Real-Life Augmentation of PC Games

Justin Schmelzer, Christy Swartz

December 13, 2011

Abstract

Our project is an augmentation of PC games. PC games today require the user to be sedentary in order to play the game. Our project allows the user to have a more immersive experience by removing the need for the keyboard and mouse by detecting and translating the user's motion into computer keyboard and mouse inputs. We used various sensors on the feet, head, and a prop gun to represent these inputs. For example, accelerometers on the feet detect movement that maps to keys in the keyboard that correspond to the forward, backward, left, and right direction of the player, and the trigger on the prop gun maps to the left click of the mouse, corresponding to firing a gun in the game.

Overview

PC games would have the user sit almost motionless if the game did not require the mouse to be moved and keys to be pressed on the keyboard to play. Our project aims to give the user a more immersive experience by removing the need for the user to manipulate the keyboard and mouse. With motion-detecting hardware, it is possible to augment the current PC game system by translating the users own movements into movements in the game, completely removing the need for the use of keyboards and mice.

The system augments the PC interface by implementing the communication protocols used by PS/2 keyboards and mice in the PS/2 module, allowing the system to communicate with a PC. The Walk module will be responsible for producing the keyboard information used in the PC game Halo (e.g., the W, A, S, D keys correspond to walking forward, left, back, and right in the game), and the Look module will be responsible for producing the mouse information used in the game (e.g., forward movement of the mouse corresponds to looking up in the game).

In this system, an accelerometer on the legs will detect movement in the x, y, and z directions (i.e., forward, backward, right and left, and up). The analog voltage signals from the accelerometer, after being processed by 3 separate Analog to Digital (A/D) converters, will be converted to PS/2-compatible keyboard commands in the Walk module and sent to the PS/2 module. Two gyros on the head will act similarly as the accelerometers on the legs, except the gyros will detect rotational movement of the head (i.e., left and right, up and down). Two Analog to Digital converters will be used to process the analog signals from the gyros, and the information will be converted to PS/2-compatible keyboard commands in the Look module and sent to the PS/2 module.

Description

Halo Game Dynamics (Author: Justin)

Halo came out for the Xbox in 2001, and was the platforms killer app. It success marked the beginning of a legacy. On the surface, it looks like any other first person shooter. However, upon playing one can quickly appreciate the careful attention to detail. Anyhow, we will now discuss the basic functionality of the game as it pertains to our system.

Like most other FPS, the user can move their character with 3 degrees of freedom (forward/backwards, left/right, and jump/crouch). Additionally, the user can look around in the game by controlling the pitch and yaw of the first person perspective. There are a myriad of weapons in the game, but the user may only carry 2 at once. There are also 2 types of grenades, which the user is free to carry and use at their own discretion.

PS/2 Protocol (Author: Christy)

The PS/2 protocol involves manipulating clock and data lines to transmit data in a serial fashion. Data is sent in 11-bit frames, where the 8-bits of data are surrounded by a start bit, an odd parity bit, and an end bit. Data is transmitted with respect to a clock line, which must be delayed by 5-25ns with respect to the data line in order for the protocol to work properly. The host (the computer) and the device (the module, what we are emulating) are the two players in the PS/2 protocol. There are separate rules for device-to-host and host-to-device communication. The default sampling rate is 100 samples/second to send to the computer; however, this rate can be changed by the host.

The PS/2 protocol was chosen because the alternative, the USB protocol, is extremely difficult to implement, since it is far more complex than the PS/2 protocol. An implementation of a keyboard and mouse emulator requires only 2 manipulated lines (the data and clock lines), and runs at frequencies far below the available clock frequencies on the labkit (10-16kHz versus 27MHz on labkit), making precise timing easier to obtain.

PS/2 Modules

Description

The PS/2 modules control the communication between the computer and our augmentation by implementing the PS/2 protocol for keyboards and mice. The modules will have internal counters to provide the necessary delays for the protocol. The modules will be split up into a module for the mouse and a module for the keyboard.

Host-to-device communication (the ability to process information from the computer) was not implemented for several reasons. First of all, the rules for host-to-device communication are more difficult to implement. Second of all, after the computer has booted up, the computer very rarely attempts to communicate with the device. Because the host-to-device communication was only needed for such a short time, we used switches to flip the clock and data line connections of the computer to the labkit between the PS/2 modules and a partially disassembled PS/2 mouse and keyboard, so the mouse and keyboard could handle the boot-up communication.

Inputs and Outputs

The PS/2 Mouse module receives a 2-bit input from the accessory module and an 18-bit input from the Look module. The 2 bits from the accessory module correspond to right and left clicks, which correspond to firing and throwing grenades in the game. The first 2 bits from the Look module are the sign bits of the left and right, up and down movement of the head, and the other 16 bits are the 2s-complement (minus the top sign bit) change in position of the head in the left and right, up and down directions.

The PS/2 Keyboard module receives a 5-bit input from the Walk module, along with a 6-bit (up to 12) bit input from the Accessory module. Each of the 5 bits from the Walk module correspond to one of the following walking directions: forward, back, left, right, and jumping. The reason for having one bit correspond to each movement is because it is easier to detect change (only have to test for 1 bit to see it the person stopped walking in a specific direction). The 6 bits from the Accessory module correspond to several of up to 12 function in the game that were selected, for instance, turning on a flashlight, melee attack, etc. (see the section Halo Game Dynamics for more information). Both the PS/2 Mouse and Keyboard module output a data and clock line to the computer.

Halo Integration

Since this project is designed with Halo in mind, only 11 keyboard keys were implemented in the PS/2 Keyboard module. See the diagram below for the mapping of functions to keys:

Figure 1: Mapping of keyboard strokes to functions in the game Halo.

Testing

Various Modelsim tests were essential to find bugs in the code. The ultimate test was connecting a PS/2 keyboard and mouse cable from the labkit to the computer. Several switches and buttons corresponded to keyboard and mouse test inputs. The PS/2 Keyboard module produced text output to a PC text editor, and the PS/2 Mouse module produced cursor movement on the computer screen.

Challenges

The PS/2 protocol sounds much easier than it is to implement. Exact precision in the implementation is crucial for it to work; even the slightest error will cause the computer to ignore all inputs from your module.

There is an abundance of documentation online for the PS/2 protocol (see citations for our reference), but some aspects of the protocol are not obvious. One particular issue was that the necessity to allow the computer (the host, where the module is the device), to hold the clock line low after the module is finished transmitting. Trying to simply implement how a PS/2 mouse manipulates its line given from an oscilloscope reading is not enough to understand what the waveform should look like; there is some degree of host-to-device interaction that cannot be ignored.

Another issue with understanding the protocol was the use of the odd parity bit. Odd parity does not mean that the parity bit is 1 if the number of 1-bits in the 8-bit message is odd; in fact, odd parity means that the parity bit is 1 if there are an even number of 1-bits! If the bit is transmitted incorrectly, the computer will not receive the message, and will ask for a retransmission, which the module could not do because we did not implement host-to-device communication (see description for reasoning behind this decision).

The sampling rate became more of an issue after integration. Because the gyro is extremely sensitive, an overflow of data was being sent to the PS/2 Mouse module, causing some commands to be interpreted as random text and other keyboard commands (yes, from the mouse). By implementing a delay, the PS/2 Mouse module performed much better.

In order to move diagonally in the game, which requires the interpretation of two start codes, it was necessary to keep track of the Walk and Accessory module outputs. Flags for each of the directions of movement and each of the functions of the accessory module were raised whenever a change was detected. The flags were cycled through to send their respective codes, allowing two or more start flags to be sent in sequence, corresponding to combinational movement in the game.

Analog Signal Processing (Author: Justin)

Movement Filters (walk, strafe, and jump)

There are 3 movement filters corresponding to the 3 independent ways in which users can move. They work off the assumption that if a user makes a movement, (forward, backwards, left, right, or jump) then the acceleration waveform of the relevant axis has a very distinct shape and duration. However, there are other waveforms that arise due to the non idealities of walking that might incorrectly be interpreted as movement. The filters attempt to decide whether or not the observed acceleration actually corresponds to movement, or is a result of noise, non-idealities, or other reasons that arent movement. They also try to be resilient in the face of sensor noise, which is fairly high.

These filters are implemented as a state machine. When the labkit sees an acceleration above a certain threshold, it goes into a state where it counts while it tracks the acceleration. While counting, it continually samples the acceleration and compares it to the threshold. If at anytime while counting, the total number of samples above the threshold falls below 98% of the total samples, that signal gets discarded (interpreted as no movement) and the state machine resets. If, however, the number of samples above the threshold remains above 98% of the total samples and the state machine counts up to 0x149970 (50 milliseconds), then the state machine interprets that as a movement.

This part was very difficult, as you need to have a very good understanding of what kind of signals you are dealing with. I spent many hours in front of the oscilloscope walking back and forth trying to understand exactly what part of the signal was me walking and what part wasnt. It wasnt easy to see. Once I figured it out, I still had the problem of tuning the filter modules. There are 3 degrees of freedom in the filter modules. One has to do with the signal thresholds, the other is the 98% mentioned earlier (I played around with values ranging between 80% and 99%). The last one has to do with the length the state machine counts for. Even now, while I feel my values for the 3 degrees of freedom give pretty good operation, I am by no means confident it is close to perfect. I spent many hours fiddling around with the values trying to get the system to operate better, and Im sure I could have spent many more.

If I were to do this differently, and I had sufficient time, I would try the following approach instead. First take lots of user data to determine the kinds of acceleration signals that correspond to different movements. Then determine some kind of averaged ideal acceleration signal. Then, in the FPGA, take the observed acceleration and compute some heavy duty signal correlation metrics (in both time and frequency) between the observed and ideal signal.

Gyro Filtering

There are two independent ways in which the user can look around, pitch and yaw. In order to implement this functionality we are using a 2 axis gyro. The gyro is plagued by

the same SNR problem as the accelerometer. Initially, it was thought that since the mouse asks for dx, and dy, then it would be wise to integrate the velocity data over a particular range of motion. However, implementing that scheme proved to be too troublesome, so we decided to just send velocity data instead. This scheme performs quite well. There was some scaling that needed to be done in order to make sure there was accurate mapping from the actual user movement to the perspective movement in the game. One thing we did not figure out for a while, is that the mouse actually asks for a 9-bit 2's-compliment number for x and y movement. We initially thought it was an 8 bit magnitude with 1 sign bit. This caused some rather unintended operation, but we caught it in time. Also, just like with the movement filters, a threshold needs to be chosen such that, if the signal is inside the threshold, no movement is sent. This makes the scheme more noise insensitive.

ADC Controllers

Both the gyro and the accelerometer are multi output and analog. This necessitates the use of ADCs in order to interface with the labkit. The ADC we used was a 4 input multiplexed ADC. On a high level, what the controllers do is to tell the ADC to do a conversion on a channel, and then change the channel and ask it to do another conversion, etc... looping over all the different inputs (3 for the accelerometer and 2 for the gyro). This was pretty straightforward. There were no significant problems encountered.

Terminator

This module was initially called False detection eliminator. However, a module name that long did not compile, so this was changed to terminator. The goal of this module is to fix the following problem. If the user wears the accelerometer on their right foot (as I did throughout all the testing) and then tries to strafe left, the movement filters interpret that as left AND backwards. This is because of the non-ideal nature of movement. When you strafe left you dont actually move your foot straight left. You pick up your heel, pivot on your toe, and then sweep your foot inward. After the pivot, the foot is pointed away from the user and the resulting inward sweep looks much like a backward movement and is interpreted as such. In order to assess the viability of this module, I spent lots of time in front of the oscilloscope making the aforementioned movement. I determined that when the user strafes left, the backward movement is sent around 50ms before the left movement is sent. Given this information, I decided to make the terminator module do the following. When a backwards command is received, hold that command for up to 50ms. If no left command is sent within that 50ms window, then the actual intention of the user was probably to go backwards, so output backwards. If, however, a left command was seen within the 50ms window, then the user was most likely just strafing left, so output only left.

Demonstration Pictures

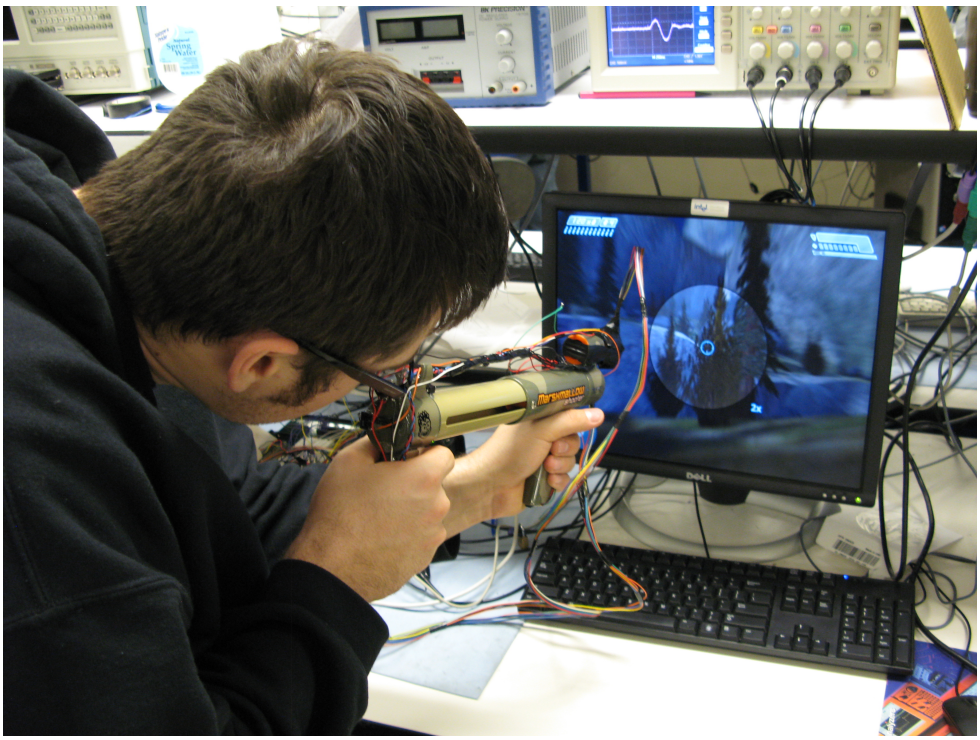

Figure 2: Justin playing Halo with the prop gun. One of the push buttons on the prop gun corresponded to the key with the function of scope zoom on a computer keyboard.

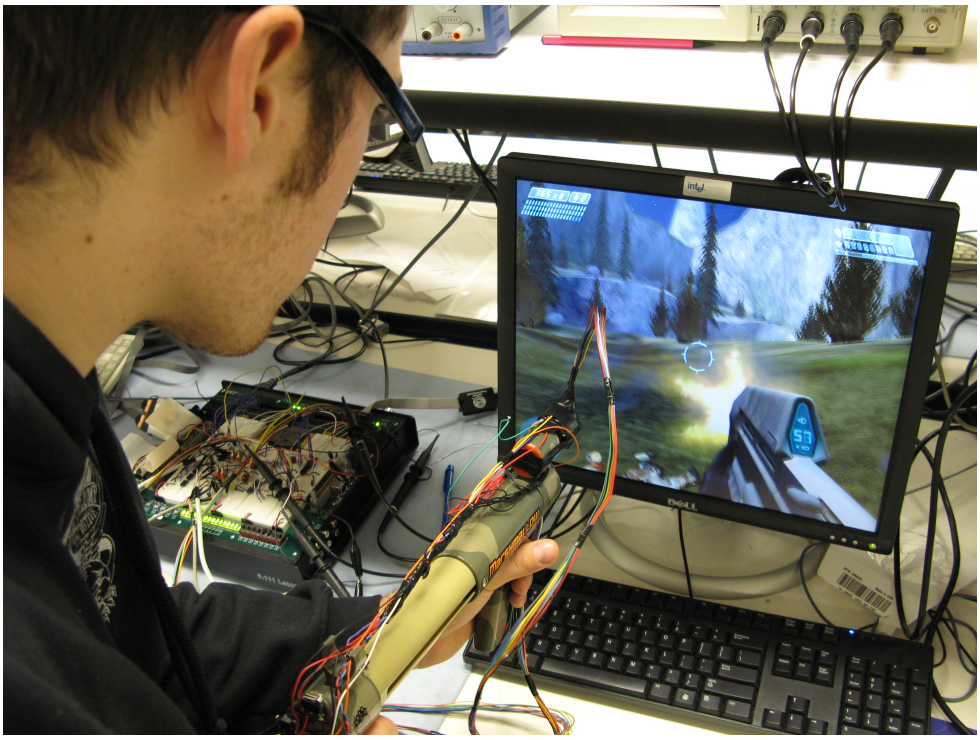

Figure 3: Justin playing Halo with the prop gun. One of the push buttons on the prop gun corresponded to a left click, mapping to the function of firing the gun on a computer mouse.

Oscilloscope Readings

A number of oscilloscope shots of the analog data from the accelerometers and gyros demonstrate the difficulty of processing the signals. As you can see, some of these signals do not seem to be very difficult from each other, which is why such complex analog and digital processing was needed.

Gyro and Accelerometer Readings

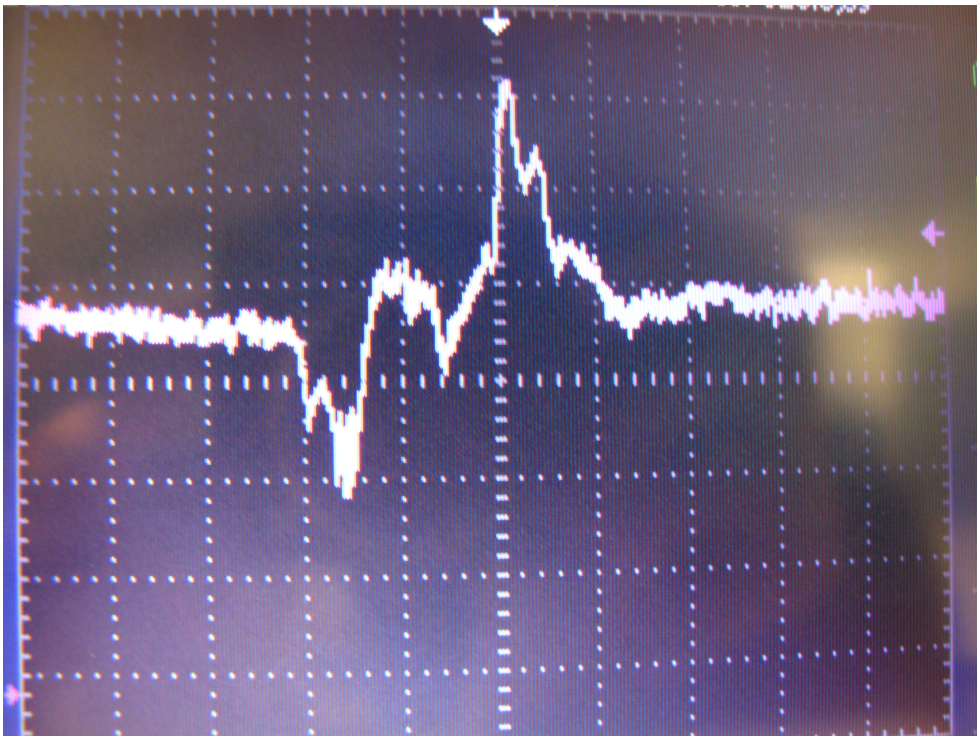

Figure 4: Moving right, example 1

PS/2 Modules

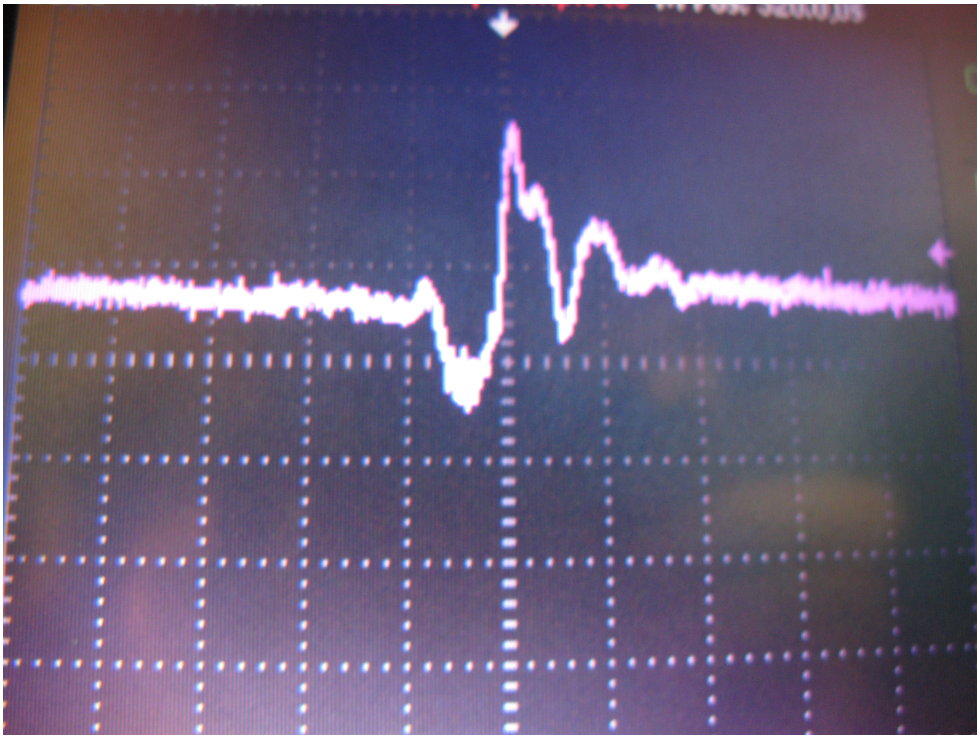

Figure 5: Moving right, example 2

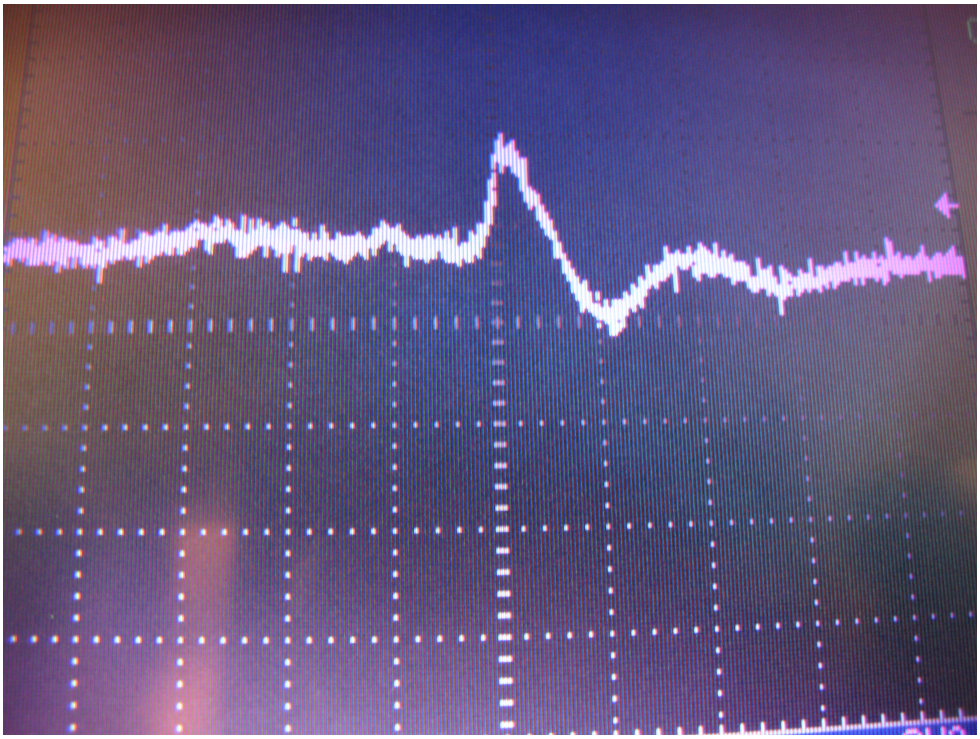

Figure 6: Moving left, example 1

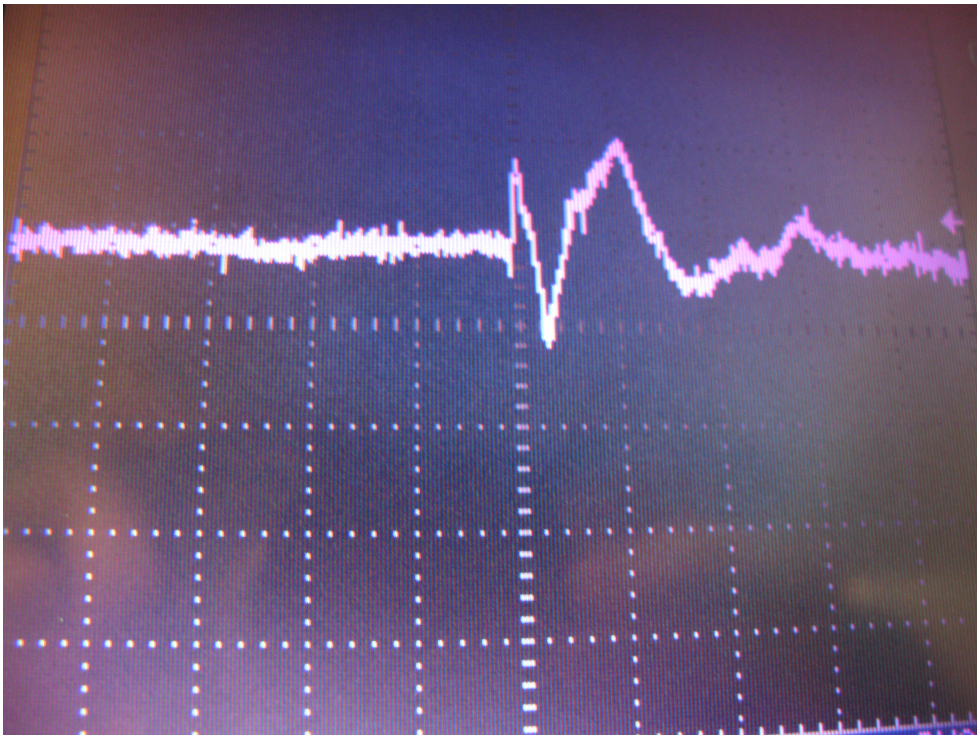

Figure 7: Moving left, example 2

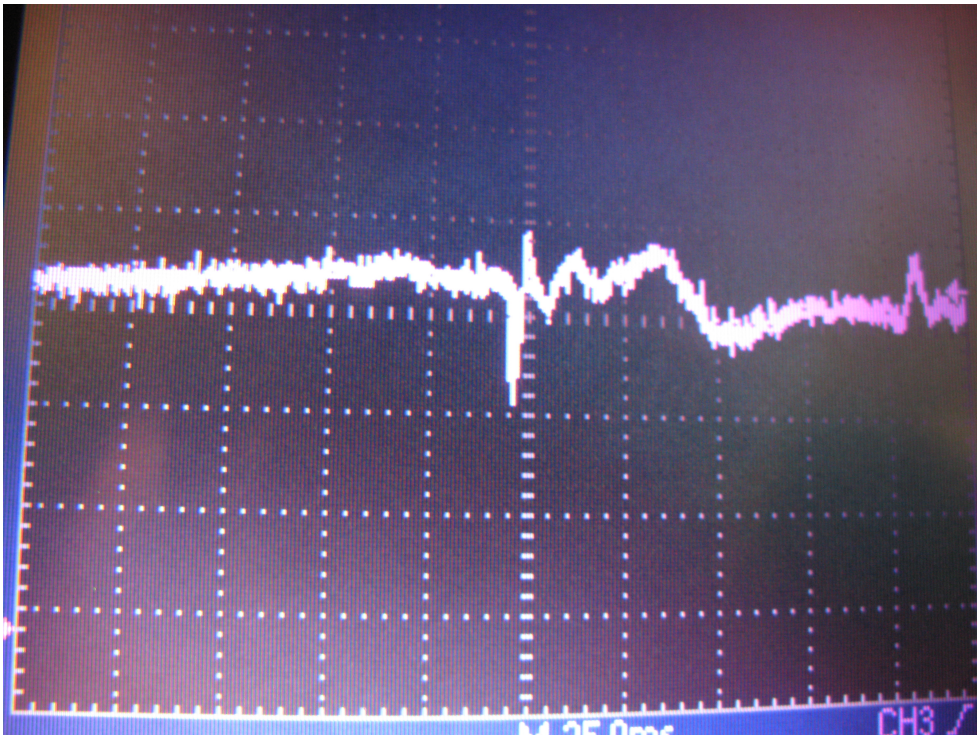

Figure 8: Moving backward, example 1

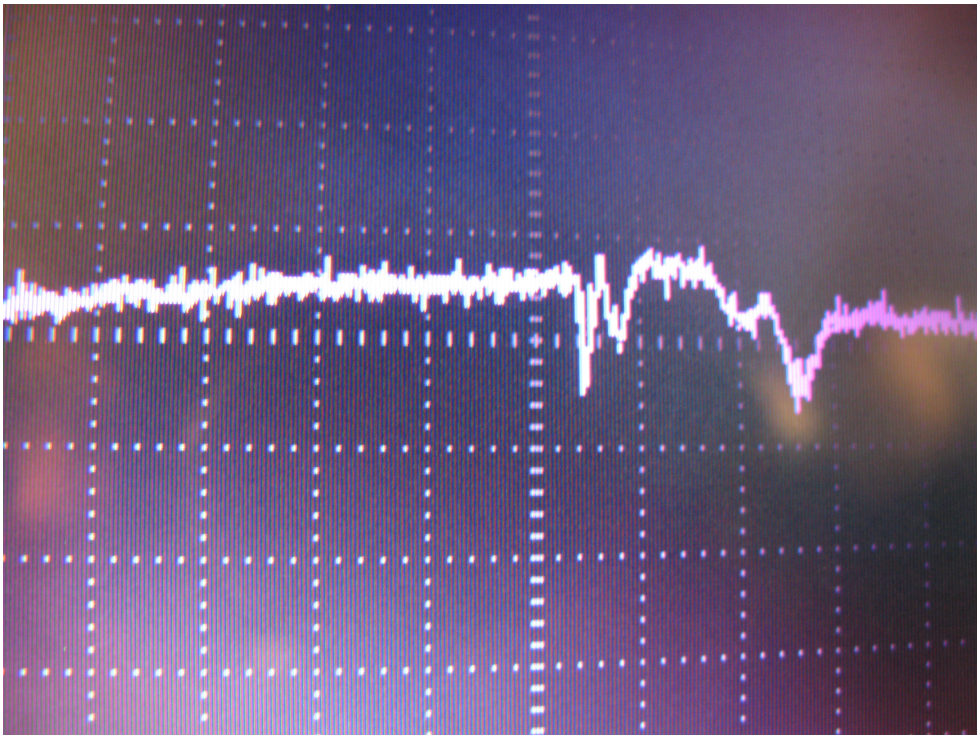

Figure 9: Moving backward, example 2

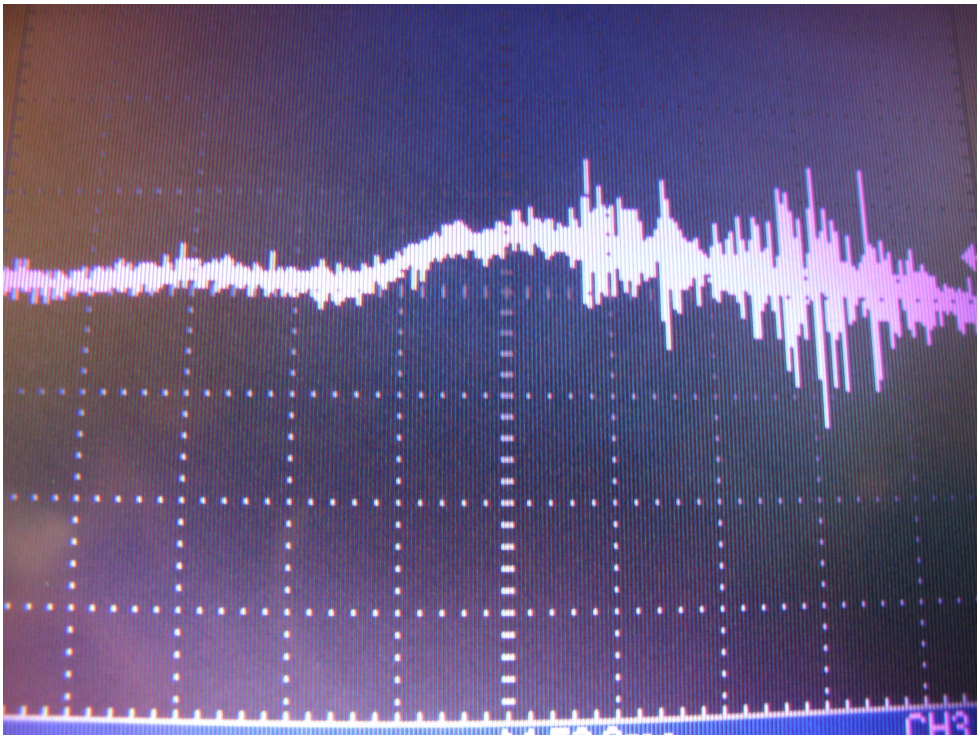

Figure 10: Moving forward, example 1

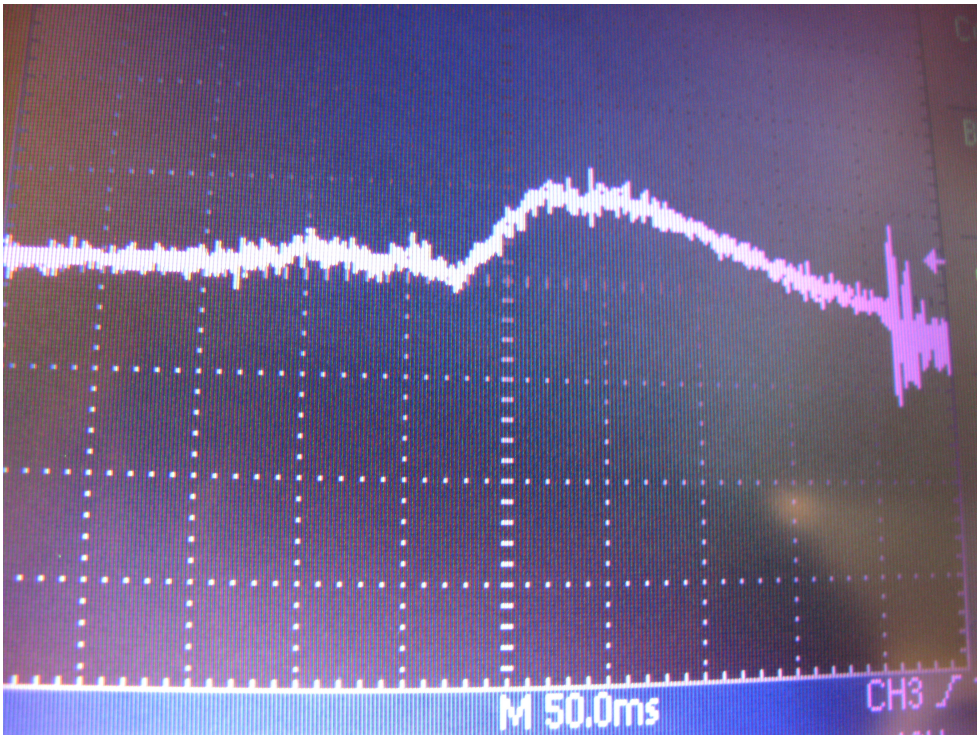

Figure 11: Moving forward, example 2

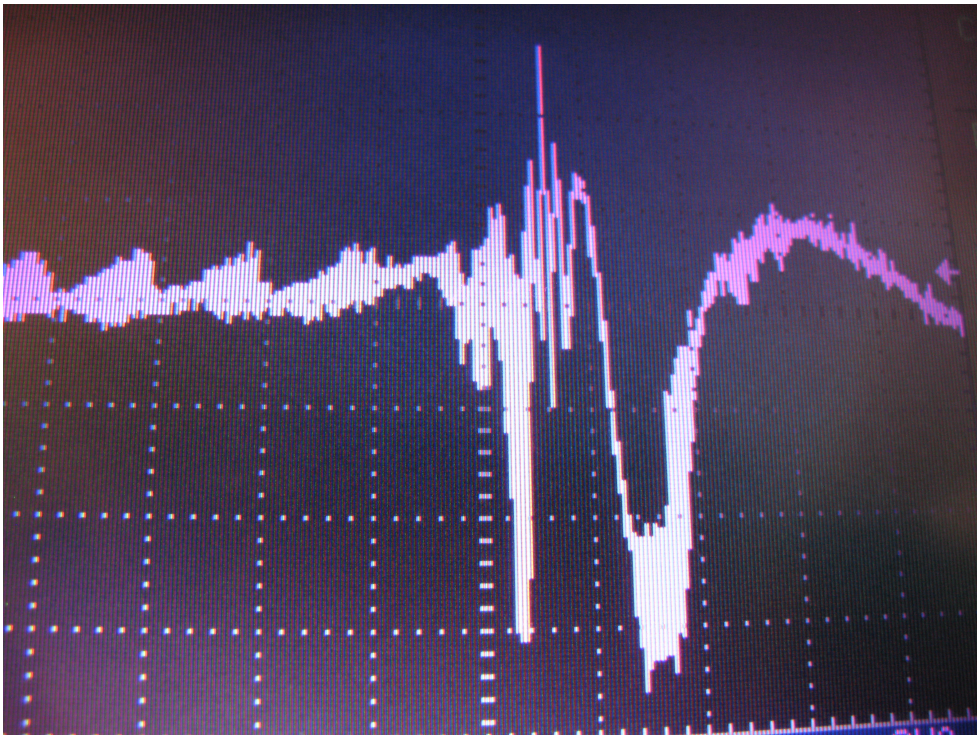

Figure 12: Jumping, example 1

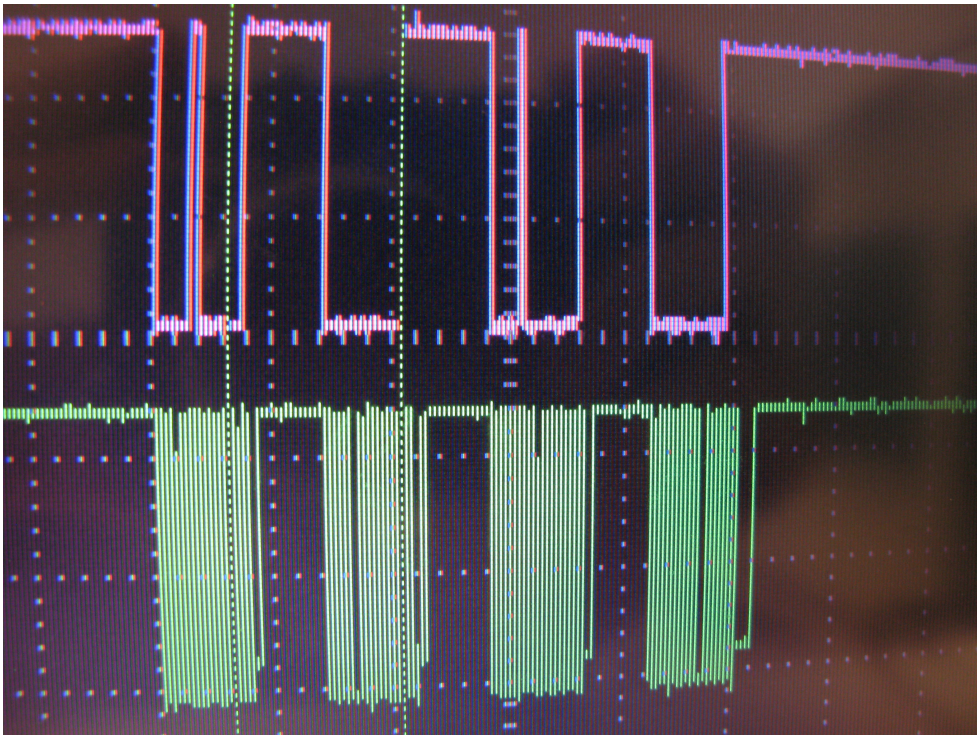

Figure 13: An example of a PS/2 Mouse data and clock transmission (data is the top signal). A PS/2 mouse transmits 4 bytes at a time.

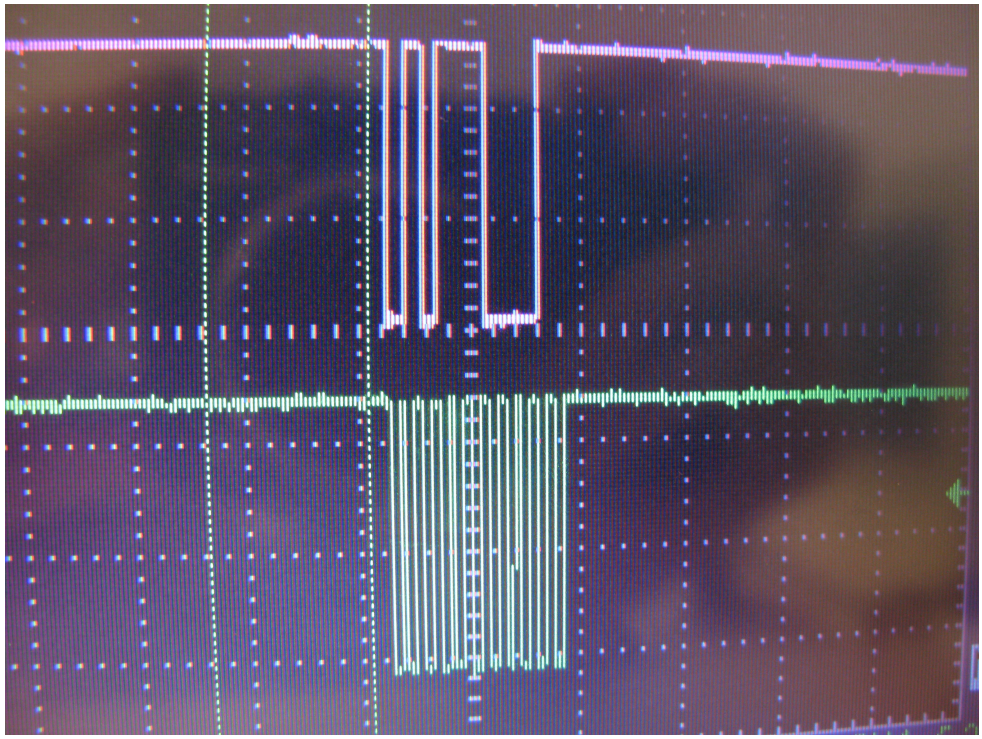

Figure 14: An example of a PS/2 Keyboard data and clock transmission (data is the top signal). A PS/2 keyboard transmits 1 byte at a time.

Block Diagram

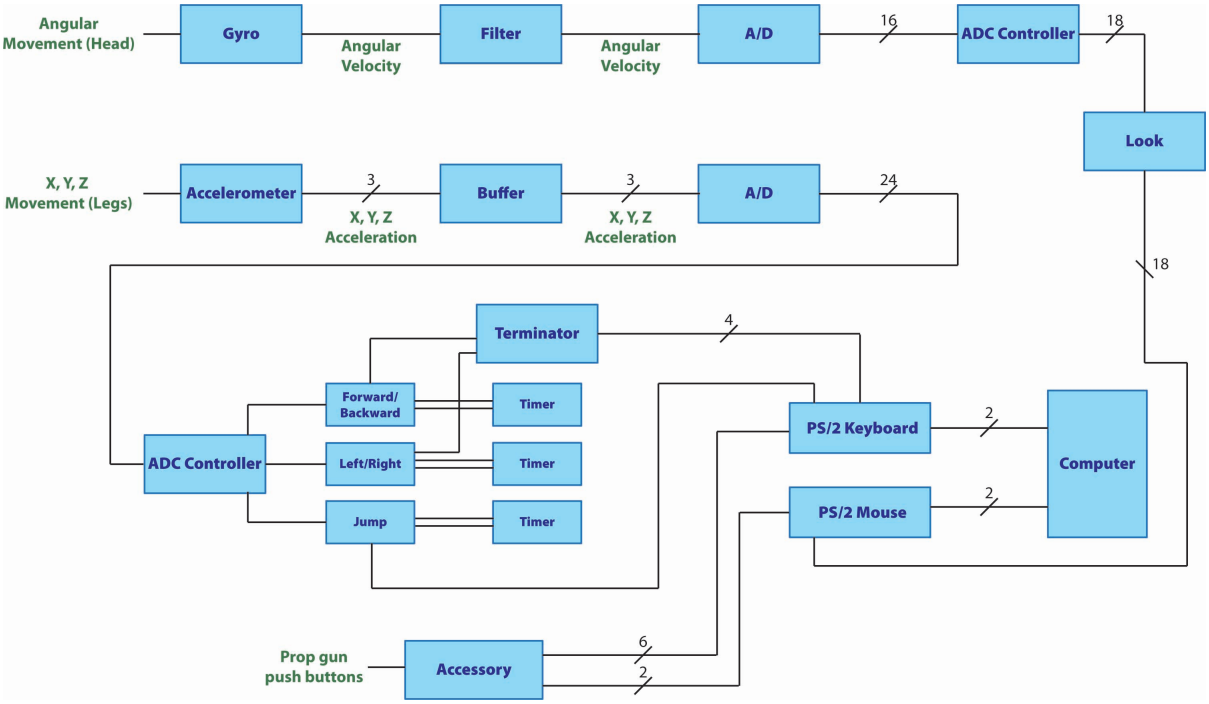

Figure 15: The high-level flow diagram of the entire project.

Conclusion

From here, the project could take many directions. In the beginning stages, it was proposed that a LCD screen be intergrated into a helmet of some sort, and the entire setup made wireless, so the user would be completely free of the computer. Given that there is nothing quite like this, this project could potentially integrated into, and possibly regenerate, the world of arcade games today. Most games are accessible from a computer or XBox. While portable, these game systems do not have a very immersive experience, since that would make them more expensive. If a truly thrilling experience existed at the arcade, users would be more inclined to leave their dark rooms and enter the real world of gaming.

Materials

Accelerometer (1): 3-axis (Dimension Engineering, DE-ACCM3D)
Gyro (1): 2-axis, from Sparkfun.com (STMicroelectronics LPR530AL)
Analog to Digital converters (5): Analog Devices, AD7824
7 push-buttons
1 prop gun (Marshmallow Shooter)

References

Chapweske, A. (2003, May 03). *The PS/2 Mouse/Keyboard protocol.* Retrieved from http://www.computer-engineering.org/ps2protocol/

Code Appendix

0.1 ADC_Controller

```
module ADC_Controller(
    output reg inv_RD,
    output reg inv_CS,
     output reg [1:0] addr,
     output reg [7:0] ch1_out, ch2_out, ch3_out,
    input inv_interrupt,
     input clk,
     input [7:0] data_in
    );

     parameter idle = 0;
     parameter kick_ch1 = 1;
     parameter have_ch1 = 2;
     parameter kick_ch2 = 3;
     parameter have_ch2 = 4;
     parameter kick_ch3 = 5;
     parameter have_ch3 = 6;

     reg [2:0] state,next_state;

     always @(*) begin
        case (state)
        idle: next_state = kick_ch1;
        kick_ch1: if (!inv_interrupt) next_state = have_ch1;
                    else next_state = kick_ch1;
        have_ch1: next_state = kick_ch2;

        kick_ch2: if (!inv_interrupt) next_state = have_ch2;
                    else next_state = kick_ch2;
```

```
   have_ch2: next_state = kick_ch3;

   kick_ch3: if (!inv_interrupt) next_state = have_ch3;
                else next_state = kick_ch3;
   have_ch3: next_state = idle;
   endcase
end

always @(posedge clk) state <= next_state;

always @(posedge clk) begin
   if (state == have_ch1) begin
       ch1_out <= data_in;
       inv_RD <= 1;
       inv_CS <= 1;
   end
   else if (state == have_ch2) begin
       ch2_out <= data_in;
       inv_RD <= 1;
       inv_CS <= 1;
   end
   else if (state == have_ch3) begin
           ch3_out <= data_in;
           inv_RD <= 1;
           inv_CS <= 1;
   end
   else if (state == kick_ch1) begin
       inv_RD <= 0;
       inv_CS <= 0;
       addr <= 2'b00;
   end
   else if (state == kick_ch2) begin
       inv_RD <= 0;
       inv_CS <= 0;
       addr <= 2'b01;
   end
   else if (state == kick_ch3) begin
       inv_RD <= 0;
       inv_CS <= 0;
       addr <= 2'b10;
   end
```

```
     end


endmodule
```

0.2 ADC_Controller2

```
module ADC_Controller2(
    output reg inv_RD,
    output reg inv_CS,
    output reg [1:0] addr,
    output reg [7:0] ch1_out,
    output reg [7:0] ch2_out,
    input inv_interrupt,
    input clk,
    input [7:0] data_in
    );

     parameter idle = 0;
     parameter kick_ch1 = 1;
     parameter have_ch1 = 2;
     parameter kick_ch2 = 3;
     parameter have_ch2 = 4;

     reg [2:0] state,next_state;

     always @(*) begin
        case (state)
        idle: next_state = kick_ch1;
        kick_ch1: if (!inv_interrupt) next_state = have_ch1;
                     else next_state = kick_ch1;
        have_ch1: next_state = kick_ch2;

        kick_ch2: if (!inv_interrupt) next_state = have_ch2;
                     else next_state = kick_ch2;
        have_ch2: next_state = idle;
        endcase
     end

     always @(posedge clk) state <= next_state;
```

```
     always @(posedge clk) begin
        if (state == have_ch1) begin
            ch1_out <= data_in;
            inv_RD <= 1;
            inv_CS <= 1;
        end
        else if (state == have_ch2) begin
            ch2_out <= data_in;
            inv_RD <= 1;
            inv_CS <= 1;
        end
        else if (state == kick_ch1) begin
            inv_RD <= 0;
            inv_CS <= 0;
            addr <= 2'b00;
        end
        else if (state == kick_ch2) begin
            inv_RD <= 0;
            inv_CS <= 0;
            addr <= 2'b01;
        end
     end

endmodule
```

0.3 Terminator

```
module terminator(
    input [1:0] y_dir_in,
    input [1:0] x_dir_in,
    input clk,
    output reg [1:0] y_dir_out,
    output reg [1:0] x_dir_out
    );

    //version 1.0
     //because of the nuances of moving, strafing left
     //is interpreted as left+back
     //this module attempts to fix that
```

```
//states
parameter idle = 0;
 parameter counting = 1;
 parameter fixing = 2;

 parameter window = 1350000;        //50ms for now

 reg [1:0] state, next_state;
 reg [20:0] count;

 //main updating block
always @(posedge clk) begin
    state <= next_state;
    if (state == counting) count <= count + 1;
    else if (state != counting) count <= 0;
 end

//state xsition diagram
always @(*) begin
    case (state)
        idle: if (y_dir_in == 2'b10) next_state = counting; //if moving backward
                 else next_state = idle;
        counting: begin
                    if (count >= window) next_state = idle; //left was not received wit
                    else if (x_dir_in == 2'b01) next_state = fixing; //left was receive
                    else next_state = counting;
                  end
        fixing: if (x_dir_in == 2'b00) next_state = idle;
                    else next_state = fixing;
    endcase
 end

 //figure out what output should be
 always @(posedge clk) begin
    if (state == idle) begin
        y_dir_out <= y_dir_in;
        x_dir_out <= x_dir_in;
    end
    if (state == fixing) begin
        y_dir_out <= 2'b00;
```

```
            x_dir_out <= 2'b01;
        end
     end
endmodule
```

0.4 y_filter

```
module y_filter(
    input [7:0] y_in,
    input clk,
    output reg [1:0] cmd_out,
     output reg start_timer,
     input expire,
     input calibrate,
     output reg[7:0] fwrd_thr,
     output reg [7:0] bkwd_thr
     );

    reg [7:0] forward_threshold = 8'h28;
    reg [7:0] backward_threshold = 8'h22;

     //output the threshold
     always @(*) begin
        fwrd_thr = forward_threshold;
        bkwd_thr = backward_threshold;
     end

     //pressing calibrate while not moving will refigure f/b thresholds
     always @(*) begin
        if (calibrate) begin
            forward_threshold = y_in + 5;
            backward_threshold = y_in - 6;
        end
     end

    parameter idle = 0;
    parameter counting_f = 1;
    parameter counting_b = 2;
    parameter forward = 3;
    parameter backward = 4;
```

```
reg [20:0] count, good;
reg [2:0] state, next_state;

 wire [27:0] count_pct, good_pct;
 assign count_pct = count*98;
 assign good_pct = good*100;

 //state transition diagram
always @(*) begin
if (calibrate) next_state = idle;
else case (state)
    idle: begin
            if (y_in > forward_threshold)
                next_state = counting_f;
            else if (y_in < backward_threshold)
                next_state = counting_b;
            else next_state = idle;
          end
counting_f: begin
                if ((count > 100) && (count_pct > good_pct)) next_state = idle;
                else if ((count >= 22'h149970) && (count_pct < good_pct))
                    next_state = forward;
                //else if ((count >= 1000) && (count_pct < good_pct))
                    next_state = forward;
                else next_state = counting_f;
            end
 counting_b: begin
                if ((count > 100) && (count_pct > good_pct))
                    next_state = idle;
                else if ((count >= 22'h149970) && (count_pct < good_pct))
                    next_state = backward;
                //else if ((count >= 1000) && (count_pct < good_pct))
                    next_state = backward;
                else next_state = counting_b;
                 end
        forward: if (expire) next_state = idle;
                     else next_state = forward;
        backward: if (expire) next_state = idle;
                  else next_state = backward;
        default: next_state = idle;
```

```
        endcase
    end

     //update state, count, and good
     always @(posedge clk) begin
        state <= next_state;
        if ((state == counting_f) || (state == counting_b)) count <= count + 1;
        //if we are in either counting state, increment count

       if ((state == counting_f) && (y_in > forward_threshold)) good <= good + 1;
       //if we are in either counting state, and y_in

        else if ((state == counting_b) && (y_in < backward_threshold)) good <= good + 1;
        //is above/below threshold, then increment good

        if (state == idle) begin //if in idle, reset count and good
            count <= 0;
            good <= 0;
        end
    end

     //kick timer on transition into forward or backward
     always @(*) begin
        if (state != forward && next_state == forward) start_timer = 1;
        else if (state != backward && next_state == backward) start_timer = 1;
        else start_timer = 0;
    end

     //decide what cmd output is depending on state
     always @(*) begin
        if (state == forward) cmd_out = 1;
        else if (state == backward) cmd_out = 2;
        else cmd_out = 0;
    end
endmodule
```

0.5 Gyro_filter

```
module Gyro_filter(
    input [7:0] adc_yaw_in,
```

```
input [7:0] adc_pitch_in,
input clk,
 input pause,
output reg [7:0] yaw_out,
output reg [7:0] pitch_out,
 output reg [7:0] dx,
 output reg [7:0] dy,
 output reg sx,
 output reg sy
);

reg [7:0] dx_int, dy_int;  //intermediate registers used for computing actual dx/dy
reg [19:0] acc_yaw;
 reg [19:0] next_acc_yaw;
 reg [19:0] acc_pitch;
 reg [19:0] next_acc_pitch;
 reg [11:0] count;
 reg [11:0] next_count;
 reg [1:0] state, next_state;

 parameter idle = 0;
 parameter counting = 1;
 parameter done = 2;

 parameter yaw_ref = 8'h75;
 parameter pitch_ref = 8'h7B;

 //update state,count,accumulators, and some outputs
 always @(posedge clk) begin
    state <= next_state;
    count <= next_count;
    acc_yaw <= next_acc_yaw;
    acc_pitch <= next_acc_pitch;

    if (next_state == idle) begin
        yaw_out <= acc_yaw[19:12];
        pitch_out <= acc_pitch[19:12];
    end
 end

 //figure out dx,dy,sx,sy
```

```
always @(posedge clk) begin
if (pause) begin
       dx <= 0;
       dy <= 0;
       sx <= 0;
       sy <= 0;
end
 else begin
   if (yaw_out > yaw_ref) begin
       dx_int <= (yaw_out - yaw_ref);
       sx <= 0;
       if (dx_int > 8'h08) dx <= dx_int/4;
       else dx <= 0;
   end
   else if (yaw_out < yaw_ref) begin
       dx_int <= (yaw_ref - yaw_out);
       sx <= 0;
       if (dx_int > 8'h08) begin
           dx <= ~(dx_int/4)-2;
           sx <= ~sx;
       end
       else dx <= 0;
   end
   else begin
       dx_int <= 0;
       sx <= 0;
   end

   if (pitch_out > pitch_ref) begin
       dy_int <= (pitch_out - pitch_ref);
       sy <= 0;
       if (dy_int > 8'h08) dy <= dy_int/4+2;
       else dy <= 0;
   end
   else if (pitch_out < pitch_ref) begin
       dy_int <= (pitch_ref - pitch_out);
       sy <= 0;
       if (dy_int > 8'h08) begin
           dy <= ~(dy_int/4)-2;
           sy <= ~sy;
       end
```

```
            else dy <= 0;
        end
        else begin
            dy_int <= 0;
            sy <= 0;
        end
     end
    end

    //main xition loop
    always @(*) begin
        case (state)
            idle: begin
                        next_state = counting;
                        next_count = 0;
                        next_acc_yaw = 0;
                        next_acc_pitch = 0;
                    end
            counting: begin
                            next_count = count + 1;
                            next_acc_yaw = acc_yaw + adc_yaw_in;
                            next_acc_pitch = acc_pitch + adc_pitch_in;
                           if (count == 4095) next_state = done;
                            else next_state = counting;
                          end
            done: next_state = idle;

        endcase
    end

endmodule
```

0.6 horizontal_filter

```
module horizontal_filter(
    input [7:0] x_in,
    input clk,
    output reg [1:0] cmd_out,
    output reg start_timer,
    input expire,
```

```
input calibrate,
output reg [7:0] left_thr,
output reg [7:0] right_thr
);

 reg [7:0] left_threshold = 8'h30;
reg [7:0] right_threshold = 8'h22;

 //output the threshold
 always @(*) begin
    left_thr = left_threshold;
    right_thr = right_threshold;
 end

 //pressing calibrate while not moving will refigure f/b thresholds
 always @(*) begin
    if (calibrate) begin
        left_threshold = x_in - 3;
        right_threshold = x_in + 12;
    end
 end

parameter idle = 0;
parameter counting_l = 1;
parameter counting_r = 2;
parameter left = 3;
parameter right = 4;

reg [20:0] count, good;
reg [2:0] state, next_state;

 wire [27:0] count_pct, good_pct;
 assign count_pct = count*98;
 assign good_pct = good*100;

 //state transition diagram
 always @(*) begin
   if (calibrate) next_state = idle;
    else case (state)
        idle: begin
                    if (x_in < left_threshold) next_state = counting_l;
```

```
                    else if (x_in > right_threshold) next_state = counting_r;
                    else next_state = idle;
                end
        counting_l: begin
                        if ((count > 100) && (count_pct > good_pct))
                            next_state = idle;
                        else if ((count >= 22'h149970) && (count_pct < good_pct))
                            next_state = left;
                        //else if ((count >= 1000) && (count_pct < good_pct))
                            next_state = left;
                        else next_state = counting_l;
                    end
     counting_r: begin
                        if ((count > 100) && (count_pct > good_pct))
                            next_state = idle;
                        else if ((count >= 22'h149970) && (count_pct < good_pct))
                            next_state = right;
                        //else if ((count >= 1000) && (count_pct < good_pct))
                            next_state = right;
                        else next_state = counting_r;
                    end
        left: if (expire) next_state = idle;
                     else next_state = left;
        right: if (expire) next_state = idle;
                  else next_state = right;
        default: next_state = idle;
    endcase
end

 //update state, count, and good
 always @(posedge clk) begin
    state <= next_state;
    if ((state == counting_l) || (state == counting_r)) count <= count + 1;
    //if we are in either counting state, increment count

   if ((state == counting_l) && (x_in < left_threshold)) good <= good + 1;
   //if we are in either counting state, and y_in

    else if ((state == counting_r) && (x_in > right_threshold)) good <= good + 1;
     //is above/below threshold, then increment good
```

```
        if (state == idle) begin //if in idle, reset count and good
            count <= 0;
            good <= 0;
        end
    end

     //kick timer on transition into forward or backward
     always @(*) begin
        if (state != left && next_state == left) start_timer = 1;
        else if (state != right && next_state == right) start_timer = 1;
        else start_timer = 0;
    end

     //decide what cmd output is depending on state
     always @(*) begin
        if (state == right) cmd_out = 2;
        else if (state == left) cmd_out = 1;
        else cmd_out = 0;
    end
endmodule
```

0.7 vertical_filter

```
module vertical_filter(
    input [7:0] z_in,
    input clk,
    output jump_out,
    input calibrate,
     input expire,
     output reg start_timer,
     output reg [7:0] jump_thr
    );

     reg [7:0] jump_threshold = 8'h42;

    //output the threshold
     always @(*) begin
        jump_thr = jump_threshold;
    end
```

```
 //pressing calibrate while not moving will refigure f/b thresholds
 always @(*) begin
    if (calibrate) begin
        jump_threshold = z_in + 48;
    end
 end

parameter idle = 0;
parameter counting_j = 1;
parameter jump = 2;

reg [20:0] count, good;
reg [2:0] state, next_state;

 wire [27:0] count_pct, good_pct;
 assign count_pct = count*98;
 assign good_pct = good*100;

 //state transition diagram
 always @(*) begin
   if (calibrate) next_state = idle;
    else case (state)
        idle: begin
                    if (z_in > jump_threshold) next_state = counting_j;
                    else next_state = idle;
                end
        counting_j: begin
                        if ((count > 100) && (count_pct > good_pct))
                            next_state = idle;
                        else if ((count >= 22'h5265C) && (count_pct < good_pct))
                            next_state = jump;
                        //else if ((count >= 1000) && (count_pct < good_pct))
                            next_state = jump;
                        else next_state = counting_j;
                    end
        jump: if (expire) next_state = idle;
                     else next_state = jump;
        default: next_state = idle;
    endcase
end
```

```
     //update state, count, and good
     always @(posedge clk) begin
        state <= next_state;
        if (state == counting_j) count <= count + 1;
            //if we are in either counting state, increment count
       if ((state == counting_j) && (z_in > jump_threshold)) good <= good + 1;
        if (state == idle) begin //if in idle, reset count and good
            count <= 0;
            good <= 0;
        end
    end

     //kick timer on transition into forward or backward
     always @(*) begin
        if (state != jump && next_state == jump) start_timer = 1;
        else start_timer = 0;
    end

     //decide what cmd output is depending on state
     assign jump_out = (state == jump);

endmodule
```

0.8 timer

```
module Timer(
    input start,
     output expire,
     input clock,
     input reset
    );

    reg [25:0] count, next_count = 0;
     reg [1:0] state, next_state;

     parameter idle = 0;
     parameter counting = 1;
     parameter expired = 2;
```

```
    always @(*) begin
       if (reset) begin
           next_state = 0;
           next_count = 0;
       end
       else case (state)
           idle: if (start) next_state = counting;
           counting: begin
                           if (count == 23000000) begin
                           //if (count == 18) begin
                               next_state = expired;
                               next_count = 0;
                           end
                           else    begin
                               next_state = counting;
                               next_count = count + 1;
                           end
                       end
           expired: next_state = idle;
           default: next_state = idle;
       endcase
    end

    always @(posedge clock) begin
       state <= next_state;
       count <= next_count;
    end

    assign expire = (state == expired);
endmodule
```

0.9 PS/2 Mouse

```
module ps2mouse(
    input clock_27MHz,
    input [7:0] lookdx,
    input [7:0] lookdy,
    input [3:0] look_extra,
   input reset,
   input ps2_clock_input,
```

```
 output reg data,
 output reg ps2_clock_delay
  );

  //send codes
  parameter extra_default = 4'b1000;//4th bit always 1 in protocol

//used to coordinate sending of bits
 reg [7:0] counter = 0;
  reg [3:0] buf_counter = 0;

  //to record previous walk, acc information
  reg [7:0] lookdx_delay = 0;
  reg [7:0] lookdy_delay = 0;
 reg [3:0] look_extra_delay = 0;

  //flags to send a message
  reg send_look = 0;
 reg send_extra = 0;
 reg sent_left = 0;
 reg sent_right = 0;

  //send buffer -- place to put send codes
  reg [7:0] send_buf_extra = extra_default;
  reg [7:0] send_bufx = 0;
  reg [7:0] send_bufy = 0;
  reg [7:0] send_bufz = 0;
  reg [7:0] send_buf = 0;

  //check to see if currently sending message
  reg sending = 0;

 wire bit_clock; //13.5kHz clock

 counter #(1000) c1(
  .clock(clock_27MHz),
  .reset(reset),
  .enable(bit_clock)
 );

 reg timer_start = 0;
```

```
 reg timer_start_delay = 0;
 reg timer_start_signal = 0;
 reg [15:0] timer_value = 0;
 wire [15:0] count;
 wire timer_expire;
 reg timer_expired = 0;

//used to delay clock pulse
timer_christy t1(
  .clock_27mhz(clock_27MHz),
  .reset(reset),
  .enable(bit_clock),//enable on bit_clock to cut down time
  .start(timer_start_signal),//start pulse
  .value(timer_value),   // timer values from 0-15
  .count(count),
  .expired(timer_expire)  // pulse high when timer value is reached
  );

 reg [270:0] sync = 0;//shift register for clock delay
 reg ps2_clock = 0;


 reg timer_sample_start = 0;
 reg timer_sample_start_delay = 0;
 reg timer_sample_start_signal = 0;
 reg [15:0] timer_sample_value = 0;
 wire [15:0] sample_count;
 wire timer_sample_expire;
 reg timer_sample_expired = 0;

//used to delay clock pulse
timer_christy tsample(
  .clock_27mhz(clock_27MHz),
  .reset(reset),
  .enable(bit_clock),//enable on bit_clock to cut down time
  .start(timer_sample_start_signal),//start pulse
  .value(timer_sample_value),   // timer values from 0-15
  .count(sample_count),
  .expired(timer_sample_expire)  // pulse high when timer value is reached
  );
```

```
always @(posedge clock_27MHz) begin
  //pulse-maker for after-sending delay
   timer_start_delay <= timer_start;
   timer_start_signal <= (!timer_start_delay && timer_start);

   //pulse-maker for sample rate
   timer_sample_start_delay <= timer_sample_start;
   timer_sample_start_signal <= (!timer_sample_start_delay && timer_sample_start);

   //shift registers to delay clock by 5-25us
  {ps2_clock_delay,sync} <= {sync[270:0],ps2_clock};

   if (timer_expire) begin
    timer_expired <= 1;
   end

   if (timer_sample_expire) begin
    timer_sample_expired <= 1;
   end

 if (!send_look && bit_clock) begin
   look_extra_delay <= look_extra[1:0];

   //flags to tell if any data must be sent (any input has changed)
   if ((lookdx != 0) || (lookdy != 0) || (look_extra_delay != look_extra[1:0])) begin
      send_look <= 1;
      send_buf_extra <= {look_extra[3:2],2'b10,look_extra[1:0]};//anded with flag
      send_bufx <= lookdx;
      send_bufy <= lookdy;
   end

    ps2_clock <= 1;
    data <= 1;

    $display("not sending");
  end //not sending

    //if currently sending a message
    else begin
    if (bit_clock) begin
      //11-bit frame sending sequence
```

```
$display("in sending sequence");
case(counter)
        0: begin
      ps2_clock <= 0;
      data <= 0;
      counter <= 1;
           end
  1: begin
      ps2_clock <= 1;
      counter <= 2;
    end
        2: begin
              ps2_clock <= 0;
              data <= send_buf[0];
              counter <= 3;
           end
  3: begin
      ps2_clock <= 1;
      counter <= 4;
    end
        4: begin
              ps2_clock <= 0;
              data <= send_buf[1];
              counter <= 5;
           end
  5: begin
      ps2_clock <= 1;
      counter <= 6;
    end
        6: begin
              ps2_clock <= 0;
              data <= send_buf[2];
              counter <= 7;
           end
  7: begin
      ps2_clock <= 1;
      counter <= 8;
    end
        8: begin
              ps2_clock <= 0;
              data <= send_buf[3];
```

```
            counter <= 9;
         end
9: begin
    ps2_clock <= 1;
    counter <= 10;
  end
      10: begin
            ps2_clock <= 0;
            data <= send_buf[4];
            counter <= 11;
         end
11: begin
    ps2_clock <= 1;
    counter <= 12;
  end
      12: begin
            ps2_clock <= 0;
            data <= send_buf[5];
            counter <= 13;
         end
13: begin
    ps2_clock <= 1;
    counter <= 14;
  end
      14: begin
            ps2_clock <= 0;
            data <= send_buf[6];
            counter <= 15;
         end
15: begin
    ps2_clock <= 1;
    counter <= 16;
  end
      16: begin
            ps2_clock <= 0;
            data <= send_buf[7];
            counter <= 17;
         end
17: begin
    ps2_clock <= 1;
    counter <= 18;
```

```
      end
          18: begin
                ps2_clock <= 0;
                data <= !(^send_buf); //xor, parity bit
                counter <= 19;
              end
    19: begin
        ps2_clock <= 1;
        counter <= 20;
      end
          20: begin $display("in counter, case 20");
                ps2_clock <= 0;
        data <= 0;//end bit
                counter <= 21;
              end
    21: begin $display("in counter, case 21");
        ps2_clock <= 1;
        data <= 1;
        timer_value <= 15;//clock high-time
        if (timer_expired && !timer_sample_start) begin//if timer over
          $display("in counter, case 21, if stmt");
          counter <= 0;
          timer_expired <= 0;
          timer_start <= 0;
        end
        else timer_start <= 1;
      end
          default: $display("error in counter");
      endcase

//used to coordinate sending 4 bytes of information
case(buf_counter)
    0: begin
         if (counter == 0) begin
            send_buf <= send_buf_extra;
            buf_counter <= 1;
         end
      end
    1: begin $display("in buf counter, case 1");
         if (counter == 21 && timer_expired) begin
            send_buf_extra <= extra_default;
```

```
                send_buf <= send_bufx;
                buf_counter <= 2;
             end
          end
        2: begin $display("in buf counter, case 2");
             if (counter == 21 && timer_expired) begin
                send_bufx <= 0;
                send_buf <= send_bufy;
                buf_counter <= 3;
             end
          end
        3: begin $display("in buf counter, case 3");
             if (counter == 21 && timer_expired) begin
                send_bufy <= 0;
                send_buf <= send_bufz;
                buf_counter <= 4;
             end
          end
        4: begin $display("in buf counter, case 4");
             if (counter == 21 && timer_expired) begin
                send_buf <= 0;
                buf_counter <= 5;
             end
             else if (counter == 20) begin
                timer_sample_start <=1;
                timer_sample_value <=255;//0.01s, sampling delay
             end
          end
        5: begin $display("in buf counter, case 5");
             if (timer_sample_expired) begin
                send_look <= 0;//end sending sequence
                timer_sample_start <= 0;
                buf_counter <= 0;
                timer_sample_expired <= 0;
             end
          end
        default: $display("error in buf_counter");
      endcase//buf_counter
    end//bit_clock
  end//sending
end//always
```

```
endmodule
```

0.10 PS/2 Keyboard

```
module ps2keyboard(
    input clock_27MHz,
    input [4:0] walk,
    input [5:0] accessory,
   input reset,
   output reg data,
   output reg ps2_clock_delay
    );

  //scan codes
  parameter switch_weapon = 8'h0D; //tab
  parameter flashlight = 8'h15; //q
  parameter forward = 8'h1D; //w
  parameter action = 8'h24; //e
  parameter reload = 8'h2D; //r
  parameter everyone_chat = 8'h2C; //t
  parameter team_chat = 8'h35; //y
  parameter left = 8'h1C; //a
  parameter backward = 8'h1B; //s
  parameter right = 8'h23; //d
  parameter melee_attack = 8'h2B; //f
  parameter switch_grenade = 8'h34; //g
  parameter vehicle_chat = 8'h33; //h
  parameter scope_zoom = 8'h1A; //z
  parameter exchange_weapon = 8'h22; //x
  parameter jump = 8'h29; //space
  parameter crouch = 8'h14; //l ctrl
  parameter stop = 8'hF0; //stop code

  //used to coordinate sending of bits
  reg [7:0] counter = 0;
  reg [7:0] buf_counter = 0;

  //to record previous walk, acc information
  reg [4:0] walk_delay = 0;
  reg [5:0] acc_delay = 0;
```

```
//to save previous walk, acc state
reg [4:0] walk_prev = 0;
reg [5:0] acc_prev = 0;

//flags to send a message
reg send_walk = 0;
reg send_acc = 0;

//send buffer -- place to put send codes
//default is zero
reg [7:0] send_buf = 0;
reg [7:0] stop_buf = 0;

//flags to check which one has changed
reg flag_forward = 0;
reg flag_backward = 0;
reg flag_left = 0;
reg flag_right = 0;
reg flag_jump = 0;
reg flag_reload = 0;
reg flag_scope_zoom = 0;
reg flag_switch_weapon = 0;
reg flag_flashlight = 0;
reg flag_melee_attack = 0;
reg flag_switch_grenade = 0;

reg stopflag_forward = 0;
reg stopflag_backward = 0;
reg stopflag_left = 0;
reg stopflag_right = 0;
reg stopflag_jump = 0;
reg stopflag_reload = 0;
reg stopflag_scope_zoom = 0;
reg stopflag_switch_weapon = 0;
reg stopflag_flashlight = 0;
reg stopflag_melee_attack = 0;
reg stopflag_switch_grenade = 0;

reg sent_forward = 0;
reg sent_backward = 0;
```

```
reg sent_left = 0;
reg sent_right = 0;
reg sent_jump = 0;
reg sent_reload = 0;
reg sent_scope_zoom = 0;
reg sent_switch_weapon = 0;
reg sent_flashlight = 0;
reg sent_melee_attack = 0;
reg sent_switch_grenade = 0;

//check to see if currently sending message
reg sending = 0;
reg stopsending = 0;
reg sequence = 0;

wire bit_clock;//13.5kHz clock

counter #(1000) c2(
  .clock(clock_27MHz),
  .reset(reset),
  .enable(bit_clock)
 );

 //timer regs
 reg timer_start = 0;
 reg timer_start_delay = 0;
 reg timer_start_signal = 0;
 reg [7:0] timer_value = 0;
 wire [7:0] count;
 wire timer_expire;
 reg timer_expired = 0;

 //after sending delay timer
timer_christy t2(
  .clock_27mhz(clock_27MHz),
  .reset(reset),
  .enable(bit_clock),//enable on bit_clock to cut down time
  .start(timer_start_signal),//start pulse
  .value(timer_value),   // timer values from 0-15
  .count(count),
  .expired(timer_expire)  // pulse high when timer value is reached
```

```
  );

 reg [270:0] sync = 0;//shift register for clock delay
 reg ps2_clock = 0;

always @(posedge clock_27MHz) begin
   //shift registers to delay clock by 5-25us
   {ps2_clock_delay,sync} <= {sync[270:0],ps2_clock};

  //pulse-maker
   timer_start_delay <= timer_start;
   timer_start_signal <= (!timer_start_delay && timer_start);

  //flag for expired timer
   if (timer_expire) begin
    timer_expired <= 1;
   end

   if (reset) begin
    sequence <=0;
    send_buf <= 0;
    stop_buf <= 0;
   end

  walk_delay <= walk;
  acc_delay <= accessory;

  //flags to tell if any data must be sent (any input has changed)
  if (walk_delay != walk) begin
      send_walk <= 1;
      walk_prev <= walk_delay;
  end
  if (acc_delay != accessory) begin
      send_acc <= 1;
      acc_prev <= acc_delay;
  end

  //default, keep both lines high
  if (!sequence) begin
    ps2_clock <= 1;
    data <= 1;
```

```
end

//set flags
 if (send_walk || send_acc) begin
       if (send_walk) begin
    $display("in sendwalk");
      if (walk[0] ^ walk_prev[0]) begin
      $display("in sendwalk, case 0");
        if (!walk[0]) begin
          stopflag_forward <= 1;
        end
        else flag_forward <= 1;
      end
      if (walk[1] ^ walk_prev[1]) begin
      $display("in sendwalk, case 1");
        if (!walk[1]) begin
          stopflag_backward <= 1;
        end
        else flag_backward <= 1;
      end
      if (walk[2] ^ walk_prev[2]) begin
      $display("in sendwalk, case 2");
        if (!walk[2]) begin
          stopflag_left <= 1;
        end
        else flag_left <= 1;
      end
      if (walk[3] ^ walk_prev[3]) begin
      $display("in sendwalk, case 3");
        if (!walk[3]) begin
          stopflag_right <= 1;
        end
        else flag_right <= 1;
      end
      if (walk[4] ^ walk_prev[4]) begin
      $display("in sendwalk, case 4");
        if (!walk[4]) begin
          stopflag_jump <= 1;
        end
        else flag_jump <= 1;
      end
```

```
     send_walk <= 0;
    end

    else if (send_acc) begin
$display("in sendacc");
    //throw flags for individual keys
        if (accessory[0] ^ acc_prev[0]) begin
  $display("in sendacc, case 0");
     if (!accessory[0]) begin
       stopflag_reload <= 1;
     end
     else flag_reload <= 1;
    end

    if (accessory[1] ^ acc_prev[1]) begin
  $display("in sendacc, case 1");
     if (!accessory[1]) begin
       stopflag_scope_zoom <= 1;
     end
     else flag_scope_zoom <= 1;
    end

    if (accessory[2] ^ acc_prev[2]) begin
  $display("in sendacc, case 2");
     if (!accessory[2]) begin
       stopflag_switch_weapon <= 1;
     end
     else flag_switch_weapon <= 1;
    end

    if (accessory[3] ^ acc_prev[3]) begin
  $display("in sendacc, case 3");
     if (!accessory[3]) begin
       stopflag_flashlight <= 1;
     end
     else flag_flashlight <= 1;
    end

    if (accessory[4] ^ acc_prev[4]) begin
  $display("in sendacc, case 4");
     if (!accessory[4]) begin
```

```
          stopflag_melee_attack <= 1;
        end
        else flag_melee_attack <= 1;
       end

       if (accessory[5] ^ acc_prev[5]) begin
      $display("in sendacc, case 5");
        if (!accessory[5]) begin
          stopflag_switch_grenade <= 1;
        end
        else flag_switch_grenade <= 1;
       end
           send_acc <= 0;
       end
 end //end message check

if (bit_clock) begin
 if (sequence) begin
   $display("in sequence, counter");
   //11-bit frame sending sequence
   case(counter)
     0: begin
          ps2_clock <= 0;
          data <= 0;
          counter <= 1;
        end
     1: begin
           ps2_clock <= 1;
           counter <= 2;
        end
     2: begin
           ps2_clock <= 0;
           data <= send_buf[0];
           counter <= 3;
        end
     3: begin
           ps2_clock <= 1;
           counter <= 4;
        end
     4: begin
           ps2_clock <= 0;
```

```
      data <= send_buf[1];
      counter <= 5;
  end
5: begin
      ps2_clock <= 1;
      counter <= 6;
  end
6: begin
      ps2_clock <= 0;
      data <= send_buf[2];
      counter <= 7;
  end
7: begin
      ps2_clock <= 1;
      counter <= 8;
  end
8: begin
      ps2_clock <= 0;
      data <= send_buf[3];
      counter <= 9;
  end
9: begin
      ps2_clock <= 1;
      counter <= 10;
  end
10: begin
      ps2_clock <= 0;
      data <= send_buf[4];
      counter <= 11;
  end
11: begin
      ps2_clock <= 1;
      counter <= 12;
  end
12: begin
      ps2_clock <= 0;
      data <= send_buf[5];
      counter <= 13;
  end
13: begin
      ps2_clock <= 1;
```

```
      counter <= 14;
  end
14: begin
      ps2_clock <= 0;
      data <= send_buf[6];
      counter <= 15;
  end
15: begin
      ps2_clock <= 1;
      counter <= 16;
  end
16: begin
      ps2_clock <= 0;
      data <= send_buf[7];
      counter <= 17;
  end
17: begin
      ps2_clock <= 1;
      counter <= 18;
  end
18: begin
      ps2_clock <= 0;
      data <= !(^send_buf); //!xor, odd parity bit
      counter <= 19;
  end
19: begin
      ps2_clock <= 1;
      counter <= 20;
   end
20: begin
      ps2_clock <= 0;
      data <= 1;//end bit
      counter <= 21;
   end
21: begin
$display("in counter, case 21");
      ps2_clock <= 1;
      data <= 1;
      timer_value <= 15;//clock high-time
      if (timer_expired) begin//if timer over
        counter <= 0;//reset counter
```

```
            timer_expired <= 0;
            timer_start <= 0;
            if (stopsending) begin //sends stop code next
                stopsending <= 0;
                send_buf <= stop_buf;
            end
            else sequence <= 0;//if a start code, just go to next in series
          end
          else timer_start <= 1;
       end
    endcase
  end//sequence

//check all the flags, and send the message if they are true
case(buf_counter)
  0: begin
  $display("in buf_counter, case 0");
    if (!sequence) begin
      if (flag_forward && !sent_forward) begin
          send_buf <= forward;
          flag_forward <= 0;
          sequence <= 1;
          sent_forward <= 1;
      end
      else if (stopflag_forward && sent_forward) begin
          send_buf <= stop;
          stop_buf <= forward;
          stopsending <= 1;
          stopflag_forward <= 0;
          sequence <= 1;
          sent_forward <= 0;
      end
      buf_counter <= 1;
     end
    end
  1: begin
  $display("in buf_counter, case 1");
      if (!sequence) begin
        if (flag_backward && !sent_backward) begin
            send_buf <= backward;
            flag_backward <= 0;
```

```
            sequence <= 1;
            sent_backward <= 1;
        end
        else if (stopflag_backward && sent_backward) begin
            send_buf <= stop;
            stop_buf <= backward;
            stopsending <= 1;
            stopflag_backward <= 0;
            sequence <= 1;
            sent_backward <= 0;
        end
        buf_counter <= 2;
      end
    end
  2: begin
  $display("in buf_counter, case 2");
      if (!sequence) begin
        if (flag_left && !sent_left) begin
            send_buf <= left;
            flag_left <= 0;
            sequence <= 1;
            sent_left <= 1;
        end
        else if (stopflag_left && sent_left) begin
            send_buf <= stop;
            stop_buf <= left;
            stopsending <= 1;
            stopflag_left <= 0;
            sequence <= 1;
            sent_left <= 0;
        end
        buf_counter <= 3;
      end
    end
  3: begin
  $display("in buf_counter, case 3");
      if (!sequence) begin
        if (flag_right && !sent_right) begin
            send_buf <= right;
            flag_right <= 0;
            sequence <= 1;
```

```
            sent_right <= 1;
        end
        else if (stopflag_right && sent_right) begin
            send_buf <= stop;
            stop_buf <= right;
            stopsending <= 1;
            stopflag_right <= 0;
            sequence <= 1;
            sent_right <= 0;
        end
        buf_counter <= 4;
      end
    end
4: begin
$display("in buf_counter, case 4");
      if (!sequence) begin
        if (flag_jump && !sent_jump) begin
            send_buf <= jump;
            flag_jump <= 0;
            sequence <= 1;
            sent_jump <= 1;
        end
        else if (stopflag_jump && sent_jump) begin
            send_buf <= stop;
            stop_buf <= jump;
            stopsending <= 1;
            stopflag_jump <= 0;
            sequence <= 1;
            sent_jump <= 0;
        end
        buf_counter <= 5;
      end
    end
5: begin
$display("in buf_counter, case 5");
      if (!sequence) begin
        if (flag_switch_grenade && !sent_switch_grenade) begin
            send_buf <= switch_grenade;
            flag_switch_grenade <= 0;
            sequence <= 1;
            sent_switch_grenade <= 1;
```

```
      end
      else if (stopflag_switch_grenade && sent_switch_grenade) begin
          send_buf <= stop;
          stop_buf <= switch_grenade;
          stopsending <= 1;
          stopflag_switch_grenade <= 0;
          sequence <= 1;
          sent_switch_grenade <= 0;
      end
      buf_counter <= 6;
    end
  end
6: begin
$display("in buf_counter, case 6");
    if (!sequence) begin
      if (flag_reload && !sent_reload) begin
          send_buf <= reload;
          flag_reload <= 0;
          sequence <= 1;
          sent_reload <= 1;
      end
      else if (stopflag_reload && sent_reload) begin
          send_buf <= stop;
          stop_buf <= reload;
          stopsending <= 1;
          stopflag_reload <= 0;
          sequence <= 1;
          sent_reload <= 0;
      end
      buf_counter <= 7;
    end
  end
7: begin
$display("in buf_counter, case 7");
    if (!sequence) begin
      if (flag_scope_zoom && !sent_scope_zoom) begin
          send_buf <= scope_zoom;
          flag_scope_zoom <= 0;
          sequence <= 1;
          sent_scope_zoom <= 1;
      end
```

```
      else if (stopflag_scope_zoom && sent_scope_zoom) begin
          send_buf <= stop;
          stop_buf <= scope_zoom;
          stopsending <= 1;
          stopflag_scope_zoom <= 0;
          sequence <= 1;
          sent_scope_zoom <= 0;
      end
      buf_counter <= 8;
    end
  end
8: begin
$display("in buf_counter, case 8");
    if (!sequence) begin
      if (flag_switch_weapon && !sent_switch_weapon) begin
          send_buf <= switch_weapon;
          flag_switch_weapon <= 0;
          sequence <= 1;
          sent_switch_weapon <= 1;
      end
      else if (stopflag_switch_weapon && sent_switch_weapon) begin
          send_buf <= stop;
          stop_buf <= switch_weapon;
          stopsending <= 1;
          stopflag_switch_weapon <= 0;
          sequence <= 1;
          sent_switch_weapon <= 0;
      end
      buf_counter <= 9;
    end
  end
9: begin
$display("in buf_counter, case 9");
    if (!sequence) begin
      if (flag_flashlight && !sent_flashlight) begin
          send_buf <= flashlight;
          flag_flashlight <= 0;
          sequence <= 1;
          sent_flashlight <= 1;
      end
      else if (stopflag_flashlight && sent_flashlight) begin
```

```
                send_buf <= stop;
                stop_buf <= flashlight;
                stopsending <= 1;
                stopflag_flashlight <= 0;
                sequence <= 1;
                sent_flashlight <= 0;
            end
            buf_counter <= 10;
          end
        end
      10: begin
      $display("in buf_counter, case 10");
          if (!sequence) begin
            if (flag_melee_attack && !sent_melee_attack) begin
                send_buf <= melee_attack;
                flag_melee_attack <= 0;
                sequence <= 1;
                sent_melee_attack <= 1;
            end
            else if (stopflag_melee_attack && sent_melee_attack) begin
                send_buf <= stop;
                stop_buf <= melee_attack;
                stopsending <= 1;
                stopflag_melee_attack <= 0;
                sequence <= 1;
                sent_melee_attack <= 0;
            end
            buf_counter <= 11;
          end
        end
      11: begin
          buf_counter <= 0;
        end
    endcase //buf_counter
  end//bit_clock
end//always
endmodule
```

0.11 timer_christy

```
module timer_christy(
    input wire clock_27mhz, // system clock
    input wire enable,  // one second enable signal from Part A
    input wire start,  // start timer
    input wire reset,    // system reset
    input wire [7:0] value,   // timer values from 0-255 seconds
    output reg [7:0] count,
    output reg expired  // pulse high when timer value is reached
    );

    reg started, myreset;

    always @ (posedge clock_27mhz) begin
        if (reset || myreset) begin
            started <= 0; expired <= 0; count <= 0; myreset <= 0;
        end

        else begin
            if (start) started <= 1; //if timer has been started
            if ((count == value) && started) begin //if the timer value has been reached
                expired <= 1;
                started <= 0;
                count <= 0;
                myreset <= 1;
            end
            else if (enable & started) count <= count + 1; //increment counter

        end
    end

endmodule
```