ClipOn EMG V-0.0

Akshay Jain

I. Abstract

Surface EMG signals are electrical signals produced due to activity in muscles. These signals can be captured by placing electrodes over the muscles and then processed to detect activity. They can be used for people with disabilities who are not able to control their body parts, but still can move some of the muscles. This project develops a ClipOn EMG system which taps the EMG signals using electrodes connected to muscles of a person, processes them using a simple electronic circuit and transmits them to a host computer using TS 7250 board over a local network wirelessly. The host computer plots the received signals and processes them for detecting two types of gestures performed by the person. Finally, we show an application where the clip on EMG system is used along with another system which helps a speech disabled person to convey his messages through a speak synthesizer.

II. Introduction

There are some wireless sEMG sensor systems available in the market. One developed by a company named Delsys called Trigno wireless [10]. This system supports multiple wireless sensors and each of these has an EMG sensor and an accelerometer attached. The output of these sensors are sent over network to a remote station which is connected to a computer. Though, this seems to be a very well integrated system, it is very expensive. For an application that just requires a raw sEMG signal, it will be an underutilized solution. A system called bio radio [11] is another wireless EMG system available. This system requires the user wearing EMG sensors to keep a processing box along. There are some other systems in the market, but most of them suffer from one or the other drawbacks mentioned above. Our aim is to develop a system similar to Trigno system, but the solution should be simpler in terms of using it and cheaper, which can increase the scope of the application.

III. Theory

Figure 1 shows the different blocks of the system. This section presents a description of each of these hardware and software blocks in detail.

Signal Acquisition

The surface EMG signals is acquired by placing electrodes over the muscles. Using these signals directly is not possible because their amplitude is very small and they contain a lot of noise. Therefore, before providing these signals to a digital system, they are first processed using an analog signal processing circuit. The noise present in an EMG signal can be reduced by acquiring them differentially, i.e, we capture two signals from a muscle by placing two electrodes very close to each other. Because the noise present in a small local neighborhood is considered to be uniform, capturing a differential signal helps us to reduce the noise. The signal acquisition system consists of a differential amplifier and a low pass filter.

Differential Amplifier: The differential amplifier takes the differential EMG signal and converts it to an amplified single signal. The differential amplifier subtracts the two signals provided as input and amplifies the difference. The noise from distant power lines and signals produced by distant muscles are considered to be noise and uniform for the two signals captured. However, the signals produced by the muscle over which the sEMG electrodes are placed is considered to be different for the two channels [1]. Hence, to eliminate the long distance noise and amplify the subtracted signal a differential amplifier is used.

Low Pass Filter: Even after the reduction of noise by the differential, there may still be some high noise present in the signal. This noise is attenuated by using a low pass filter with a cutoff frequency in the range of 100 – 500 Hz which is the frequency range of the EMG signals produced by human body.

EP9301

The EP9301 microprocessor available on the TS 7250 board accepts the digital signal from Max 197 ADC and transmits it over a local network wirelessly with the help of a wireless dongle connected to one of its USB ports. The EP9301 is based on ARM architecture, and specifically it contains 166 MHz ARM920T processor. We use the Ethernet, USB and ADC peripheral interfaces available in EP9301. EP9301 contains a 12 bit, 5 channel successive approximation Analog to Digital convertor with a

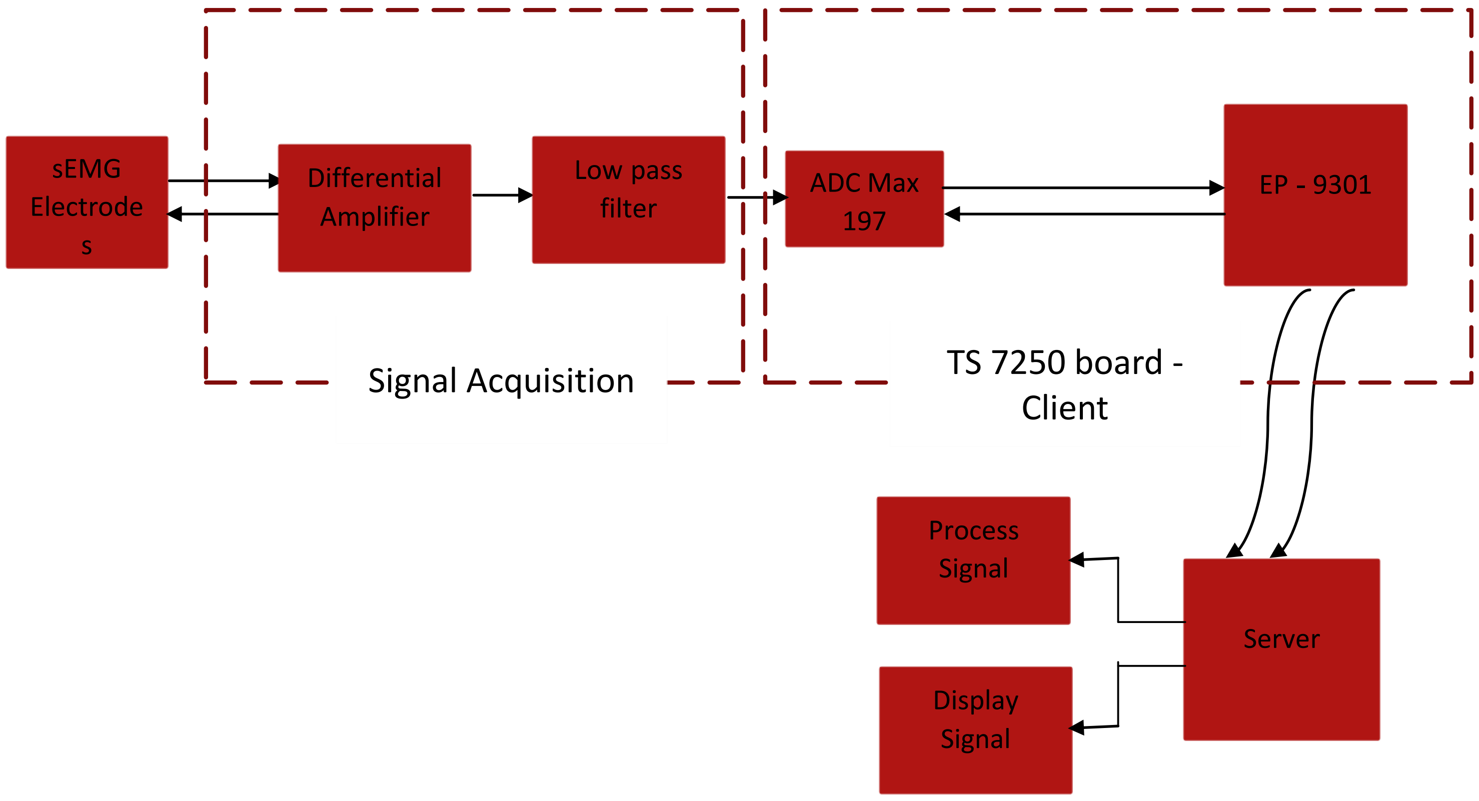


Figure 1: System Overview

maximum sampling frequency of 925 samples per second. We carried out some tests with this ADC to digitize the EMG signal data, but we later switched on to a faster ADC, Max 197.

ADC – Max 197

TS 7250 has an external Max 197, 12 bit, 8 channel successive approximation ADC with a maximum sampling frequency of 60,000 samples per second [3]. It can be configured for both unipolar and bipolar mode with voltage range from +5 to -5 volts in bipolar mode and 0 to +10 volts in unipolar mode. We use this ADC for digitizing our data due to its high sampling frequency.

TCP / IP communication

To transmit the digital signal data over the local network we use TCP / IP communication protocol. This is a reliable communication protocol, therefore every data packet sent is verified by using a checksum, and resent if the data received is not equal to the data sent. In case of missing packet, the packets are sent again. Due to an efficient layer structure of this protocol we only have to deal with the application layer, while the protocol stack implemented in the two systems takes care of the communication. The data is transmitted in packets of 1500 bytes each. Out of these 1500 bytes, 1448 bytes is for data while the remaining 52 bytes are for header.

Circular Buffers

The data transmission over a network can be of non – uniform speed depending on the congestion in the network, but the data transmission rate over a long time can be considered as constant. To avoid data loss due to this communication behavior we use circular buffers to send and receive data. Circular buffers are fixed size single buffers which can be programmed to behave as if the two ends of the buffers were connected to each other forming a ring like structure.

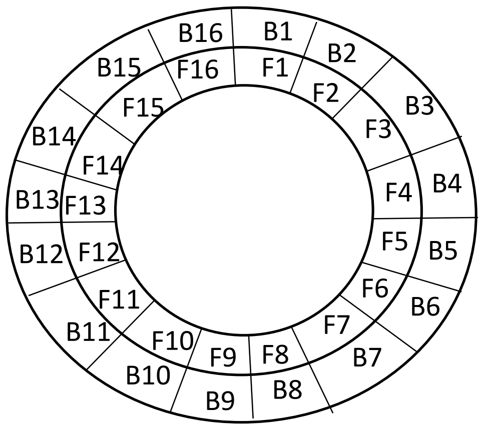


Figure 2: Circular Buffer Design

Figure 2 shows a pictorial representation of a circular buffer we used. The buffer actually is a fixed size traditional buffer. The buffer is divided in N = 16 blocks of ‘B’ bytes each (for example). Each block has a flag (F) associated with it which keeps track whether it is ready to write or read data. This is done to protect the data in a block.

Client / Server Model

We implement the communication between the host computer and TS 7250 in a sever client model. The host, which is the server opens a port to initiate communication while the client which knows the IP address of the server connects to it using sockets.

IV. Implementation

This section presents the implementation details of our system.

Signal Acquisition

Figure 3 shows the circuit diagram [4] used to realize a differential amplifier and a low pass filter. The +-Vcc is used as +-6Volts.

INA 126 [5] is a micro power instrumentation amplifier. It is used to implement a differential amplifier. The output of the differential amplifier from pin 6 of INA 126 is given by

$$V_o = (V_{in}^{+} - V_{in}^{-})*G$$

Where G is the gain, given by

$$G = 5 + 80\text{Kohm}/(R_{17}+R_{18})$$

$$R_{17} = R_{18} = 2.74\text{Kohm}$$

Therefore, $G = 5 + 80/(2.74+2.74) = 19.59$

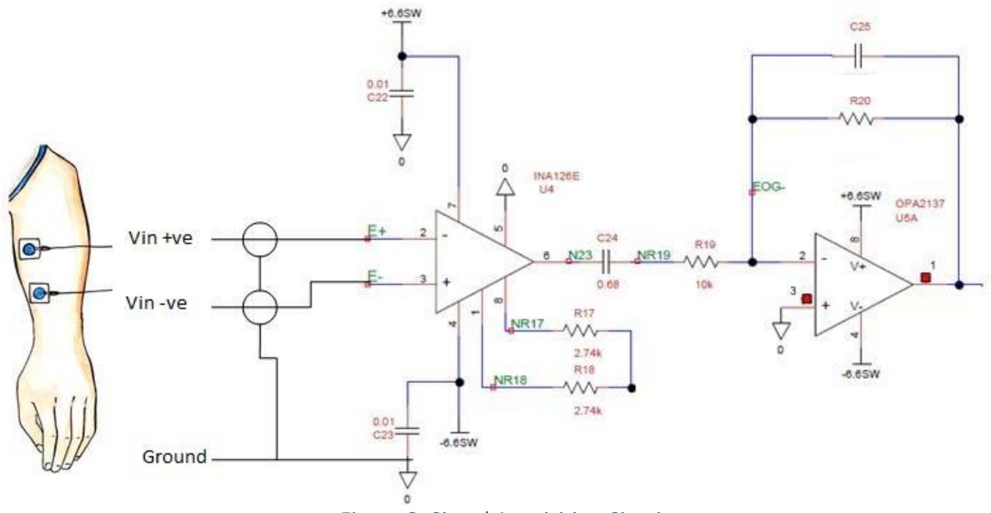


Figure 3: Signal Acquisition Circuit

OPA 2137 [6] is a general purpose operational amplifier which is implemented as a low pass filter. The cut of frequency of the LPF is given by

$$F_c = 1/(2\text{pi}R_{20}*C_{25})$$

$$R_{20} = 470\text{Kohm},\ C_{25} = 1000\text{pf}$$

Therefore, $F_c = 338.6\text{Hz}$

Figure 4 shows a picture of the circuit implemented on a bread board.

The output of differential amplifier is provided to the Max 197 ADC present on the TS 7250 board.

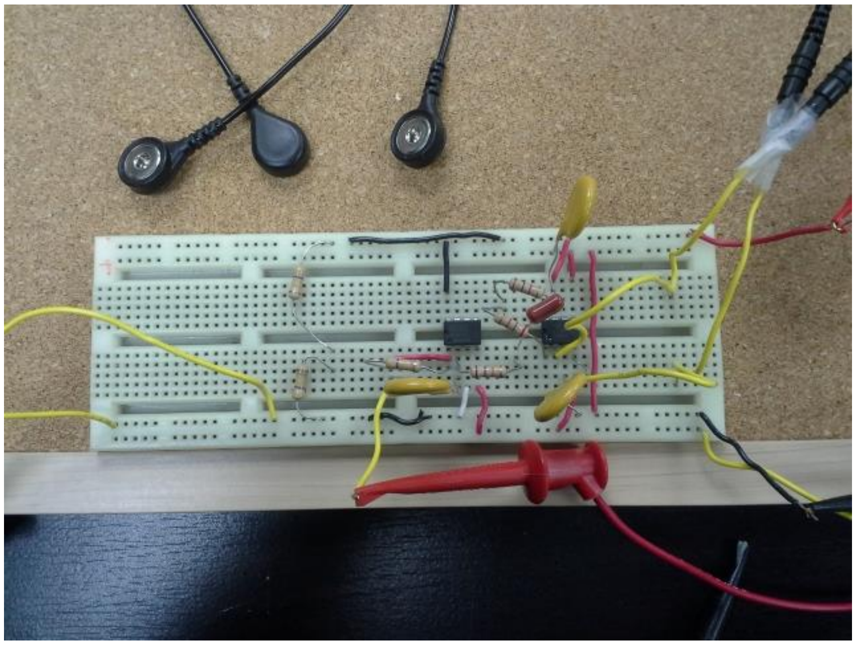


Figure 4: Signal acquisition circuit realized on a breadboard

Client – TS 7250

Figure 5, 6 and 7 shows flowcharts of the main program, thread used to transmit data to the host and the ADC receive data thread respectively.

Main Program

1. The ADC registers are mapped to the OS memory using mmap function. The registers are then configured to accept input signal of range -5 to +5 V on channel 7.
2. The next step is to create a socket to communicate with the server.
3. The program then creates a Circular buffer and initializes the memory to zero to make all the blocks in circular buffer empty. The circular buffer is created using the following structure. The Circular buffer size is configured as 400*500 shorts.

```
typedef struct

{

        short data[BUFFER_SIZE*CIRCULAR_BUFFER_SIZE];

        char full[CIRCULAR_BUFFER_SIZE];

}CIRCULAR_BUFFER;
```

4. Two pthreads are created, one to fetch ADC data and the other to transmit it over TCP IP.

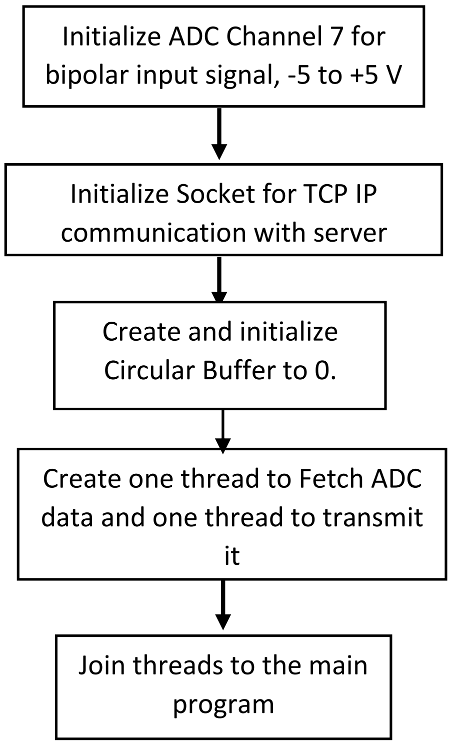

Initialize ADC Channel 7 for bipolar input signal, -5 to +5 V
Initialize Socket for TCP IP communication with server
Create and initialize Circular Buffer to 0.
Create one thread to Fetch ADC data and one thread to transmit it
Join threads to the main program

Figure 5: Flowchart, Client: Main program

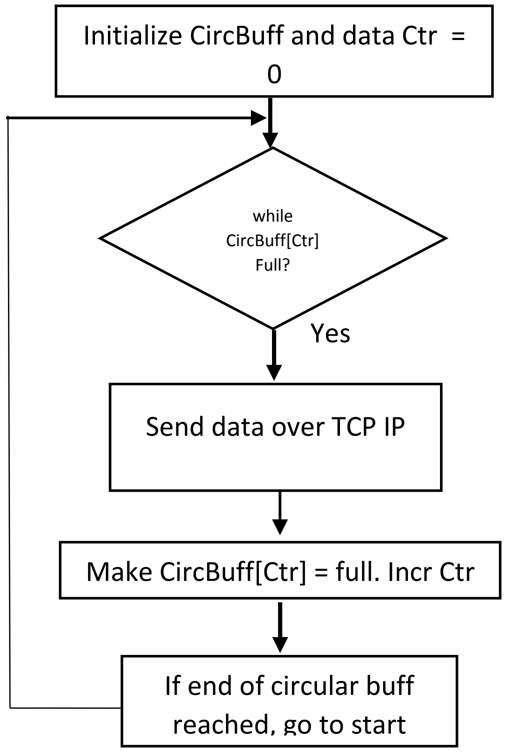

Initialize CircBuff and data Ctr = 0
while CircBuff[Ctr] Full?
Yes
Send data over TCP IP
Make CircBuff[Ctr] = full. Incr Ctr
If end of circular buff reached, go to start

Figure 6: Flowchart, Client: Transmit ADC Data thread

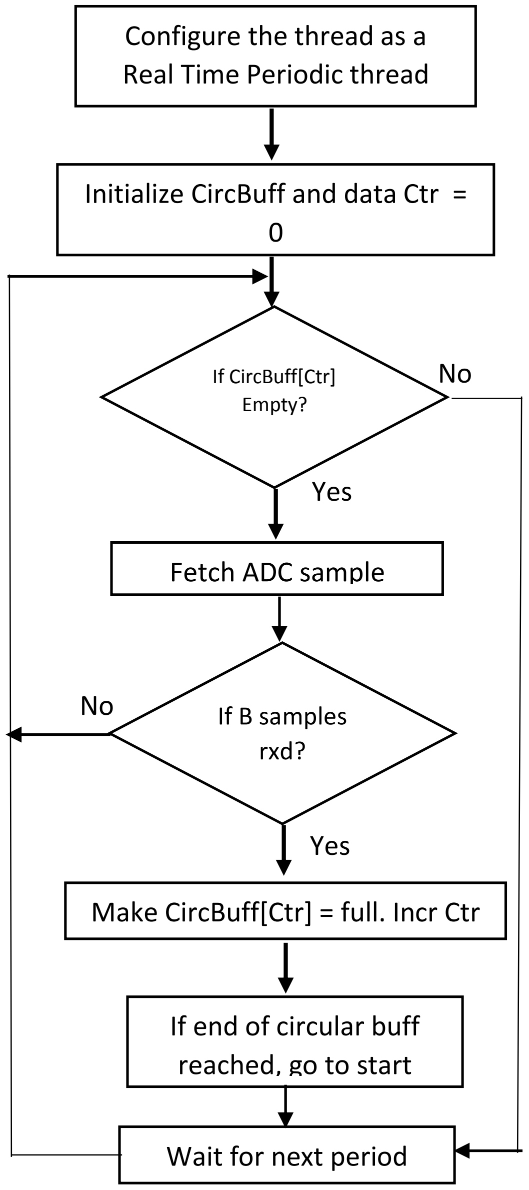

Configure the thread as a Real Time Periodic thread
Initialize CircBuff and data Ctr = 0
If CircBuff[Ctr] Empty?
No
Yes
Fetch ADC sample
If B samples rxd?
No
Yes
Make CircBuff[Ctr] = full. Incr Ctr
If end of circular buff reached, go to start
Wait for next period

Figure 7: Flowchart, Client: Fetch ADC data thread

ADC data Receive thread

This thread fetches samples from ADC data and stores them into circular buffer. Figure 7 shows a flowchart of this thread. The resolution of ADC is 2048/5 = 7.3 mVolts

1. The thread is configured as a real time periodic thread. The period of this thread decides at what sampling frequency we want to grab ADC data. To set a sampling frequency to 4KHz, the period of this thread is set to 1/4000 seconds.

2. On the first execution, the Circular Buffer Counter and data counter are initialized to 0. The Circular Buffer Counter keeps track of which block of circular buffer to fill and data counter keeps track of which index of the current circular buffer block is to be filled with ADC data.
3. Before writing ADC data, we check whether the Circular Buffer block is empty or not. This is to protect data in case when the transmit thread has not transmitted the data till the time when circular buffer completes one cycle.
4. If the Buffer is empty, then we grab the ADC data.
5. Once we have one block worth of data, we make the status flag of the circular buffer block as true, to indicate the transmit thread that the block is ready to be sent.
6. The thread then waits for the next period to arrive.

Data transmit thread

This thread transmits the ADC data stored in the circular buffer. Figure 6 shows a flowchart.

1. On the first execution, the Circular Buffer Counter is initialized to 0. The Circular Buffer Counter keeps track of which block of circular buffer to transmit.
2. Wait until the next block to be sent is made full by the Receive ADC data thread.
3. The data is then sent over TCP IP.
4. The status of the buffer is then made empty.

Server

Main program

Figure 8 shows the flowchart of the main program.

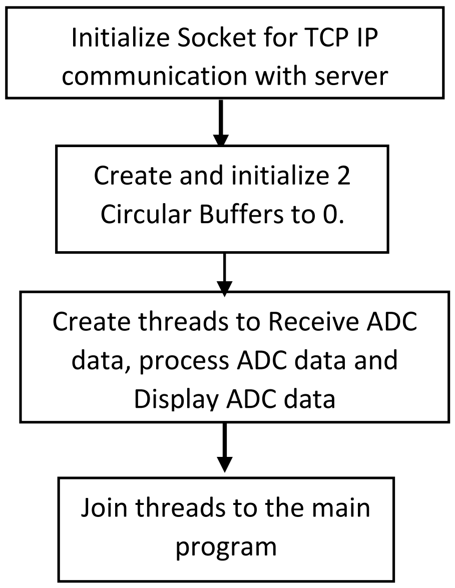

Initialize Socket for TCP IP communication with server
Create and initialize 2 Circular Buffers to 0.
Create threads to Receive ADC data, process ADC data and Display ADC data
Join threads to the main program

Figure 6: Flowchart, Server: Main program

1. The main program initializes the TCP IP socket for communication with the client.

2. Two circular buffers are created. One is used by processing thread while the other is used by display thread. The circular buffer structure is defined as following. The size of the each Circular buffer is 200*500 shorts.

```
typedef struct
{
        short data[BUFFER_SIZE*CIRCULAR_BUFFER_SIZE];
        bool full[CIRCULAR_BUFFER_SIZE];
}CIRCULAR_BUFFER;
```

3. It creates 3 threads, one for receiving data over TCP IP, one to display the received data and one to process the data to detect the gestures. The address of first buffer is passed to the display thread, while the address of second buffer is passed to process thread.

Receive Thread

This thread is responsible for receiving the data over TCP IP and store the data in two circular buffers. Figure 9 shows the flow chart of the receive thread.

1. Initialize the Circular buffer counters for 2 circular buffers.
2. It waits to receive the data in a local buffer.
3. If the current block of circular buffer1 is empty, then copy the data to that block and indicate by changing the status flag to full.
4. If the current block of circular buffer 2 is empty, then copy the data to that block and indicate by changing the status flag to full.

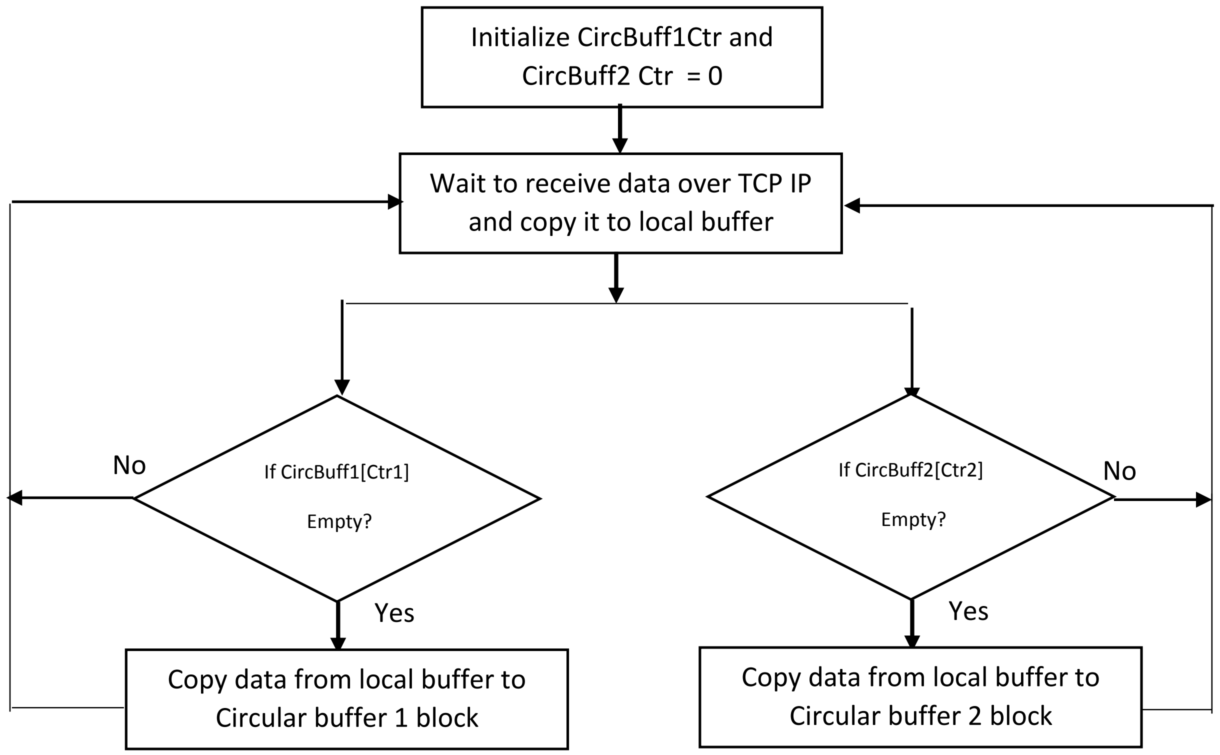

Initialize CircBuff1Ctr and CircBuff2 Ctr = 0
Wait to receive data over TCP IP and copy it to local buffer
If CircBuff1[Ctr1] Empty?
No
Yes
Copy data from local buffer to Circular buffer 1 block
If CircBuff2[Ctr2] Empty?
No
Yes
Copy data from local buffer to Circular buffer 2 block

Figure 7: Flowchart, Server: Receive data over TCP IP thread

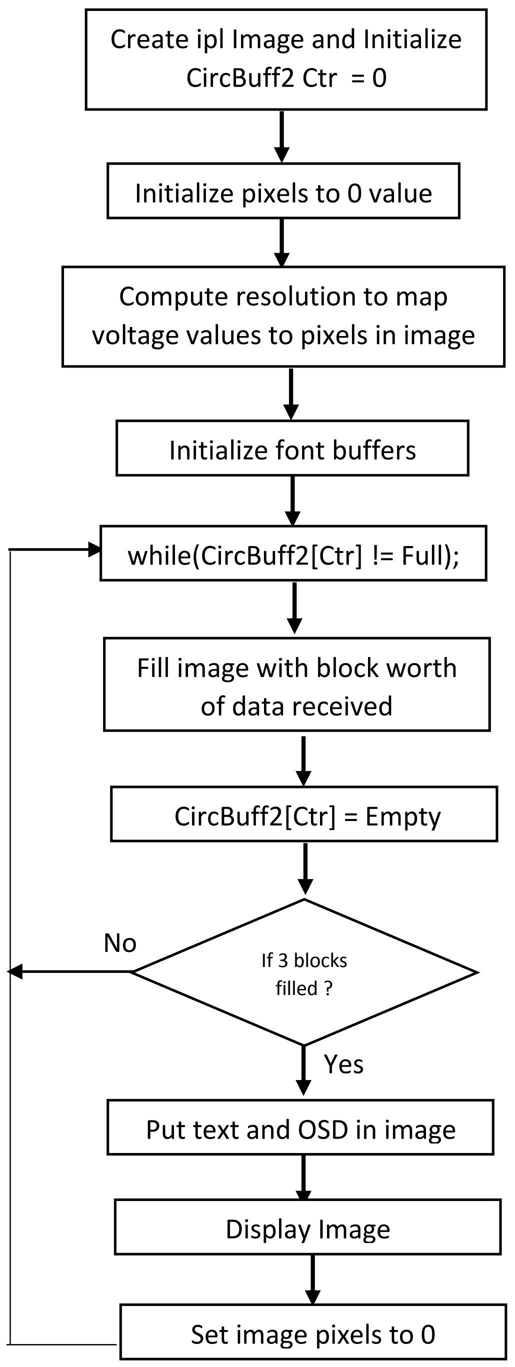

Create ipl Image and Initialize CircBuff2 Ctr = 0
Initialize pixels to 0 value
Compute resolution to map voltage values to pixels in image
Initialize font buffers
while(CircBuff2[Ctr] != Full);
Fill image with block worth of data received
CircBuff2[Ctr] = Empty
If 3 blocks filled ?
No
Yes
Put text and OSD in image
Display Image
Set image pixels to 0

Figure 8: Flowchart, Server: Display thread

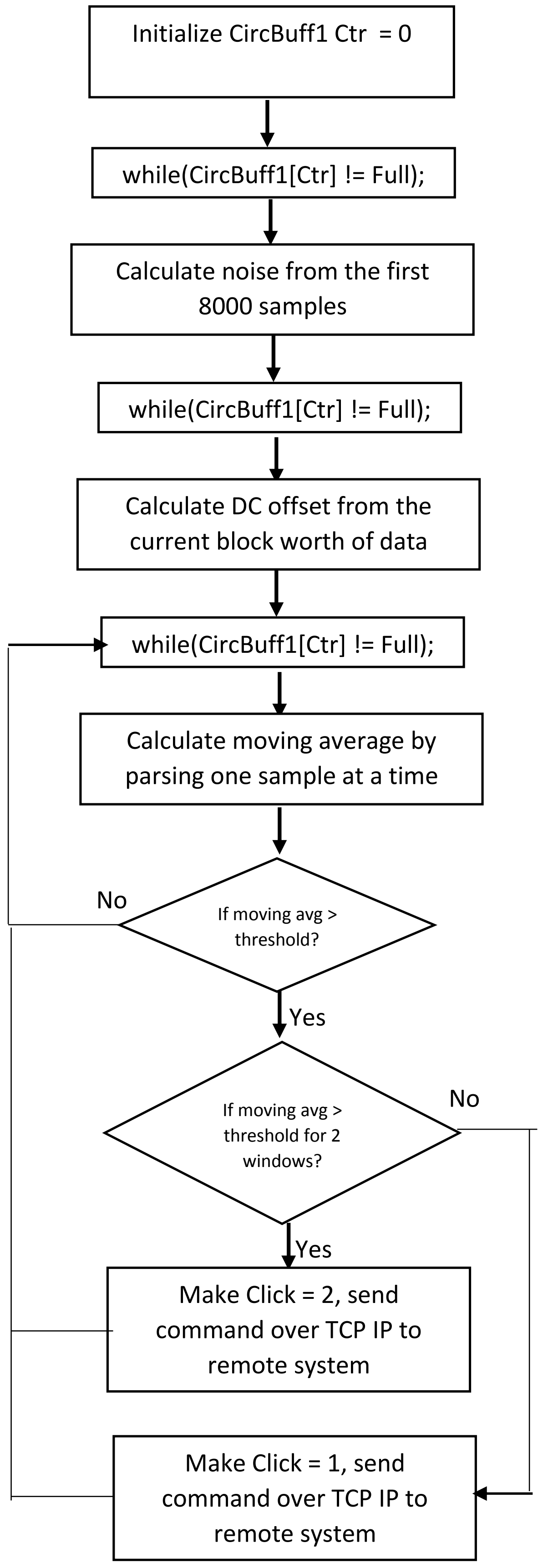

Initialize CircBuff1 Ctr = 0
while(CircBuff1[Ctr] != Full);
Calculate noise from the first 8000 samples
while(CircBuff1[Ctr] != Full);
Calculate DC offset from the current block worth of data
while(CircBuff1[Ctr] != Full);
Calculate moving average by parsing one sample at a time
If moving avg > threshold?
No
Yes
If moving avg > threshold for 2 windows?
No
Yes
Make Click = 2, send command over TCP IP to remote system
Make Click = 1, send command over TCP IP to remote system

Figure 9: Flowchart, Server: Process thread

Display Thread

The display thread performs the task of displaying the received data using OpenCV. Figure 10 shows a flowchart.

1. An IPL image is created of size 1500x500.
2. The image is initially configured to display signals between min and max voltage.
3. After receiving 3 blocks worth of data, the image is updated.

Process Thread

This thread is responsible for processing the received data to detect activity in the received signal. We recognize two kinds of activities. On recognizing an activity a message is displayed on the signal plot and a command is sent over a socket to a remote system. Figure 11 shows a flowchart.

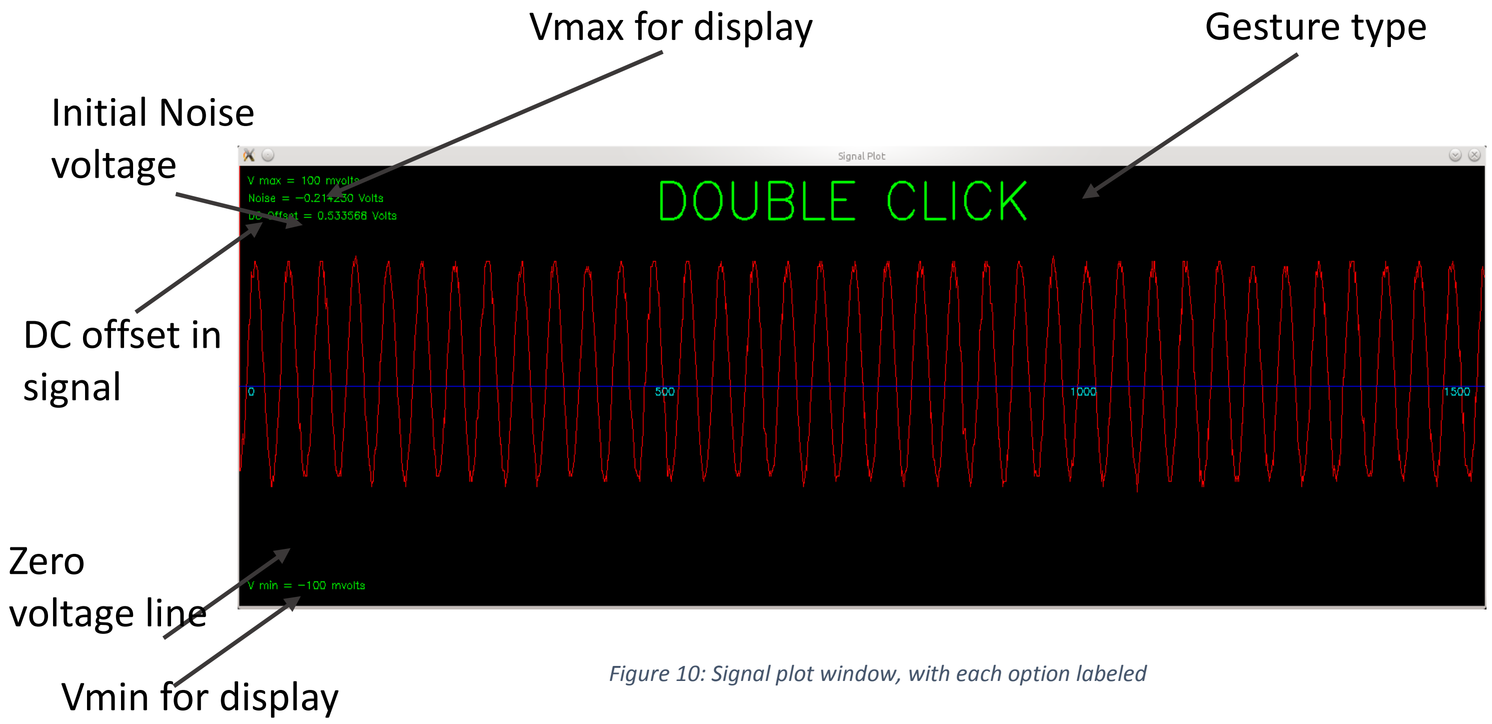
1. The first 8000 samples are used to get an estimate of the noise in the signal.
2. The next 500 samples are used to estimate the DC offset in the signal. This is useful when the input signal does not have a mean of 0 volts.
3. The thread then waits for new samples and for each sample received, moving average is calculated over a window size of 500 samples. If the moving average reaches a threshold, then a gesture is detected. If this detection remains for 2 consecutive window, then a double gesture is detected, otherwise a single gesture is detected. The threshold set is 0.01 V above or below the estimated noise voltage.
4. On detecting an activity a command identifying the activity is sent to a remote system using socket communication.

V. Results

This section presents the results obtained. Figure 12 shows a screenshot of the plot window which displays the received signal. The plot window has various labels as described in the screenshot.

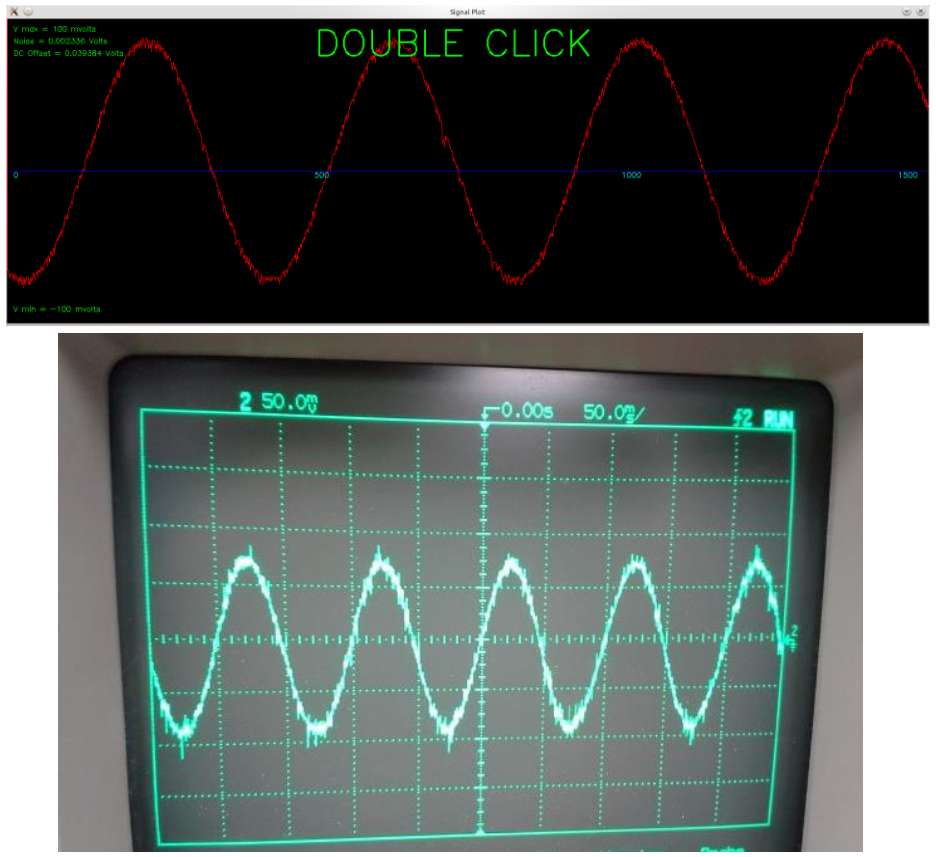
Figure 13 shows comparison between sinusoidal of 80 mv peak to peak voltage and 10 Hz frequency signals as viewed on an Oscilloscope and our plot window.

Figure 14 shows comparison between sinusoidal of 80 mv peak to peak voltage and 70 Hz frequency signals as viewed on an Oscilloscope and our plot window. Also displayed is a FFT plot of the digitized signal.



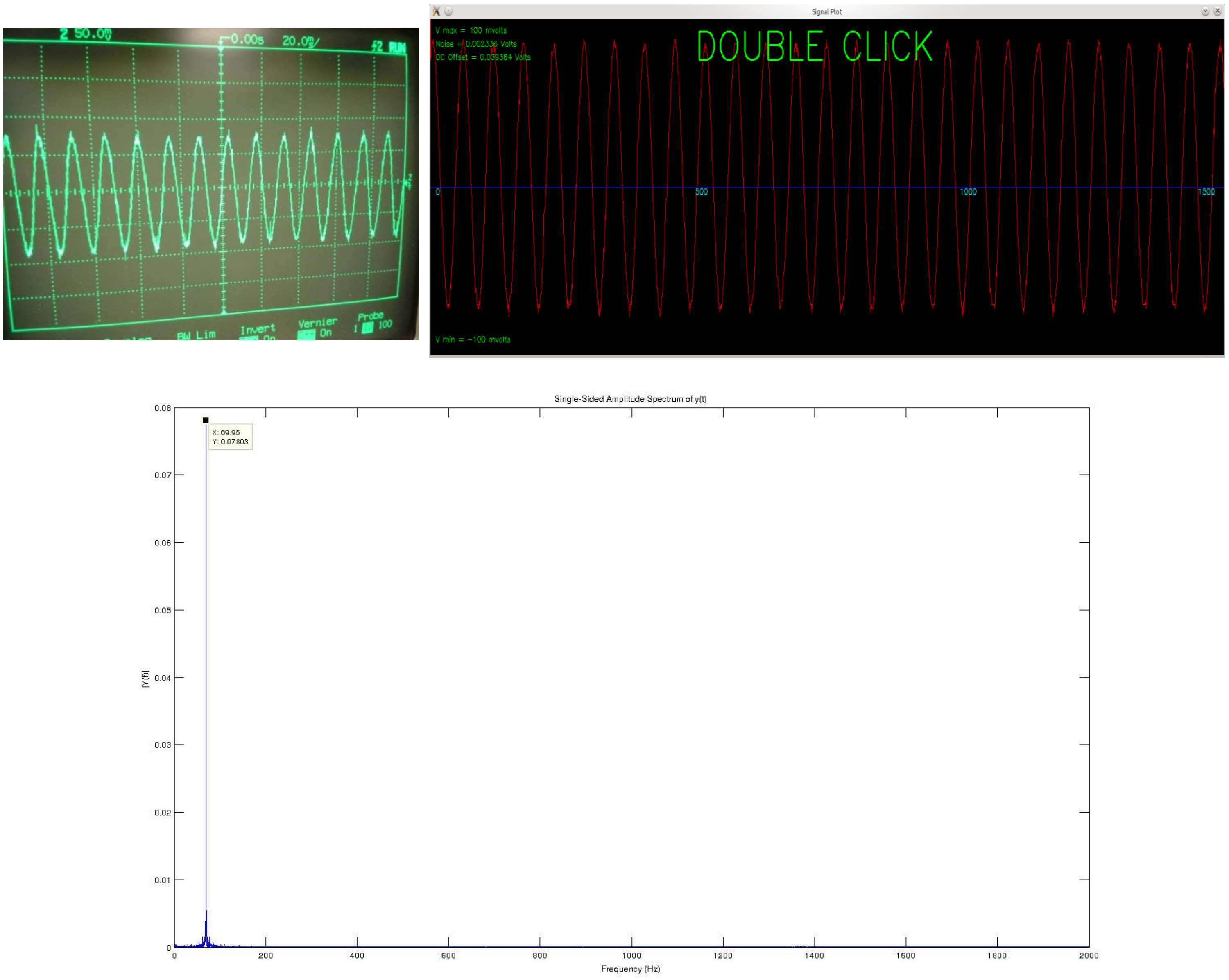
```
V max = 100 mvolts
Noise = -0.214230 Volts
DC Offset = 0.533568 Volts
V min = -100 mvolts
```

Figure 10: Signal plot window, with each option labeled



```
V max = 100 mvolts
Noise = 0.002336 Volts
DC Offset = 0.039384 Volts
V min = -100 mvolts
```

Figure 11: Sinusoid of 80mV p-p and 10 Hz frequency. Top signal on our plot, Bottom – Signal on the oscilloscope



V max = 100 mvolts
Noise = 0.002336 Volts
DC Offset = 0.039384 Volts
V min = −100 mvolts

Figure 12: Sinusoid of 80 mV p-p, 70 Hz frequency. Top left Signal on Oscilloscope, Top Right- Signal on our plot, Bottom - FFT of the received digital signal

VI. Conclusion

It can be observed that the received signal is similar with respect to noise from the original signal as viewed on the Oscilloscope. Because of some inherent noise in the signal, some noise is seen in the received signal. The FFT plot of the received samples show a high peak at the actual frequency of sinusoidal signal.

Please see the attached videos to see that the system is able to detect activities in muscles when some gestures are performed. It is also able to distinguish a single and a double gesture. The system was also integrated to a pillow talker system which receives single and double gesture command and acts accordingly.

VII. Future Work

1. To increase the number of gestures to be recognized we plan to add a longer duration gesture which will be useful for a person using the pillow talker system.
2. Another area of focus is to port the tasks running on a TS 7250 board to a smaller and cheaper board. This is possible because the TS 7250 board is only responsible for receiving analog signal, digitizing it and sending it over local network. Along with this, exploring a communication interface simpler than the currently used WiFi interface should also be useful.
3. We also plan to port the signal acquisition circuit on a PCB, which should give us a less noisy signal.

VIII. References

[1] De Luca, Carlo J. "Surface electromyography: Detection and recording." DelSys Incorporated 10 (2002): 2011.

[2] EP 9301 datasheet

[3] Max 197 datasheet

[4] Tinkertron EMG box

[5] INA 126 datasheet

[6] OPA 2137 datasheet

[7] TS 7250 user manual

[8] Open CV user manual

[9] Rivera, L.A.; DeSouza, G.N., "A power wheelchair controlled using hand gestures, a single sEMG sensor, and guided under-determined source signal separation," Biomedical Robotics and Biomechatronics (BioRob), 2012 4th IEEE RAS & EMBS International Conference on , vol., no., pp.1535,1540, 24-27 June 2012

[10] http://glneurotech.com/TheBioRadio/

[11] http://www.delsys.com/products/trignowireless.html

IX. Appendix – Code

Server Program

Provide the port number as input argument.

```
/**
 *  Copyright (C) 2013 Vision-Guided and Intelligent Robotics Lab
 *  Written by Akshay Jain <aj4g2@mail.missouri.edu>c
 *    Receives ADC data form TS 7250 and process and display
 *  Provide the port number as an arguement to the executable
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation. Meaning:
 *          keep this copyright notice,
 *          do  not try to make money out of it,
 *          it's distributed WITHOUT ANY WARRANTY,
 *          yada yada yada...
 *
 *  You can get a copy of the GNU General Public License by writing to
 *  the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
 *  Boston, MA  02111-1307  USA
 */
#include <sys/stat.h>
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <assert.h>
#include <semaphore.h>
#include <pthread.h>
#include <sys/time.h>
#include <stdbool.h>
#include <cv.h>
#include <highgui.h>
//Size of each block of circular buffer
#define BUFFER_SIZE
      (500)
//Number of blocks per circular buffer
#define CIRCULAR_BUFFER_SIZE                                      (200)
//To convert ADC reading to vtg
#define ADC_TO_VTG
      (0.0024)
//Number of samples to estimate noise
#define THR_WIN
8000/2      // For the threshold window. Set to ~ 2 sec.
#define DEBUG
#ifdef DEBUG
```

```
# define DEBUG_PRINT(x) printf x;fflush(stdout)
#else
# define DEBUG_PRINT(x) do {} while (0)
#endif
//Circular buffer data structure
typedef struct
{
      short data[BUFFER_SIZE*CIRCULAR_BUFFER_SIZE];
      bool full[CIRCULAR_BUFFER_SIZE];
}CIRCULAR_BUFFER;

//Global variables to connect sockets
int sockfd, newsockfd, portno, sockfd2, newsockfd2, portno2;
socklen_t clilen, clilen2;
struct sockaddr_in serv_addr, cli_addr, serv_addr2, cli_addr2;
unsigned int Click = 0, DisplayClickCtr = 0;
void error(const char *msg)
{
    perror(msg);
    exit(1);
}
//Receive the data over socket
void *Rx_adc_data(void *temp)
{
      int n, BlockCtr1 = 0, BlockCtr2 = 0;
      short *pdata1, *pdata2;
      short pdata[BUFFER_SIZE];
      bool *pstatus1, *pstatus2;
      //get address of two circular buffers
      CIRCULAR_BUFFER *temp1 = (CIRCULAR_BUFFER *)(temp);
      CIRCULAR_BUFFER *temp2 = temp1+1;

    while(1)
    {
      //Wait to receive data over network
            n =
recvfrom(newsockfd,pdata,BUFFER_SIZE*sizeof(pdata[0]),MSG_WAITALL,(struct
sockaddr *)&cli_addr,&clilen);
            if (n < 0) error("ERROR reading from socket");
            pstatus1 = ((temp1->full) + (BlockCtr1));
            //If the circular buffer1 block is empty then copy the data to
process
            if (*pstatus1 == false)
            {
                  pdata1 = (short *)(temp1->data) + (BlockCtr1)*BUFFER_SIZE;
                  memcpy(pdata1, pdata, BUFFER_SIZE*sizeof(pdata[0]));
                  *pstatus1 = true;
                  BlockCtr1 = (BlockCtr1 + 1)%CIRCULAR_BUFFER_SIZE;
            }
            pstatus2 = ((temp2->full) + (BlockCtr2));
            //If the circular buffer1 block is empty then copy the data to
display
            if (*pstatus2 == false)
            {
                  pdata2 = (short *)(temp2->data) + (BlockCtr2)*BUFFER_SIZE;
                  memcpy(pdata2, pdata, BUFFER_SIZE*sizeof(pdata[0]));
                  *pstatus2 = true;
```

```
                    BlockCtr2 = (BlockCtr2 + 1)%CIRCULAR_BUFFER_SIZE;
              }
      }
      return(0);
}
/*
 *
 * Inspired by the code given by Luis
 */

// Implements the absolute value function for floating point variables. Not
necessary if the
// function abs() is available.
float absf(float value)
{
      if (value < 0)
            return (-1.0*value);
      else
            return (1.0*value);
}
/**
 * This function processes the data to detect and recognize the two gestures.
 */
double NoiseVtg = 0, DC_Offset = 0;
void *Process_adc_data(void *temp)
{
      unsigned int BlockCtr = 0;
      short *pdata, *pstart;
      double ADCvtg, noiseV = 0, thresholdV = 0,dc_offset = 0,mov_sum =
0,last_vtg = 0,mov_ave = 0;
      unsigned int DetectNoise = 1, Calculate_dc_Offset = 1,FirstMovingAvg =
1,first_vtg_index = 0, RejectBlocks = 0;
      double gaV = 0.010;
      pstart = (short *)temp;
      bool ActivityDetected = false, PreviousActivityDetected = false;
      CIRCULAR_BUFFER *temp1 = (CIRCULAR_BUFFER *)temp;
      temp = (void *)temp1->data;
      bool *pstatus;
    int temp_sock;socklen_t fromlen;
    fromlen = sizeof(struct sockaddr_in);

    while(1)
    {
      //Get the noise estimate from first 8 sample blocks.
      if (DetectNoise == 1)
      {
            DetectNoise = 0;
            while(BlockCtr < 8)
            {
            pstatus = ((temp1->full) + (BlockCtr));
                  while (*pstatus != true);
                  {
                        pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                              BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
                              for (int i = 0;i<BUFFER_SIZE;i++)
                              {
                                    ADCvtg = (double)pdata[i] *
(double)0.0024;
                                    noiseV += ADCvtg/THR_WIN;
                              }
                              *pstatus = false;
                  }
            }
            printf("Noise Voltage %f\n",noiseV);
            NoiseVtg = noiseV;
            thresholdV = noiseV + gaV;

      }
      //Get the DC offset in the signal from the next block
      if (Calculate_dc_Offset == 1)
      {
            Calculate_dc_Offset = 0;
                  //Calculate the DC offset from one window
            pstatus = ((temp1->full) + (BlockCtr));
            while (*pstatus != true);
            {
                  pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                        BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
                        for (int i = 0;i<BUFFER_SIZE;i++)
                        {
                              ADCvtg = (double)pdata[i] * (double)0.0024;
                              dc_offset += ADCvtg;
                        }
                        *pstatus = false;
                        dc_offset = dc_offset / BUFFER_SIZE;
                        mov_sum = 0.0;
                        last_vtg = 0;
                        printf("dc_offset %f\n", dc_offset);
                        DC_Offset = dc_offset;
            }
      }

            //Now calculate the moving average with window length as
BUFFER_SIZE
      while (RejectBlocks != 0)
      {
            pstatus = ((temp1->full) + (BlockCtr));
            while (*pstatus != true);
            {
                  pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                        BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
                        RejectBlocks--;
                        *pstatus = false;
            }
      }
      //The first time system runs, then calculate the
      //average of the complete window if size BUFFER_SIZE
            if (FirstMovingAvg == 1)
            {
            pstatus = ((temp1->full) + (BlockCtr));
            while (*pstatus != true);
            {
                        first_vtg_index = BlockCtr*BUFFER_SIZE;
                        pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                        BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
                        FirstMovingAvg = 0;
                        for (int i = 0; i < BUFFER_SIZE; i++)
                        {
                              ADCvtg = (double)pdata[i] * (double)0.0024;
                              mov_sum += absf(ADCvtg - dc_offset);
                        }
                        *pstatus = false;
            }
//                printf("mov_avg %f\n",mov_sum/BUFFER_SIZE);
            }
            pstatus = ((temp1->full) + (BlockCtr));
            while (*pstatus != true);
            {
                  pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                  //This is to check if two gestures or one gesture
                  if (ActivityDetected)
                  {
                        ActivityDetected = false;
                        PreviousActivityDetected = true;
                  }
                  //Now keep on adding 1 sample and update moving average.
                  for (int i = 0; i < BUFFER_SIZE; i++)
                  {
                        last_vtg = (double)(pstart[first_vtg_index]) *
(double)0.0024;
                        first_vtg_index = (first_vtg_index +
1)%(CIRCULAR_BUFFER_SIZE*BUFFER_SIZE);
                        ADCvtg = (double)pdata[i] * (double)0.0024;
                        mov_sum += absf(ADCvtg - dc_offset) - absf(last_vtg -
dc_offset);
                        mov_ave = mov_sum/BUFFER_SIZE;
                        //Check if moving average is greater than threshold
                        if (mov_ave > thresholdV)
                        {
                              //Activity Detected
                              fflush(stdout);
                              mov_sum = 0;
                              RejectBlocks = 2;
                              FirstMovingAvg = 1;
                              //If two consecutivr activities detected, then
double click
                              if (PreviousActivityDetected)
                              {
                                    printf("Double Click\n\n");
                                    Click = 2;
                                    DisplayClickCtr = 0;
                                    //Send the click activity over network to
Pillow Talker
                                    temp_sock = sendto(newsockfd2, &Click,
sizeof(Click), 0, (struct sockaddr *)&serv_addr2, fromlen);
                              }
                              //Otherwise single click
                              else
                                    ActivityDetected = true;
                              PreviousActivityDetected = false;
                              //If activity detected once, then break and
start the process again.
                              //Note that DC offset and noise are only
calculated once
                              break;
                        }
                  }
                  if (PreviousActivityDetected)
                  {
                        printf("Single Click\n\n");
                        Click = 1;
                        DisplayClickCtr = 0;
                        PreviousActivityDetected = false;
                        //Send the click activity over network to Pillow
Talker
                      temp_sock = sendto(newsockfd2, &Click, sizeof(Click),
0, (struct sockaddr *)&serv_addr2, fromlen);

                  }
                  //Make the block status empty
                  *pstatus = false;
                  BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
            }
    }
      return(0);
}
void InitializeSocket(char **);
void *Display_adc_data(void *);
int main(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
      InitializeSocket(argv);
      //Create two circular buffers
      CIRCULAR_BUFFER Buffer[2];
      //Make all the blocks empty
      memset(Buffer,0,sizeof(CIRCULAR_BUFFER)*2);
      //Here goes the thread creation
      pthread_t thread_rx, thread_process, thread_display;
      if (pthread_create(&thread_rx, NULL, &Rx_adc_data, (void *)Buffer) !=
0){
            printf("Unable to create thread\n"); exit(-1);}
      if (pthread_create(&thread_process, NULL, &Process_adc_data, (void
*)(&Buffer[0])) != 0){
            printf("Unable to create thread\n"); exit(-1);}
      if (pthread_create(&thread_display, NULL, &Display_adc_data, (void
*)(&Buffer[1])) != 0){
            printf("Unable to create thread\n"); exit(-1);}

      pthread_join(thread_rx, NULL);
      pthread_join(thread_process, NULL);
      pthread_join(thread_display, NULL);

      close(newsockfd);
      close(newsockfd2);
      close(sockfd);
      close(sockfd2);
      return 0;
}

void InitializeSocket(char *argv[])
{
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
       error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr)) < 0)
             error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0)
         error("ERROR on accept");

    sockfd2 = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd2 < 0)
       error("ERROR opening socket 2");
    bzero((char *) &serv_addr2, sizeof(serv_addr2));

    portno2 = atoi(argv[2]);
    serv_addr2.sin_family = AF_INET;
    serv_addr2.sin_addr.s_addr = INADDR_ANY;
    serv_addr2.sin_port = htons(portno2);
    if (bind(sockfd2, (struct sockaddr *) &serv_addr2,
             sizeof(serv_addr2)) < 0)
             error("ERROR on binding");
    listen(sockfd2,5);
    clilen2 = sizeof(cli_addr2);
    newsockfd2 = accept(sockfd2, (struct sockaddr *) &cli_addr2, &clilen2);
    if (newsockfd2 < 0)
         error("ERROR on accept");

}
#define BUFFERS_IN_ONE_PLOT         3
#define PLOT_WIDTH               BUFFER_SIZE*BUFFERS_IN_ONE_PLOT
#define PLOT_HEIGHT              500
#define VOLTAGE_MIN              -0.1
#define VOLTAGE_MAX              0.1

void Display(const char *name, IplImage *Image)
{
      // create a window
      cvNamedWindow(name, CV_WINDOW_AUTOSIZE);
      cvMoveWindow(name, 100, 100);
      // show the image
      cvShowImage(name, Image);
}
//This function displays the plot
```

```
void *Display_adc_data(void *temp)
{
      CvSize size = cvSize(PLOT_WIDTH, PLOT_HEIGHT);
      IplImage *plot_img = cvCreateImage(size,IPL_DEPTH_8U,3);
      cvZero(plot_img);
      //Get the resolution, volts/pixel to plot
      float reso_y = (float)(VOLTAGE_MAX - VOLTAGE_MIN)/(float)PLOT_HEIGHT;
      DEBUG_PRINT(("Reso y %f",reso_y));
      unsigned int draw_y;
      CvPoint previous = cvPoint(0,0);
      short *pdata;
      double ADCvtg;
      unsigned int BlockCtr = 0;
      unsigned int Offset=0,PlotCtr = 0;
      char vmin[50], vmax[50], Noise[50], DCOffet[50];
      sprintf(vmin,"V min = %d mvolts",(int)(VOLTAGE_MIN*1000));
      sprintf(vmax,"V max = %d mvolts",(int)(VOLTAGE_MAX*1000));
      CvFont *font = (CvFont *)calloc(100,4);
      CvFont *font_large = (CvFont *)calloc(100,4);
      cvInitFont(font, CV_FONT_HERSHEY_SIMPLEX, 0.4, 0.4, 0, 1, 8);
      cvInitFont(font_large, CV_FONT_HERSHEY_SIMPLEX, 2, 2, 0, 2, 8);
      CIRCULAR_BUFFER *temp1 = (CIRCULAR_BUFFER *)temp;
      temp = (void *)temp1->data;
      bool *pstatus;
      while(1)
      {
            pstatus = ((temp1->full) + (BlockCtr));
            while (*pstatus != true);
            {
                  pdata = (short *)temp + (BlockCtr)*BUFFER_SIZE;
                  BlockCtr = (BlockCtr + 1)%CIRCULAR_BUFFER_SIZE;
                  for (int i = 0; i < BUFFER_SIZE; i++)
                  {
                        ADCvtg = (double)pdata[i] * (double)0.0024;
                        //Draw the pixel in the image
                        draw_y = (unsigned int)(ADCvtg/reso_y) - (unsigned
int)((float)VOLTAGE_MIN/reso_y);
                        draw_y = PLOT_HEIGHT - draw_y;
                        if ((draw_y > 0) && (draw_y < PLOT_HEIGHT))
                        {
                              cvLine(plot_img, previous,
cvPoint(i+Offset,draw_y), CV_RGB(255,0,0), 1, 8, 0);
                              previous = cvPoint(i+Offset,draw_y);
                        }
                  }
                  *pstatus = false;
                  PlotCtr++;
                  Offset = PlotCtr*BUFFER_SIZE;
            }
            //Now prepare the image to display
            if (PlotCtr == BUFFERS_IN_ONE_PLOT)
            {
                  cvPutText(plot_img,vmin,cvPoint(10,PLOT_HEIGHT-
20),font,CV_RGB(0,255,0));

      cvPutText(plot_img,vmax,cvPoint(10,20),font,CV_RGB(0,255,0));
                  cvLine(plot_img,
cvPoint(0,PLOT_HEIGHT/2),cvPoint(PLOT_WIDTH,PLOT_HEIGHT/2), CV_RGB(0,0,255),
1, 8, 0);

      cvPutText(plot_img,"0",cvPoint(10,PLOT_HEIGHT/2+10),font,CV_RGB(0,255,2
55));

      cvPutText(plot_img,"500",cvPoint(500,PLOT_HEIGHT/2+10),font,CV_RGB(0,25
5,255));

      cvPutText(plot_img,"1000",cvPoint(1000,PLOT_HEIGHT/2+10),font,CV_RGB(0,
255,255));
                  cvPutText(plot_img,"1500",cvPoint(1500-
50,PLOT_HEIGHT/2+10),font,CV_RGB(0,255,255));
                  sprintf(Noise,"Noise = %f Volts",(NoiseVtg));
                  sprintf(DCOffet,"DC Offset = %f Volts",(DC_Offset));

      cvPutText(plot_img,Noise,cvPoint(10,40),font,CV_RGB(0,255,0));

      cvPutText(plot_img,DCOffet,cvPoint(10,60),font,CV_RGB(0,255,0));
                  if ((Click == 1) && (DisplayClickCtr < 5))
                  {
                        cvPutText(plot_img,"SINGLE
CLICK",cvPoint(500,60),font_large,CV_RGB(0,255,0));
                        DisplayClickCtr++;
                  }
                  else if ((Click == 2) && (DisplayClickCtr < 5))
                  {
                        cvPutText(plot_img,"DOUBLE
CLICK",cvPoint(500,60),font_large,CV_RGB(0,255,0));
                        DisplayClickCtr++;
                  }
                  Display("Signal Plot", plot_img);
                  cvSet(plot_img, cvScalar(0,0,0));
                  PlotCtr = 0;
                  Offset = 0;
                  previous = cvPoint(0,0);
                  cvWaitKey(1);
            }
      }
      return(0);
}
```

Client Program

Provide the host address and port number as input argument 1 and 2 respectively.

```
/**
 *  Copyright (C) 2013 Vision-Guided and Intelligent Robotics Lab
 *  Written by Akshay Jain <aj4g2@mail.missouri.edu>
 *    Grabs ADC data form TS 7250 and sends over TCP IP
 *    Provide the host address and port number as argv[1] and argv[2] resp.
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation. Meaning:
 *          keep this copyright notice,
 *          do  not try to make money out of it,
 *          it's distributed WITHOUT ANY WARRANTY,
 *          yada yada yada...
 *
 *  You can get a copy of the GNU General Public License by writing to
 *  the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
 *  Boston, MA  02111-1307  USA
 */
#include "Declarations_v2.h"
#include <stdbool.h>
//Global variables to initialize ADC and set up socket communication.
unsigned char     *chk_ad_reg, *option_lsb_reg, *chk_sts_reg;
int fd;
int sockfd, portno, n;
struct sockaddr_in serv_addr;
struct hostent *server;

//Sampling Frequency of ADC data
#define F                                                   4000
#define ADC_SAMPLE_RATE_NSEC                    (1000000000/F)//(2500000)
//Size of each block in circular buffer
#define BUFFER_SIZE                                         (500)
//Number of blocks in circular buffer
#define CIRCULAR_BUFFER_SIZE                    (400)
//#define SLEEP                                             (30)
sem_t send_data_sem;
//Data structure for circular buffers.
typedef struct
{
      short data[BUFFER_SIZE*CIRCULAR_BUFFER_SIZE];
      volatile char full[CIRCULAR_BUFFER_SIZE];
}CIRCULAR_BUFFER;
//Function to return the value read from ADC.
adc_datatype GetADCVal(void)
{
      adc_datatype *ADCval = (short *)option_lsb_reg;
      *option_lsb_reg = CHANNEL_7;
      /* Wait for ADC to complete the conversion process. MAX197 takes 12us
to convert.*/
      while ((*chk_sts_reg & 0x80) != 0x0)
            usleep(10);
      return((adc_datatype)*(ADCval));

}

void InitializeMax197ADC(void);
void InitializeSocket(char **);
//Function to get ADC data and stor it in circular buffer
void *FetchNStore_adc_data(void *temp)
{
      RTIME period = start_rt_timer(nano2count(ADC_SAMPLE_RATE_NSEC));
      //Initialize the real time task, with name thrd1
      RT_TASK* rttask1 = rt_task_init(nam2num("thrd1"), 0, 512, 256);
      //Make the task periodic with Start time and repeat interval.
      rt_task_make_periodic(rttask1, rt_get_time(), period);
      CIRCULAR_BUFFER *temp1 = (CIRCULAR_BUFFER *)temp;
      adc_datatype *pdata = (adc_datatype *)temp1->data;
	volatile char *status = temp1->full;
	unsigned int Ctr = 0;
	unsigned int CircularCtr = 0, BlockCtr = 0;
	while(1)
	{
		//If the current block of circular buffer is empty.
		if (status[CircularCtr] == false)
		{
			pdata[Ctr] = GetADCVal();
			fflush(stdout);
			Ctr++;
			//if one block worth of data received
			if (Ctr == BUFFER_SIZE)
			{

				fflush(stdout);
				BlockCtr++;
				//make the status of the block as true to indicate
that it is full
				status[CircularCtr] = true;
				Ctr = 0;
				//If end reached, go back to the start
				if (CircularCtr == (CIRCULAR_BUFFER_SIZE-1))
				{
					CircularCtr = 0;
					pdata = (adc_datatype *)temp1->data;
				}
				//Else, just go back to the next block
				else
				{
					pdata += BUFFER_SIZE;
				}
				CircularCtr++;
			}
		}
		rt_task_wait_period();
	}
	pthread_exit(0);
	return(NULL);
}
//Thread to transmit the data over socket
void *Tx_adc_data(void *temp)
{

	printf("In Tx Thread");
	CIRCULAR_BUFFER *temp1 = (CIRCULAR_BUFFER *)temp;
	adc_datatype *pdata = (adc_datatype *)temp1->data;
	volatile char  *status = temp1->full;
	unsigned int CircularCtr = 0;
	while(1)
	{
		//Wait for the block of circular buffer to fill
		while (status[CircularCtr] != 1);
		n = write(sockfd,(void *)pdata,BUFFER_SIZE*sizeof(adc_datatype));
		//Once the data is written over network, make the block status as
empty
		status[CircularCtr] = false;
```

```
          if (n < 0) printf("ERROR writing to socket");
          if (CircularCtr == (CIRCULAR_BUFFER_SIZE-1))
          {
            CircularCtr = 0;
            pdata = (adc_datatype *)temp1->data;
          }
          else
            pdata += BUFFER_SIZE;
          CircularCtr++;
      }
      pthread_exit(0);
      return(NULL);
}



int main(int argc, char *argv[])
{
      InitializeMax197ADC();
      InitializeSocket(argv);
      //Create a circuar buffer
      CIRCULAR_BUFFER Buffer;
      //Initialize to 0, so that all the blocks are
      //indicated empty
      memset(&Buffer,0,sizeof(CIRCULAR_BUFFER));
      pthread_t thread_adc, thread_tx;
      int i=0;
      if (pthread_create(&thread_adc, NULL, &FetchNStore_adc_data, (void
*)&Buffer) != 0){
            printf("Unable to create thread\n"); exit(-1);}
      if (pthread_create(&thread_tx, NULL, &Tx_adc_data, (void *)&Buffer) !=
0){
            printf("Unable to create thread\n"); exit(-1);}

      pthread_join(thread_adc, NULL);
      pthread_join(thread_tx, NULL);

      close(fd);
      return(0);
}

void InitializeMax197ADC(void)
{
      fd = open("/dev/mem", O_RDWR|O_SYNC);
      if (fd < 0) {
            printf("Can't open /dev/mem\n");
            exit(1);
      }

      /* Get a pointer to register that contains the information about ADC
status.*/
      chk_ad_reg         = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
MAP_SHARED, fd, CHK_AD_REG);
      assert(chk_ad_reg != MAP_FAILED);

      /* Pointer to register used to set options for input voltage type &
range. The same
```

```
       * pointer is used to access the LSB of the result in the Read mode.*/
      option_lsb_reg    = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
MAP_SHARED, fd, OPTION_LSB_REG);
      assert(option_lsb_reg != MAP_FAILED);

      /* Pointer to register used to check the conversion status of ADC. If
bit 7 is 1, then
       * the conversion process is still in progress.*/
      chk_sts_reg       = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
MAP_SHARED,fd, CHK_STATUS_REG);
      assert(chk_sts_reg != MAP_FAILED);

      if( !(*chk_ad_reg & CHK_AD_BIT) ){
            printf("ERROR in installing A/D on 7250");
            exit(2);
      }

}
void InitializeSocket(char *argv[])
{
      //Get the port number to communicate to
    portno = atoi(argv[2]);
    //Instantiate a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        printf("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        printf("ERROR connecting");

}
```