Locating Radio Signals Using Software Defined Radio

by
Benjamin Johnson

A THESIS

submitted to
Oregon State University
Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Electrical and Computer Engineering
(Honors Scholar)

Presented August 27, 2021
Commencement June 2022
AN ABSTRACT OF THE THESIS OF


Abstract approved:_____________________________________________________

Kevin McGrath

Locating sources of radio signals can be useful in many applications. From pointing antennas at nearby radio towers to fox hunting, being able to determine the relative location or direction to a radio transmitter can be very useful. Generally, it is necessary to have the current GPS locations of both the transmitter and the receiver. This can sometimes be infeasible, leading to the need to determine the relative direction to a radio transmitter given only the radio signal itself. This thesis designs and implements an embedded system that is able to automatically determine the direction towards a given signal relative to a receiver using only the given signal. The system is able to update to the current relative direction of the radio signals source in real time and display the physical direction.

Key Words: radio frequency, direction finding, embedded system, SDR, software defined radio, FPGA

Corresponding e-mail address: johnsbe3@oregonstate.edu
Locating Radio Signals Using Software Defined Radio

by
Benjamin Johnson

A THESIS

submitted to
Oregon State University
Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Electrical and Computer Engineering
(Honors Scholar)

Presented August 27, 2021
Commencement June 2022
Honors Baccalaureate of Science in Electrical and Computer Engineering project of Benjamin Johnson presented on August 27, 2021.

APPROVED:

____________________
Kevin McGrath, Mentor, representing Electrical Engineering and Computer Science

____________________
Mike Bailey, Committee Member, representing Electrical Engineering and Computer Science

____________________
Sam Quinn, Committee Member, External Committee Member

____________________
Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

____________________
Benjamin Johnson, Author
Acknowledgments

Firstly, I would like to thank Kevin McGrath for being willing to spend his time mentoring me on this thesis and giving me all of the resources and guidance needed to complete it. He was always very supportive and able to guide me in the right direction.

I would also like to thank Mike Bailey for taking the time to meet with me and recommend Kevin McGrath as a thesis mentor and then be willing to serve on the committee.

Next, I want to thank Sam Quinn for his enthusiasm and interest in this thesis from the beginning.

In addition, I would again like to thank Kevin McGrath, Mike Bailey, and Sam Quinn for being so flexible and open with the project.

Lastly, I would like to thank my mom, dad, and brother for always supporting me in my pursuit of an honors thesis.
## Contents

1 Introduction 1

2 Background 1  
   2.1 Time of Flight Triangulation 1  
   2.2 Modulated Signals 4  
   2.3 Software Defined Radio 5  

3 Design 5  
   3.1 Overview 5  
   3.2 FPGA Algorithm 7  
   3.3 Physical Representation of Direction 8  

4 Results 9  
   4.1 Demonstration Results 9  
   4.2 Limitations 10  

5 Conclusion 11  

6 References 11  

7 Appendix 11  
   7.1 FPGA Code 11  
   7.2 C++ Resampling Program Code 18  
   7.3 Arduino Code 23
1 Introduction

Locating where a radio signal is being transmitted from in relation to where it is being received can be useful for a range of applications. Whether it is for positioning an antenna towards a nearby radio tower or finding rouge base stations, it is desirable to be able to locate the source of a radio signal. Generally, if a radio receiver wants to locate the relative location of a radio transmitter to itself the receiver needs to know the GPS location of both itself and the transmitter. With this information the receiver then calculates the relative location. However, obtaining GPS locations can increase the complexity requirements of the receiver or be infeasible in cases where the foreign transmitter does not want to disclose its location. As such, it is desirable to find the relative location of a certain radio transmitter without needing to know the GPS location of the transmitter.

This thesis developed and tested a system that can automatically determine the location of a radio transmitter relative to the physical system given only the transmitter’s radio signal and its carrier frequency. The system is a proof-of-concept design that can calculate the relative location of a radio signal’s transmitter in real time. The system takes advantage of software defined radio to process its direction finding.

2 Background

2.1 Time-of-Flight Triangulation

There are several ways to triangulate the relative position of a radio signal depending on the setup of the receiver. The triangulation method focused on in this thesis is the time-of-flight method. This method takes advantage of the fact that light takes different amounts of time to travel different total distances.

If two antennas are placed at a fixed distance apart and are setup to receive the same signal you can compare the two signals that are received by the antennas to determine the relative direction that the signal is coming from. While the contents of the two received signals are the same there may be a time shift between them. Depending on where the source of the signal is relative to the two antennas will determine how much of, if any, offset there is between the two signals received by the antennas.

Figure 1a shows the case where the source of the radio signal, S, is equal distance in between the two antennas, A and B. Since the distances between the two antennas and the transmitter are equal there will be no offset between the two received signals as the transmitted signal would take the same amount of time to reach both antennas.
Figure 1a: Time-of-Flight antenna setup with source equal distance from receivers (Angle and phase not to scale)

Figure 1b shows the case where the source of the radio signal is closer to antenna A than to antenna B. The radio signal will arrive at antenna A before it arrives at antenna B as it has to travel a further distance. As such, the received signal at antenna A will be time shifted forward relative to the received signal at antenna B. The further the source of the signal goes to the left in the diagram the greater the time shift will be between the two signals.
Finally, Figure 1c shows the case where the source of the radio signal is close to antenna B than it is to antenna A. This time the signal will arrive at antenna B before it arrives at antenna A which means that the received signal at antenna B will be ahead of the received signal at antenna A.
The magnitude of the time shift between the two received signals will be equal when the signal source is placed at equal distances away from the dividing axis between the two receiving antennas. In addition, this method of direction finding relies on the assumption that the signal transmitter is in front of the two receiving antennas. This is because the time shifts would look identical if the transmitter at a certain spot in front of the antennas or that spot mirrored across the axis of the antennas.

**2.2 Modulated Signals**

Most radio signals are not directly transmitted at their base frequency. Instead, they are modulated in some way and then transmitted. There are three main methods of modulation, amplitude, phase, and frequency. You can modulate any of the three properties of a radio signal to encode information within the signal.

This can be taken one step further with quadrature signals. Two signals that have a phase difference of 90 degrees are said to be in quadrature. The in-phase component is said to be the I component while the signal that is phase shifted 90 degrees is said to be the Q component. Figure 2 shows an example of quadrature signals.
Quadrature signals can in turn be individually amplitude modulated by any information signal and then added together to generate the final modulate radio signal. This thesis relied on this principle to analyze a given radio signal.

2.3 Software Defined Radio

Traditionally, radios are implemented in hardware and operate entirely in the analog space. This can be useful for specific use cases where the properties of the radio are somewhat basic and predefined. However, if there is any processing that the radio must do that is not common it can be very difficult to implement in hardware. This is where software defined radio has an advantage over traditional radios implemented in hardware. A software defined radio (SDR) is a radio implemented in software instead of hardware. There is still a physical antenna or array of antennas and receivers that sample the radio signal but all of the samples are digital and processed in software. This allows for any complex processing of radio signals to happen on almost any computer at the software level and can be adjusted or iterated on very easily.

3 Design

3.1 Overview

The system designed and implemented in this thesis is an embedded system consisting of an SDR module with a radio transceiver and FPGA, an SDR program on a computer, and a microcontroller module that drives a servo. All of the software running on the computer and the accompanying microcontroller and servo are for demonstration purposes to display the processed direction data. The actual process of locating the source of a given radio signal is
done entirely in the SDR module at the level of the transceiver and the FPGA. Figure 3 shows a diagram of the top level design of the system.

![Top Level System Block Design](image1)

**Figure 3: Top Level System Block Design**

The SDR module used for the system is the “bladeRF 2.0 micro xA9” from nuand. The bladeRF contains an Analog Devices AD9361 RF Transceiver with a frequency range of 47MHz to 6GHz and a maximum sampling rate of 61.44MHz. The bladeRF also contains an Intel Cyclone V E FPGA that is fully programmable and handles the actual processing of the signal data. All of the calculations for the direction finding are done on the FPGA itself.

![bladeRF 2.0 micro xA9](image2)

**Figure 4: Image of bladeRF 2.0 micro xA9 from nuand product site**

Once the data is processed on the bladeRF’s FPGA it is then sent via a USB connection to a GNU Radio program on a normal computer where it is resampled and averaged before being
sent off to a microcontroller, in this case an Arduino Uno Rev3, which drives a servo motor, an SG90, to point in the relative direction of the radio signal’s source.

The embedded system locates the relative direction of a given radio signals source in real time with the main limiter being the time it takes for the servo motor to rotate to the desired position. The main direction finding algorithm implemented on the FPGA runs continuously several thousand times a second.

### 3.2 FPGA Algorithm

The main algorithm that determines the relative location of the radio signal source is done entirely inside the bladeRF. The bladeRF takes in the radio signal through its two receiver antennas attached to its RX1 and RX2 ports. The base radio signal is then heterodyned with a local oscillator at the carrier frequency of the radio signal, for example 313MHz. The signal from the local oscillator is split into two signals and phase offset by 90 degrees, these signals are referred to as quadrature signals. Then, the received signals from the antenna are split into two and multiplied by both of the reference quadrature signals. The result of this heterodyne is two sets of I and Q values, one set per RX channel, that represent the two signals that carry the actual information being transmitted by the radio source. Figure 5 shows a diagram of the process used to demodulate the received radio signals.

![Figure 5: Receiver demodulation process](image)

At this point all of the I and Q signals are still analog and must be converted into digital signals to then be processed. This is done with analog to digital converters (ADCs). The ADCs in the bladeRF output 12-bit digital samples at the rising edge of the RX sample clock. This sample rate must be close enough to the frequency of the I and Q signals being transmitted by the source. Some experimenting may be required initially to determine the best sample rate for a specific signal.
After the transceiver in the bladeRF converts the base radio signal into digital I and Q samples the samples are read into the Cyclone V FPGA on the bladeRF. The FPGA clocks in IQ samples on the rising edge of each RX clock pulse. The default bitstream for the FPGA simply took the IQ samples and passed them through the USB interface to whatever SDR program is running on the respective computer. For this thesis the FPGA was custom programmed to implement the direction finding algorithm and pass on the results to the connected computer.

The custom FPGA bitstream reads in the IQ samples from both RX channels then stores them into system memory. It does this for each sample in a window of 120 samples. After the 120 samples have been stored in memory the cross correlation of each corresponding I or Q signal is taken. The cross correlation of two signals calculates the how similar the two signals are when one is time shifted a certain amount. Figure 6 shows an example result of a cross correlation of two signals.

![Figure 6: Example result of cross correlation by TinyPebble from Wikipedia](image)

The result of the cross correlation is a signal whose value shows how much the signals match when offset by a given amount. The cross correlation of either the two I signals or the two Q signals are taken in the index range of -60 to +60. Then the maximum value of the cross correlation is found, the index of the maximum value is the time offset between the two signals relative to which is defined as the reference signal. In this case the signal from the RX1 port is used as the reference signal so time offsets are relative to it. This time offset is directly correlated to the direction of the radio signal source simply offset and scaled but it is dependent on the assumption that the source is in front of the axis created by the two antennas.

The resulting time offset generated from the cross correlation is then sent off to the computer to scale and display.
3.3 Physical Representation of Direction

Once the FPGA on the bladeRF has calculated the scaled version of the relative distance it is sent off to the connected computer to be resampled and scaled before being used to drive the position of a servo motor. Initially, the direction information is read into the connected computer by a GNU Radio program that reads it in at the sample rate of the bladeRF. The GNU Radio program then resamples the data so it is only receiving direction a few thousand times a second. Figure 7 shows the flowgraph of the GNU Radio program used.

![GNU Radio program flowgraph](image)

Figure 7: GNU Radio program flowgraph

After the GNU radio resampled the direction data it sent it through a UDP socket to a C++ program that averages the direction data over a period of half a second to get rid of minor errors or noise in the cross correlation. After it has averaged the direction data the C++ program sends it through a serial port to the Arduino microcontroller. The microcontroller then scales and offsets the direction data to be physically accurate before driving the servo motor to point in that direction. The averaging over a period of half a second is arbitrary and could be made less or more depending on noise.

4 Results

4.1 Demonstration Results

For the main demonstration of the design a Toyota Camry key fob was used as the given radio signal source. Based on its FCC ID it is known that it transmits a modulated signal with a carrier frequency between 312.1MHz and 314.35MHz. After some initial experimentation it
was determined that the ideal sampling rate for the IQ signals was at 40MHz. Next, more experimentation was done to determine the ideal distance between the receiving antennas. It was found that placing the antennas in a line about 74cm apart provided a good, clean signal that had a strong cross correlation.

The system was then setup with the above mentioned parameters and was able to accurately locate the key fob when it was transmitting a signal in front of the antennas. It was able to update its calculated direction for the key fob correctly every half a second and move the servo motor to point to where the key fob was. Figure 8 shows the physical layout of the demonstration system.

![Diagram of demonstration system](image.png)

Figure 8: Layout of demonstration system

### 4.2 Limitations

While the implemented system was able to accurately locate and display the relative direction the key fob was in for the demonstration there was a lot of experimentation and fine-tuning that needed to happen before a clean result was achieved. The main issue with the system was noise. It was very difficult to read in a clean signal to the bladeRF. The main issues seemed to be that the power output of the key fob was relatively low and that there was exceptional amount of multipathing. The antennas were not able to pick up a clean signal when the key fob was too far away. In addition, reflections off of nearby surfaces caused the signal to appear as if it was coming from a different place than it actually was. The number of reflections seen in the signal varied greatly with the exact location the system was at. The best locations tended to have very little metal in the surrounding area and was relatively wide open.
In addition to the limitations in obtaining a noise free signal, there is the issue of only being able to receive signals that operate in the bladeRF’s frequency and sampling range. If the carrier or modulated signals are too high or too low in frequency the transceiver simply will not be able to sample it accurately.

5 Conclusion

As the demonstration shows, it is possible to create an embedded system that is able to automatically determine the relative direction of a given radio signal’s source. The most notable part of the specific design for this thesis is that the bulk of the SDR processing is done on a custom programmed FPGA. The speed and low latency of the FPGA allows for reliable timing information associated with the received signals. This, in turn, allows for accurate calculation of the time offset between the two receiver channels. While the design implemented in this thesis does function properly there are very distinct limitations to it that would need to be accounted for in future implementations of this design.

6 References


7 Appendix

7.1 FPGA Code

-- This file defines the main VHDL module that
-- runs the algorithm to find the actual relative
-- angle of the radio source relative to the
-- antenna

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use ieee.math_complex.all;

library work;
use work.bladerf;
use work.bladerf_p.all;
use work.fifo_readwrite_p.all;

entity algorithm is
  port (  
    -- main sample clock  
    rx_clock : in std_logic;
    sys_clock : in std_logic;
    
    -- data streams  
    adc_streams : in sample_streams_t(0 to 1):=(others => ZERO_SAMPLE);
    
    -- modified output data streams, has my angle measurements  
    out_streams : out sample_streams_t(0 to 1):=(others => ZERO_SAMPLE)
  );
end entity;

architecture alg of algorithm is
  
  -- final angle value as a result of algorithm  
  -- initial value should be half of data registers length  
  signal angle : signed(15 downto 0) := (others => '0');
  signal final_angle : signed(15 downto 0) := (others => '0');
  
  -- intermediate values for the algorithm  
  -- signal i_offset : signed(15 downto 0) := (others => '0');
  -- signal j_offset : signed(15 downto 0) := (others => '0');
  signal product : signed(31 downto 0) := (others => '0');
  signal max : signed(31 downto 0) := (others => '0');
  signal running_total : signed(31 downto 0) := (others => '0');
  signal i : signed(15 downto 0) := (others => '0');
  signal j : signed(15 downto 0) := (others => '0');
  
  -- q stuff  
  signal q_product : signed(31 downto 0) := (others => '0');
signal q_max : signed(31 downto 0) := (others => '0');
signal q_running_total : signed(31 downto 0) := (others => '0');
signal q_angle : signed(15 downto 0) := (others => '0');
signal q_final_angle : signed(15 downto 0) := (others => '0');

-- counters for finding max value in algorithm search
signal counter : signed(15 downto 0) := (others => '0');

-- generated output stream that impersonates the RX stream
signal out_stream : sample_streams_t(0 to 1) := (others => ZERO_SAMPLE);

--ram module signals
signal address_a : std_logic_vector(7 downto 0) := (others => '0');
signal address_b : std_logic_vector(7 downto 0) := (others => '0');
signal ram_in_a : std_logic_vector(15 downto 0) := (others => '0');
signal ram_in_b : std_logic_vector(15 downto 0) := (others => '0');
signal wren_a : std_logic := '1';
signal wren_b : std_logic := '1';
--signal ram_inclock : std_logic := '0';
--signal ram_outclock : std_logic := '0';
signal ram_out_a : std_logic_vector(15 downto 0) := (others => '0');
signal ram_out_b : std_logic_vector(15 downto 0) := (others => '0');

--Q ram module signals
signal q_address_a : std_logic_vector(7 downto 0) := (others => '0');
signal q_address_b : std_logic_vector(7 downto 0) := (others => '0');
signal q_ram_in_a : std_logic_vector(15 downto 0) := (others => '0');
signal q_ram_in_b : std_logic_vector(15 downto 0) := (others => '0');
--signal q_ram_inclock : std_logic := '0';
--signal q_ram_outclock : std_logic := '0';
signal q_ram_out_a : std_logic_vector(15 downto 0) := (others => '0');
signal q_ram_out_b : std_logic_vector(15 downto 0) := (others => '0');

--stuff to check if good signal
signal sum : signed(31 downto 0) := (others => '0');
signal valid : std_logic := '1';
signal valid_out : std_logic := '0';

begin

-- populate output stream
--out_stream(1).data_i <= adc_streams(1).data_i;
--out_stream(1).data_q <= adc_streams(1).data_q;
out_stream(1).data_v <= adc_streams(1).data_v;

out_stream(0).data_v <= adc_streams(0).data_v;
--out_stream(0).data_q <= (others => '0');
--leave the channel 0 i and q data to be filled by algorithm

-- assign final output stream
out_streams <= out_stream;

-- instantiate ram module
ram_module : entity work.ram2port
  port map (  
    address_a => address_a,  
    address_b => address_b,  
    inclock => rx_clock,  
    outclock => rx_clock,  
    data_a => ram_in_a,  
    data_b => ram_in_b,  
    wren_a => wren_a,  
    wren_b => wren_b,  
    q_a => ram_out_a,  
    q_b => ram_out_b
  );

-- instantiate Q ram module
Q_ram_module : entity work.ram2port
  port map (  
    address_a => q_address_a,  
    address_b => q_address_b,  
    inclock => rx_clock,  
    outclock => rx_clock,  
    data_a => q_ram_in_a,  
    data_b => q_ram_in_b,  
    wren_a => wren_a,  
    wren_b => wren_b,  
    q_a => q_ram_out_a,  
    q_b => q_ram_out_b
  );

-- assign rx streams to RAM inputs
ram_in_a <= std_logic_vector(adc_streams(0).data_i);
ram_in_b <= std_logic_vector(adc_streams(1).data_i);
q_ram_in_a <= std_logic_vector(adc_streams(0).data_q);
q_ram_in_b <= std_logic_vector(adc_streams(1).data_q);

-- set angle outputs
out_stream(0).data_i <= final_angle;
out_stream(0).data_q <= q_final_angle;
-- out_stream(1).data_i <= adc_streams(0).data_i;
-- out_stream(1).data_q <= adc_streams(0).data_q;
out_stream(1).data_i <= (out_stream(1).data_i('RIGHT' => '1', others => '0') when valid_out = '1' else others => '0');
out_stream(1).data_q <= adc_streams(0).data_i;

cross_correlation : process(rx_clock)
  constant HALF_WINDOW_WIDTH : integer := 60;
  constant WINDOW_WIDTH : integer := HALF_WINDOW_WIDTH * 2;
  constant HALF_WINDOW_WIDTH_NEG : integer := HALF_WINDOW_WIDTH - WINDOW_WIDTH;
  constant WINDOW_MAX_VALUE : integer := WINDOW_WIDTH + 1;
  constant HALF_RAM : integer := 128;
begin
  if(rising_edge(rx_clock)) then
    -- make sure in write mode
    if(wren_a = '1' and wren_b = '1') then
      -- data will be clocked into ram
      if(counter <= WINDOW_WIDTH and counter >= 0) then
        -- move to next address
        address_a <= std_logic_vector(to_unsigned(to_integer(counter+1), address_a'length));
        address_b <= std_logic_vector(to_unsigned(to_integer(counter+1), address_b'length) + HALF_RAM);
        -- move to next address for Q values
        q_address_a <= std_logic_vector(to_unsigned(to_integer(counter+1), q_address_a'length));
        q_address_b <= std_logic_vector(to_unsigned(to_integer(counter+1), q_address_b'length) + HALF_RAM);
        counter <= counter + 1;
        -- check good signal and not noise
        sum <= sum + abs(adc_streams(0).data_i);
        -- set valid out bit
        if(valid = '1' and counter >= 1 and counter <= 10) then
          valid_out <= '1';
        else
          valid_out <= '0';
        end if;
      else
        valid_out <= '0';
      end if;
  end if;
end process;
--reset address and disable write
address_a <= (others => '0');
address_b <= std_logic_vector(to_unsigned(HALF_WINDOW_WIDTH + HALF_RAM, address_b'length));
--q value stuff
q_address_a <= (others => '0');
q_address_b <= std_logic_vector(to_unsigned(HALF_WINDOW_WIDTH + HALF_RAM, address_b'length));
counter <= (others => '0');
i <= (others => '0');
j <= (others => '0');
max <= (max'LEFT => '1', others => '0');
angle <= to_signed(HALF_WINDOW_WIDTH_NEG, angle'length);
running_total <= (others => '0');
--q stuff
q_max <= (max'LEFT => '1', others => '0');
q_angle <= to_signed(HALF_WINDOW_WIDTH_NEG, angle'length);
q_running_total <= (others => '0');
wren_a <= '0';
wren_b <= '0';
--check if good signal
if(sum > 12000) then
  --valid signal
  sum <= (others => '0');
  valid <= '1';
else
  --just noise
  sum <= (others => '0');
  valid <= '0';
end if;
end if;
--make sure in calculate mode
elsif(wren_a = '0' and wren_b = '0') then
  --multiple and add to running total
  --product <= signed(ram_out_a) * signed(ram_out_b);
  running_total <= running_total + signed(ram_out_a) * signed(ram_out_b);
  --q stuff
  q_running_total <= q_running_total * signed(q_ram_out_a) * signed(q_ram_out_b);
  --update angle value
  if(j + HALF_WINDOW_WIDTH - i = WINDOW_WIDTH or j = WINDOW_WIDTH) then
    --compare to previous max
    if(running_total > max) then
      --set new max and angle values
      max <= running_total;
      running_total <= (others => '0');
  end if;
angle <= i - HALF_WINDOW_WIDTH;
end if;
--q stuff
if(q_running_total > q_max) then
--set new max and angle values
q_max <= q_running_total;
q_running_total <= (others => '0');
q_angle <= i - HALF_WINDOW_WIDTH;
end if;
end if;
--update address values and j for next time
if(j + HALF_WINDOW_WIDTH - i = WINDOW_WIDTH or j = WINDOW_WIDTH) then
--end of j loop, reset running total
running_total <= (others => '0');
--q stuff
q_running_total <= (others => '0');
if(HALF_WINDOW_WIDTH - i - 1 < 0) then
--start in negative numbers, bring up to zero
address_a <= std_logic_vector(to_unsigned(to_integer(i + 1 - HALF_WINDOW_WIDTH), address_a'length));
address_b <= std_logic_vector(to_unsigned(HALF_RAM, address_b'length));
--q stuff
q_address_a <= std_logic_vector(to_unsigned(to_integer(i + 1 - HALF_WINDOW_WIDTH), q_address_a'length));
q_address_b <= std_logic_vector(to_unsigned(HALF_RAM, q_address_b'length));
j <= i + 1 - HALF_WINDOW_WIDTH;
else
--starts in range
address_a <= (others => '0');
address_b <= std_logic_vector(to_unsigned(to_integer(HALF_WINDOW_WIDTH - i - 1 + HALF_RAM), address_b'length));
--q stuff
q_address_a <= (others => '0');
q_address_b <= std_logic_vector(to_unsigned(to_integer(HALF_WINDOW_WIDTH - i - 1 + HALF_RAM), q_address_b'length));
j <= (others => '0');
end if;
i <= i + 1;
else
--not end of j loop, increment
address_a <= std_logic_vector(to_unsigned(to_integer(j + 1), address_a'length));
address_b <= std_logic_vector(to_unsigned(to_integer(j + 1 + HALF_WINDOW_WIDTH - i + HALF_RAM), address_b'length));

-- q stuff
q_address_a <= std_logic_vector(to_unsigned(to_integer(j + 1), q_address_a'length));
q_address_b <= std_logic_vector(to_unsigned(to_integer(j + 1 + HALF_WINDOW_WIDTH - i + HALF_RAM), q_address_b'length));

j <= j + 1;
end if;

-- update i value
if (i = WINDOW_WIDTH and (j + HALF_WINDOW_WIDTH - i = WINDOW_WIDTH or j = WINDOW_WIDTH)) then
    -- done with calculation
    i <= (others => '0');
j <= (others => '0');
wren_a <= '1';
wren_b <= '1';

-- check if good signal
if (valid = '1') then
    -- good signal
    final_angle <= angle;
    -- q stuff
    q_final_angle <= q_angle;
    end if;
    end if;
    end if;
end if;
end process;
end architecture;

7.2 C++ Resampling Program Code

#include <iostream>
#include <tchar.h>

/*
   Simple UDP Server
*/

#define _WINSOCK_DEPRECATED_NO_WARNINGS

#include<stdio.h>
```c
#include <winsock2.h>
#include <WS2tcpip.h>

#pragma comment(lib, "ws2_32.lib") //Winsock Library

#define BUFLEN 512 //Max length of buffer
#define PORT 8888 //The port on which to listen for incoming data

//helper function for printing Serial port state
void PrintCommState(DCB dcb)
{
    // Print some of the DCB structure values
    _tprintf(TEXT("\nBaudRate = %d, ByteSize = %d, Parity = %d, StopBits = %d\n"),
                dcb.BaudRate,
                dcb.ByteSize,
                dcb.Parity,
                dcb.StopBits);
}

int main()
{
    // Open serial port
    HANDLE serialHandle;
    BOOL success;
    TCHAR comport[100];
    _tcscpy_s(comport, _countof(comport), _T("COM3"));

    serialHandle = CreateFile(comport, GENERIC_READ | GENERIC_WRITE, 0, 0, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, 0);

    //error handling
    if (serialHandle == INVALID_HANDLE_VALUE)
    {
        // Handle the error.
        printf("CreateFile failed with error %d.\n", GetLastError());
        return (1);
    }

    // Do some basic settings
    DCB serialParams = { 0 };
    serialParams.DCBlength = sizeof(serialParams);

    success = GetCommState(serialHandle, &serialParams);
    if (!success)
    {
        // Do something
    }
```
{  
    // Handle the error.
    printf("GetCommState failed with error %d.\n", GetLastError());
    return (2);
}

PrintCommState(serialParams);

// fill in values
serialParams.BaudRate = 9600;
serialParams.ByteSize = 8;
serialParams.StopBits = ONESTOPBIT;
serialParams.Parity = NOPARITY;

success = SetCommState(serialHandle, &serialParams);
if (!success)
{
    // Handle the error.
    printf("SetCommState failed with error %d.\n", GetLastError());
    return (3);
}

// Set timeouts
COMMTIMEOUTS timeout = { 0 };
timeout.ReadIntervalTimeout = 50;
timeout.ReadTotalTimeoutConstant = 50;
timeout.ReadTotalTimeoutMultiplier = 50;
timeout.WriteTotalTimeoutConstant = 50;
timeout.WriteTotalTimeoutMultiplier = 10;

success = SetCommTimeouts(serialHandle, &timeout);
if (!success)
{
    // Handle the error.
    printf("SetCommTimeouts failed with error %d.\n", GetLastError());
    return (2);
}

// check the serial config again
success = GetCommState(serialHandle, &serialParams);
if (!success)
{
    // Handle the error.
    printf("GetCommState failed with error %d.\n", GetLastError());
    return (2);
}
PrintCommState(serialParams);       // Output to console
_tprintf(TEXT("Serial port %s successfully reconfigured.\n"), comport);

//open UDP socket
SOCKET s;
struct sockaddr_in server, si_other;
int slen, recv_len;
char buf[BUFLEN];
WSADATA wsa;

slen = sizeof(si_other);

//Initialise winsock
printf("\nInitialising Winsock...");
if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
{
    printf("Failed. Error Code : %d", WSAGetLastError());
    exit(EXIT_FAILURE);
}
printf("Initialised.\n");

//Create a socket
if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
{
    printf("Could not create socket : %d", WSAGetLastError());
}
printf("Socket created.\n");

//Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(PORT);

//Bind
if (bind(s, (struct sockaddr*)&server, sizeof(server)) == SOCKET_ERROR)
{
    printf("Bind failed with error code : %d", WSAGetLastError());
    exit(EXIT_FAILURE);
}
puts("Bind done");

//variables for averaging angle measurements
SYSTEMTIME time;
int counter = 0;
double runningtotal = 0;
long starttime = 0;
long currenttime = 0;

GetSystemTime(&time);
starttime = (time.wMinute * 60000) + (time.wSecond * 1000) + time.wMilliseconds;

//keep listening for data
while (1) {

    //clear the buffer by filling null, it might have previously received data
    memset(buf, '\0', BUFLEN);

    //try to receive some data, this is a blocking call
    if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr*)&si_other, &slen)) == SOCKET_ERROR) {
        printf("recvfrom() failed with error code : %d", WSAGetLastError());
        exit(EXIT_FAILURE);
    }

    //convert data to float
    unsigned char buffer[4];
    buffer[0] = buf[0];
    buffer[1] = buf[1];
    buffer[2] = buf[2];
    buffer[3] = buf[3];
    float num = *(float*)buffer;

    //add number to running total
    runningtotal += num;

    //increment count
    counter++;

    //if 1 second has passed, average total with count and send to arduino, reset running total and count
    GetSystemTime(&time);
    currenttime = (time.wMinute * 60000) + (time.wSecond * 1000) + time.wMilliseconds;
    if (abs(currenttime - starttime) >= 500) {
        
    }
//send data to arduino serial port
int angle = (int)(runningtotal / counter); //printf("number to send: %d\n", angle);
char charangle[50];
itoa_s(angle, charangle, 10);
//printf("char to send: %s\n", charangle);
DWORD bytesWritten = 0;
success = WriteFile(serialHandle, charangle, 50, &bytesWritten, NULL);
if (!success) {
    // Handle the error.
    printf("Failed to write data to serial port with error %d.\n", GetLastErr); //s();
}
//printf("byteswritten: %d\n", bytesWritten);
//reset averaging values
counter = 0;
runningtotal = 0;
starttime = currenttime;
}

//close UPD socket
 closesocket(s);
 WSACleanup();

 //close serial port
 CloseHandle(serialHandle);

 return 0;
}

7.3 Arduino Code

#include <Servo.h>

Servo servo;

// runs once on startup
void setup() {
    Serial.begin(9600);
    //attach servo motor to pin on board, any PWM pin will work
    servo.attach(8);
    //set initial angle of servo
```c
void loop() {
    //get angle from computer through serial port
    int angle = 90;
    if (Serial.available() > 0) {
        angle = Serial.parseInt();
        angle = (int)(angle * 1.3);
        angle = angle + 90 - 10;

        delay(50);
        int leftinbuffer = Serial.available();
        int junk = 0;
        if (leftinbuffer > 0) {
            for (int i = 0; i < leftinbuffer; i++) {
                junk = Serial.read();
            }
        }
        //int junk = Serial.read();
        //Serial.write(angle);
        servo.write(angle);
        delay(400);
    }
}
```