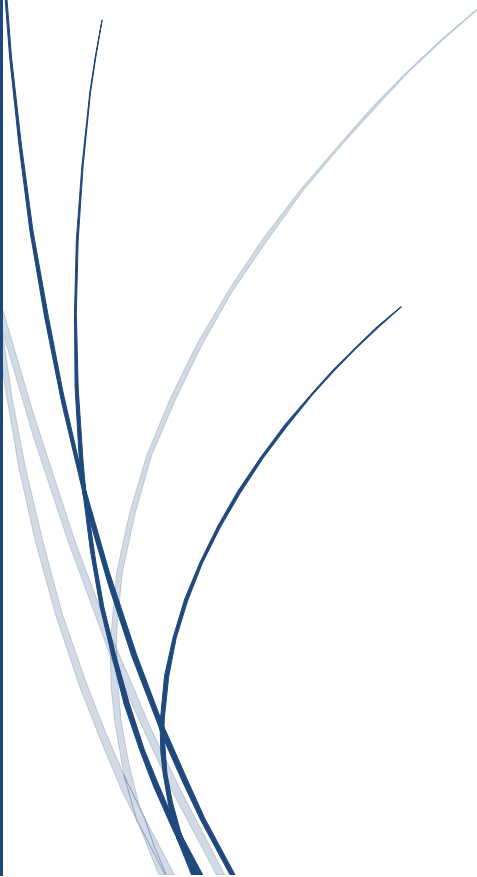




May 04, 2018

# Real Time Embedded Computing Final Report

ECE 4220, University of Missouri



Hotrabhavananda, Benjarit  
Li, Zhengyu  
Yao, Yifu

Professor: Dr. Luis Rivera

## Table of Contents

|   |    |
|---|----|
| Abstract.....   | 3  |
| Introduction.....   | 3  |
| Background.....   | 4  |
| Implementation .....  | 4  |
| Experiment and Results .....                                | 8  |
| Clock synchronization tests. ....                           | 8  |
| Proper ADC test.....  | 8  |
| “Line overload” events can be tested.....                   | 9  |
| “No power” events can be tested .....                       | 9  |
| Control commands test. ....                                 | 10 |
| Sporadic switch changes / button pushes can be tested ..... | 10 |
| Visualization tests. ....                                   | 10 |
| Discussion and Conclusions .....                            | 10 |
| Appendices .....  | 11 |
| Arduino_sine_wave.ino .....                                 | 11 |
| BTNKernel.c.....  | 12 |
| final_historian.c .....                                     | 16 |
| final_server.c .....  | 18 |

Table of Figures

Figure 1. RTU Flowchart..... 7

Figure 2. Historian Flowchart..... 8

Figure 3. ADC Value ..... 9

Figure 4. Voltage sine wave ..... 9

Figure 5. Overload test ..... 9

Figure 6. No power test ..... 10

## **Abstract**

Our project is a supervisory control and data acquisition (SCADA) system. This system involves the creation of two remote terminal units (RTUs) and a historian client. This is a system that will continuously receive data from RTUs and printing the status of RTUs in historian client, and if needed, sending commands to RTUs to control two RTUs. This project addresses the issue of industrial monitor and control. This system is widely used for many industrial enterprises and power systems. The approach for this implementation will be using Raspberry Pi 3, Arduino and auxiliary boards as components of RTUs, and a lab computer as console operator. Programming on the RPi 3 to receive digital signals that converted by Arduino from auxiliary boards, and RPi 3 will be sending the signals to console operator so that the status of auxiliary board will be received and will be shown on the human-machine interface which is the terminal in our machines. The RPi 3 will be also receiving commands from console and then make some changes according to the commands it received on auxiliary board.

## **Introduction**

The scope of this project revolves around the creation of supervisory control and data acquisition system. This is a system that will record all changes of RTUs, and return all status to the console operator and printed those status it received through the human-machine interface. To be able to achieve the function of this system, the project needs materials that consists of two microcontrollers, two auxiliary boards, two programmable logic controllers, wires and two computers. More specifically, we are using Raspberry Pi 3 as microcontrollers, BCM2835 as auxiliary boards, and Arduino as programmable logic controllers. The purpose of this project is to provide a safer environment and an efficiency working tool for people in industry enterprises and power systems. Therefore, first responders could see the current status of all RTUs so that they are

able to know which parts have problems and which parts are working properly, and then fixing the problem as soon as possible to avoid tremendous losses.

## **Background**

In modern manufacturing and industrial processes, mining industries, public and private utilities, leisure and security industries, supervisory control is often needed to connect equipment and systems separated by large distance. This can range from a few meters to thousands of kilometers. Supervisory control is used for sending commands, programs and receives monitoring information from the remote terminal units. SCADA refers to the combination of supervisory control and data acquisition. SCADA encompasses the collecting of the information, transferring it back to the central site, carrying out any necessary analysis and control and then displaying that information on a number of operator screens or displays. The required control actions are then conveyed back to the process.

## **Implementation**

The fundamental knowledge of this project is based on what we had learned from Lab1 to Lab6. Basically, this project can be divided into two parts. The first part is a program for RTUs that checking status of the auxiliary boards and sending these information to the workstation. The other part is a historian program that receiving commands from RTUs and it is able to send command to RTUs as well to control LEDs on auxiliary boards.

In general, we used wiringPi functions such as *digitalRead()* to check the status (ON/OFF) of switches, buttons and LEDs, and we used kernel module where we implemented ISR to check if buttons had been pressed. Specifically, in the first part, we set up wiringPi and wiringPi SPI at

first. Then we initialized switches, buttons as input and set LEDs as output by *pinMode()*. We used some flags, such as *switch1*, *button1* and *LED1*, to store the status of switches, buttons and LEDs. 1 means ON and 0 means OFF. And we also have some arguments such as *preSwitch1*, *preButton1* and *preLED1* to store the previous status. At the same time, we initialized a semaphore to protect such checking flags when we changed it and set each checking flag to 0 as initialization. What's more, opening char device thus we could implement communication between kernel space and user space and creating socket so that the communication between server and client can be done. Then we created two threads to get RTU status and get information of pressed buttons from kernel. In our periodic update thread, we checked status every 1 second. First, we got the RTU's time by time functions and sent it to client. Then, we checked the status of switches, buttons and LEDs at first by *digitalRead()* in a protection of semaphore. Then we wrote these status as string messages and we sent them to client through socket. The client will be able to print the messages out to the terminal which is our human-machine interface. Also, we used the provided functions to read ADC values that we created by using Arduino and we also sent it to client. After getting the status information, we checked if there were some events happened. If ADC value is equals to 0, there will be a "No Power" event to be shown on terminal, or if ADC value is not between 1 and 2, there will be an "Overload" event to be shown. As for switches and LEDs, we compared their current status with their previous status. If they are not equal, we are able to know that they had been changed, then we sent messages of events about changing to client. For buttons, we got messages from kernel space. The messages would be able to show if the buttons had been pressed. Meanwhile, when an event happened, we used 7-segment display to show which event happened. 0 means "No Power". 1 means "Overload". 2 and 3 means switches changed. 4 and 5 means buttons changed. 7, 8 and 9 means LEDs changed. We used *digitalWrite()* to implement such

functions. At last, we copied current status to previous status. Then the thread would keep working in the same way every 1 second.

As for the kernel thread, the message it received is used to decide if there is a button event from kernel space by *read()*. If it received message which is “button1”, for example, it would know that the button 1 had been pressed. Then there was a flag changed to 1 meaning the button 1 had been pressed so in the update thread, the pressed button event could be found. As for the kernel module, just like what we did in lab6 part2, we used interrupt to record if buttons had been pressed. If buttons had been pressed, the kernel would send a message to the user space, such as “button1”. In this way, these two threads would continue to update RTUs’ log.

In the other part of the project, our program continues to receive commands from client. There are six commands that can be used to control LEDs which are “LED1ON”, “LED2ON”, “LED3ON”, “LED1OFF”, “LED2OFF”, “LED3OFF”. If received command equal to one of them, the relevant LED will be turned on or turned off by *digitalWrite()* function.

As for the historian program, we implemented it as a client. In the initialization, we made socket connection. After that, the program scanned what command the user typed and sent it as command to RTUs. At the same time, we created a thread to read messages from RTUs and we printed them out.

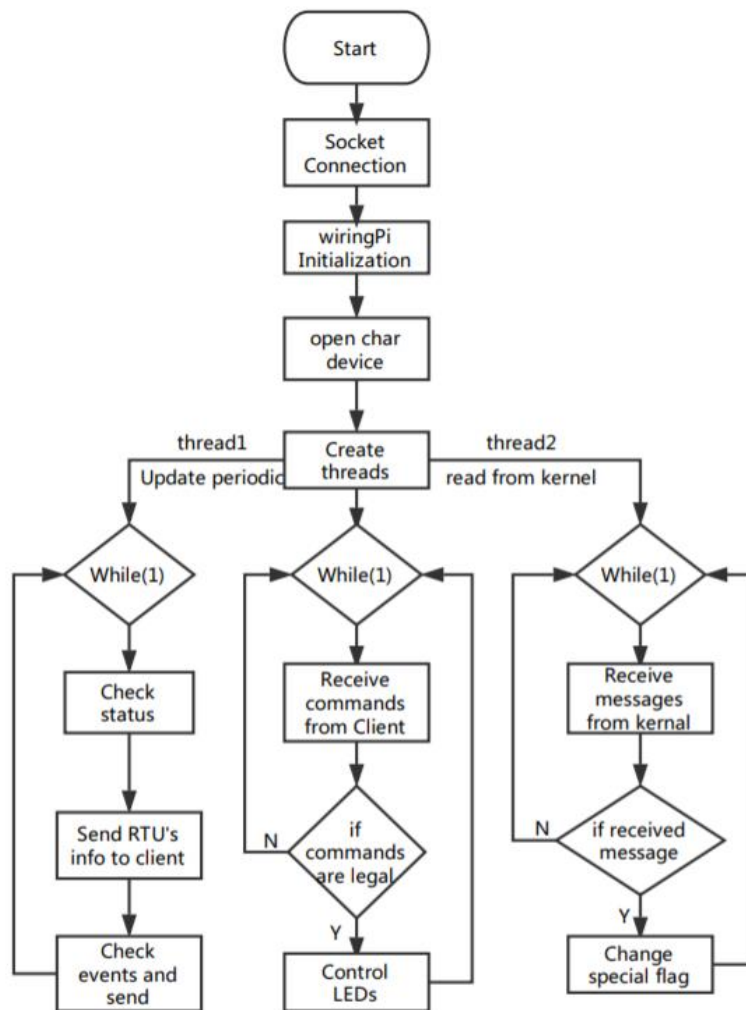
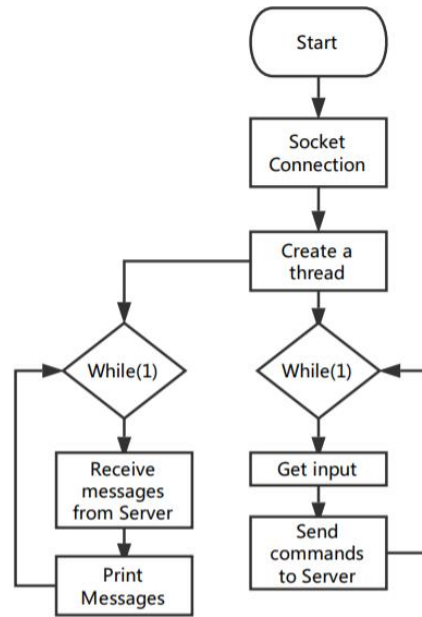


Figure 1. RTU Flowchart





*Figure 2. Historian Flowchart*

## Experiment and Results

We have completed following experiments:

### **Clock synchronization tests.**

We tested the time of our RPis, and every RPi has the same time. The result was showed in our demo video.

### **Proper ADC test.**

We printed out the ADC values we got and make them as plots in a graph to see if they are a proper ADC. And the results showed that the ADC value we had is great.

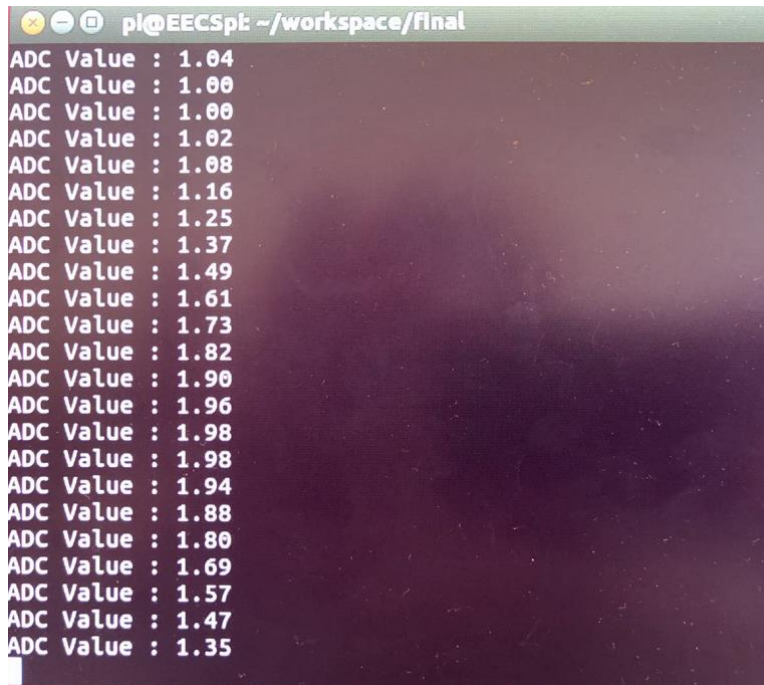


Figure 3. ADC Value

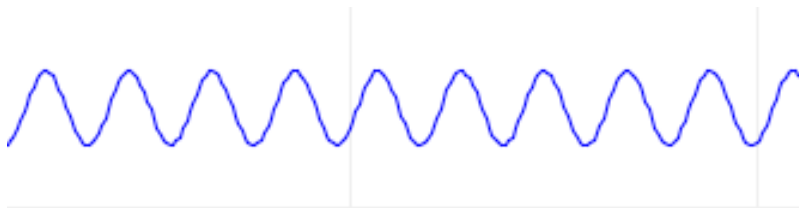


Figure 4. Voltage sine wave

### **“Line overload” events can be tested**

We tested if the “Line overload” event could be checked when we increased the amplitude of the signals and the result showed that our program could handle it.

```
RTU1: 11:48:54
Switch 1: OFF Switch 2: OFF
Button 1: OFF Button 2: OFF
LED 1: OFF LED 2: OFF LED 3: OFF
ADC Value: 0.96
Event: Overload!
RTU 1 finished
```

Figure 5. Overload test

### **“No power” events can be tested**

We tested no power event by stopping generating the power wave and the result showed that our program could handle that as well.



```
This was received: RTU1: 15:02:48
This was received: Switch 1: OFF Switch 2: ON
This was received: Button 1: OFF Button 2: OFF
This was received: LED 1: OFF LED 2: OFF LED 3: OFF
This was received:
This was received: Event: No Power!
This was received: RTU 1 finished
```

*Figure 6. No power test*

### **Control commands test.**

We tested the commands “LED1ON”, “LED2ON”, “LED3ON”, “LED1OFF”, “LED2OFF” and “LED3OFF” to test if three LEDs could be controlled by these six commands. And results showed that these commands work.

### **Sporadic switch changes / button pushes can be tested**

We tested all buttons and switches on each auxiliary board to see if their status could be checked when we changed switches and pressed buttons. The results shows that they all could be checked.

### **Visualization tests.**

Our output looks very clear and readable. The result can be found in our demo video.

## **Discussion and Conclusions**

In our project, the first problem we met was how to create voltage signals. At first, we wanted to use some chips and capacitances to create it but we failed. Then we tried to use Arduino Uno to create. We learned how to use Arduino online such as google some methods to create voltage signal. Then we tried to use  $\sin()$  function and transformed the result to get the supposed voltage signal. Finally, we succeeded on creating sine wave voltage signals.

And another problem we had was about the IP address. When we implemented communication between server and client, at first our server could not receive commands from client. But if we used the workstation from lab5 or lab6, the server could receive it. Then we found we made a

wrong target address when we used *sendto()*. After fixing it, we found that the server still could not receive commands from client but the client could receive messages from server. We didn't find any problems in our server program so we rewrite the config part of socket connection then it could work. We thought that there existed some slight mistakes in the original config section.

What we learned from this project is we know the basic operations about Arduino and we get how to make ADC. And we are more familiar with the socket now. Such project make us understand more deeply about server/client.

## Appendices

### Arduino\_sine\_wave.ino

This code is used for generating a sin wave as input voltage.

```
int sine[255];
void setup()
{
    float x;
    float y;
    for(int i=0;i<255;i++)
    {
        x=(float)i;
        y=sin((x/255)*2*10*PI); // generate sine wave
        sine[i]=int(y*31)+76.6;
    }
}
void loop()
{
    for (int i=0;i<255;i++)
    {
        PORTD=sine[i];
        delayMicroseconds(10);
    }
}
```

## BTNKernel.c

This is a program for our button press kernel module

```
#ifndef MODULE
#define MODULE
#endif
#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kthread.h> // for kthreads
#include <linux/sched.h> // for task_struct
#include <linux/time.h> // for using jiffies
#include <linux/timer.h>
#include <linux/delay.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <linux/fs.h>
#include <linux/string.h>

#define MSG_SIZE 50
#define CDEV_NAME "Final" // "YourDevName"

MODULE_LICENSE("GPL");

static int major;
static char msg[MSG_SIZE];

unsigned long *BPTR;

// structure for the kthread.
static struct task_struct *kthread1;
unsigned long *EVENT, *PUD, *PUD_CLK, *EDGE;

int mydev_id; // variable needed to identify the handler
int button1, button2;

static irqreturn_t button_isr(int irq, void *dev_id)
{
    // In general, you want to disable the interrupt while handling it.
    disable_irq_nosync(79);

    unsigned long tmp = *EVENT & 0x1F0000;
```

```

if(tmp == 0x10000) //0000 0001 0000 0000 0000 0000
{
    button1 = 1;
    printk("button 1 pushed\n");
}
else if(tmp == 0x20000) //0000 0010 0000 0000 0000 0000
{
    button2 = 1;
    printk("button 2 pushed\n");
}

*EVENT = *EVENT | 0x1F0000;//clear it
    printk("Interrupt handled\n");
    enable_irq(79);          // re-enable interrupt

return IRQ_HANDLED;
}

// Function to be associated with the kthread; what the kthread executes.
int kthread_fn(void *ptr)
{
    printk("Before loop\n");

    // The ktrhead does not need to run forever. It can execute something
    // and then leave.
    while(1)
    {
        // In an infinite loop, you should check if the kthread_stop
        // function has been called (e.g. in clean up module). If so,
        // the kthread should exit. If this is not done, the thread
        // will persist even after removing the module.
        if(kthread_should_stop()) {
            do_exit(0);
        }

        // comment out if your loop is going "fast". You don't want to
        // printk too often. Sporadically or every second or so, it's okay.
    }

    return 0;
}

// Function called when the user space program reads the character device.
static ssize_t device_read(struct file *filp, char __user *buffer, size_t len, loff_t *offset)
{
    msg[0] = "\0";

    if (button1 == 1)

```

```

{
    strcpy(msg, "button1");
    button1 = 0;
}
if (button2 == 1)
{
    strcpy(msg, "button2");
    button2 = 0;
}
    ssize_t dummy = copy_to_user(buffer, msg, len);
    printk("%s\n", msg);

    return len;
}

// Function called when the user space program writes to the Character Device.
static ssize_t device_write(struct file *filp, const char __user *buff, size_t len, loff_t *off)
{
    ssize_t dummy;

    return len;          // the number of bytes that were written to the Character Device.
}

// structure needed when registering the Character Device. Members are the callback
// functions when the device is read from or written to.
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
};

int cdev_module_init(void)
{
    // register the Character Device and obtain the major (assigned by the system)
    major = register_chrdev(0, CDEV_NAME, &fops);
    if (major < 0) {
        printk("Registering the character device failed with %d\n", major);
        return major;
    }
    printk("Lab6_cdev_kmod example, assigned major: %d\n", major);
    printk("Create Char Device (node) with: sudo mknod /dev/%s c %d 0\n", CDEV_NAME, major);

    char kthread_name[11]="my_kthread"; // try running ps -ef | grep my_kthread
                                         // when the thread is
active.
    printk("In init module\n");

    int dummy = 0;
    BPTR = (unsigned long *) ioremap(0x3F200000, 4096);

```

```

EVENT = BPTR + 0x40/4; //even

//pull-down setting
PUD = BPTR + 0x94/4; //point at gppud register
*PUD = *PUD | 0x155; //pud as pull down control

udelay(150); //wait

PUD_CLK = BPTR + 0x98/4; //point at gppudclk0
*PUD_CLK = *PUD_CLK | 0x1F0000; //asynchronous falling EDGE

udelay(150); //wait

*PUD &= 0xFFFFFEAA; //off off
    *PUD_CLK &= 0xFFE0FFFF; //off now

// EnableRising EDGE detection for all buttons.
EDGE = BPTR + 0x4C/4; //point at rising EDGE
*EDGE = *EDGE | 0x1F0000;

button1 = 0;
button2 = 0;
    // Request the interrupt / attach handler (line 79, doesn't match the manual...)
    // The third argument string can be different (you give the name)
    dummy = request_irq(79, button_isr, IRQF_SHARED, "Button_handler", &mydev_id);

kthread1 = kthread_create(kthread_fn, NULL, kthread_name);

if((kthread1)) // true if kthread creation is successful
{
    printk("Inside if\n");
        // kthread is dormant after creation. Needs to be woken up
    wake_up_process(kthread1);
}
    return 0;
}

void cdev_module_exit(void)
{
    // Once unregistered, the Character Device won't be able to be accessed,
    // even if the file /dev/YourDevName still exists. Give that a try...
    unregister_chrdev(major, CDEV_NAME);
    printk("Char Device /dev/%s unregistered.\n", CDEV_NAME);

    free_irq(79, &mydev_id);
    printk("Button Detection disabled.\n");

    int ret;

```



```

        // the following doesn't actually stop the thread, but signals that
        // the thread should stop itself (with do_exit above).
        // kthread should not be called if the thread has already stopped.
        ret = kthread_stop(kthread1);

        if(!ret)
            printk("Kthread stopped\n");
    }

module_init(cdev_module_init);
module_exit(cdev_module_exit);

```

## final\_historian.c

This is a program for our historian/console operator

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <pthread.h>
#include <semaphore.h>

#define MSG_SIZE 40                // message size

sem_t sem;

void* readMessages(void*);
void* sendMessages(void*);

void displayChoice();
void sendCommand();
void printHistory();

char buffer[MSG_SIZE],ip[MSG_SIZE],receivedIP[MSG_SIZE], ipHolder[MSG_SIZE];
char commandMSG[MSG_SIZE];

int sock, n, r;
unsigned int length;
int boolval = 1;                // for a socket option
// Use this to find IP

```

```

struct ifreq ifr;
char eth0[] = "wlan0";
socklen_t fromlen;
struct sockaddr_in server;
struct sockaddr_in from; // From the client
int port_number;
int flagg = 0;

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[]){
    int option;
    pthread_t sender, reader;

    if (argc == 2){
        port_number = atoi(argv[1]);
    } else {
        port_number = 2000; // set default if not provided
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0); // Creates socket. Connectionless.
    if (sock < 0)
        error("socket");

    // change socket permissions to allow broadcast
    if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &boolval, sizeof(boolval)) < 0)
    {
        printf("error setting socket options\n");
        exit(-1);
    }

    length = sizeof(server); // determines lenght of the structure
    bzero(&server, length); // set all valus = 0
    server.sin_family = AF_INET; //constant for internet domain
    server.sin_addr.s_addr = INADDR_ANY; // MY IP address
    server.sin_port = htons(port_number);

    if (bind(sock, (struct sockaddr *)&server, length) < 0){
        printf("binding error dumby\n");
    }

    if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &boolval, sizeof(boolval)) < 0){
        printf("error setting socket option\n");
        exit(-1);
    }
}

```

```

        fromlen = sizeof(struct sockaddr_in);

//Start the pthreads
pthread_create(&reader, NULL, readMessages, NULL);

while(1)
{
    bzero(commandMSG, MSG_SIZE);
    scanf("%s",commandMSG);
    server.sin_addr.s_addr = inet_addr("128.206.19.255");
    n = sendto(sock, commandMSG, MSG_SIZE, 0, (const struct sockaddr *)&server,fromlen);
    if (n<0){
        error("Sendto");
    }
}
return 0;
}

void* readMessages(void* agr){
    int count = 1;
    while(1){
        bzero(&buffer,MSG_SIZE); // clear the buffer to NULL

        n = recvfrom(sock, buffer, MSG_SIZE, 0, (struct sockaddr *)&from, &fromlen);
        if (n<0)
        {
            error("recv");
        }
        printf("%s\n",buffer);
    }
}

```

## final\_server.c

This is a program for our two RTUs

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>
#include <wiringSerial.h>

#include <sys/time.h>
#include <sys/types.h>

```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <time.h>
#include <string.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>

#define MSG_SIZE 50

//PIN
#define BTN1 27
#define BTN2 0
#define S1 26
#define S2 23
#define RED 8
#define YELLOW 9
#define GREEN 7
#define SEVENABLE 29
#define A 5
#define B 6
#define C 25
#define D 2

#define SPI_CHANNEL    0    // 0 or 1
#define SPI_SPEED      2000000    // Max speed is 3.6 MHz when VDD = 5 V
#define ADC_CHANNEL    1    // Between 0 and 3

#define CHAR_DEV "/dev/Final"

//RTU infomation
int RTUNum;
int switch1, switch2;
int button1, button2;
int LED1, LED2, LED3;
float ADCValue=0;
int preSwitch1, preSwitch2;
int preButton1, preButton2;
int preLED1, preLED2, preLED3;
int buttonOneFlag, buttonTwoFlag;

//socket arguments
int sock, length, n, num;
int boolval = 1;           // for a socket option

```

```

socklen_t fromlen;
struct sockaddr_in server;
struct sockaddr_in addr;
struct sockaddr_in from; // From the client

//IP arguments
struct ifreq ifr;
char eth0[] = "wlan0";
char ip[MSG_SIZE], ipHolder[MSG_SIZE];

//ISR arguments
unsigned long *EVENT, *PUD, *PUD_CLK, *EDGE;
unsigned long *BPTR;
int mydev_id; // variable needed to identify the handler

uint16_t get_ADC(int channel); // prototype
time_t timep; //time record
char buffer[MSG_SIZE]; //to store received command
char buffer2[MSG_SIZE]; //to store sent status
sem_t my_sem;

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int parseIP(char* IP)
{
    char *saveptr;
    char* temp = strtok_r(IP, ".", &saveptr);
    int i = 1, numBoard = 0;
    while(temp != NULL)
    {
        if(i == 4)
            numBoard = atoi(temp);
        temp = strtok_r(NULL, ".", &saveptr);
        i++;
    }
    return numBoard;
}

uint16_t get_ADC(int ADC_chan)
{
    uint8_t spiData[3];
    spiData[0] = 0b00000001;
    spiData[1] = 0b10000000 | (ADC_chan << 4);

    spiData[2] = 0;
    srand(time(NULL));

```

```

        wiringPiSPIDataRW(SPI_CHANNEL, spiData, 3);

        return ((spiData[1] << 8) | spiData[2]);
    }

```

```

void getTime()
{
    strcpy(buffer2, "RTU1: "); //or RTU2

    time(&timep);
    char s_temp[MSG_SIZE], s_temp2[MSG_SIZE] = "";
    strcpy(s_temp, ctime(&timep));
    int i=11; // we only need 11~18 of the time value
    while(i<=18)
    {
        s_temp2[i-11]=s_temp[i];
        i++;
    }
    strcat(buffer2, s_temp2);
    return;
}

```

```

void getSwitch()
{
    strcpy(buffer2, "Switch 1: ");
    if (switch1 == 1) strcat(buffer2, "ON ");
    else strcat(buffer2, "OFF ");

    strcat(buffer2, "Switch 2: ");
    if (switch2 == 1) strcat(buffer2, "ON");
    else strcat(buffer2, "OFF");

    return;
}

```

```

void getButton()
{
    strcpy(buffer2, "Button 1: ");
    if (button1 == 1) strcat(buffer2, "ON ");
    else strcat(buffer2, "OFF ");

    strcat(buffer2, "Button 2: ");
    if (button2 == 1) strcat(buffer2, "ON");
    else strcat(buffer2, "OFF");

    return;
}

```

```

void getLED()

```

```

{
    strcpy(buffer2, "LED 1: ");
    if (LED1 == 1) strcat(buffer2, "ON ");
    else strcat(buffer2, "OFF ");

    strcat(buffer2, "LED 2: ");
    if (LED2 == 1) strcat(buffer2, "ON ");
    else strcat(buffer2, "OFF ");

    strcat(buffer2, "LED 3: ");
    if (LED3 == 1) strcat(buffer2, "ON");
    else strcat(buffer2, "OFF");

    return;
}

void getADCValue()
{
    ADCValue = -1;
    ADCValue = get_ADC(ADC_CHANNEL);
    ADCValue = ((3.300/1023)*ADCValue)/2.0;
    sprintf(buffer2, "%s %.2f", "ADC Value: ", ADCValue);
    return;
}

void* readFromKernal(void *arg)
{
    printf("readFromKernal Thread begin.\n");
    char buf[MSG_SIZE];
    while(1)
    {
        bzero(buf, MSG_SIZE);
        n = read(mydev_id, buf, sizeof(buf));
        if(n != sizeof(buf))
        {
            printf("Read failed, leaving...\n");
            break;
        }
        if (strcmp(buf, "button1") == 0)
        {
            buttonOneFlag = 1;
        }
        else if (strcmp(buf, "button2") == 0)
        {
            buttonTwoFlag = 1;
        }
    }
    pthread_exit(0);
}

void* periodicUpdate(void *arg)

```

```

{
printf("Periodic Update thread begin.\n");
while(1)
{
    /*****Check Status*****/
    sem_wait(&my_sem);
    //switch status
    switch1 = digitalRead(S1);
    switch2 = digitalRead(S2);
    //button status
    button1 = digitalRead(BTN1);
    button2 = digitalRead(BTN2);
    //LED status
    LED1 = digitalRead(RED);
    LED2 = digitalRead(YELLOW);
    LED3 = digitalRead(GREEN);
    sem_post(&my_sem);
    /*****/

    /*****Sent message to Client*****/
    //get status info
    bzero(buffer2, MSG_SIZE);
    getTime();
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    bzero(buffer2, MSG_SIZE);
    getSwitch();
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    bzero(buffer2, MSG_SIZE);
    getButton();
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    bzero(buffer2, MSG_SIZE);
    getLED();
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    bzero(buffer2, MSG_SIZE);
    getADCValue();
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);

    //if some events happend
    if (ADCValue == 0)
    {
        bzero(buffer2, MSG_SIZE);
        strcpy(buffer2, "Event: No Power!");
        n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
        digitalWrite(A, 0);
        digitalWrite(B, 0);
        digitalWrite(C, 0);
        digitalWrite(D, 0);
    }
    else if (ADCValue>2 || ADCValue<1)
    {

```



```

    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: Overload!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    digitalWrite(A, 1);
    digitalWrite(B, 0);
    digitalWrite(C, 0);
    digitalWrite(D, 0);
}

if (preSwitch1 != switch1)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: Switch 1 Change!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    preSwitch1 = switch1;
    digitalWrite(A, 0);
    digitalWrite(B, 1);
    digitalWrite(C, 0);
    digitalWrite(D, 0);
}

if (preSwitch2 != switch2)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: Switch 2 Change!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    preSwitch2 = switch2;
    digitalWrite(A, 1);
    digitalWrite(B, 1);
    digitalWrite(C, 0);
    digitalWrite(D, 0);
}

if (buttonOneFlag == 1)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: Button 1 Pressed!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    buttonOneFlag = 0;
    digitalWrite(A, 0);
    digitalWrite(B, 0);
    digitalWrite(C, 1);
    digitalWrite(D, 0);
}

if (buttonTwoFlag == 1)
{

```

```

    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: Button 2 Pressed!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    buttonTwoFlag = 0;
    digitalWrite(A, 1);
    digitalWrite(B, 0);
    digitalWrite(C, 1);
    digitalWrite(D, 0);

}

if (preLED1 != LED1)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: LED 1 Change!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    preLED1 = LED1;
    digitalWrite(A, 0);
    digitalWrite(B, 1);
    digitalWrite(C, 1);
    digitalWrite(D, 0);

}

if (preLED2 != LED2)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: LED 2 Change!\n");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    preLED2 = LED2;
    digitalWrite(A, 1);
    digitalWrite(B, 1);
    digitalWrite(C, 1);
    digitalWrite(D, 0);

}

if (preLED3 != LED3)
{
    bzero(buffer2, MSG_SIZE);
    strcpy(buffer2, "Event: LED 3 Change!");
    n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
    preLED3 = LED3;
    digitalWrite(A, 0);
    digitalWrite(B, 0);
    digitalWrite(C, 0);
    digitalWrite(D, 1);

}

    bzero(buffer2, MSG_SIZE);

```

```

        strcpy(buffer2, "RTU 1 finished\n");
        n = sendto(sock, buffer2, MSG_SIZE, 0, ( struct sockaddr*)&from, fromlen);
        /*****/

        sleep(1);
    }
    pthread_exit(0);
}

int main(int argc, char *argv[])
{

    /*****Socket Connection*****/
    int r;
    int port_number;

    // set the port number
    if (argc == 2)
    {
        port_number = atoi(argv[1]);
    } else
    {
        port_number = 2000; // set default if not provided
    }

    // Creates socket. Connectionless.
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
    {
        error("Opening socket");
    }

    length = sizeof(server);          // length of structure
    bzero(&server,length);           // sets all values to zero. memset() could be used
    server.sin_family = AF_INET;      // symbol constant for Internet domain
    server.sin_port = htons(port_number); // port number
    server.sin_addr.s_addr = INADDR_ANY; // IP address of the machine on which the server is running

    // gets the host name and the IP
    bzero(&ifr, sizeof(ifr)); // Set all values to zero
    ifr.ifr_addr.sa_family = AF_INET; // Type of address to retrieve - IPv4 IP address
    strncpy((char*)&ifr.ifr_name, eth0, IFNAMSIZ-1); // Copy the interface name in the ifreq
    structure
    // Get IP address
    if (ioctl(sock,SIOCGIFADDR,&ifr) == -1)
    {
        error("Cannot get IP address");
        close(sock);
        exit(0);
    }
}

```

```

    }
    // Converts the internet host address in network byte order to a string in IPv4 dotted-decimal notation
    strcpy(ip,inet_ntoa(((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr));
    strcpy(ipHolder,ip);
    int current_board_number = parseIP(ipHolder);
    printf("IP : %s\n",ip);
    printf("Board Number : %d\n",current_board_number);

    // binds the socket to the address of the host and the port number
    if (bind(sock, (struct sockaddr *)&server, length) < 0)
    {
        error("binding");
    }
    // set broadcast option
    if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &boolval, sizeof(boolval)) < 0)
    {
        printf("error setting socket options\n");
        exit(-1);
    }
    fromlen = sizeof(struct sockaddr_in);    // size of structure
    /***/

    /***Setup***/
    if (wiringPiSetup()<0)
    {
        printf("wiringPi Setup failed!\n");
        exit(-1);
    }

    // Configure the SPI
    if(wiringPiSPISetup(SPI_CHANNEL, SPI_SPEED) < 0)
    {
        printf("wiringPiSPISetup failed\n");
        return -1 ;
    }
    /***/

    /***WiringPi Initialization***/
    pinMode(S1, INPUT); //awitch 1 as input
    pinMode(S2, INPUT); //switch 2 as input

    pinMode(BTN1, INPUT); //button 1 as input
    pinMode(BTN2, INPUT); //button 2 as input

    pinMode(RED, OUTPUT); //red LED as output
    pinMode(YELLOW, OUTPUT); //yellow LED as output
    pinMode(GREEN, OUTPUT); //green LED as output

    pullUpDnControl(S1, PUD_DOWN);
    pullUpDnControl(S2, PUD_DOWN);

```

```

pullUpDnControl(BTN1, PUD_DOWN);
pullUpDnControl(BTN2, PUD_DOWN);

digitalWrite(RED, 0);
digitalWrite(YELLOW, 0);
digitalWrite(GREEN, 0);

pinMode(SEVENABLE, OUTPUT);
pinMode(A, OUTPUT);
pinMode(B, OUTPUT);
pinMode(C, OUTPUT);
pinMode(D, OUTPUT);
digitalWrite(SEVENABLE, 1);

sem_init(&my_sem, 0, 1); //init semaphore

button1 = 0;
button2 = 0;
preButton1=0;
preButton2=0;
preLED1=0;
preLED2=0;
preLED3=0;
preSwitch1=0;
preSwitch2=0;
buttonOneFlag=0;
buttonTwoFlag=0;

digitalWrite(A, 0);
digitalWrite(B, 0);
digitalWrite(C, 0);
digitalWrite(D, 0);
/*****/

/*****Open char device*****/
if ((mydev_id = open(CHAR_DEV, O_RDONLY)) == -1)
{
    printf("Cannot open device%s\n", CHAR_DEV);
    exit(1);
}
/*****/

/*****Thread to send current RTU logs and read button from kernal*****/
pthread_t ptr,ptr2;
pthread_create(&ptr, NULL, periodicUpdate, NULL);
pthread_create(&ptr2, NULL, readFromKernal, NULL);
/*****/

while (1)

```

```

{
    // receive from client
    bzero(&buffer,MSG_SIZE); // clear the buffer to NULL
    n = recvfrom(sock, buffer, MSG_SIZE, 0, (struct sockaddr *)&from, &fromlen);
    if (n < 0)
        error("recvfrom");

    printf("Received a command. It says: %s\n",buffer);

    if(strcmp(buffer, "LED1ON") == 0)
    {
        digitalWrite(REDA, 1);
        sem_wait(&my_sem);
        LED1 = 1;
        sem_post(&my_sem);
    }

    if(strcmp(buffer, "LED2ON") == 0)
    {
        digitalWrite(YELLOW, 1);
        sem_wait(&my_sem);
        LED2 = 1;
        sem_post(&my_sem);
    }

    if(strcmp(buffer, "LED3ON") == 0)
    {
        digitalWrite(GREEN, 1);
        sem_wait(&my_sem);
        LED3 = 1;
        sem_post(&my_sem);
    }

    if(strcmp(buffer, "LED1OFF") == 0)
    {
        digitalWrite(REDA, 0);
        sem_wait(&my_sem);
        LED1 = 0;
        sem_post(&my_sem);
    }

    if(strcmp(buffer, "LED2OFF") == 0)
    {
        digitalWrite(YELLOW, 0);
        sem_wait(&my_sem);
        LED2 = 0;
        sem_post(&my_sem);
    }

    if(strcmp(buffer, "LED3OFF") == 0)

```

```
{
    digitalWrite(GREEN, 0);
    sem_wait(&my_sem);
    LED3 = 0;
    sem_post(&my_sem);
}

return 0;
}
```