Dwayne Williams

Final Project Report

Abstract

For my project, I created a SCADA (supervisory control and data acquisition) system. The system contains an historian that receives updates from RTU's that monitors their connected sensors.

Introduction

In this project, I used the TS-7250 boards in the lab, these boards acted as RTU's. Connected to the board were a simple circuit I found online that produced a sine wave and an auxiliary board with five buttons that simulated switches. The RTU's monitored if the voltage was over or under its normal operating values, if the voltage remained constant which would mean that there is no power and the status of four of the button (on or off). Every second the RTU's collects all the data, the time of the update and any events that may have occurred and the time they occurred in that second and sends it to the historian. The historian than logs the events in to files. If no events occur the RTU's only send the status of all that it monitors and the time of the update. The historian logs the events into one file and the periodic updates into another to better keep track of all the updates. If a user asks the historian reads both files, copies its content, sorts them and prints the contents together in order of occurrence.

Background

For most of this project, I used what we did in the labs. For instance, "Proj_SW" is almost exactly the module I created in lab 6. For the socket connection, I used the code provided for us for use in lab five to create client/server tcp connection between the RTU and historian. Throughout the time is spent I coding this project I used the internet to find the best way of performing certain small task. such as sorting or converting variables.

Implementation

I used four different programs for this project two kernel modules and two user space programs. One of the kernel modules, “Proj_ADC”, strictly handled the Analog to Digital conversion. Reading values produced by circuit connected to the board and converted it to numbers and sends it through a fifo to the RTU. The other kernel module, “Proj_SW”, simply uses interrupts to detect the pressing of buttons on the auxiliary boards. When it detects a button press it send a number representing the button which was pressed on the board to the RTU through a fifo. this number corresponds to the order of the buttons in on the board. “Proj_RTU” receives the data from the two kernel modules within pthreads and stores them in a global structure that keeps track of the data.

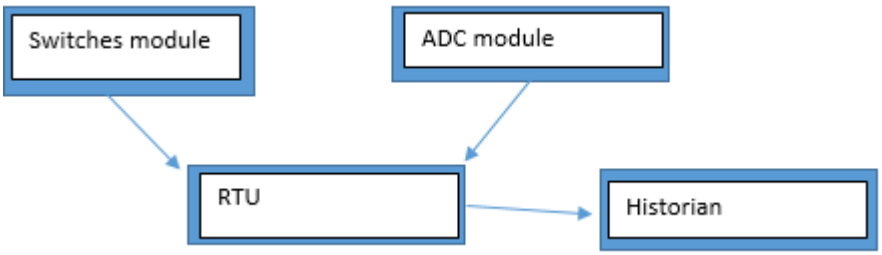
One pthread waits for “Proj_SW” to send a number, when it receives that number it records the time the number arrived and changes the globaly declared variable that monitors the position of the button. All buttons are initially off (set to 0) when the RTU receives a number It checks if the variable is 0 or 1. If the variable is one it is changed to zero and the messages says that the switch was turned off, if the variable is zero the variable is changed to one and the messages says that the switch was turned on.

Another pthread receives the values from “Proj_ADC”, and saves them to a global variable that is constantly updated every time a value is received. If the it receives a value that is above a value that is above an indicated threshold a message saying the line is being overloaded is created. If the value is below an indicated value a message saying it is below normal operating operations is created. It also keeps track of if the voltages have remained constant for a couple seconds, if it has it creates a message saying that there is no power at line. The messages will occur every update until the values go back to normal.

In the main function the program grabs the values stored within the global variables and stores them in to a data structure. This structures contains the statues of every button, the currant voltage at the line, the board number, the time of the update and the event messages along with their timestamps. This data is sent to “Proj_H”. If the RTU is disconnected from the network, it goes into offline mode and stores the updates into an array and sends them when it reconnects.

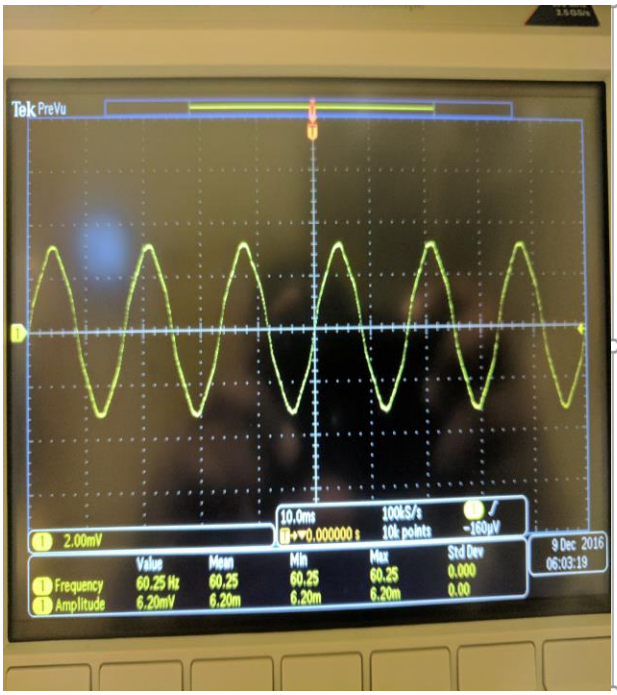
Throughout the RTU I used two semaphores and two flags so if one event is being written it would not stop another event from being written in another part of the program so no events would be missed.

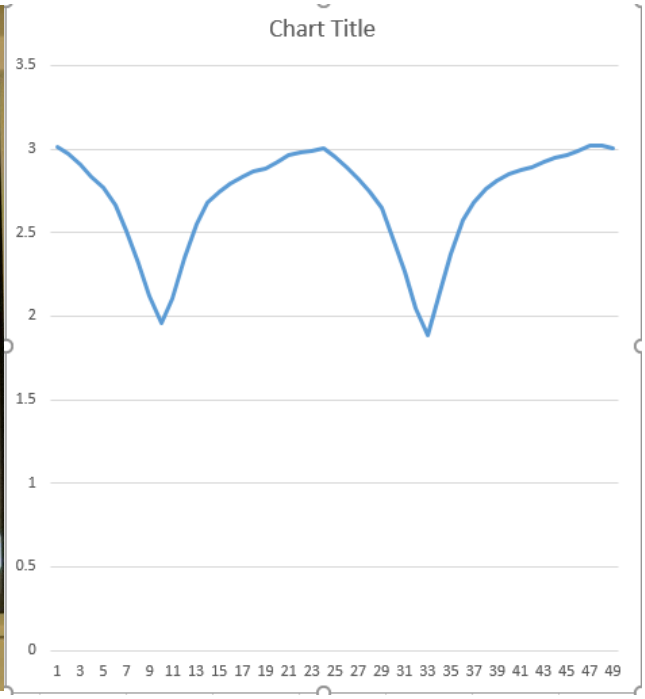
"Proj_H" receives the data from the RTU and creates two log files. One log file contains the periodic updates, the standard update, the other log contains the events. When the data is received, the data is appended to the files. These files are cleared every time the program is ran. If the user requests it the program copies the contents of both files and stores them in arrays. The arrays are sorted individually, using select sort, in order of the time of the update, then they are sorted together and displayed in the terminal. If the user wishes he can also clear the logs.



Experiment and Results

While coding this project, I have tested my code a lot, I can't provide a number, to test my ADC I used a test program that simply received a hundred voltage values and plotted the graph below and compared it to a wave I received testing my circuit using an oscilloscope. The values from the test program aren't received as fast as in the oscilloscope so the graphs are not exactly alike but they are similar in that they oscillate. To make sure my program was working properly I constantly tested with every part running. This helped me determine where any mistake made may have occurred.





Discussion and Conclusion

My program mostly runs as intended. While coding, I ran into a problem with the Analog to Digital conversion. At first the when I ran the program I received large numbers event though I correctly converted it in the user-space, I found out I was incorrectly declaring the variables being sent and received. I corrected this by better declaring them. Also, I also had issues when sorting the logs sorting the periodic logs and events originally caused a segmentation fault when I attempted to sort with one file. To fix this I found a selection sort algorithm online, which was better than what I was originally doing, and I also split the logs into two files which also helped with keeping track of the events. The code is also not perfect. At random times the program fails to connect and if this happens the board must reset. And if the buttons are pressed excessively it will cause a segmentation fault. I am also not sure how effect my approach with the semaphores are ideally it works as intended but I was not sure how to test my implementation. My use of the flags to make sure that the log is being written to could also be improved but I did not have the time to test alternatives for this. Also, when an event occurs which involves the voltage it prints out more than times than necessary this creates more events than are necessary and litters the

array. I've tried to correct this but I often ran into issues with my changes and though it's not efficient it works. Over all though not perfect the program for the most part runs correctly and I gained experience with this project, I feel I have a better understanding of the concepts learned in this class and how combined they can be very useful.

Code

Proj_RTU.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <sys/time.h>
#include <semaphore.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_lxrt.h>
#include <signal.h>

#define MSG_SIZE 100
#define EVNT_SIZE 10000
```

```
//holds event information
struct event
{
	double stmp;
	char evnt_type[MSG_SIZE];
};

//struct that holds information for log file
typedef struct
{
	int rtu;
	double stmp;
	float volts;
	struct event events[50];
	int S1;
	int S2;
	int S3;
	int S4;
	int num_events;
	int act_events;
}log;

void switch1(void* ptr);
void adc1(void* ptr);
void exit1(void* ptr);
void printlog(int a);
double getstmp(struct timeval now);
//void child( int sox, struct sockaddr_in server, socklen_t serverlength, struct timeval start);

char buffer[MSG_SIZE];
```

```
struct timeval start;

sem_t sem;
sem_t sem2;

int ex=1;

log log1;
log logArray[EVNT_SIZE];

int evntnum = 0;
int dc_evntnum = 0;

int cyc = 0;

int num = 0;
int but1 = 0;
int but2 = 0;
int but3 = 0;
int but4 = 0;

float volts = 0;
float sent_volts = 0;
int cnt = 0;

int flag = 0;

RT_TASK *rt_process;
RTIME period;

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

```
{

        pthread_t thrd1, thrd2, thrd3;

        if (sem_init(&sem,0, 1) == -1)
        {
                printf("Error creating Semaphore");
                exit(2);
        }

        if (sem_init(&sem2,0, 1) == -1)
        {
                printf("Error creating Semaphore");
                exit(2);
        }

        int sox, length, n;//, flag = 0
        char addr[MSG_SIZE], ip[MSG_SIZE] , serveraddr[MSG_SIZE];
        char boardnum[MSG_SIZE];
        struct hostent *serv;
        int chk = 0;

        printf("Enter Historian board #: "); // get board number of the historian
        scanf("%d", &num);

        //get rtu address
        gethostname(ip, sizeof(ip));
        struct hostent *ipADDR;
        struct in_addr **boardIPlist;
        ipADDR = (struct hostent *) gethostbyname(ip);
        boardIPlist = (struct in_addr **) ipADDR -> h_addr_list;
        strcpy(addr, inet_ntoa(*boardIPlist[0]));
```

```
        //unsigned long convaddr = nam2num(serveraddr);

        char tmp[MSG_SIZE];
        strcpy( tmp, addr);

        char* addrtoken[4]; //parse address of RTU for parts
        addrtoken[0] = strtok(tmp, ".");
        addrtoken[1] = strtok(NULL, ".");
        addrtoken[2] = strtok(NULL, ".");
        addrtoken[3] = strtok(NULL, ".");
        sprintf(boardnum, "%d", num);
        printf("Board IP: %s\n", addr);

        // creat the string that contains the historian address
        strcpy(serveraddr, addrtoken[0]); // construct the string containing the client address
        strcat(serveraddr, ".");
        strcat(serveraddr, addrtoken[1]);
        strcat(serveraddr, ".");
        strcat(serveraddr, addrtoken[2]);
        strcat(serveraddr, ".");
        strcat(serveraddr, boardnum);

        printf("connecting to board %s through port %s\n",serveraddr,argv[1]);
//      printf("size of addr %d\n",strlen(serveraddr));
        struct sockaddr_in server; //from;
        //struct in_addr sox_in_ad;

        log1.rtu = atoi(addrtoken[3]);

        sox = socket(AF_INET, SOCK_STREAM, 0); // socket file descriptor
```

```
length = sizeof(server);                    // length of structure

serv = gethostbyname(serveraddr);
if (serv == NULL)
{
        printf("ERROR, no such host\n");
        exit(2);
}
bzero(&server,length);

server.sin_port = htons(atoi(argv[1]));
bcopy((char *)serv->h_addr, (char *)&server.sin_addr.s_addr, serv->h_length);
server.sin_family = AF_INET;

//      bind( sox, (struct sockaddr *) &server, length); // bind socket

socklen_t serverlength = sizeof(struct sockaddr_in);
if(connect(sox,(struct sockaddr *) &server, serverlength) < 0)
{
        printf("Not connected\n");
        close(sox);
        exit(2);
}

struct timeval now;

//create pthreads
pthread_create(&thrd3, NULL, (void *)&exit1, argv[1]);
pthread_create(&thrd1, NULL, (void *)&switch1, NULL);
pthread_create(&thrd2, NULL, (void *)&adc1, NULL);

while (ex != 0)
```

```
{


        if(flag == 0)
        {

                sem_wait(&sem); // wait for semaphore


                gettimeofday(&now, NULL); // get time of the log being sent


                log1.stmp = getstmp(now); // store time
                log1.volts = volts; // store current voltage values
                if ( volts == sent_volts) // if the current voltage is equal to the previous voltage increment
counter
                {
                        cnt++;
                }

                else if ( volts != sent_volts) // if voltage is not the same as the previous voltage reset the
counter
                {
                        cnt = 0;
                }

                sent_volts = log1.volts; // save the current voltage in to the previous voltage variable
                log1.act_events += evntnum; // increment total number of events
                log1.num_events = evntnum; // store the number of events that occured this period
                //store status of the buttons
                log1.S1 = but1;
                log1.S2 = but2;
                log1.S3 = but3;
                log1.S4 = but4;
```

```
n = sendto(sox, &log1, sizeof(log),0,(const struct sockaddr *) &server, serverlength);//
send struct to historian
if(n < 0 && chk == 0)
{
	if(connect(sox,(struct sockaddr *) &server, serverlength) < 0) // if the
connection is lost try to reconnect
	{
		printf("Error connecting to socket\n Entering offline mode...\n");
		chk++;
	}
}

else if(n < 0 && chk == 1)
{
	if(connect(sox,(struct sockaddr *) &server, serverlength) < 0)
	{
		printf("log saved");
		logArray[dc_evntnum] = log1;
		dc_evntnum++;
	}
}

else if(n == sizeof(log) && chk == 1)
{
	chk =0;

	while ( dc_evntnum > 0)
	{

		printf("Sending Lost Data...\n");
		n = sendto(sox, &log1, sizeof(log),0,(const struct sockaddr *) &server,
serverlength);
		if (n < 0)
```

```
                                        {
                                                printf("Error writing socket 2\n");
                                                exit(2);
                                        }

                                        dc_evntnum--;
                                        rt_task_wait_period();

                                }

                        }

                        else
                        {

//                              printlog(1);

                                while (evntnum > 0) // clear the current number of events
                                {
                                        bzero(log1.events[evntnum].evnt_type,1000);
                                        evntnum--;
                                }

                        }

                        flag = 1; // change the flag

                }

                // same as above
                else if (flag == 1)
                {
```

```
sem_wait(&sem2);

gettimeofday(&now, NULL);

log1.stmp = getstmp(now);
log1.volts = volts;
if ( volts == sent_volts)
{
        cnt++;
}

else if ( volts != sent_volts)
{
        cnt = 0;
}

sent_volts = log1.volts;
log1.act_events += evntnum;
log1.num_events = evntnum;
log1.S1 = but1;
log1.S2 = but2;
log1.S3 = but3;
log1.S4 = but4;

n = sendto(sox, &log1, sizeof(log),0,(const struct sockaddr *) &server, serverlength);//
send message
if(n < 0 && chk == 0)
{
        if(connect(sox,(struct sockaddr *) &server, serverlength) < 0)
        {
                printf("Error connecting to socket\n Entering offline mode...\n");
                chk++;
```

```
                }
        }

        else if(n < 0 && chk == 1)
        {
                if(connect(sox,(struct sockaddr *) &server, serverlength) < 0)
                {
                        printf("log saved");
                        logArray[dc_evntnum] = log1;
                        dc_evntnum++;
                }
        }

        else if(n == sizeof(log) && chk == 1)
        {
                chk =0;

                while ( dc_evntnum > 0)
                {

                        printf("Sending Lost Data...\n");
                        n = sendto(sox, &log1, sizeof(log),0,(const struct sockaddr *) &server,
serverlength);
                        if (n < 0)
                        {
                                printf("Error writing socket 2\n");
                                exit(2);
                        }

                        dc_evntnum--;
                        rt_task_wait_period();
```

```
                                        }

                                }

                                else
                                {

//                                      printlog(2);

                                        while (evntnum > 0)
                                        {

                                                bzero(log1.events[evntnum].evnt_type,1000);
                                                evntnum--;

                                        }

                                }

                                flag = 0;

                        }

                        usleep(1000000); // wait a second

                                sem_post(&sem); //release semaphores
                                sem_post(&sem2);

        }

        close(sox); // close the socket file descriptor
```

```
	return 0;
}

void switch1(void* ptr)
{



	int fifo_in, swch;
	struct timeval now;
	fifo_in = open("/dev/rtf/2", O_RDWR);//open fifo for reading

	while(ex != 0)
	{

		read(fifo_in, &swch, sizeof(swch)); // wait for the kernel to send a button number throug the fifo



		if (flag == 0)
		{
			sem_wait(&sem);
			flag = 1;
			gettimeofday(&now, NULL); // get the time the number was recieved
			log1.events[evntnum].stmp = getstmp(now); // save time
			if ( swch == 1 && but1 == 0) // if the button is off and the fifo send its number change its status to on and create an event message for it
			{
				but1 = 1;
				sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on ", swch);
				//				printf("%s",log1.events[evntnum].evnt_type);


			}
```

```
                else if ( swch == 1 && but1 == 1)// if the button is on and the fifo send its number change its status to off and create an event message for it
                {
                        but1 = 0;
                        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off ", swch);
                        //                              printf("%s",log1.events[evntnum].evnt_type);

                }

                else if ( swch == 2 && but2 == 0)
                {
                        but2 = 1;
                        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on ", swch);
                        //                              printf("%s",log1.events[evntnum].evnt_type);

                }

                else if ( swch == 2 && but2 == 1)
                {
                        but2 = 0;
                        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off ", swch);
                        //                              printf("%s",log1.events[evntnum].evnt_type);

                }

                else if ( swch == 3 && but3 == 0)
                {
                        but3 = 1;
                        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on ", swch);
                        //                              printf("%s",log1.events[evntnum].evnt_type);

                }
```

```
        else if ( swch == 3 && but3 == 1)
        {
                but3 = 0;                       rt_task_wait_period();
                sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off ", swch);
                //                              printf("%s",log1.events[evntnum].evnt_type);

        }

        else if ( swch == 4 && but4 == 0)
        {
                but4 = 1;
                sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on ", swch);
                //                              printf("%s",log1.events[evntnum].evnt_type);

        }

        else if ( swch == 4 && but4 == 1)
        {
                but4 = 0;
                sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off ", swch);
                //                              printf("%s",log1.events[evntnum].evnt_type);

        }

        evntnum++; // increment the event counter
        sem_post(&sem);
}

if (flag == 1)
{
        sem_wait(&sem2);
```

```
gettimeofday(&now, NULL);
log1.events[evntnum].stmp = getstmp(now);
if ( swch == 1 && but1 == 0)
{
        but1 = 1;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 1 && but1 == 1)
{
        but1 = 0;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 2 && but2 == 0)
{
        but2 = 1;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 2 && but2 == 1)
{
        but2 = 0;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);
```

```
}

else if ( swch == 3 && but3 == 0)
{
        but3 = 1;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 3 && but3 == 1)
{
        but3 = 0;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 4 && but4 == 0)
{
        but4 = 1;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned on", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);

}

else if ( swch == 4 && but4 == 1)
{
        but4 = 0;
        sprintf(log1.events[evntnum].evnt_type, "Switch%u: turned off", swch);
        //                      printf("%s",log1.events[evntnum].evnt_type);
```

```
                        }


                        evntnum++;

                        sem_post(&sem2);
                }


        }

        pthread_exit(0);

}


void adc1(void* ptr)

{


        int fifo_in;

        int noP = 2; // the number of second the program waits until it creates  no power event wait = noP+1

        float overld = 4, Low = 1.5; // above overld and a line overload event is created under low and a below normal operation event is created

        unsigned short volt;

        struct timeval now;


        fifo_in = open("/dev/rtf/3", O_RDWR); // open fifo


        while(ex != 0)

        {


                if (read(fifo_in, &volt, sizeof(unsigned short)) != sizeof(unsigned short)) //read fifo

                {

                        printf("Error with ADC fifo\n");

                        exit(2);

                }


                gettimeofday(&now, NULL); // get time voltager recieved
```

```
volts = (float)(5 * (float)volt) / (float)4095; // vonvert the value receved from fifo


if (cnt >= noP) // of the wait counter is greater or equal to noP then creat an event message
{

        if (flag == 0)
        {

                sem_wait(&sem);
                log1.events[evntnum].stmp = getstmp(now);// get time of event
                sprintf(log1.events[evntnum].evnt_type, "No Power"); // create no power event
message
                evntnum++; // increament number of events
                sem_post(&sem);

        }

        if (flag == 1)
        {

                sem_wait(&sem2);
                log1.events[evntnum].stmp = getstmp(now);
                sprintf(log1.events[evntnum].evnt_type, "No Power");
                evntnum++;
                sem_post(&sem2);

        }

        while(cnt >= noP ) // while the wait conter is greater than noP continou to produce the
event message
```

```
{
        if (read(fifo_in, &volt, sizeof(unsigned short)) != sizeof(unsigned short))
        {
                printf("Error with ADC fifo\n");
                exit(2);
        }

        gettimeofday(&now, NULL);

        volts = (float)(5 * (float)volt) / (float)4095;

        if (flag == 0)
        {
                if (cnt >= noP)
                {
                        sem_wait(&sem);
                        log1.events[evntnum].stmp = getstmp(now);
                        sprintf(log1.events[evntnum].evnt_type, "Still No Power");
                        evntnum++;
                        sem_post(&sem);
                }
        }

        else if (flag == 1)
        {
                if (cnt >= noP)
                {
                        sem_wait(&sem2);
                        log1.events[evntnum].stmp = getstmp(now);
                        sprintf(log1.events[evntnum].evnt_type, "Still No Power");
                        evntnum++;
                        sem_post(&sem2);
```

```
                        }
                }

        }
}

else if (volts > overld) // if the volatage is above normal operations
{
        if (flag == 0)
        {
                sem_wait(&sem);
                log1.events[evntnum].stmp = getstmp(now);
                sprintf(log1.events[evntnum].evnt_type, "Line overload: V > %f", overld);
                evntnum++;
                sem_post(&sem);
        }

        if (flag == 1)
        {
                sem_wait(&sem2);
                log1.events[evntnum].stmp = getstmp(now);
                sprintf(log1.events[evntnum].evnt_type, "Line overload: V > %f", overld);
                evntnum++;
                sem_post(&sem2);
        }

}

else if (volts < Low)// if the voltage is below normal operations
{
        if (flag == 0)
        {
```

```
                                        sem_wait(&sem);
                                        log1.events[evntnum].stmp = getstmp(now);
                                        sprintf(log1.events[evntnum].evnt_type, "Line below normal operation: V < %f",
Low);
                                        evntnum++;
                                        sem_post(&sem);
                                }

                                if (flag == 1)
                                {
                                        sem_wait(&sem2);
                                        log1.events[evntnum].stmp = getstmp(now);
                                        sprintf(log1.events[evntnum].evnt_type, "Line below normal operation: V < %f",
Low);
                                        evntnum++;
                                        sem_post(&sem2);
                                }

                        }


        }

        pthread_exit(0);

}


void exit1(void* ptr) //to exit
{

        while (ex != 0)
        {
```

```
		if(num != 0)
		{
			printf("Enter 0 to exit\n");
			scanf("%d",&ex);
		}

		else if (num != 0 && ex != 0)
		{
			ex =20;
			printf("Enter 0 to exit\n");
			scanf("%d",&ex);
		}
	}



}



void printlog(int a) // used for debugging
{
	int i;

	printf("(log %d) %d, %lf, %f, %d %d\n", a,log1.rtu,log1.stmp, log1.volts, log1.act_events, log1.num_events);

	for (i = 0; i < evntnum; i++)
	{
		//			printf(" %ld.%06ld %ld.%06ld\n"log1.events[i].end.tv_sec,
log1.events[i].end.tv_usec, log1.events[i].start.tv_sec , log1.events[i].start.tv_usec);

		printf("(log %d) %lf %s\n", a, log1.events[i].stmp,log1.events[i].evnt_type);
	}
```

```
	pthread_exit(0);

}

double getstmp(struct timeval now) // converts struct timeval into a number of type double
{

	double stmp =0;

	stmp = (double) (now.tv_usec) ;
	stmp /= 1000000;
	stmp += (double)now.tv_sec;

	return stmp;

}
```

Proj_SW.c

```
#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
```

```
#include <linux/time.h>
#include <asm/delay.h>

MODULE_LICENSE("GPL");

//global vars
static RT_TASK tsk1;
RTIME period;

unsigned long *PFDR,*PFDDR;
unsigned long *PBDR,*PBDDR;
unsigned long *IntEn, *Type1,*Type2,*EOI ,*RawInt,*DeBo;
unsigned long *VIC2_soft,*VIC2_IntEn,*VIC2_clear;

char input;

typedef struct timeval TIME;

void ISR_HW(unsigned irq_num, void *cookie)
{

	rt_disable_irq(59);  //disable 59
	int output;



	if((*RawInt & 0x01))// check button 1
	{
//			printk("sent 1");
			output = 1;
			rtf_put(2, &output, sizeof(int)); //send button number through fifo to RTU
	}
	if((*RawInt & 0x02))
	{
			output = 2;
```

```
                rtf_put(2, &output, sizeof(int));
        }
        if((*RawInt & 0x04))
        {
                output = 3;
                rtf_put(2, &output, sizeof(int));
        }
        if((*RawInt & 0x08))
        {
                output = 4;
                rtf_put(2, &output, sizeof(int));
        }


        *EOI |= 0x1F;//clear interrupts
        rt_enable_irq(59);  //enable 59
}

int init_module(void)
{

        unsigned long *ptr;

        ptr = (unsigned long*) __ioremap(0x80840000, 4096, 0); // map 0x80840000

        //point to registers
        PBDDR = ptr + 5;
        PBDR = ptr + 1;
        PFDDR = ptr + 13;
        PFDR = ptr + 12;
   *PBDDR |= 0xE0;
        *PFDDR |= 0x02;

        Type1 = ptr + 43;//GPIOBTYPE2
```

```
	Type2 = ptr + 44;//GPIOBTYPE2
	EOI = ptr + 45;//GPIOBEOI
	IntEn = ptr + 46;//GPIOBINTEN
	RawInt = ptr + 48;   //RAWINTSTSB
	DeBo = ptr + 49; //GPIOBDebounce

	*DeBo |= 0x1F;
	*IntEn |= 0x1F;
	*EOI |= 0x1F;
	*Type2 &= 0xE0;
	*Type1 |= 0x1F;

	rtf_create(2, sizeof(int)); // create fifo

	rt_request_irq(59, ISR_HW, 0, 1);
	*EOI |= 0x1F;
	*IntEn |= 0x1F;
	rt_enable_irq(59);

	return 0;

}

void cleanup_module(void)
{

	rt_task_delete(&tsk1);//deletes real time task
	rtf_destroy(2);
	rt_disable_irq(59);
	rt_release_irq(59);
	stop_rt_timer();//stops timer

}
```

Proj_H.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <sys/time.h>
#include <semaphore.h>
#include <signal.h>

#define MSG_SIZE 100
#define LOG_SIZE 10000

int buffLoc1;
sem_t sem;

struct event
{
        double stmp;
        char evnt_type[MSG_SIZE];
};

//struct that holds information for log file
typedef struct
{
        int rtu;
        double stmp;
```

```
        float volts;
        struct event events[50];
        int S1;
        int S2;
        int S3;
        int S4;
        int num_events;
        int act_events;
}log;

int pr = 3;

void recvLog(int sox);
void exit1(void* ptr);
void print();
const char* On_off(int i);


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

        pthread_t thrd;

        FILE *fp = fopen("log.txt", "w"); // clear log files
        fclose(fp);
        FILE *fp2 = fopen("events_log.txt", "w");
        fclose(fp2);

        pthread_create(&thrd, NULL, (void *)&exit1, NULL);

        int sox,sox2, length,i=0,pid;//, flag = 0

        struct sockaddr_in server, client; //from;
        //struct in_addr sox_in_ad;
```

```
sox = socket(AF_INET, SOCK_STREAM, 0); // socket file descriptor

length = sizeof(server);                                // length of structure
bzero(&server,length);

server.sin_port = htons(atoi(argv[1]));
//          printf("%d\n", server.sin_port);
server.sin_addr.s_addr = INADDR_ANY;
server.sin_family = AF_INET;

bind( sox, (struct sockaddr *) &server, length); // bind socket

listen(sox, 5);
socklen_t clientlength = sizeof(struct sockaddr_in);

while (pr != 0)
{
        sox2 = accept(sox, (struct sockaddr *) &client, &clientlength); // wait for connection to be made

        i++;
        pid = fork(); // fork process

        if(sox2 < 0)
        {
                printf("error on accept\n");
                exit(2);
        }

        if (pid == 0)
        {
                printf("Connection #%d created\n",i);
                close(sox);
                recvLog(sox2);
```

```
                exit(0);
        }

        if (pr == 0)
        {
                break;
        }

        else
        {
                close(sox2);
                signal(SIGCHLD,SIG_IGN);       // to avoid zombie problem
        }

    }

    close(sox);
    return 0;
}

void recvLog(int sox)
{
    int i=0;

    log buffer;
    buffer.rtu = 0;
    buffer.stmp = 0;

    while( pr != 0)
    {

        if(recvfrom(sox, &buffer, sizeof(log), MSG_WAITALL, NULL, NULL) == sizeof(log))// recvfrom() could be used
        {
```

```
                                FILE *fp = fopen("log.txt", "a");//open file in append mode

                                fprintf(fp, "%lf: RTU %d, Voltage at line %f, Events %d, Switches %s %s %s %s\n", buffer.stmp,
buffer.rtu, buffer.volts, buffer.act_events,On_off(buffer.S1),On_off(buffer.S2),On_off(buffer.S3),On_off(buffer.S4)); //write to
file

                                fclose(fp);

                                //append each event to the end of the file

                                FILE *fp2 = fopen("events_log.txt", "a");//open file in append mode

                                for(i=0; i<buffer.num_events; i++)

                                {

                                        fprintf(fp2, "%lf: RTU %d \"%s\"\n", buffer.events[i].stmp, buffer.rtu,
buffer.events[i].evnt_type);

                                }

                                //append each event to the end of the file

                                fclose(fp2);


                        }


                        else

                        {

                                close(sox);

                        }

                }


                close(sox);


}


void exit1(void* ptr) // waits for user input

{




                while(pr != 0) //

                {

                        if (pr != 0 || pr != 1 || pr != 2)

                        {
```

```
                printf("Enter 0 to exit, 1 to print log or 2 to clear log.\n");
                scanf("%d", &pr);
        }

        if (pr == 1)
        {
                print();

                pr = 3;
        }

        else if (pr == 2)
        {
                FILE *fp = fopen("log.txt", "w");
                fclose(fp);
                FILE *fp2 = fopen("events_log.txt", "w");
                fclose(fp2);
                printf("Log cleared.\n");
        }

                usleep(1000000);

    }
    pthread_exit(0);
}

void print( )
{

    char tmp[LOG_SIZE];

    int i=0, j,k;

    char buffer[MSG_SIZE];
```

```
char bufferArray[LOG_SIZE][MSG_SIZE];

double stmp1 = 1;
double stmp2 = 1;



FILE *fp = fopen("log.txt", "r");

while(fgets(tmp, sizeof(tmp), fp) != NULL) // copy contents of file
{

        sprintf(bufferArray[i], "%s", tmp);

        if (i >= LOG_SIZE)
        {
                printf("Error to many elements");
                fclose(fp);
                pr = 1;
                usleep(1000000);
        }

        i++;

}

fclose(fp);



int rtu;
int evntnum=0;
float volt;
char S1[4];
char S2[4];
```

```
	char S3[4];
	char S4[4];



	for(j=0; j < i; j++) // selection sort
	{
		k = j;


		sscanf(bufferArray[k],"%lf: RTU %d, Voltage at line %f, Events %d, Switches %s %s %s %s\n", &stmp1, &rtu,
&volt, &evntnum,S1,S2,S3,S4);
		sscanf(bufferArray[k-1],"%lf: RTU %d, Voltage at line %f, Events %d, Switches %s %s %s %s\n", &stmp2,
&rtu, &volt, &evntnum,S1,S2,S3,S4);


		while(k > 0 && stmp1 < stmp2)
		{
			strcpy(buffer, bufferArray[k]);
			strcpy(bufferArray[k], bufferArray[k-1]);
			strcpy(bufferArray[k-1], buffer);
			k--;
		}


	}


	char tmp2[LOG_SIZE];


	int i2=0, j2,k2;


	char buffer2[MSG_SIZE];
	char bufferArray2[LOG_SIZE][MSG_SIZE];


	static double allstmp[LOG_SIZE];
	static double allstmp2[LOG_SIZE];


	//repeat for event log file
```

```
        FILE *fp2 = fopen("events_log.txt", "r");

        while(fgets(tmp2, sizeof(tmp2), fp2) != NULL)
        {
                sprintf(bufferArray2[i2], "%s", tmp2);

                if (i2 >= LOG_SIZE)
                {
                        printf("Error to many elements");
                        fclose(fp2);
                        pr = 1;
                        usleep(1000000);
                }

                i2++;

        }

        fclose(fp2);

//      int rtu;
        char events[MSG_SIZE];

        for(j2=0; j2 < i2; j2++)
        {
                k2 = j2;

                sscanf(bufferArray2[k2],"%lf: RTU %d \"%s\"\n", &stmp1, &rtu, events);
                sscanf(bufferArray2[k2-1],"%lf: RTU %d \"%s\"\n", &stmp2, &rtu, events);

                while(k2 > 0 && stmp1 < stmp2)
                {
                        strcpy(buffer2, bufferArray2[k2]);
                        strcpy(bufferArray2[k2], bufferArray2[k2-1]);
```

```
                    strcpy(bufferArray2[k2-1], buffer2);
                    k2--;
                }

        }

        for(j2=0; j2 < i2; j2++)
        {

                sscanf(bufferArray2[j2],"%lf: RTU %d \"%s\"\n", &allstmp2[j2], &rtu, events); //get timestamp of periodic values and store them into an array

        }

        for(j=0; j < i; j++)
        {
                sscanf(bufferArray[j],"%lf: RTU %d, Voltage at line %f, Events %d\n", &allstmp[j], &rtu, &volt, &evntnum);//get timestamp of periodic values and store them into an array

        }

        j= j2 = 0;
        //sort and print the both sorted arrays in order of time
          while (j < i && j2 < i2)
          {

                if(allstmp[j] == i)
                    printf("%s", bufferArray[j]);

                else if(allstmp2[j2] == i2)
                    printf("%s", bufferArray[j2]);

                else if (allstmp[j] < allstmp2[j2])
            {
              printf("%s", bufferArray[j]);
```

```
      j++;
     }

     else
     {
      printf("%s", bufferArray2[j2]);
      j2++;
     }

    }

    /* Print remaining elements of the larger array */
    while(j < i)
    {
     printf("%s", bufferArray[j]);
     j++;
    }
    while(j2 < i2)
    {
     printf("%s", bufferArray2[j2]);
     j2++;
    }


}

const char* On_off(int i) // used to convert the button values recieved from the RTU and sends on or off
{
        if (i == 1)
                return "On";

        else if (i== 0)
                return "off";
```

```
        else
                return "N/A";

}
```

Proj_ADC.c

```
#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include <linux/time.h>



MODULE_LICENSE("GPL");

static RT_TASK tsk1;
RTIME period, period2;

char* High;
short* Low;
unsigned long* install;
unsigned char* Conversion;
```

```
static void rt_process(int t)
{
	unsigned short volt = 2300;

	if((*install & 0x1) == 1)
	{

		while(1)
		{

			*High = 0x40; //determine channel

			while((*Conversion >> 7) == 1) // poll bit seven until conversion is complete
			{
				rt_sleep(nano2count(100000));//wait .1ms
			}

			volt = *Low & 0xFFF; // save the voltage avaliable in the least significant bit

			rtf_put(3, &volt, sizeof(unsigned short)); // semd voltage through fifo
			rt_task_wait_period(); // wait period
		}
	}

}


int init_module(void)
{
	rtf_create(3, sizeof(unsigned short));
```

```
	High = (char*) __ioremap(0x10F00000, 4096, 0);
	Low = (short*) __ioremap(0x10F00000, 4096, 0);
	install = (unsigned long*) __ioremap(0x22400000, 4096, 0);
	Conversion = (unsigned char*) __ioremap(0x10800000, 4096, 0);

	rt_set_periodic_mode();//set to periodic mode
	period = start_rt_timer(nano2count(100000000));//.1ms
	rt_task_init(&tsk1, rt_process, 0, 256, 0, 0, 0);
	rt_task_make_periodic(&tsk1, rt_get_time(), period2);

	return 0;
}

void cleanup_module(void)
{

	rt_task_delete(&tsk1);//deletes real time task
	rtf_destroy(3);
	stop_rt_timer();//stops timer

}
```