = sqrt(x*x + y*y + z*z);
  retVec[0] = x / mag;
  retVec[1] = y / mag;
  retVec[2] = z / mag;

}
```

```
void delay(int milis){
        float start = clock();
        while(clock() < start + milis * 1000);
}



// Dot Product of two vectors
float dotProduct(float* vec1, float* vec2) {

  float dotProd = 0;
  int i;
  for (i = 0; i < 3; i++) {
    dotProd += vec1[i]*vec2[i];
  }
  return dotProd;

}

// Use normal vector to assign designs to orb
void setLedRings(float* normalVec, char colorSpectrum[], int spectrumLength) {
   char stop = *"~";
   float avg;

   int j = 0;
  for (int i = 0; i < X_NUM; i++) {
    for (int led = 0; led < X_LENGTHS[i]; led++) {
      if(j == 0){
              avg = dotProduct(xVectors[0], normalVec);
      }
      else{
        float distance1 = dotProduct(xVectors[j-1], normalVec);
        float distance2 = dotProduct(xVectors[j], normalVec);
        avg = (distance1 + distance2)/2;
      }
      int segment = 0.5*(avg + 1.0)*spectrumLength;

      char color = colorSpectrum[segment];
      if(!(j%2)){
        xLeds[j/2] = color;
      }
      j++;
    }
    }
    j=0;
    for (int i = 0; i < Y_NUM; i++) {
    for (int led = 0; led < Y_LENGTHS[i]; led++) {
      if(j == 0){
              avg = dotProduct(yVectors[0], normalVec);
      }
      else{
        float distance1 = dotProduct(yVectors[j-1], normalVec);
        float distance2 = dotProduct(yVectors[j], normalVec);
        avg = (distance1 + distance2)/2;
      }
      int segment = 0.5*(avg + 1.0)*spectrumLength;

      char color = colorSpectrum[segment];
      if(!(j%2)){
        yLeds[j/2] = color;
      }
      j++;
    }
  }
    for (int i = 0; i < 74; i++) {
        char received = spiSendReceive(xLeds[i]);
    }
    for (int i = 0; i < 87; i++) {
        char received = spiSendReceive(yLeds[i]);
    }
    char received = spiSendReceive(stop);
}


void main(void) {

  pioInit();
  spiInit(244000,0);
  int j = 0;

  // Precompute vectors
  for (int i = 0; i < X_NUM;i++) {
  for (int led = 0; led < X_LENGTHS[i]; led++) {
        setLedVec(X_OFFSETS[i], X_ERRORS[i] + (led - X_LENGTHS[i]/2) * LED_INTERVAL, X_DIR,
xVectors[j]);
          j++;
         }
        }
   j = 0;
   for (int i = 0; i < Y_NUM; i++) {
   for (int led = 0; led < Y_LENGTHS[i]; led++) {
         setLedVec(Y_OFFSETS[i], Y_ERRORS[i] + (led - Y_LENGTHS[i]/2) * LED_INTERVAL, Y_DIR,
yVectors[j]);
         j++;
          }
        }

  float normalVec[3] = {0,0,1};

while(1==1){
        thetaR += 5;
        normalVec[0] = sin(thetaR/100.0);
        normalVec[1] = cos(thetaR/100.0);
        normalVec[2] = 0;

        char colorSpectrum[7] = "\x66\x66\x66\x88\x88\x99\x99";
        setLedRings(normalVec, colorSpectrum, 7);
        delay(10);
  }
}
```

```
# final_project.py
# E155 - FInal Project - 2017
# ckotcherha@hmc.edu

import enum
import re
import struct
import sys
import time
import myo_raw

from subprocess import call

global LEDupdateCounter
global mode
global highlighted_mode
global num_modes
global rings
global cycles_to_ignore
global ring_index
LEDupdateCounter = 0
mode = 0
highlighted_mode = 1
num_modes = 2
rings = [1,2,3,4,5,6,7,7]
cycles_to_ignore = 0
ring_index = 0

if __name__ == '__main__':

    #handles motion data and games
    def data_IMU(quat, acc, gyro, times=[]):
        global LEDupdateCounter
        global mode
        global rings
        global cycles_to_ignore
        global ring_index
        global highlighted_mode
        frame_delay = 5
        acc_x = acc[0]
        acc_y = acc[1]
        acc_z = acc[2]
        gyro_x = gyro[0]
        gyro_y = gyro[1]
        gyro_z = gyro[2]

        #LED update delay (all game modes)
        LEDupdateCounter += 1
        if LEDupdateCounter > frame_delay: #reset count
            LEDupdateCounter = 0

        if mode == 1: #spectrum height mode
            if (acc_x<-1500):
                if LEDupdateCounter == frame_delay:
                    print('magenta')
                    call(["sudo", "../project/magenta"])
            elif (acc_x<-1000):
                if LEDupdateCounter == frame_delay:
                    print('violet')
                    call(["sudo", "../project/violet"])
            elif (acc_x<-500):
                if LEDupdateCounter == frame_delay:
                    print('blue')
                    call(["sudo", "../project/blue"])
            elif (acc_x<0):
                if LEDupdateCounter == frame_delay:
                    print('indigo')
                    call(["sudo", "../project/indigo"])
            elif (acc_x <500):
                if LEDupdateCounter == frame_delay:
                    print('green')
                    call(["sudo", "../project/green"])
            elif (acc_x <1000):
                if LEDupdateCounter == frame_delay:
                    print('yellow')
                    call(["sudo", "../project/yellow"])
            elif (acc_x <1500):
                if LEDupdateCounter == frame_delay:
                    print('orange')
                    call(["sudo", "../project/orange"])
            elif (acc_x <2054):
                if LEDupdateCounter == frame_delay:
                    print('red')
                    call(["sudo", "../project/red"])
            else:
                if LEDupdateCounter == frame_delay:
                    print('clear')
                    call(["sudo", "../project/clear"])

        if mode == 2: #ring game
            #check for victory condition, prevent movement (base case)
            if (rings[0:7] == [5,5,5,5,5,5,5]):
                call(["sudo", "../project/magenta"])
                call(["sudo", "../project/violet"])
                call(["sudo", "../project/blue"])
                call(["sudo", "../project/indigo"])
                call(["sudo", "../project/green"])
                call(["sudo", "../project/yellow"])
                call(["sudo", "../project/orange"])
                call(["sudo", "../project/red"])
                call(["sudo", "../project/clear"])
                mode = 0 #exit to menu
                #first menu option
                highlighted_mode = 1
                print('returning_to_menu')
                call(["sudo", "../project/indigo"])
                print('Spectum_Height_Mode')

            #otherwise, manipulate ring orientations
            else:
                if LEDupdateCounter == frame_delay:
                    #initial delta values
                    orientation_change = 0
                    ring_change = 0

                    #cycle counter for ignoring data
                    cycles_to_ignore -= 1
                    if cycles_to_ignore < 0: #reset count
                        cycles_to_ignore = 0


                    if cycles_to_ignore == 0: #don't check if recent detection
                        #check for pitch-twitch to shift selected ring
                        if acc_x < -2000:
                            ring_change = 1  #shift up
                            orientation_change = 0
                            cycles_to_ignore = 3
                            print('+1_up')
                        elif acc_x > 1700:
                            ring_change = -1 #shift down
                            orientation_change = 0
                            cycles_to_ignore = 3
                            print('-1_down')
                        else:
                            ring_change = 0
                            #check for roll-twitch to shift subring value
                            if gyro_x < -5400: #neg is right
                                orientation_change = -1  #shift right
                                cycles_to_ignore = 3
                                print('-1 right')
                            elif gyro_x > 3300:
                                orientation_change = 1 #shift left
                                cycles_to_ignore = 3
                                print('+1 left')
                            else:
                                orientation_change = 0


                        #Update selected ring values:
                        ring_index += ring_change #update selected ring
                        #limit ring indexing range
                        for i in range(0,len(rings)):
                            if ring_index < 0:
                                ring_index = 0
                            if ring_index > 6:
                                ring_index = 6

                        #finally, give selected ring updated value
                        rings[ring_index] += orientation_change
                        #limit subring led value range
                        if rings[ring_index] < 1:
                            rings[ring_index] = 1
                        elif rings[ring_index] > 8:
                            rings[ring_index] = 8

                        print(rings)
                        #update LED rings
                        call(["sudo", "../project/ring_game", str(rings[0]), str(rings[1]),
                        str(rings[2]), str(rings[3]), str(rings[4]), str(rings[5]),
                        str(rings[6]), str(ring_index)])



def data_Pose(p):
    global mode
    global highlighted_mode
    global num_modes
    global rings

    #MENU mode (0)
    if mode == 0:
        #determine highlighting
        if highlighted_mode == 1: #spectrum hight mode
            call(["sudo", "../project/indigo"])
            print('Spectum Height Mode')
        elif highlighted_mode == 2: #ring game mode
            call(["sudo", "../project/green"])
            print('Ring Game Mode')
        else: #shouldn't happen,
            call(["sudo", "../project/indigo"])
            highlighted_mode == 1
        #check for highlighting switch
        if p == Pose.WAVE_OUT: #highlight next mode
            highlighted_mode +=1
            if highlighted_mode > num_modes: #wrap around
                highlighted_mode = 1
        if p == Pose.WAVE_IN: #highlight previous mode
            highlighted_mode -=1
            if highlighted_mode < 1: #wrap around
                highlighted_mode = num_modes
        if p == Pose.FIST: #ENTER highlighted mode
            print('ENTER mode:',highlighted_mode)
            mode = highlighted_mode #leave menu, ENTER highlighted mode
                if mode == 2: #starting ring oreintations
                    rings = [1,3,6,2,4,8,6,7]

        #when not in menu (in a game mode), check for EXIT to menu
        elif p == Pose.WAVE_IN:
            print('EXIT_to_menu')
            mode = 0


m = MyoRaw(sys.argv[1] if len(sys.argv) >= 2 else None) #nitialize Myo object
m.connect() #connect to Myo

m.add_imu_handler(data_IMU) #initialize IMU handler, motion processing
m.add_pose_handler(data_Pose) #menu and game state control


try:
    while True:
        m.run(1)

except KeyboardInterrupt:
    pass
finally:
    m.disconnect()
    print('disconnected')
```