Thomas Nabelek
Portfolio: SCADA simulator
Fall 2015

The following is a final project completed for the Real Time Embedded Computing course at the University of Missouri in Fall 2015. The project was a supervisory control and data acquisition (SCADA) system simulator written in C and compiled for desktop computers and TS-7250s (single board computers with a EP9302 ARM9 CPU). The TS-7250 boards ran a non-RT Linux OS that had been patched with a Real Time Application Interface (RTAI) to provide real time services.

The project was meant to incorporate topics covered in the class including periodic and realtime processing, TCP socket network communication, client-server model, multithreading, task and thread cooperation and synchronization, intertask pipe communication, kernel modules, user space tasks, and hardware and software service interrupts.

A video demo of the completed project can be found at nabelekt.com/course_projects/ECE4220/SCADA_sim/demo.mp4.

Thomas Nabelek
November 16, 2015
ECE 4220 - Project Proposal

I propose the creation of a supervisory control and data acquisition (SCADA) system capable of managing several remote terminal units (RTUs) for the purpose of collecting sensor data in near real time and controlling instrumentation. This system will log all activity – with time values indicating when a given event occurs, will allow for synchronous execution of tasks so that data can be received an acted upon immediately,

Implementation, Assumptions, Constraints, Specifications

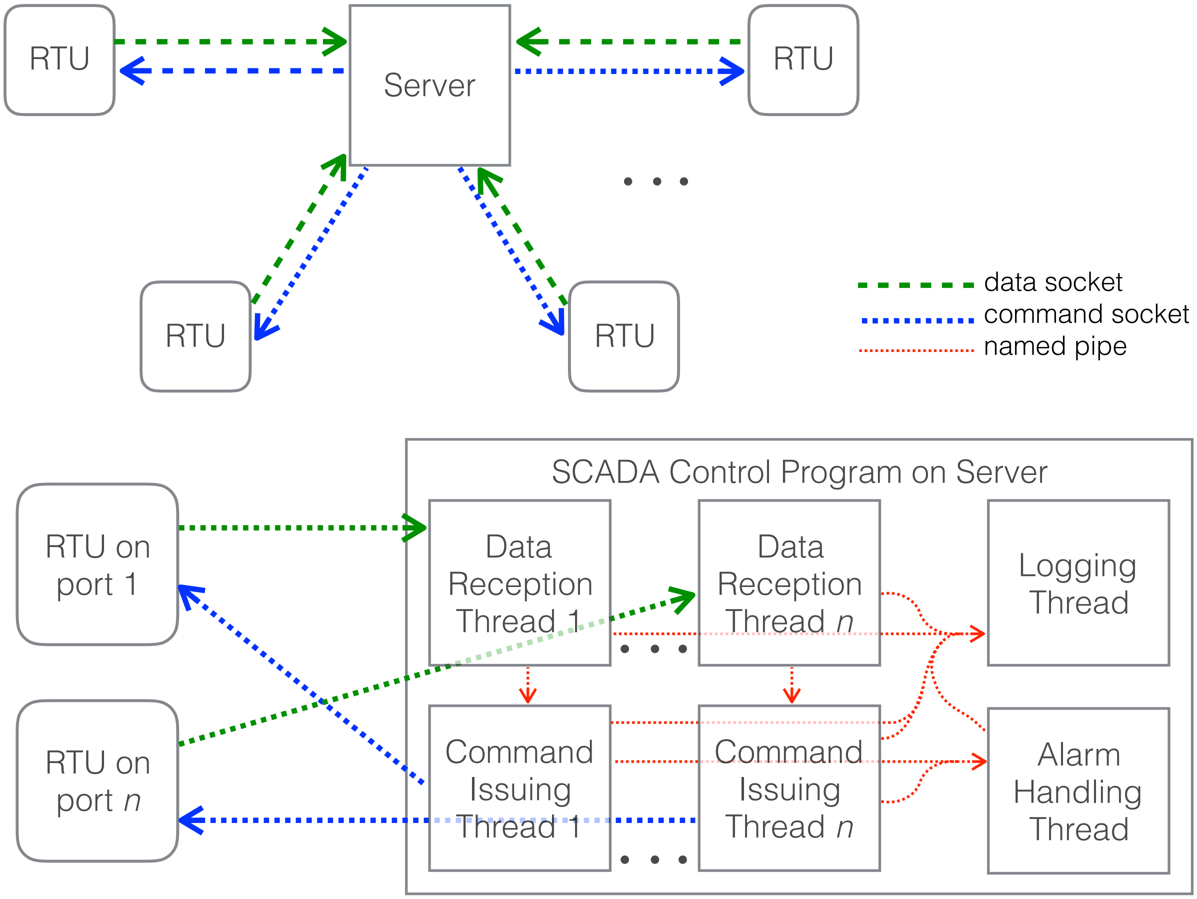
This SCADA system will be implemented using a typical client/server model. A single computer will be used to communicate with each RTU, collecting data and issuing commands, using Transmission Control Protocol (TCP) sockets. It is assumed that each RTU is on the same local area network (LAN). The server will run a program written in the C programming language and having a command line interface.

Predetermined ports and statically assigned IP addresses will allow for easily established network connections. Separate threads will be used for the reception of data from RTUs, sending commands to RTUs, logging data, and handling alarms. A separate thread will be used for each communication port in use so that the reception of data from a particular set of RTUs using the same port does not hold up the receiving of information from other RTUs using a different port. Likewise, separate threads will be used to send commands to the RTUs – one thread per port being used. A single thread will be used to maintain the log file and will receive the information it need from the data reception threads and from an alarm handling thread via named pipes.

The RTUs will write a given data point to the specified socket along with the time stamp pertaining to the data point so that the information can be logged together by the server.

Once a data point is received by a data reception thread, in addition to being passed to the logging thread via a named pipe, the data reception thread will also be pass the data to a corresponding command issuing thread – both using the same port and communicating with the same RTU(s). The command issuing thread will decide based on the data if a command needs to be issued to the RTU and if the data meets an alarm condition. If a command needs to be sent, the command issuing thread will write to the socket being ready by the RTU(s). If an alarm condition is met, the command issuing thread will write to a named pipe being read by the alarm handling thread. Commands will also be shared with the logging thread's named pipe. The alarm handling thread will take whatever action is necessary to "sound" an alarm. The purpose of this is simply to notify the operator than an alarm event is occurring so that he can act quickly. The command issuing thread is where the determination, that emergency action must be taken automatically by the system, happens and is where the alarm event is processed so that commands are issued to the RTUs as necessary.

Design elements discussed in class that will be used include the client/server model; socket communication; multithreading; and task communication, synchronization, and cooperation.



Assuming threads 1-*n* are used for communication between server and RTUs. Actual port numbers would be different.

Benefits Over Related Systems

Where other systems may lack multithreading, I treat it as an essential component. Without multithreading, significant problems could arise. For example, the reception of information from an RTU that should be triggering an alarm event may be held up by the reception of less important routine information.

Limitations

The proposed implementation assumes a constant and stable LAN connection. Additionally, execution of the various threads is limited by the number of processing cores available on the server. Ideally, each thread would have its own core for execution. Failing this, cores supporting hyper-threading would minimize any processing bottlenecks.

Implementation Methodology, Timeline, and Goals

Given a two week implementation period, this SCADA system would be built in stages. In the first week, the data reception thread functions would first be developed so that data can be received to be used for testing. The logging subsystem would then be implemented. The command issuing thread functions allowing simple command communication with the RTUs would then be implemented. In the second week, logic allowing for the processing of alarm events would be added to the command issuing thread functions. Finally, the alarm handling thread functions would be completed. Testing would occur throughout the development process, making use of RTUs along the way. The individual modules would first be tested on their own and then in cooperation with the other pieces of the SCADA system that they will interact with.

ECE 4220 Final Project
Report

Thomas Nabelek
December, 2015

1) Abstract

This project, incorporating the major topics covered in class, and functioning as a simulation of a SCADA system with two RTUs, was largely successful and functions as intended, though without the implementation of the ADC, voltage, current, and power portion.

2) Introduction

This project incorporated the topics covered in class in a way that we were made to integrate them successfully, requiring sufficient understanding of their functionality. The SCADA system simulation was used as a way to showcase implementation of these concepts.

3) Background

The primary model used in my implementation is the common client/server model with one device receiving data from all other devices and processing it as needed. The necessary processing done on each device and communication between devices and system tasks was done using TCP network communication, FIFO pipes, multithreading, a linked list, realtime processing, a kernel module, and semaphores. These methods were employed to ensure reliable execution and completion of tasks, and avoidance of errors.

4) Proposed method / System description / Implementation

My implementation was comprised of three primary elements:

- Controller/Historian user space program
- RTU user space program
- RTU kernel module

The basic structure of each of these elements was:

Controller:

```
main() {
   launch RTU1 thread - receive from RTU on port A
   launch RTU2 thread - receive from RTU on port B
   launch UI (user interface) thread - present user interface
}

receive from RTU(port number) {
   setup socket connection with supplied port number
   open log file
   while (1) {
      get info type
      if (info type == update) {
         clear structures
         read time and info string from socket into temporary
                structures
         copy to new structures

         if (verbose_mode is on)
             print information to screen
         wait on and then lock semaphore
         write information to file
         unlock semaphore
      }
      // Do basically the same thing, with some adjustments for (info
                type == event)
   }
   close log file
}

present user interface() {
   while(1) {
      get user input
      switch(user input) {
         case 1: If log file exists, print it out
         case 2: Delete log file
         case 3: Switch user mode on/off
      }
   }
}
```

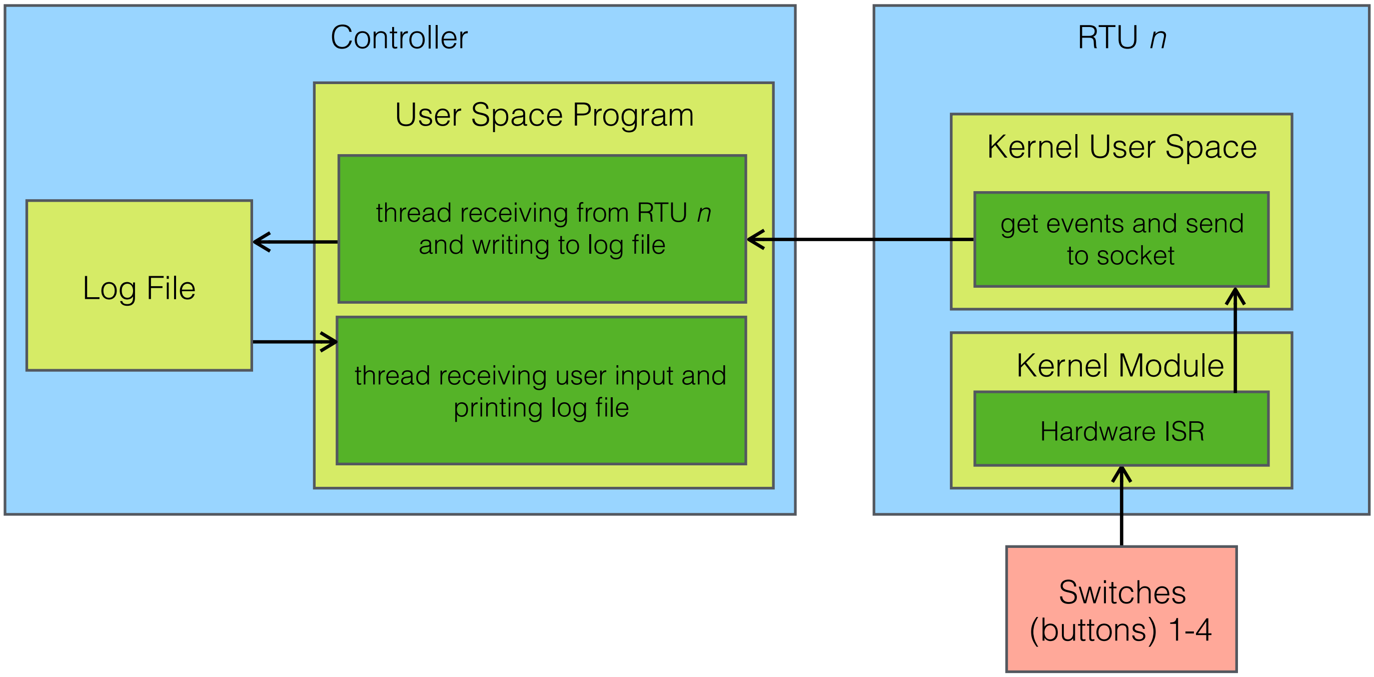
RTU - user space:

```
main() {
  setup socket connection with supplied port number
  launch thread to get threads from pipe - get events from pipe
  start periodic task
  initialize switch statuses to 1 and power to -10
  while (1) {
     while (there are events to send) {
           send info type through socket
           send event time through socket
           send event time microseconds through socket
           send info string through socket
           move to next event in queue
     }
     send info type through socket
     send update time through socket
     get update info and send info string through socket

     wait till end of realtime period
  }
}

get events from pipe() {
  open event time pipe
  open event switch pipe
  while (1) {
     read event time from kernel module event time pipe
     read event switch from kernel module event switch pipe
     read event switch status from kernel module event switch pipe

     put information into event queue linked list
     record switch information for next regular update
  }
}
```

RTU - kernel module:

```
hardware interrupt service routine() {
  disabled interrupt
  get current time
  get switch (button) register status
  switch (register value) {
     case switch 1–4:
         flip switch value
  }

  write event time to FIFO time pipe
  write switch designation to FIFO switch pipe
  write switch status to FIFO switch pipe

  enable interrupt
}

init_module() {
  get and configure ports for buttons (to act as switches)
  get and configure registers for hardware interrupts
  create FIFO time pipe and switch pipe
}

cleaup_module {
  disable interrupts
  close pipes
}
```

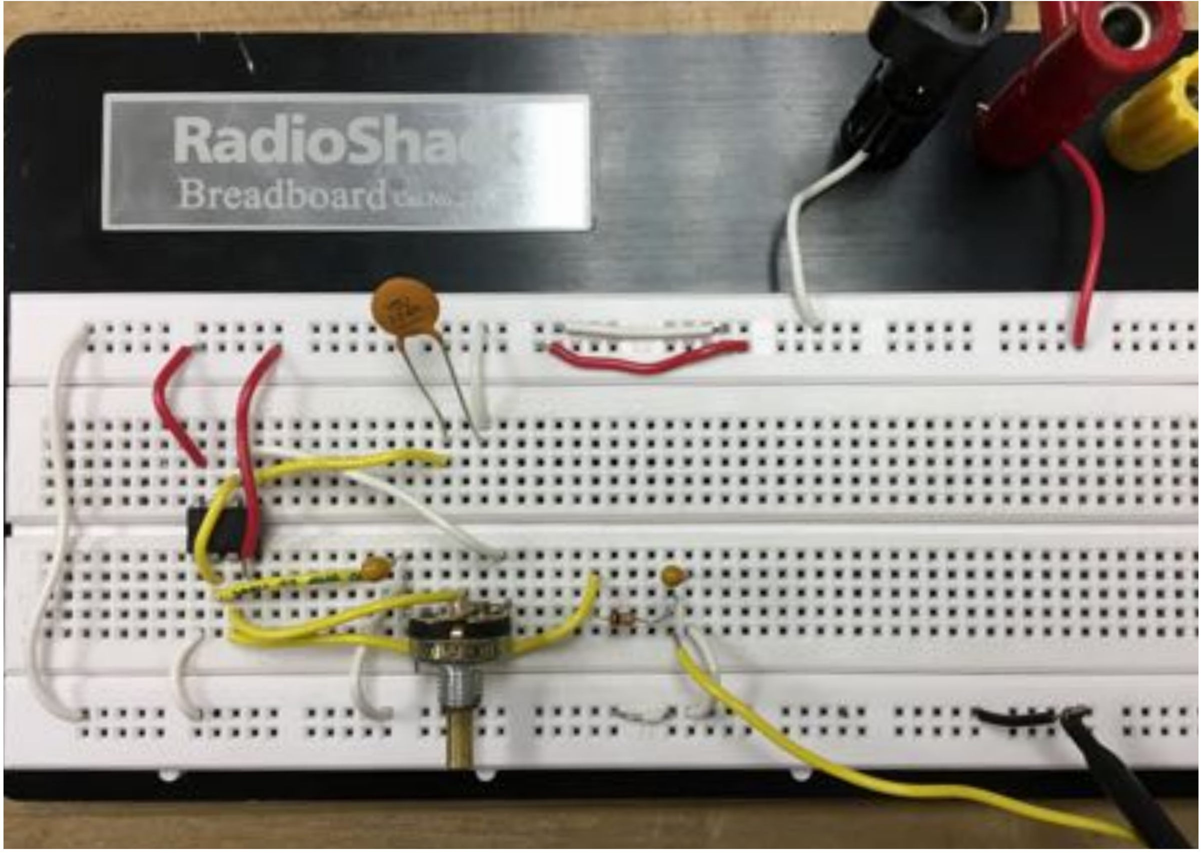
Basic communication implemented between processes and devices:

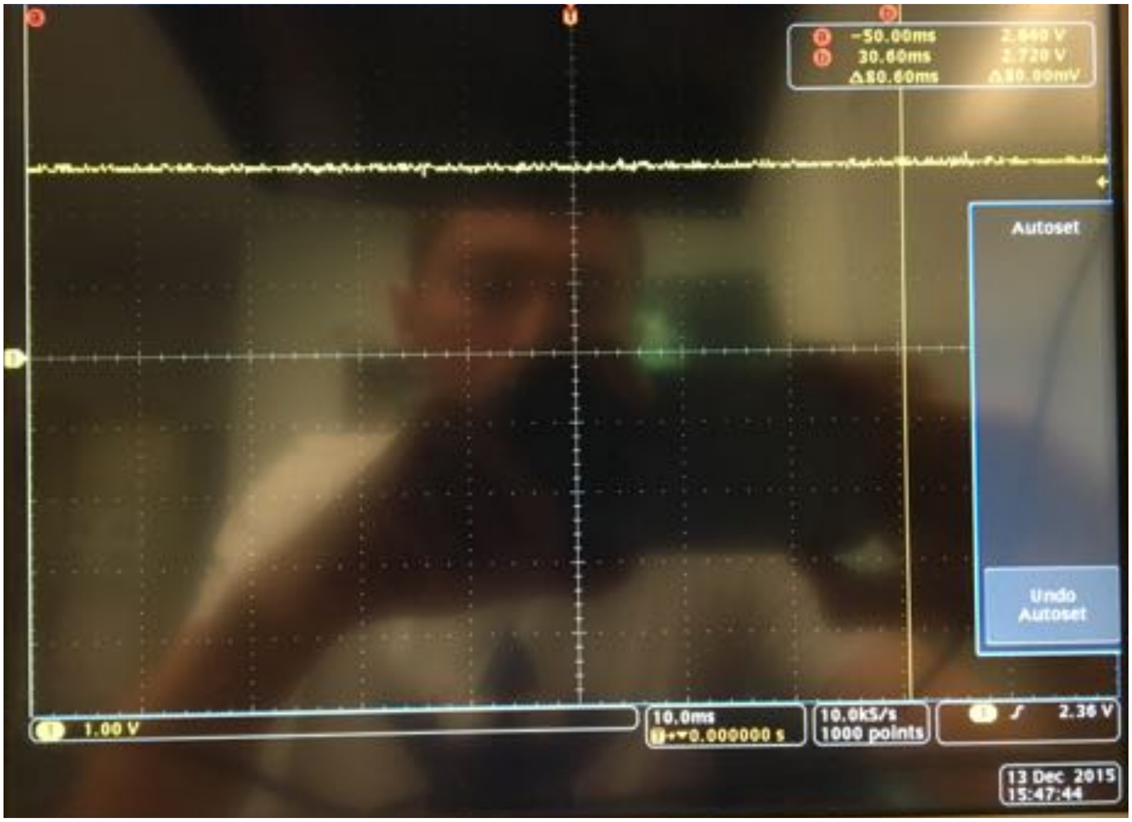

Controller
User Space Program
Log File
thread receiving from RTU *n* and writing to log file
thread receiving user input and printing log file
RTU *n*
Kernel User Space
get events and send to socket
Kernel Module
Hardware ISR
Switches (buttons) 1-4

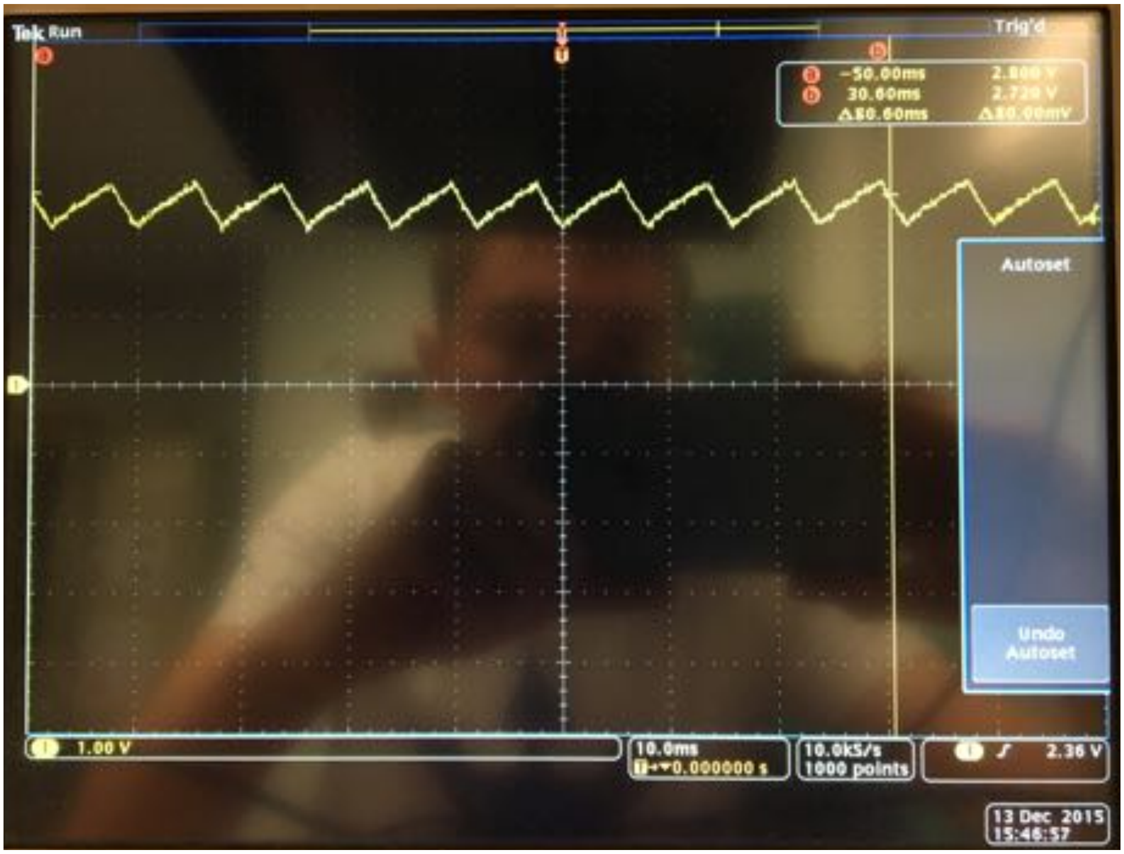
It is important to note here that each RTU operates on a separate TCP port and separate TCP connection to the Controller. Additionally, each RTU thread running on the Controller is able to write to the log file, but access to the file is protected with a semaphore so that only one process writes at a time.

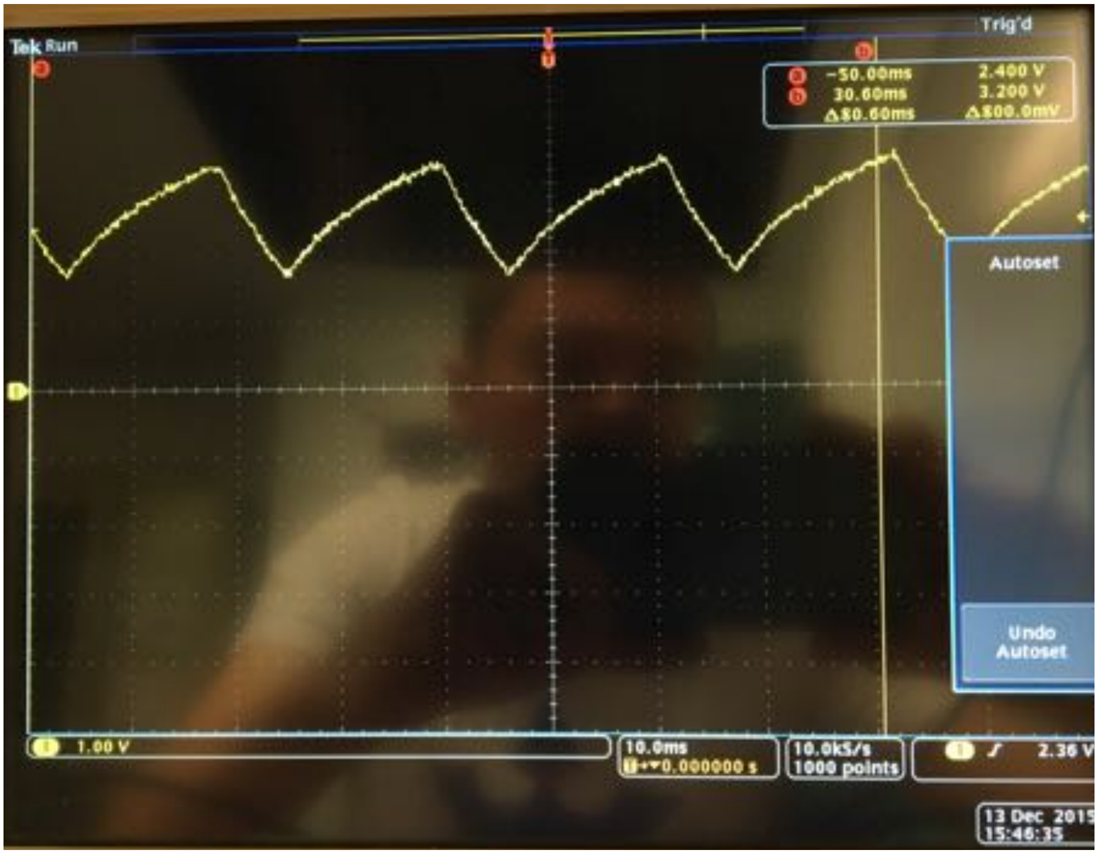
Unfortunately I was not able to complete the ADC, voltage, current, and power portion of the project, but if I had, I think my implementation would have worked in the following way. The kernel module would have contained a real time task checking check the appropriate ADC voltage and current registers every millisecond or less. If these values were ever outside of their limits, an event would be triggered, writing to a user space pipe the time and voltage/current information. I would probably set it so that no more than one overload event could be triggered every second, so that a single overload occurrence does not tigger many alerts. Current values would be checked with previous values to determine if there is power and a condition of no power would be reported in a similiar way.

The signal generating circuit for this portion was completed and functioned as intended:




a -50.00ms 2.800 V
b 30.60ms 2.720 V
Δ80.60ms Δ80.00mV
Autoset
Undo Autoset
1 1.00 V
10.0ms
T→▼0.000000 s
10.0kS/s
1000 points
1 ∫ 2.36 V
13 Dec 2015
15:47:44


Tek Run
Trig'd
a -50.00ms 2.800 V
b 30.60ms 2.720 V
Δ80.60ms Δ80.00mV
Autoset
Undo Autoset
1 1.00 V
10.0ms
T→▼0.000000 s
10.0kS/s
1000 points
1 ∫ 2.36 V
13 Dec 2015
15:46:57



5) Experiments and Results

The implementation of my design was very successful, and functions as I believe the project required, except for the ADC, voltage, current, and power portion.

I implemented my design first by ensuring good communication between processes and devices, and then moving to logging, and then the switch flips. I tested along the way, each piece at a time, and integrated the pieces together before testing the whole thing. The programs were run many times in order to slowly eliminate bugs. The system was tested with a single RTU, two RTUs, no button presses, a few slow button presses, and many rapid button presses. These cases all functioned as intended.

Please see the separately included video for demonstration.

6*) Discussion and Conclusion*

In the end, the implementation functioned as expected and the results made sense. There were a few issues that I ran into along the way:

The information received by the Controller from the socket, other than the time, was sent and received as a string because, while I would have preferred to pass the actual data values from the RTUs to the controller, when I sent the information structures that I had originally over the TCP socket, the values were received as 0s on the Controller end. I think the issue might have something to do with network/CPU byte order. I do not have much experience working with structures in sockets or pipes, so I can not be sure abut the problem here.

When using fopen() to open the log file, I had to use "ab+" rather than "a+" for the second argument. This sets the file to be written as a binary file rather than a standard text file. When setting it as a standard text file, no text was written to the file, though fprintf() reported that it was. Using a binary file instead fixed this and I was able to read from the binary file fine.

A remaining issue is that the sometimes when a switch is flipped for the first time, it is flipped from 1 to 1 when it should be flipped from 1 to 0. I was not able to resolve this with limited time. Additionally, I am not sure if I free all of the memory that is dynamically allocated as I did not look through the code thoroughly checking for that and did not analyze the program for memory leaks.

Not a problem, but a note: The information was read into temporary structures and then copied to newly allocated structures by the Controller because I originally planned to send the information to a separate thread to print to the log. This required a separately allocated structure so that they information would persist in memory.

I enjoyed this project quite a bit and wish that I had been able to spend more time completing it, or simply that I had started earlier. I think that this project represents the most integration I have done between devices and processes and the topics learned in class. I wish I understood the reasoning behind the problems above, as these are things that I have not encountered before. I believe that I was able to complete the project without burning any CPU cycles as every while loop has a blocking function where it waits or is a realtime periodic loop.

7) Appendices - Full Code (also included separately)

Controller - user space:

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

#define LOG_FILE "scada_log.txt" // Do not make longer than 45 characters

#define BUFFER_SIZE 30 // Adjust this

pthread_attr_t thread_attribute;
sem_t log_file_sem;
int verbose_mode = 1;

void *receive_from_RTU(char *);
void *user_interface();

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

  if (argc < 3) { // Check if port number was provided in command line
     printf("ERROR: ports not provided.\nUsage: %s <RTU 1 port> <RTU 2 port>\nExiting.
\n\n", argv[0]);
     return 1;
  }

  // ----- SETUP AND LAUNCH RTU THREADS AND UI THREAD ----- //
  pthread_t RTU1_thread, RTU2_thread, UI_thread;
  pthread_attr_t thread_attribute;
  pthread_attr_init(&thread_attribute);
  pthread_attr_setdetachstate(&thread_attribute, PTHREAD_CREATE_JOINABLE);

  sem_init(&log_file_sem, 0, 1); // Intialize semaphore for writing to log file

  pthread_create(&RTU1_thread, &thread_attribute, (void *)receive_from_RTU, (void *)
argv[1]); // Launch thread for RTU 1
  pthread_create(&RTU2_thread, &thread_attribute, (void *)receive_from_RTU, (void *)
argv[2]); // Launch thread for RTU 2
  pthread_create(&UI_thread, &thread_attribute, user_interface, NULL); // Launch
thread for user interface and log printing
  // -------------------------------- //

  // Cleanup
  pthread_join(RTU1_thread, NULL);
  pthread_join(RTU2_thread, NULL);
  pthread_join(UI_thread, NULL);
  sem_destroy(&log_file_sem); // Destroy semaphor
//  fclose(log_file);

  return 0;
}

// Communicate with RTUs and write to log file
```

```
void *receive_from_RTU(char *port_number) {

  // ----- SETUP TCP CONNECTION ----- //
  int socket_desc, new_socket_desc;
  socklen_t client_length;
  struct sockaddr_in server_addr, client_addr;

  socket_desc = socket(AF_INET, SOCK_STREAM, 0); // Creat socket
  if (socket_desc < 0) printf("Error opening socket.\n\n");

  // Initialize structures
  memset((void *) &server_addr, 0, sizeof(server_addr)); // Clear server address
structure
  server_addr.sin_family = AF_INET; // Symbol constant for Internet domain
  server_addr.sin_addr.s_addr = INADDR_ANY; // IP address of the machine on which the
server is running
  server_addr.sin_port = htons(atoi(port_number));  // Port number, converted to
network byte order, from command line

  if (bind(socket_desc, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
printf("Error on binding.\n\n");

  listen(socket_desc, 10); // Set socket to listen for connections
  client_length = sizeof(client_addr); // Get size of structure

  new_socket_desc = accept(socket_desc, (struct sockaddr *) &client_addr,
&client_length); // Accept new socket conenction
  if (new_socket_desc < 0) printf("Error on accept\n");
  // -------------------------------- //

  // ----- RECEIVE INFORMATION FROM SOCKET AND WRITE TO LOG FILE ----- //
  struct tm temp_time;
  float temp_time_microseconds;
  char temp_buffer[BUFFER_SIZE];
  FILE *log_file = fopen(LOG_FILE, "ab+"); // Open log file for updating (create if it
doesn't exist)

  while (1) {
    // Recieve info type from socket
    int info_type = -1;
    read(new_socket_desc, &info_type, sizeof(info_type));

    if (info_type == 0) { // If info type is a regular update
             // Clear and load temp structures from socket
             memset((void *) &temp_time, 0, sizeof(temp_time)); // Clear time
structure
             memset((void *) temp_buffer, 0, BUFFER_SIZE); // Clear buffer
             read(new_socket_desc, &temp_time, sizeof(temp_time)); // Fill structure
with update info
             read(new_socket_desc, temp_buffer, BUFFER_SIZE); // Fill structure with
update info

             // Allocate new info structures to pass to logging thread
             struct tm *update_time = (struct tm *) malloc(sizeof(struct tm));
             char *buffer = (char *) malloc(sizeof(char) * BUFFER_SIZE);
             memcpy((void *) update_time, &temp_time, sizeof(struct tm)); // Clear
time structure
             memcpy((void *) buffer, temp_buffer, BUFFER_SIZE); // Clear buffer

             if (verbose_mode)
               printf("UPDATE %4d %02d %02d %02d:%02d:%02d,%s\n", update_time->tm_year
+1900,update_time->tm_mon+1, update_time->tm_mday,
                      update_time->tm_hour, update_time->tm_min, update_time->tm_sec,
buffer);
```

```
             char *log_buffer = (char *) malloc(sizeof(char) * BUFFER_SIZE + 20); //
Add 20 characters for date and time
             sprintf(log_buffer, "UPDATE %4d %02d %02d %02d:%02d:%02d,%s",
update_time->tm_year+1900,update_time->tm_mon+1, update_time->tm_mday,
                      update_time->tm_hour, update_time->tm_min, update_time->tm_sec,
buffer);

             sem_wait(&log_file_sem);
             fprintf(log_file, "%s\n", log_buffer);
             fflush(log_file);
             sem_post(&log_file_sem);
    }
    else if (info_type == 1) { // If info type is an event
             // Clear and load temp structures from socket
             memset((void *) &temp_time, 0, sizeof(temp_time)); // Clear time
structure
             memset((void *) &temp_time_microseconds, 0,
sizeof(temp_time_microseconds)); // Clear microseconds time
             memset((void *) temp_buffer, 0, BUFFER_SIZE); // Clear buffer
             read(new_socket_desc, &temp_time, sizeof(temp_time)); // Fill structure
with update info
             read(new_socket_desc, &temp_time_microseconds,
sizeof(temp_time_microseconds)); // Get microseconds
             read(new_socket_desc, temp_buffer, BUFFER_SIZE); // Fill structure with
update info

             // Allocate new info structures to pass to logging thread
             struct tm *event_time = (struct tm *) malloc(sizeof(struct tm));
             float *event_microseconds = (float *) malloc(sizeof(float));
             char *buffer = (char *) malloc(sizeof(char) * BUFFER_SIZE);
             memcpy((void *) event_time, &temp_time, sizeof(struct tm)); // Copy time
from temporary structure
             memcpy((void *) event_microseconds, &temp_time_microseconds,
sizeof(float)); // Copy time from temporary structure
             memcpy((void *) buffer, temp_buffer, BUFFER_SIZE); // Copy buffer from
temporary buffer

             if (verbose_mode)
               printf("*EVENT* %4d %02d %02d %02d:%02d:%02d.%03.0f,%s\n", event_time-
>tm_year+1900, event_time->tm_mon+1, event_time->tm_mday,
                             event_time->tm_hour, event_time->tm_min, event_time-
>tm_sec, (float)*event_microseconds/1000, buffer);

               char *log_buffer = (char *) malloc(sizeof(char) * BUFFER_SIZE + 20); //
Add 20 characters for date and time
               sprintf(log_buffer, "*EVENT* %4d %02d %02d %02d:%02d:%02d.%03.0f,%s",
event_time->tm_year+1900, event_time->tm_mon+1, event_time->tm_mday,
                             event_time->tm_hour, event_time->tm_min, event_time-
>tm_sec, (float)*event_microseconds/1000, buffer);

               sem_wait(&log_file_sem);
               fprintf(log_file, "%s\n", log_buffer);
               fflush(log_file);
               sem_post(&log_file_sem);
    }
  }
  // ----------------------------------------------------------------- //

  fclose(log_file);
  pthread_exit(NULL);
}

// Present user interface and print log file
```

```
void *user_interface() {
  printf("\nWelcome to the SCADA console.\n");
  int user_option;
  while (1) {
    printf("\nOptions\n  1) Print log file\n  2) Delete log file\n  3) Switch verbose
mode on\\off\n\n");
    scanf("%d", &user_option);
    switch(user_option) {
      case 1: {// Print log file
        int character;
        FILE *log_file = fopen(LOG_FILE, "r");
        if (log_file) { // If log file was opened
          printf("The log file reads:\n");
          printf("%9s%9s%5s%16s%10s\n", "Date", "Time", "RTU","Switches 1-4",
"Power");
          while ((character = getc(log_file)) != EOF)
            putchar(character);
          fclose(log_file);
        }
        else // If log file was not opened
          printf("The log file was not found.\n");
        break;
         }
      case 2: // Delete log file
        if (access(LOG_FILE, F_OK) == 0) { // Check if log file exists
          char command_buffer[50];
          sprintf(command_buffer, "rm %s", LOG_FILE);
          system(command_buffer);
          printf("The log file has been deleted.\n");
        }
        else
          printf("The log file could not be found.\n");
        break;
      case 3: // Switch verbose mode on/off
       if (verbose_mode == 0) verbose_mode = 1; else verbose_mode = 0;
       break;
      default:
        printf("Invalid option.\n");
    }
  }

  pthread_exit(NULL);
}
```

RTU - user space

```
/*
 ============================================================================
 Name        : Project-RTU.c
 Author      : Thomas Nabelek (tntp9)
 Version     :
 Copyright   :
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <rtai.h>
#include <rtai_lxrt.h>
#include <arpa/inet.h>
#include <rtai_fifos.h>
#include <pthread.h>

#define BUFFER_SIZE 30 // Adjust this

int socket_desc;
struct sockaddr_in serv_addr;
struct hostent *server;

typedef struct switch_event {
	struct timeval event_time;
	int switch_flipped;
	int switch_status;
	struct switch_event *next_event;
} switch_event;

switch_event *event_queue_head = NULL;
switch_event *event_queue_tail = NULL;

int RTU_ID;
RTIME base_count;

time_t raw_update_time;
struct tm update_time;
time_t raw_event_time;
struct tm event_time;

int switch_status_1;
int switch_status_2;
int switch_status_3;
int switch_status_4;
float power;

char buffer[BUFFER_SIZE];

void error(const char *); // Print error messages
void *get_events_from_pipe();

int main(int argc, char** argv) {
  // ----- SETUP TCP CONNECTION ----- //
  if (argc < 4) { // Check if port number was provided in command line
     printf("\nERROR: no port and/or RTU_ID provided.\nUsage: %s <server IP> <port>
<RTU ID>\nExiting.\n\n", argv[0]);
     return 1;
  }

  socket_desc = socket(AF_INET, SOCK_STREAM, 0); // Create socket
  if (socket_desc < 0) error("ERROR opening socket");

  server = gethostbyname(argv[1]); // Set server address
  if (!server) printf("Error with server IP\n");

  // Initialize structures
  memset((char *) &serv_addr, 0, sizeof(serv_addr)); // Clear server address structure
  serv_addr.sin_family = AF_INET; // Symbol constant for Internet domain
  bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
// Copy IP address
  serv_addr.sin_port = htons(atoi(argv[2]));  // Port number, converted to network
byte order, from command line

  if (connect(socket_desc,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
error("ERROR connecting");  // Establish connection to the server
  // -------------------------------- //

  // ----- SETUP AND LAUNCH EVENT THREAD ----- //
  pthread_t get_events;
  pthread_attr_t thread_attribute;
  pthread_attr_init(&thread_attribute);
  pthread_attr_setdetachstate(&thread_attribute, PTHREAD_CREATE_JOINABLE);

  pthread_create(&get_events, &thread_attribute, get_events_from_pipe, NULL); //
Launch thread to get events from pipe

  base_count = start_rt_timer(nano2count(100000000)); // Use 0.1 seconds for base
period
  RT_TASK *send_update_task = rt_task_init(nam2num("TASK1"), 0, 512, 256); //
Initialize real time task
  rt_task_make_periodic(send_update_task, rt_get_time(), base_count*10); // Make task
period to every 1 second and begin

  int info_type_update = 0;
  int info_type_event = 1;

  // Initialize update values
  RTU_ID = atoi(argv[3]); // Get RTU ID from command line
  switch_status_1 = switch_status_2 = switch_status_3 = switch_status_4 = 1; //
Initialize switch statuses to 1
  power = -10;

  while (1) {
        // If there were events, send them before the update
        while (event_queue_head != NULL) {
                memset(buffer, 0, BUFFER_SIZE); // Clear buffer
                event_time = *localtime(&event_queue_head->event_time.tv_sec);
                float event_time_microseconds = event_queue_head->event_time.tv_usec;
                if (write(socket_desc, (void *)&info_type_event,
sizeof(info_type_event)) < 0) error("Error writing to socket."); // Write info type to
socket
                if (write(socket_desc, (void *)&event_time, sizeof(event_time)) < 0)
error("Error writing to socket."); // Write event time to socket
                if (write(socket_desc, (void *)&event_time_microseconds,
sizeof(event_time_microseconds)) < 0) error("Error writing to socket."); // Write
event time to socket
```

```
                sprintf(buffer, "  %d,  switch %d flipped to %d", RTU_ID,
event_queue_head->switch_flipped, event_queue_head->switch_status);
                if (write(socket_desc, (void *)buffer, BUFFER_SIZE) < 0) error("Error
writing to socket.");

                // Move to next event in linked list and free current event
                switch_event *event_queue_previous = event_queue_head;
                event_queue_head = event_queue_head->next_event;
                free(event_queue_previous);
         }
         event_queue_tail = NULL; // Reset linked list

         // Send update
         memset(buffer, 0, BUFFER_SIZE); // Clear buffer
         time(&raw_update_time);
         update_time = *localtime(&raw_update_time);
         if (write(socket_desc, (void *)&info_type_update, sizeof(info_type_update)) <
0) error("Error writing to socket."); // Write info type to socket
         if (write(socket_desc, (void *)&update_time, sizeof(update_time)) < 0)
error("Error writing to socket."); // Write update time to socket
      sprintf(buffer, "  %d,  %d,  %d,  %d,  %d,  %f\n", RTU_ID, switch_status_1,
switch_status_2, switch_status_3, switch_status_4, power);
         if (write(socket_desc, (void *)buffer, BUFFER_SIZE) < 0) error("Error writing
to socket.");

         rt_task_wait_period(); // Wait till end of period to send next update
  }

  return 0;
}

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

void *get_events_from_pipe() {
       int time_fifo = open("/dev/rtf/7", O_RDWR);
       int switch_fifo = open("/dev/rtf/8", O_RDWR);

       struct timeval event_time;
       int switch_flipped;
       int switch_status;

       while (1) {
              switch_flipped = -1;
              switch_status = -1;
              read(time_fifo, &event_time, sizeof(event_time));
              read(switch_fifo, &switch_flipped, sizeof(switch_flipped));
              read(switch_fifo, &switch_status, sizeof(switch_status));
              switch_event *new_switch_event = (switch_event *)
malloc(sizeof(switch_event));
              new_switch_event->event_time = event_time;
              new_switch_event->switch_flipped = switch_flipped;
              new_switch_event->switch_status = switch_status;

              // Build linked list of events
              new_switch_event->next_event = NULL;
              if (event_queue_head == NULL) {  // If first event in list
                     event_queue_head = new_switch_event;
                     event_queue_tail = new_switch_event;
              }
              else
                     event_queue_tail->next_event = new_switch_event;
                event_queue_tail = new_switch_event;

                // Record switch flip for regular update
                switch (switch_flipped) {
                        case 1: switch_status_1 = switch_status; break;
                        case 2: switch_status_2 = switch_status; break;
                        case 3: switch_status_3 = switch_status; break;
                        case 4: switch_status_4 = switch_status; break;
                }

                printf("Switch %d flipped to %d\n", new_switch_event->switch_flipped, 
new_switch_event->switch_status);
        }
        pthread_exit(NULL);
}
```

RTU - kernel module

```
#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_sem.h>
#include <rtai_fifos.h>
#include <linux/time.h> // do_gettimeofday()

MODULE_LICENSE("GPL");

unsigned long *PADR, *PBDR, *PBDDR;
unsigned long *GPIOBIntType1, *GPIOBIntType2, *RawIntStsB, *GPIOBEOI, *GPIOBIntEn,
*GPIOBDB, *VIC2IRQStatus, *VIC2_INTCLEAR, *VIC2_INTEN;
unsigned long *VIC2IRQStatus, *VIC2IntEnable, *VIC2SoftIntClear;

enum switch_status{OFF,ON};
enum switch_status switch1 = ON;
enum switch_status switch2 = ON;
enum switch_status switch3 = ON;
enum switch_status switch4 = ON;

static RT_TASK check_power_task;

RTIME count;

void ISR_HW(unsigned irq, void *cookie);

void check_power() {
  while (1) {
    rt_task_wait_period();
  }
}

void ISR_HW(unsigned irq, void *cookie) {
  rt_disable_irq(irq);

  struct timeval event_time;
  do_gettimeofday(&event_time); // Get current time

  int button_value = *RawIntStsB & 0xFFFFFF1F; // & with 0...00011111 to get value of
just first 5 bits

  int switch_to_send = 0;
  enum switch_status status_to_send = ON;

  switch (button_value) {
    case 0x01: // button 1
      if (switch1 == ON) switch1 = OFF; else switch1 = ON; // Record status change of
switch
//      printk("switch 1 flipped\n");
      switch_to_send = 1;
      status_to_send = switch1;
      break;
    case 0x02: // button 2
      if (switch2 == ON) switch2 = OFF; else switch2 = ON; // Record status change of
switch
//      printk("switch 2 flipped\n");
      switch_to_send = 2;
      status_to_send = switch2;
      break;
    case 0x04: // button 3
      if (switch3 == ON) switch3 = OFF; else switch3 = ON; // Record status change of
switch
//      printk("switch 3 flipped\n");
      switch_to_send = 3;
      status_to_send = switch3;
      break;
    case 0x08: // button 4
      if (switch4 == ON) switch4 = OFF; else switch4 = ON; // Record status change of
switch
//      printk("switch 4 flipped\n");
      switch_to_send = 4;
      status_to_send = switch4;
      break;
  }

  if (button_value != 0x10) {
        rtf_put(7, &event_time, sizeof(event_time)); // Send event time
        rtf_put(8, &switch_to_send, sizeof(switch_to_send)); // Send switch
designation number
        rtf_put(8, &status_to_send, sizeof(status_to_send)); // Send switch status
  }

  *GPIOBEOI  |= 0x1F; // Set register bits to XXX11111 to clear RawIntStsB
  rt_enable_irq(irq);
}

int init_module(void) {

  // Get port addresses
  PADR = (unsigned long *) __ioremap(0x80840000, 4096, 0);
  PBDR = PADR + 0x04/sizeof(unsigned long); // Go to 0x0804_0004, port B data register
  PBDDR = PADR + 0x14/sizeof(unsigned long); // Go to 0x0804_0014, port B data
direction register

  // Get hardware interrupt register addresses
  GPIOBEOI = PADR + 0xB4/sizeof(unsigned long); // Go to 0x0804_00B4, (+45)
  GPIOBIntType1 = PADR + 0xAC/sizeof(unsigned long); // Go to 0x0804_00AC, (+43)
  GPIOBIntType2 = PADR + 0xB0/sizeof(unsigned long); // Go to 0x0804_00B0, (+44)
  GPIOBIntEn = PADR + 0xB8/sizeof(unsigned long); // Go to 0x0804_00B8, (+46)
  GPIOBDB = PADR + 0xC4/sizeof(unsigned long);
  RawIntStsB = PADR + 0xC0/sizeof(unsigned long); // Go to 0x0804_00C0, (+48)

  // Configure the registers for buttons and speaker
  *PBDDR &= 0xFFFFFFE0; // Set register bits to XXX00000 (bits 0-4 to read)

  // Configure the registers for hardware interrupt
  *GPIOBIntType1  |= 0x1F; // Set register bits to XXX11111 (act on edge)
  *GPIOBIntType2  &= 0xFFFFFFE0; // Set register bits to X...XXX00000  (act on falling
edge)
  *GPIOBEOI  |= 0x1F; // Set register bits to XXX11111
  *GPIOBIntEn  |= 0x1F; // Set register bits to XXX11111, sets RawIntStsB to clear
  *GPIOBDB |= 0x1F; // Set register bits to XXX11111, set to enable debouncing
  rt_request_irq(59, ISR_HW, 0, 1);
  rt_enable_irq(59);
  rtf_create(7, sizeof(struct timeval)); // Setup pipe to write event time to
  rtf_create(8, sizeof(int)); // Setup pipe to write flipped switch  number and 
switch_status to

  // Initialize and begin real time tasks
  count = nano2count(1000000); // Set base count to 0.1 milliseconds
  rt_set_periodic_mode();
  rt_task_init(&check_power_task, check_power, 0, 256, 0, 0, 0);
  rt_task_make_periodic(&check_power_task, rt_get_time(), start_rt_timer(5 * 
count)); // Check power line every half millisecond

  printk("PROJECT MODULE INSTALLED\n");
  return 0;
}

void cleanup_module(void) {
  // Remove tasks, interrupt service routines
  rt_task_delete(&check_power_task);

  rt_disable_irq(59);
  rt_release_irq(59);

  rt_disable_irq(63);
  rt_release_irq(63);

  // Close pipes
  rtf_destroy(7);
  rtf_destroy(8);

  printk("PROJECT MODULE REMOVED\n");
}
```