Auto Targetting in a Remote Sentry Turret

Jacky Chang, Stephanie Paige, Eli Stickgold

December 11, 2008

Abstract

We set out to design and implement a proof of concept laser turret. This turret would have two modes of operation. One would be a manual mode that allows users to direct the laser through use of a cursor and fire when ordered. The other mode would automatically identify and fire upon one of possibly several moving bodies in its field of vision. This was complicated by the fact that a motion detection algorithm capable of differentiating between multiple targets in hardware, using a static camera, is a problem lacking a widely accepted solution. This complication was further exacerbated by memory and timing constraints that we faced, caused by the physical limitations of the FPGA. Through the use of changed-pixel data and dual-pass histograms, we were able to produce a system that could reliably distinguish and highlight a moving target. Further refinement and testing with more reliable equipment is still necessary.

Contents

1 Overview **5**

2 Description **6**

2.1 Motion Tracking (Stephanie) 6

2.1.1 Column Histogram (vertical_histogram) 6

2.1.2 Row Histogram (horizontal_histogram) 6

2.1.3 Window-of-Interest Finders (row_finder, col_finder) 7

2.2 Pixel-to-Angles LUT (anglesLUT) (Stephanie) 8

2.3 Video Capture (Eli) 8

2.4 Video Storage (Eli) 9

2.5 Changemap Generation and Storage (Eli) 9

2.6 Display (Eli and Jacky) 10

2.7 Mouse Tracking (Jacky) 11

2.8 Target (Jacky) 12

2.9 Divider (Jacky) 12

2.10 Timer (Jacky) 12

2.11 ASCII Converter (Jacky) 13

2.12 Servo Interface (Jacky) 13

2.13 Rangefinder (Jacky) 14

2.14 Master Servo (Jacky) 15

3 Testing **16**

3.1 Motion Tracking 16

3.2 Pixel-to-Angle LUT 16

3.3 Video Processing 16

3.4 Memory 16

3.5 Mouse Tracking and Display 17
3.6 Servo Interface 17

4 Conclusion **17**

Appendices **18**

A Testing Code **18**
A.1 Motion-Tracking Test Rig (Stephanie) 18
A.2 RS-232 Transmitter Testbench (Jacky) 18
A.3 RS-232 Receiver Testbench (Jacky) 19

B Unused Code **21**
B.1 Range (Jacky) 21
B.2 RS-232 Transmitter (Jacky) 22
B.3 RS-232 Receiver (Jacky) 23

List of Figures

1 Turret Block Diagram 5

2 Schematic Representation of Pixel to Angle Calculation 8

3 Block Diagram of Video Processing and Changemap Generation 10

4 Servo Angles of Rotation [1] 13

5 Circuit Diagram for Servo Motors 14

1 Overview

The purpose of the this device is to fire upon moving bodies that enter a target area. Alternatively, a user may override automated targetting and manually fire upon priority targets. In order to facilitate installation and troubleshooting, the turret is connected to a graphical interface which provides users with multiple data overlays.

In this preliminary design, the turret is constructed of two directly-joined servo motors, one per degree of freedom, and a laser pointer. The horizontal motor controls the azimuth of the laser pointer. Directly afixed to the rotating platform of the horizontal motor is the vertical motor, which controls the elevation of the laser. The laser pointer attached to the platform of second motor is used to define and display the orientation of the turret.

The turret has two modes of operation: manual and automatic. In either mode, the module generates a target pixel, which is mapped to a particular azimuth and elevation through look up tables. These angles are then sent to the servo motor controls, which direct the laser pointer to highlight the target. In manual mode, this pixel is selected through use of the cursor.

In automatic mode, the target pixel is chosen by a multi-module video-processing algorithm. Video frames from the camera are stored in the ZBTs and used to generate a change map: a two color image, in which white pixels mark pixels that have changed from the previous frame, and black pixels mark unchanged locations. This map is then used to compile vertical and horizontal histograms, which are then used to calculate the center of a particular box of interest, which becomes the target pixel.

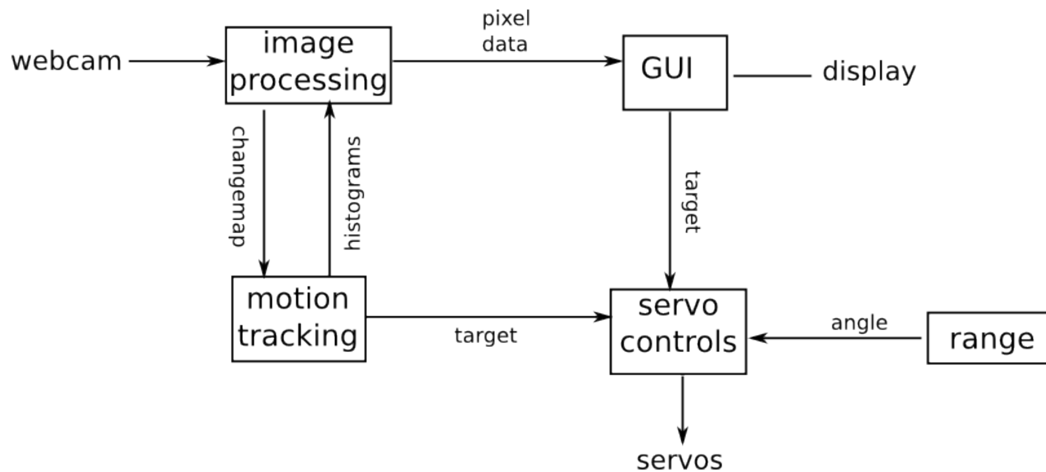

Figure 1: Turret Block Diagram

2 Description

2.1 Motion Tracking (Stephanie)

The motion tracking algorithm is excecuted in two passes: A histogram of changed pixels in each column of a frame is constructed using changemap data streaming in from the Video Processing module, then searched for a span rich in entries over a threshold number of pixels. The center of this span becomes the x-coordinate of the target pixel. The edges of this span are reported to another histogram module, which calculates a histogram of changed pixels per row, but only counts pixels that fall within the span of interest. This histogram is also searched to find a high-activity region, which determines the y-coordinate of the target pixel. Excluding pixels outside the x-span of interest in the second histogram prevents other moving bodies in the camera's few from winning the second search, thus causing the turret to aim at the x-coordinate of one body and the y-coordinate of an another.

2.1.1 Column Histogram (vertical_histogram)

inputs: hcount, vcount, clk, pixel
output: data

Because data streams in row by row, summing up by columns requires as many accumulator locations as the width of the image. These were implemented using a dual-port 640x9 BRAM with one read-only port and one write-only port. The value for each column was stored in the BRAM using the column's hcount as an address. The read-only port always provides the current histogram value for the current hcount, both to module-internal logic and to other modules through data output.

For the first line of each frame, the memory is cleared by storing 1 or 0 for white or black pixels in the changemap at the location corresponding to the current hcount. During the frame, a pixel from the change-map stream is read in and added to the sum accumulating for the current column retrieved from the BRAM the preivous cycle. The next cycle, the new sum is stored in the BRAM at the same location.

2.1.2 Row Histogram (horizontal_histogram)

inputs: hcount, vcount, clk, pixel, start, finish, request
outputs: data

Unlike the other histogram module, this module can use a single accumulator location (a bus of 10 registers) , and also accepts a start-pixel and end-pixel input to narrow the area of the frame it sums over, taken from col_finder. Each entry in the histogram is stored for one

frame in a single-port 480x10 BRAM, addressed by the line's vcount number. During the frame, the memory is written to, storing histogram data. Outside the frame, the memory outputs the histogram data for whichever row is requested at the request input.

For each line in the frame, when hcount is equal to the current value of start, the value in the accumulator registers is cleared by moving the current pixel value into it. Every clock cycle until hcount is equal to finish, the total in the accumulator is incremented by the current changemap pixel value. At hcount == finish + 1, the current total in the accumulator is written into the BRAM at the location addressed by the current vcount.

2.1.3 Window-of-Interest Finders (row_finder, col_finder)

inputs: clk, hcount, vcount, data, TH, MAX_GAP
outputs: start, finish, center

The two finder modules (col_finder, row_finder) search histogram data with the same single-pass algorithm. The algorithm searches for the highest scoring 'window', a region of high-value histogram lines that are separated by no more than MAX_GAP lines.

One line of a histogram is read into the module per clock cycle during a pass. During the pass, the module is either *searching* for a window or *extending* a window. Six parameters (best_start, best_finish, best_score, cur_start, cur_finish, cur_gap) are reset to zero at the beginning of each pass. If the first line of histogram data exceeds TH, the significance threshold, cur_score is initialized to one and the module enters the searching state. Otherwise, cur_score is initialized to 1 and the module enters the extending state.

When searching, the histogram data for the current line is compared to TH. If the data is greater than TH, the current line is recorded as cur_start and the module enters the extending state. Otherwise, it proceeds to the next line in the histogram.

In the extending state, if the current data is greater than TH, cur_finish is updated to the current line number, cur_score is incremented by one, and the module moves onto the next line. Otherwise, cur_gap is incremented, or if cur_gap is already equal to MAX_GAP, the window is closed: the module returns to the searching state, and if cur_score exceeds best_score, best_score, best_start, and best_finish are replaced by cur_score, cur_start, and cur_finish.

At the end of a pass, if the module is currently extending a window and cur_score exceeds best_score, cur_start and the maximum possible line number are pushed into start and finish. Otherwise, best_start and best_finish are pushed into start and finish. Center is calculated by averaging start and finish.

Col_finder processes histogram data during the first line of the vsync period, processing one line per clock cycle. Row_finder processes histogram data as it becomes available, one line of the histogram per line displayed on the screen, during the hsync period.

2.2 Pixel-to-Angles LUT (anglesLUT) (Stephanie)

inputs: x, y, reset
outputs: x_pulse, y_pulse

The LUT mapping horizontal and vertical coordinates to angles is simply two asynchronous ROMs. The azimuth and elevation for each x and y coordinate were calculated according to the following principles:

If V is half the viewing angle of the camera in either the horizontal direction, X is half of the width of the camera's field of view in pixels, and x is the distance between a point P on the camera image and the vertical midline of the image, then the angle V' which connects the camera's location to P can be determined with the following equation:

$\frac{\tan V'}{\tan V} = \frac{x}{X}$

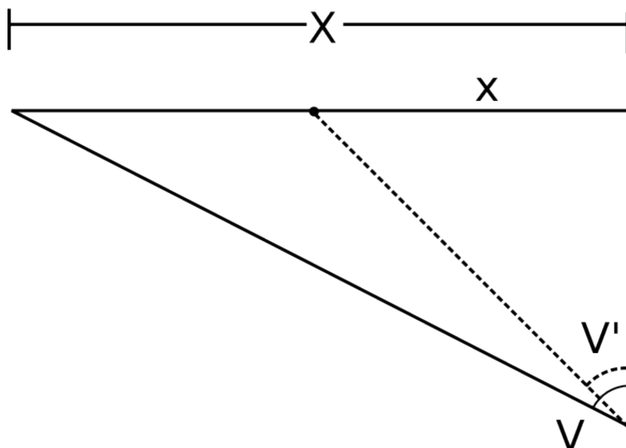

Figure 2: Schematic Representation of Pixel to Angle Calculation

If P is on the left side of the image, then the azimuth (with 0 at the far left side of the image) is V - V'. If P is on the right side of the image, then the azimuth is V + V'.

An analogous calculation can be performed in the vertical direction to obtain the elevation of any point in the image. Angles were stored in the ROM as pulse-widths rounded to the nearest μs. (See Figure 4)

2.3 Video Capture (Eli)

inputs: clk, reset, tv_in_ycrcb
outputs: ycrcb, f, v, h, data_valid

The webcam used in this project transmits a 640x480 resolution interlaced image at a rate of 30 frames per second. This data is captured by the adv7185 module and fed into an

NTSC decoder, which converts the data into a 30-bit YCrCb value, a horizontal sync signal, a vertical sync signal, and a field sync signal which is used to deinterlace the video. The field sync is used as the low-order bit for the vertical position of the current pixel, due to the design of interlaced video. Of the 30 bits of YCrCb data sent out by the decoder, only 8 bits of luminance data are used, in order to allow the storage algorithms to pack multiple pixels of data into each memory location, which results in a greyscale image being displayed.

2.4 Video Storage (Eli)

inputs: clk, vclk, fvh, dv, din, sw
outputs: ntsc_addr, ntsc_data, ntsc_we

Once the image is decoded, the 8 bits of data preserved (the high-order bits of luminance) are sent, along with a 3-bit control signal created by amalgamating the horizontal, vertical and field sync signals from the decoder, to an addressing and packing module, ntsc2zbt.

This module uses the fvh control signal along with the clock generated by the camera to generate column and row registers, tracking the location of the current pixel's data. The column, row and sync signals are then synchronized with the system pixel clock, and every time a new pixel is read in, it is added to a 32-bit cache. Every four input cycles (enough time to fully fill the cache with new pixel data) an output address is updated, assigned using the column and row (disregarding the two low-order bits of the column figure, as four pixels of data is stored to each address) and the 36-bit data output signal(four 0s followed by the cache) is also updated.

Both of the two ZBTs available on the labkit take the data output from ntsc2zbt as their write data, and are write-enabled once every four cycles of the pixel clock (accomplished by tying their behavior to the lowest 2 bits of hcount). When write-enabled, the ZBTs use the address outputted by ntsc2zbt, but when reading the two ZBTs behave differently. One still take the write address to provide the comparison pixel used to generate the change map, while the other taking an address similar in format to the one generated by ntsc2zbt, but based off hcount and vcount, generating the image that is processed and displayed to the screen.

2.5 Changemap Generation and Storage (Eli)

inputs: clk, new_write, old_read
output: changemap_write, bram_addr

The change map used by the motion-detection algorithm is generated by a comparison performed once every four pixel cycles as data is put in memory. On an early cycle, the data at the current write address is read out of the second ZBT (dedicated to producing

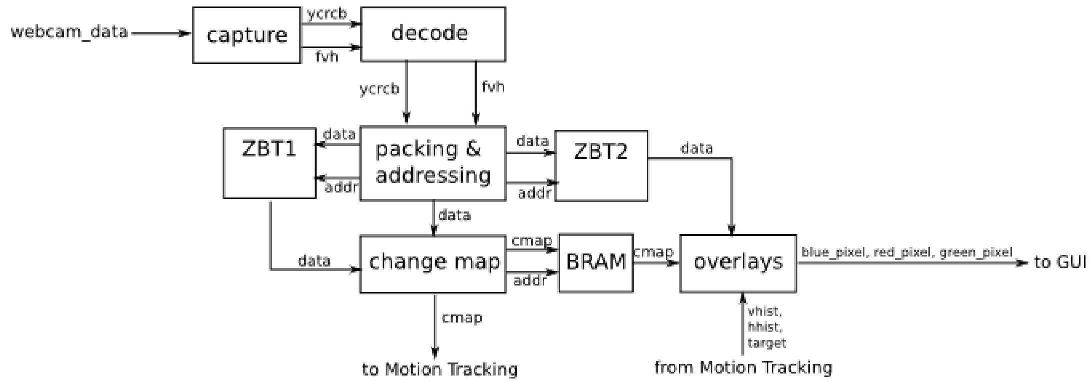

Figure 3: Block Diagram of Video Processing and Changemap Generation

the change map) and latched to a 36-bit register. When the ZBTs are write-enabled, the data being written is similarly latched, producing two 36-bit registers containing the values of four pixels at two intervals separated by one thirtieth of a second. A constant calculation checks each of the four pixels stored to determine whether the new or old value is larger, and then once per four cycles the smaller value is subtracted from the larger for each pixel, and the result compared to a fixed threshold. A 4-bit change_map register is updated to contain 1 if the corresponding pixel changed more than the threshold and 0 if it did not.

Once the change_map data is latched, it is written to a BRAM, using the same addressing scheme as the ZBTs. The BRAM is only write-enabled on one of the four cycles during which change_map is held constant to allow time for the data to be read out by the modules that implement the motion detection algorithms.

2.6 Display (Eli and Jacky)

inputs: vcount, hcount, vr_pixel, hhist_start, hhist_finish, hhist_center, hhist_data, vhist_start, vhist_finish, vhist_center, vhist_data, mode_pixel, mode2_pixel, mouse_pixel, acquired_pixel, dist_pixel
outputs: red_pixel, green_pixel, blue_pixel, gui_pixel

The first ZBT, meanwhile, writes once every four cycles into a similar 36-bit register, but must produce an output every cycle (to be displayed) rather than every four cycles. To this end, the register is latched at the beginning of each 4-cycle block and on each cycle a different byte of data is unpacked and passed as the argument vr_pixel. vr_pixel provides the base luminance value of the camera at each location, and it is then passed into a bit-wise or along with another 8-bit number that is merely 8 copies of a single value, 'red' 'green' or 'blue', to produce three separate values: red_pixel, green_pixel and blue_pixel (either the value of vr_pixel or else set to maximum value), which are given to the GUI for final sprite overlays.

Each color is used to provide one or two pieces of information. In basic mode a blue box one pixel wide is generated around the 'target of interest' box provided by the motion-detection algorithm (in the form of vhist_start, vhist_finish, hhist_start and hhist_finish) and a pair of red lines one pixel wide running vertically and horizontally intersect at the target point (provided by vhist_center and hhist_center).

Two overlays can be enabled through switches (their contribution to red and green is logically anded with the value of the switch): a change map overlay that highlights changing pixels in green, and a histogram overlay that shows the outputs of the horizontal and vertical histograms along the sides (implemented simply by checking whether the distance to the bottom and right-hand edges of the screen is less than the current horizontal and vertical histogram data being received).

The display module takes the data output from the ZBTs and overlays sprites. The code uses the mouse-tracking module to get the pixel for the cursor and the char_string_display module from the course staff[**?**, 4] The char_string_display module has been modified in order to produce font half the size of the original module. The user interface uses user input to determine the mode of the turret and chooses what words to display based on that input. The mode of the turret determines when the fire signal should actually be propogated, and when the fire signal is considered valid, the bottom righthand text on the screen switches from "Acquiring Target" to "Firing." In either mode, the distance is displayed in hex on the lower lefthand corner of the screen. In order to display the distance, the module takes the output of the fake_range module and passes it to the ASCII converter. That output is then passed to char_string_display. Also in either mode, the top righthand corner of the display informs the user of what mode the turret is currently in.

2.7 Mouse Tracking (Jacky)

inputs: vclock, reset, hcount, vcount, hsync, vsync, blank, manual_p, ps2_clock, ps2_data
outputs: phsync, pvsync, pblank, pixel, x_loc, y_loc, click_x, click_y, button_p

The mouse tracking module takes input from the PS2 mouse and calculates an x-y coordinate, where it overlays a cursor sprite. It also saves and outputs the location of the last button click from the mouse, along with one bit enable indicating if the mouse is clicked, the current mouse location, a 3-bit pixel value, and the syncs and blanking for the VGA signal.

In order to process data from the PS2 mouse, the module uses the ps2_mouse_xy module provided by the course staff[**?**, 2] The cursor sprite was created using the target module.

2.8 Target (Jacky)

inputs: x, hcount, y, vcount, button_p, manual_p
output: pixel

The target module takes the x-y location of the mouse, the hcount and vcount, button_p, and whether the turret is in manual or automatic fire mode. It returns a 3-bit value for each pixel, indicating its color. The module also has three parameters: WIDTH, HEIGHT and COLOR. The WIDTH and HEIGHT are by default set to 5, making the cursor 10-pixels by 10-pixels, and the default COLOR is red.

The module first checks which mode the device is in. If the device is in automatic fire mode, the module produces a 10-pixel by 10-pixel white box in order to indicate to users that the mouse has no affect on the behavior of the device.

If the turret is in manual fire mode, the module then checks the state of the button press, using button_p. If the button is depressed, the module produce a red 10-pixel by 10-pixel box containing red 6-pixel by 6-pixel box, which then contains a single red pixel that is exactly over the current position of the mouse. If the button is not depressed, the module produces a targetting crosshair, consisting of a red 10-pixel by 10-pixel box with a 18-pixel by 18-pixel cross centered on the current position of the mouse.

2.9 Divider (Jacky)

inputs: clk, start_timer
output: enable

By default, the divider converts the 25MHz master clock to a signal called enable at a frequency that is set by the user. The divider is reset every time start_timer is asserted so the first enable signal after the timer starts is exactly one period after the timer's started. The divider has a parameter DELAY, which changes the period (and thus the frequency) of the enable. Its default sets the enable frequency to a 1HZ signal that is asserted once every 25,000,000 cycles.

2.10 Timer (Jacky)

inputs: start, enable, clk, value
outputs: count, expired

The timer takes a 4-bit time parameter and counts down that number of seconds. It initializes its internal value when start_timer is asserted and asserts the expired signal when the countdown reaches one. The count then halts until it is reinitialized when start_timer is

asserted again.

This FSM consists of four states: S_RESET, S_0, and S_01. The RESET stateloads the value for the counter and waits for start_timer to be asserted. The FSM then moves to the S_0, where the number is decremented every time the enable signal from the divider is asserted. When the internal counter reaches zero, the expired signal is asserted and the state is moved to S_01. After one clock cycle, the FSM returns to the S_RESET state to await the next start_timer. The default state of the timer is S_RESET.

2.11 ASCII Converter (Jacky)

input: number
output: ascii

The ASCII converter takes a 4-bit number and converts it to a 8-bit ASCII value. The 4-bit input is passed to a case statement, which outputs the appropriate ASCII value.

2.12 Servo Interface (Jacky)

inputs: position, clk
outputs: range, fire, servo

The servo motors use pulse-width modulation to control their rotation. The angle of rotation correlates with the length of the pulse in microseconds according to the image in Figure 4 below.

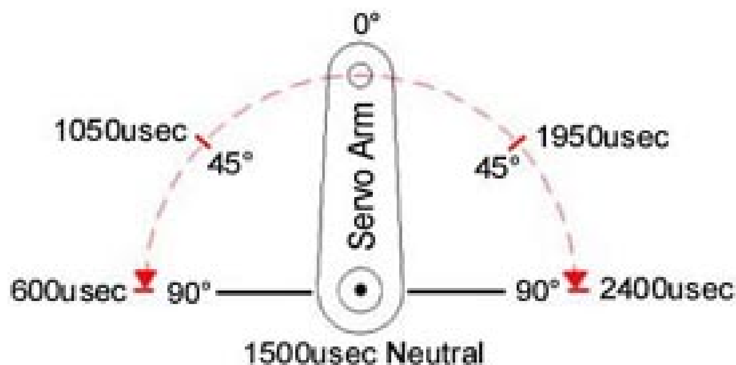

Figure 4: Servo Angles of Rotation [1]

The servo interface module takes as input a 12-bit position and the clock and outputs three enables, called range, fire and servo respectively. The module first initializes the 12-bit registers current and old_position to 1500, which is the center position of the servo motors.

The divider module is used with a DELAY of 25 in order to produce an enable signal every microsecond, which is then passed to the timer. The 12-bit position input is checked to ensure that it remains between 600 and 2400, since those are the limits of the pulse width necessary to move the motors. If it is not, it sets current to 1500.

The output enable servo is determined on servo_count, the microsecond enable from the timer, by checking the current position against the current count on the timer. This produces a signal that is the inverse of the desired enable signal of the servo motor. This signal is then fed to a inverter on the circuit board, which is wired according to the schematic below:

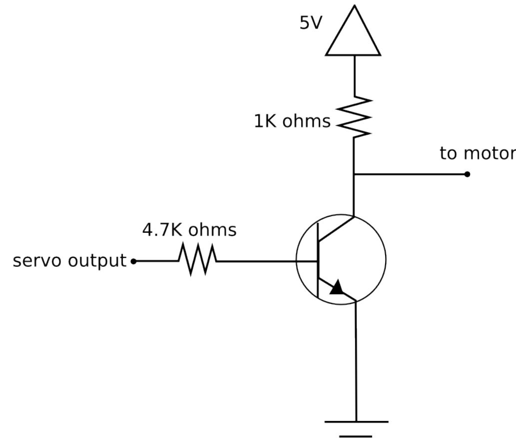

Figure 5: Circuit Diagram for Servo Motors

After the expired signal is received from the timer module, the servo interface sets old_position equal to current and checks if they were equal. If not, the fire enable is set high. The module also assigns range to expired.

2.13 Rangefinder (Jacky)

inputs: clk, up, down
output: distance

The rangefinder module went through two iterations based on unavoidable hardware problems. The first iteration was based on the assumption that would be able to obtain a functional sonic rangefinder which could receive RS-232 serial signal and transmit the same. The code for this module is in Appendix B.1 and is labelled "range."

The range module takes the clock, an enable and a serial signal and outputs a 12-bit elevation, a move enable and a serial signal called to_range. A divider uses a DELAY of 25 to create a microsecond enable signal that is used for all of the timers in the module. The enable begins the first timer, which is set to count down the amount of time necessary to transmit one byte using the RS-232 transmitter, and the first RS-232 transmitter, which sends the

the first byte containing a 0, which is the address of the rangefinder[**?**, 3] When the expired signal from the timer goes high, the second transmitter is enabled and begins sending the command, in this case 84[**?**, 3] The range module then waits for a signal back from the rangefinder. When a signal is detected, the second timer begins to countdown, and when its expired signal is asserted, the distance is considered valid. This distance is then passed to an angle look up table (unimplemented), which would output the correct elevation necessary in order to compensate for the distance to the target.

Rather than using an actual rangefinder to measure the distance to the target, the fake_range module allows users to input an approximate distance in units of decimeters. The module takes the clock, an up and a down signal, and outputs a 6-bit distance value. The default distance is set to 20 dm, or 2m, and on the positive edge of the clock, the number would increment or decrement based whether or not either of the buttons were pressed.

2.14 Master Servo (Jacky)

inputs: clk, x_pulse, y_pulse
outputs: horiz_servo, vert_servo, fire

The master servo module is a wrapper module that controls all signals to and from the servo motors. It takes as inputs the clock and two 12-bit pulse lengths, one for each servo, and outputs signals for each servo and a firing enable.

Because the movement of the servo motors is heavily dependent on the rangefinder and its accompaning modules, this module also went through two different iterations. The first iteration can still be seen in the vestigal code of the module.

In both iterations, module passes the relevant pulse width to the correct servo after adding 600, the base pulse width, to each, and takes the firing enable from the vertical servo to pass on. The current iteration also simply adds the value output from fake_range to the vertical pulse, in order to allow users to affect elevation in order to compensate for the distance to the target.

In the former iteration of this module, the range module was also instantiated, and a third servo interface was added. This output to the vertical servo motor was determined by comparing the old x and y values to the current ones. If they were found to be the same and the enable signal from the rangefinder was asserted, the output to the vertical servo would be the elevation needed to compensate for the distance to the target. Otherwise, the servos would continue to actually track the target. Additionally, the only fire signal that would be regarded was the one that came from the servo interface that actually dealt with the range compensation.

3 Testing

3.1 Motion Tracking

To observe the response of the histogram and finder modules to arbitrary inputs, the pong game from lab 4 was adapted into a test rig. The increased XVGA resolution allowed both a 640x480 input region to be displayed on screen simultaneously with graphical representation of the output from the histograms and the start/finish/center outputs from the finder modules. Because hcount and vcount above 639 and 479 are treated as part of the hsync and vsync period, operation of the modules was essentially unchanged.

3.2 Pixel-to-Angle LUT

To test the accuracy of the LUTs, the turret was placed into manual mode, the laser dot aligned with the onscreen cursor by rotating the servo setup, and the quality of tracking visually assessed by moving the mouse around the screen.

3.3 Video Processing

The basic modules for capturing and displaying video data were prebuilt and available from the code repository on the 6.111 web page, so they required relatively little debugging and were fairly transparent. The overlay displays were created largely to aid debugging, although they were kept in the final version as they seemed useful tools. The change map overlay was the most useful debugging tool for the video processing portion of the design, and it tested for successful operation of both ZBTs, the change map generation, and the BRAM.

3.4 Memory

The hex display could be hooked up to various parameters to allow intermediate-level testing, but as most signals varied at either 30hz or 25Mhz, the display was primarily useful only for testing whether two signals were equivalent and seeing if a given signal was changing at all. The hex display was used to test the two ZBTs and the change map BRAM by feeding the addressing variables and the data inputs and outputs. While this couldn't provide data slow enough for us to read, it made it clear when addresses were failing to change or data wasn't being written or read at all.

3.5 Mouse Tracking and Display

For the display, testing and debugging was relatively simple, as expected. It was easy, visually to tell if there were display problems. For the mouse, it was important to check the cursor at the edges of the screen. Additionally, the current location and the location of the last click were connected to the hex display in order to ensure that the module was tracking correctly. Outputting the distance on the bottom lefthand corner of screen also allowed for debugging of the range module.

3.6 Servo Interface

Testing the fuctionality of the servo motors was a bit more difficult. We initially used the eight switches on the labkit to manually control the pulse width to the motors. Connecting the oscilloscope to the input of the servos allowed us to check what was being sent and tell if the pulses actually increased or decreased as the switches changed. A similar method was used when debugging the faulty rangefinder, allowing us to see what was being transmitted and received by the RS-232 modules. We also created ModelSim testbenches for the unused RS-232 receiver and transmitter.

4 Conclusion

In this project we designed and prototyped a proof of concept motion tracking turret. The turret could reliably follow moving objects across its field of vision and was able to select a single target from multiple moving bodies. The graphical user interface provided a reasonable amount of feedback to users that would aid in understanding the current state of the device.

While our proof of concept was quite successful, the system has room for much improvement. The servo motors would benefit greatly from having a noise filter to reduce erractic motion. A self-correcting targetting algorithm in place of the fixed angle tables would greatly improve the accuracy of the turret because we would no longer be affected the servo motor set up deviating from ideal assumptions. We would also greatly benefit from a higher quality camera that would have reduced noise and fewer artifacts in regions of high contrast. The last thing, regarding hardware, that could have been improved, would be a functional sonic rangefinder and calcuations for real firing solutions.

The other area in which this design could have been improved would be the addition of more user customizable options. The turret would be more effective if our user interface allowed customization of the threshhold values. Additionally, a mechanism to allow users to assign more weight to targets nearer the cursor on the screen. This would allow user more control over what was targetted in the automatic firing mode, and would reduce the turret's

tendency to jump between targets.

References

[1] "HS-311 Standard," Dec. 10. http://servocity.com/html/hs-311_standard.html.

[2] "ps2_mouse_yx," Dec. 10. http://web.mit.edu/6.111/www/f2005/code/ps2_mouse.v.

[3] "SRF02 Ultrasonic range finder," Dec. 10. http://www.robot-electronics.co.uk/htm/srf02techSer.htm.

[4] "char_string_display" Dec. 10. http://web.mit.edu/6.111/www/f2005/code/cstringdisp.v.

Appendices

A Testing Code

A.1 Motion-Tracking Test Rig (Stephanie)

A.2 RS-232 Transmitter Testbench (Jacky)

When we were still expecting to use the actual rangefinder, and with it the RS-232 transmitter, we created a testbench to check its functionality.

```
module transmit_tb;

   // Inputs
   reg clk;
   reg transmit;
   reg [7:0] data;

   // Outputs
   wire      serial;

   // Instantiate the Unit Under Test (UUT)
   RS232_transmit uut (
                       .clk(clk),
                       .transmit(transmit),
```

```
                    .data(data),
                    .serial(serial)
                    );

   initial begin
      // Initialize Inputs
      clk = 0;
      transmit = 0;
      data = 0;

      forever #20 clk = ~clk; // 40 ns per cycle clock (25Mhz)
   end

   initial begin
      transmit = 0;
      #104_167; // wait the amount of time it takes for one bit to go through
      transmit = 1;
      data = 84;
      #1_145_834;
   end

endmodule
```

A.3 RS-232 Receiver Testbench (Jacky)

Similarly to the transmitter, we created a testbench for the RS-232 receiver when we expected to use it with the rangefinder.

```
module receive_tb;

   // Inputs
   reg serial;
   reg clk;

   // Outputs
   wire [7:0] data;
   wire       started;

   // Instantiate the Unit Under Test (UUT)
   RS232_receive uut (
                      .serial(serial),
```

```
                        .clk(clk),
                        .data(data),
                        .started(started)
                        );

    initial begin
        // Initialize Inputs
        serial = 1;
        clk = 0;

        forever #20 clk = ~clk;
    end

    initial begin
        serial = 1; // idle
        #104_167;
        serial = 0; // start
        #104_167;
        serial = 0; // bit 1
        #104_167;
        serial = 1; // bit 2
        #104_167;
        serial = 0; // bit 3
        #104_167;
        serial = 1; // bit 4
        #104_167;
        serial = 0; // bit 5
        #104_167;
        serial = 0; // bit 6
        #104_167;
        serial = 0; // bit 7
        #104_167;
        serial = 0; // bit 8
        #104_167;
        serial = 1; // stop 1
        #104_167;
        serial = 1; // stop 2
    end

endmodule
```

B Unused Code

B.1 Range (Jacky)

```
module range (input clk, enable, serial,
              output reg [11:0] elevation, output move, output reg to_range);
   wire micro, second, receiving, request;
   wire [7:0] distance;
   reg        low_byte = 0;
   wire       to_range1, to_range2, to_range3;

   divider #(.DELAY(25)) microsecond (.clk(clk), .start_timer(enable),
                                      .enable(micro));
   timer one_message(.clk(clk), .start(enable), .enable(micro),
                     .value(1250), .expired(second));
   RS232_transmit sender(.clk(clk), .transmit(enable), .data(0),
                         .serial(to_range1));
   RS232_transmit sender2(.clk(clk), .transmit(second), .data(84),
                          .serial(to_range2));
   RS232_receive receiver (.serial(serial), .clk(clk), .data(distance),
                           .started(receiving));
   timer receive_message(.clk(clk), .start(receiving), .enable(micro),
                         .value(1250), .expired(move));

   always @(*) to_range <= (to_range1 && to_range2);

   always @(posedge clk) begin
      if (low_byte && move) elevation[7:0] <= distance;
      else if (~low_byte && move) elevation[16:0] <= distance;
      else elevation <= elevation;
      if (move) low_byte = ~low_byte;
   end



   // waiting for the LUT ...

   //always @(posedge clk) begin
   //   if (move) begin
   //   end
   //end
```

```
endmodule // range
```

B.2 RS-232 Transmitter (Jacky)

The RS-232 transmitter was never used because it was implemented for use in the discarded range module. It took as inputs the clock, an enable signal labelled transmit, and an 8-bit data signal to send. Its output was a 1-bit signal.

The module has a divider with a DELAY of 2604, which will assert an enable called baudtick at a 9600 baudrate. RS232_transmit consists of twelve states, one for each bit it sends as well as the idle state. The module waits in the idle state, transmitting a 1, until it receives the enable signal. At this point, it moves to the start state, and transmits a 0. At this point, it changes states every time baudtick is asserted. There are eight data states, during which the module transmit the corresponding data bit. The last two states are stop1 and stop2, during which the transmitter sends two 1's as stop bits. It then returns to the idle state to await another enable signal.

```
module RS232_transmit (input clk, transmit, input [7:0] data,
                       output reg serial);
   wire baudtick;
   divider #(.DELAY(2604)) baudrate (.clk(clk), .start_timer(0),
                                     .enable(baudtick));

   parameter idle = 11;
   parameter start = 0;
   parameter data1 = 1;
   parameter data2 = 2;
   parameter data3 = 3;
   parameter data4 = 4;
   parameter data5 = 5;
   parameter data6 = 6;
   parameter data7 = 7;
   parameter data8 = 8;
   parameter stop1 = 9;
   parameter stop2 = 10;

   reg [3:0] state, next_state;
   reg [10:0] to_send;
   always @(*) begin
      case(state)
        idle: next_state = (transmit) ? start : state;
        start: next_state = (baudtick) ? data1 : state;
```

```
        data1: next_state = (baudtick) ? data2 : state;
        data2: next_state = (baudtick) ? data3 : state;
        data3: next_state = (baudtick) ? data4 : state;
        data4: next_state = (baudtick) ? data5 : state;
        data5: next_state = (baudtick) ? data6 : state;
        data6: next_state = (baudtick) ? data7 : state;
        data7: next_state = (baudtick) ? data8 : state;
        data8: next_state = (baudtick) ? stop1 : state;
        stop1: next_state = (baudtick) ? stop2 : state;
        stop2: next_state = (baudtick) ? idle : state;
        default: next_state = idle;
      endcase // case(state)
   end // always @ (posedge clk)

   always @(posedge clk) state <= next_state;

   always @(*) begin
      to_send <= {2'b11, data[7:0], 1'b0};
      if (state == idle) serial <= 1;
      else serial <= to_send[state];
   end
endmodule // RS232_transmit
```

B.3 RS-232 Receiver (Jacky)

Like the transmitter, the RS-232 receiver was never used because it was created for use in the discarded range module. It took as inputs the clock and a serial signal, and outputs an 8-bit data signal and an enable called started which indicates when a start bit has been received.

The set up of the receiver is very similar to that of the transmitter. There is a timer with a DELAY of 162 which asserts an enable at sixteen times the baudrate. The module consists of twelve states, one for each bit it receives and the idle state. The idle state is the default state, and the module moves to the start state when the sychronized signal goes low. The low signal is the start bit, and after eight baud16tick enables, it moves to the data1 state. There is a timer that counts down from sixteen, using baud16tick as its enable, and when its expired signal, sixteen_count is asserted, the module moves to the next state and the incoming signal is sampled. The module returns to the idle state after the second stop bit is received and remains there until it gets the next start bit. A ring buffer is used to store the incoming data.

```
module RS232_receive (input serial, clk, output reg [7:0] data,
```

```
                    output reg started);
wire baud16tick;
divider #(.DELAY(162)) baudclk (.clk(clk), .start_timer(0),
                                 .enable(baud16tick));

reg [1:0] synced;
always @(posedge clk) if (baud16tick) synced <= {synced[0], serial};

parameter idle = 4'b0000;
parameter start = 4'b0001;
parameter data1 = 4'b1000;
parameter data2 = 4'b1001;
parameter data3 = 4'b1010;
parameter data4 = 4'b1011;
parameter data5 = 4'b1100;
parameter data6 = 4'b1101;
parameter data7 = 4'b1110;
parameter data8 = 4'b1111;
parameter stop1 = 4'b0010;
parameter stop2 = 4'b0011;

wire [14:0] baudcount;
reg [3:0]   state, next_state;
reg [3:0]   eight_count = 0;
reg         timerp;
wire        sixteen_count;
timer sixteen(.start(timerp), .enable(baud16tick), .clk(clk), .value(16),
              .count(baudcount), .expired(sixteen_count));
always @(*) begin
   case(state)
     idle: next_state = (synced[1]) ? start : state;
     start: next_state = (eight_count == 8) ? data1 : state;
     data1: next_state = (sixteen_count) ? data2 : state;
     data2: next_state = (sixteen_count) ? data3 : state;
     data3: next_state = (sixteen_count) ? data4 : state;
     data4: next_state = (sixteen_count) ? data5 : state;
     data5: next_state = (sixteen_count) ? data6 : state;
     data6: next_state = (sixteen_count) ? data7 : state;
     data7: next_state = (sixteen_count) ? data8 : state;
     data8: next_state = (sixteen_count) ? stop1 : state;
     stop1: next_state = (sixteen_count) ? stop2 : state;
     stop2: next_state = (sixteen_count) ? idle : state;
```

```
      default: next_state = idle;
    endcase // case(state)
  end // always @ (posedge clk)

  always @(posedge clk) begin
    if (state == start && baud16tick) eight_count <= eight_count + 1;
    else eight_count <= 0;
    timerp = (state != idle);
    started <= (synced[1] && (state == idle));
    if (baud16tick) state <= next_state;
    if (baud16tick && sixteen_count && state[3])
      data <= {serial, data[7:1]};
  end
endmodule // RS232_receive
```