

Real-Time Temperature Sensing Suite

ECE 4220: Real-Time Embedded Computing

Project Report

Dylan Samson

Objectives:

The objective of my project was to use a TMP36 temperature sensor in conjunction with a TS-7250 Single Board Computer and auxiliary board located in the Linux lab to create a program for temperature sensing and acquisition. This program allows a user to set up periodic temperature readings, log those readings to a file, as well as other functionality that may be necessary when measuring temperature in a laboratory setting.

The goal was to create this temperature sensing and acquisition system was to create a system that has a high degree of flexibility and accuracy when performing temperature measurements. Such a system, if successfully implemented, would be of great use to individuals such as chemists or in industrial settings where a process needs to be consistently and accurately be monitored. Additionally, such a system incorporates many of the topics covered during this semester such as real time tasks, interrupts, named pipes, and more.

To successfully implement this system, both a user space application and kernel module were created. The kernel module takes care of the interrupts used for push button functionality, while the user space application does the vast majority of the work. This user space application enables the user to setup the data collection session, manage the temperature collection session both via a user interface displayed to the terminal and the push buttons of the auxiliary board, as well as to output data to files or the terminal.

Implementation

The overall system is composed of a Linux PC, a TMP36 temperature sensor, and a TS-7250 Single Board Computer and auxiliary board. A block diagram of the system's hardware may be seen in Figure 1.

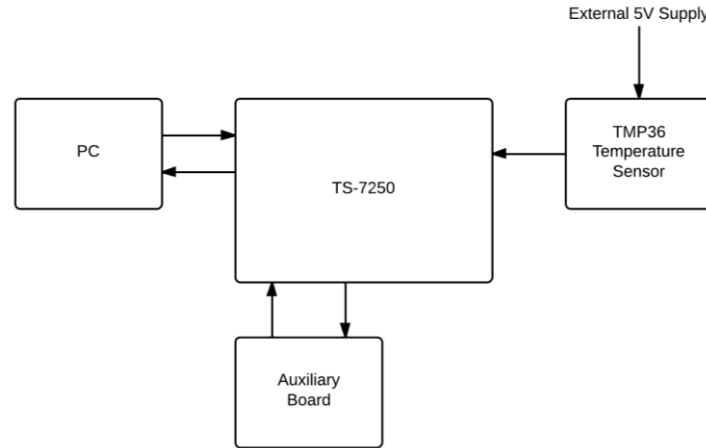


Figure 1: Hardware for Temperature Sensing Suite

The TS-7250 measures temperature readings from the TMP36 temperature sensor via the onboard MAX197 A/D Converter. This ADC has 8 channels and is 12-bit with a conversion time of 12 μ s. The TMP36 used is in a TO-92 package and has three pins. These pins are V_{DC} , Ground, and V_{OUT} . The V_{DC} pin is supplied 5V via an external supply. The ground pin is connected to the ground from the supply and to channel 0 of the ADC. The V_{OUT} pin, which has a voltage from 0 to V_{DC} that depends on the temperature measured, is connected to channel 0 of the ADC. These connections were made using a breadboard. Figure 2 depicts the hardware system.

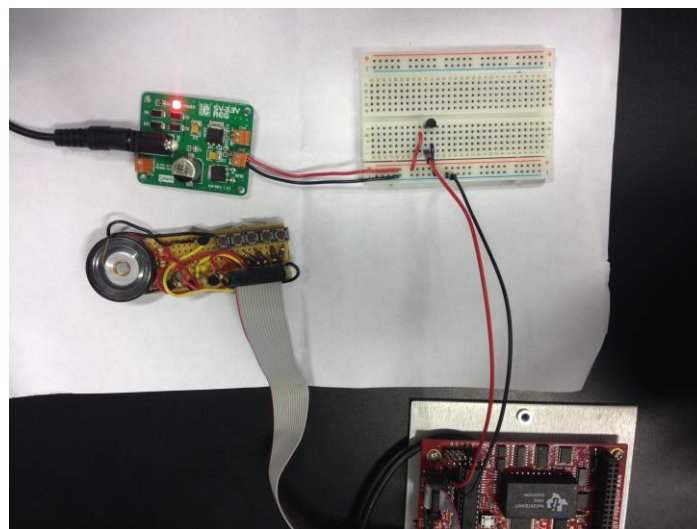


Figure 2: Hardware Setup for Temperature Sensing Suite

The auxiliary board is connected to the TS-7250 via the TS-7250's DIO1 Header. The auxiliary board has 5 push buttons, 3 LEDs (red, green, yellow) and a speaker. The auxiliary board's push buttons are used for user input and the LEDs are used to prompt the user of the button last pressed. The functionality of each button may be seen in Table 1. It is important to note that I had originally planned to use the auxiliary board's LEDs and speaker to notify the user of a particular temperature being reached. However, this feature was not successfully implemented.

B0	start / pause data collection
B1	end data collection
B2	take temperature measurement
B3	flag previous temperature measurement
B4	reset data collection

Table 1: Push Button (B0-B4) Functionality

The external supply is a 5V-3.3VReg Board from MikroElectronica. I already owned this from a previous project I had done. I decided to use this external supply because the TS-7250's main onboard 5V connector was in use and I decided it to be simpler to use this external supply than modifying the board to gain access. I found this to work very well for this system.

Both the user space application and kernel module were written in C using Eclipse on the Linux workstations in the lab. I will first begin by discussing the implementation of the kernel module. A flow chart for the kernel module may be seen in Figure 3. This kernel module is composed of `init_module`, `cleanup_module`, and `my_handler`. As previously mentioned, this kernel module is responsible for the port B interrupts which are used for the push button functionality.

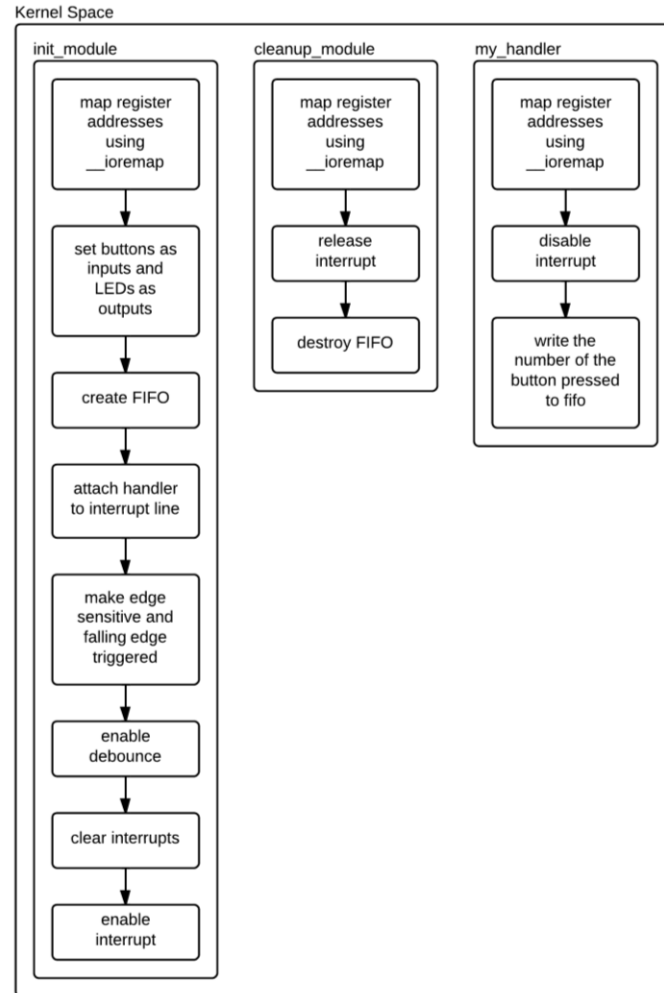


Figure 3: Kernel Module (ProjectModule.c) Flow Chart

Initially, implementing the push button functionality using polling was proposed. The decision to use interrupts was made because polling would require resources to constantly monitor the activity of the auxiliary board's push buttons while interrupts are generally more suited for such an application.

`init_module`, which executes on installation of the kernel module, begins with the mapping of the registers. This is done using `__ioremap` and then applying the respective offsets for each register. The port B data and data direction registers are used to control the LEDs. `GPIOIntType1` and `GPIOIntType2` are used to configure the interrupt. `GPIOIntEn` is used to

enable the interrupt and GPIOBDB is used to enable debounce. Lastly, GPIOBEOI is used to clear the interrupt. After these registers are mapped, a FIFO is created using `rtf_create()` and the handler is attached to the interrupt line using `rt_request_irq()`. The FIFO is used to send the button number to the user space program. Next, the registers are configured to enable the interrupt for the buttons, configure the interrupt to be falling edge sensitive, and enable debounce. Finally, `rt_enable_irq()` is used to enable the interrupt.

Once installed, `my_handler` executes each time the interrupt is triggered when a push button is pressed. This handler begins with the mapping of the port B data register, GPIOBEOI, and `RawIntStsB`. The port B register is used to light up certain LEDs depending upon the push button pressed, GPIOBEOI is used to clear the port B interrupt, and `RawIntStsB` is used to provide the status of the interrupt. After the mapping of these registers, the interrupt is disabled. This makes sure that it does not interrupt itself. Then, using an if structure, depending on the button pressed, LEDs are lit and the number of the button pressed is written to a FIFO to be read in the user space program. The interrupt finishes by clearing and enabling the interrupt.

`cleanup_module` executes whenever the kernel module is being removed. This begins by mapping the registers as was done in `init_module` which allows for proper removal of the module. `cleanup_module` completes by turning off the LEDs on the auxiliary board, releasing the interrupt, and destroying the FIFO.

The user space application (`Project.c`) main function begins by creating the named pipe and initializing the semaphores. These are used in the `getTemp` function. Then, the variables of the `Options` structure are initialized. This structure allows for the variables contained within to be used in multiple threads without any global declarations. The program then begins by asking the

user to specify a period. Then, this period is used to set the real time timer. Next, the dataCollect thread is created.

dataCollect is responsible for finishing setup and handling periodic data collection using a real time task. To finish setup, the user inputs a duration for the program to run, a specific time to take a measurement, and then asks the user what kind of output files they would like. The program does ask the user to specify a temperature to be notified of when reached. However, this functionality was not completed so doesn't really do anything. Threads are then created to handle the terminal interface, button interface, and file I/O. dataCollect then enters a real time task that has the specified period.

Each time a temperature reading is to be taken, getTemp is called. This function begins by mapping the registers for use of the MAX197 ADC and opens the named pipe for writing. To record the time of the measurement, gettimeofday() is used. To take a measurement using the ADC, 0x40 is written to the control register. 0x40 is written because this reflects channel 0 is to be used, is unipolar and with a 5 volt range. Once bit 7 of the busy register is 1, that means conversion is complete. The temperature reading is then written to the named_pipe to be used in the fileWrite thread. A semaphore is used to keep readings from being written to the named pipe at the same time or for race conditions to occur.

fileWrite begins with the opening of a .csv and/or .txt file based on the user selection during setup. Then, the named pipe is opened and read from. If this is the first temperature reading, the start time is set which allows the times to be converted from epoch time. The ADC reading is then converted to Celsius, Fahrenheit, and Kelvin. Finally, depending on the user selections, the files are written to and the temperature measurement is output to terminal.

`userInterface` manages the user interface once data collection has begun. Option 1 allows the user to start and pause data collection. Option 2 allows the user to abort data collection. Option 3 allows the user to flag the previous measurement. Option 4 allows the user to view the temperature measurements in the terminal until they press the keyboard. Lastly, Option 5 allows the user to change the time specified for a specific measurement to occur.

`buttonPress` manages the auxiliary board's push button functionality. It does so by opening and reading the FIFO for button presses and performing the actions as specified in Table 1.

`monitorTime` polls the time to check if the time specified for the specific measurement to occur has arrived. If so, a temperature measurement is taken.

`kbhit` is used to wait for a keyboard press when the temperature measurements are being displayed in the terminal. This function was found on an online forum and works great [1]. A flow chart of the user space application may be seen in Figure 4.

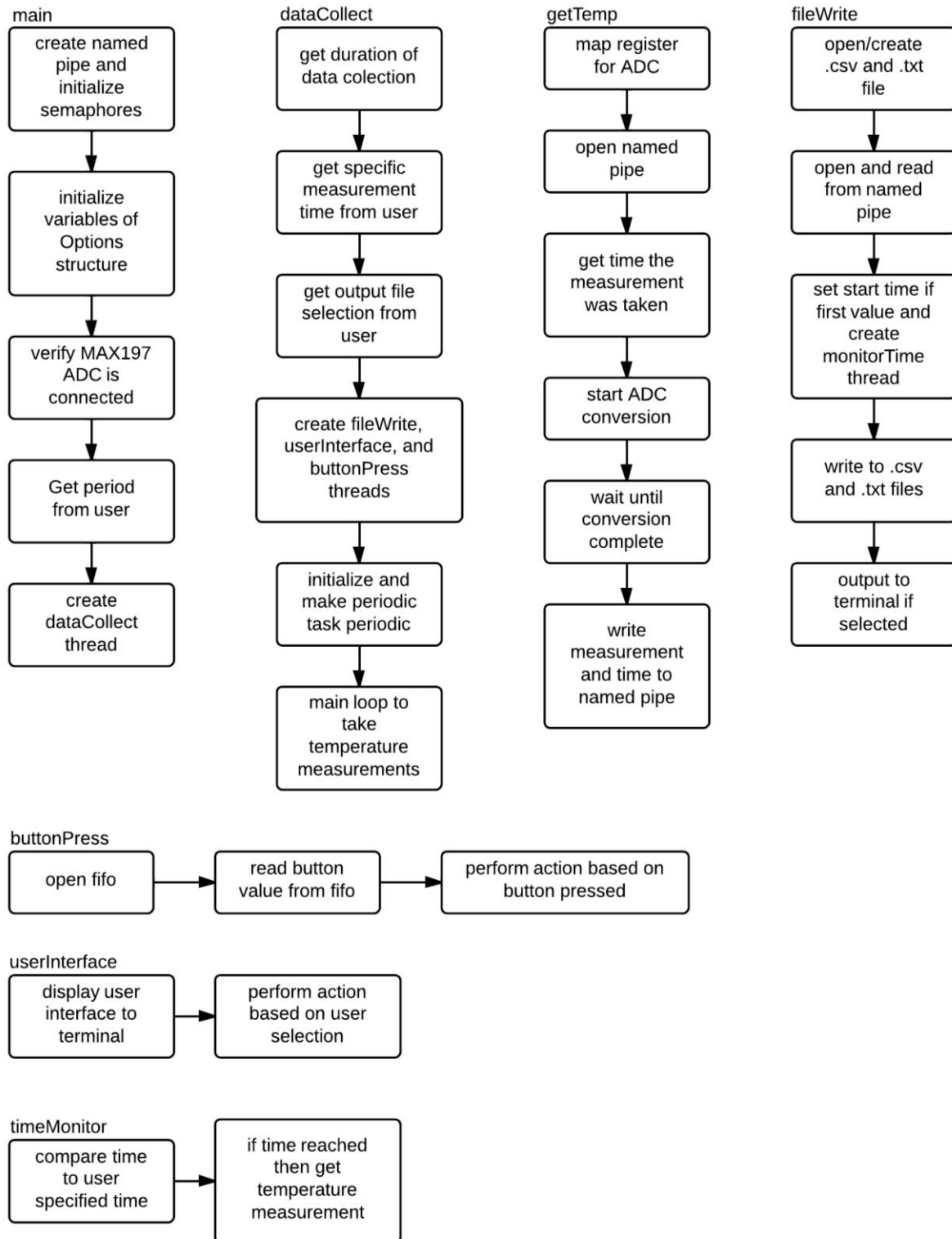


Figure 4: User Space Application (Project.c) Flow Chart

Experiments and Results

I did five different simulations to demonstrate my program during execution. Simulation 1 was just to show that the basic functionality of the system works. Period measurement were taken every 100ms and set to be taken for 10sec. Additionally, output to both a .csv and .txt file was selected. The system correctly collected temperature measurements every 100ms and ran for 10sec before stopping. The output files are also correct and show that the program performs file I/O as desired.

Simulation 2 was to demonstrate the system can take a measurement at a specific time specified by the user, as well as some of the features of the terminal interface. This simulation showed that the system was able to take the specific measurement. It also showed that the temperature measurements may be viewed using the terminal and that the temperature measurements may be paused and started again using the terminal. Lastly, it showed that measurements can be flagged using the terminal.

Simulation 3 demonstrated the ability to add a time to take a specific measurement after data collection has begun. It also shows the ability to abort data collection using the terminal. The system was able to do both of these features.

Simulation 4 was a demonstration which shows that the auxiliary board's push buttons all perform their desired actions when pressed. The collection session is first paused and unpaused. Then, data collection is reset. Next, a few measurements are taken by pressing button B2 and a few measurements are flagged using B3. Finally, data collection is aborted using push button B1. All these operations performed as desired. A video of this demonstration was recorded.

Simulation 5 was a demonstration that shows that the system is able to detect the varying temperature. This was verified by using a hair dryer to heat the temperature sensor and compare

those temperatures read by the system to those that were read on a DMM using a temperature probe. This demonstration showed that the system does indeed sense temperature change. This simulation was recorded.

Along with the submission of this report are the output files of these simulations, as well as videos and copies of the terminal output when applicable. Additionally, each simulation comes with an explanation that explains the simulation in further detail. Please see these for further information on these demonstrations and the results of testing.

Discussion

Overall, this system performs nearly all features as desired. This is with a few exceptions however. First, the ability for the user to specify a temperature to be notified of when reached was not implemented. This was due to me running out of time to complete this feature. Another issue is that the ability for the user to specify a time to take a specific measurement has a minor bug. Suppose that the period is 100ms and the user specifies the specific measurement to occur at 110ms. This specific time will be missed because the real time task will have control of the resources and not allow polling to take place and the time will be missed. However, it does work well otherwise, consistently catching the time and taking the measurement at this time.

Despite these difficulties, I was able to create a system that used many of the topics discussed in the course over the semester. These topics include real time tasks, kernel modules, named pipes, FIFOs, threads and interrupts. All of which are very important concepts in real time computing. Additionally, this system foundationally performs very well as the simulations show. There is room for improvement such as adding the ability to have more than one time specified for a specific measurement, as well as implementing the feature to specify a notification

temperature; however, I think the system performs really well overall. This system would need much more flexibility added in order to be a viable commercial option.

Works Cited

- [1] “kbhit() for Linux.” C Board RSS. Web. 01 May 2014. <<http://cboard.cprogramming.com/c-programming/62166-kbhit-linux.html>>.

```

/*
=====
Name      : Project.c
Author    : Dylan Samson
Version   :
Copyright :
Description : Real Time Data Collection Project
=====
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <pthread.h>
#include <semaphore.h>
#include <rtai_fifos.h>
#include <string.h>

RTIME BaseP;

//structure used to store temperature measurement and time of measurement
typedef struct TempPoint{
    double value_a;
    struct timeval time_a;
}TempPoint;

//structure used to share variables between threads
typedef struct Options{
    int FileChoice;
    int PrintChoice;
    int StopChoice;
    int PeriodicChoice;
    int PauseChoice;
    int Flag;
    int Reset;
    struct timeval StartTime;
    int TimeSet;
    double timeToMeasure;
    double tempToNotify;
    int tempNotifyUnit;
}Options;

sem_t sem1;
sem_t sem2;

void dataCollect(void *ptr);
void getTemp();
void fileWrite(void *ptr);
void userInterface(void *ptr);
void buttonPress(void *ptr);
void monitorTime(void *ptr);
int kbhit(void);

int main(void) {

```

```

Options Opts; //main structure declaration
pthread_t thread0; //thread that will be used for data collection
volatile unsigned char *option; //bit 0 of this register will be used to check if MAX197 installed
unsigned char *optionValue;
int fd = open("/dev/mem", O_RDWR|O_SYNC); //file descriptor for mapping
double PeriodTemp = 0; //used for setting the real time period
int PeriodCh; //stores Y/N for period choice

system("mkfifo named_pipe >& /dev/null"); //create named pipe
system("clear");
sem_init(&sem1, 0, 1); //initialize semaphores
sem_init(&sem2, 0, 0);

//initialize variables of main structure
Opts.StopChoice = 0;
Opts.PrintChoice = 0;
Opts.PeriodicChoice = 0;
Opts.PauseChoice = 0;
Opts.TimeSet = 0;
Opts.tempToNotify = 0;

//map address of option register
option = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x22400000);

//check bit 0 to see if MAX197 ADC is installed
optionValue = (unsigned char *)option;
if(*optionValue & 0x01){
    printf("MAX197 Connected\n");
}
else{
    printf("MAX197 Not Connected\n");
    pthread_exit(0);
}

//check if user would like to start periodic data collection
printf("\nWould you like to take periodic measurements (Y/N)? ");
PeriodCh = getchar();

if(PeriodCh == 'Y'){
    printf("Enter period length in milliseconds? "); //get period value from user
    scanf("%d", &Opts.PeriodicChoice);
    PeriodTemp = Opts.PeriodicChoice;
    PeriodTemp = PeriodTemp * 1000000; //convert to nanoseconds

    BaseP = start_rt_timer(nano2count(PeriodTemp)); //set base period for real time task

    //create thread for data collection
    pthread_create(&thread0, NULL, (void*)&dataCollect, (void *)&Opts);

    pthread_join(thread0, NULL);
}
else{
    printf("\n");
}

usleep(5000);
return 0;
}

```

```

//finishes setup and manages data collection
void dataCollect(void *ptr){
    Options *Opts = (Options *)ptr;
    int iterations;

    //get how long the user would like to run program
    printf("\nHow long in seconds would you like to take measurements? ");
    scanf("%d", &iterations);
    //calculate number of temperature measurements to take
    iterations = ((iterations * 1000) / Opts->PeriodicChoice) + 1;

    //get a time user would like to take a specific measurement
    printf("\nWould you like to take a measurement at a specific time?\n"
           "Specify time in milliseconds (0 = No): ");
    scanf("%lf", &Opts->timeToMeasure);

    //get a temp user would like to be notified of when it is reached
    printf("\nWould you like to be notified at a specific temp?\n"
           "Specify temp in centigrade (0 = No): ");
    scanf("%lf", &Opts->tempToNotify);

    //ask if user would like to save data to files
    Opts->FileChoice = getchar();
    printf("\nWould you like to save these measurements to a file (Y/N)? ");
    Opts->FileChoice = getchar();

    //ask user what kind of files they would like and set choice
    if(Opts->FileChoice == 'Y'){
        printf("What kind of file would you like?\n");
        printf("1 - comma-separated values (.csv)\n");
        printf("2 - general text output (.txt)\n");
        printf("3 - both a .csv and .txt file\n");
        printf("4 - cancel saving of measurments to file\n");
        scanf("%d", &Opts->FileChoice);
    }

    //create thread for writing to output files and printing to terminal
    pthread_t thread1;
    pthread_create(&thread1, NULL, (void *)&fileWrite, (void *)Opts);

    system("clear");

    //start thread for terminal interface and button interface
    pthread_t thread3, thread4;
    pthread_create(&thread3, NULL, (void *)&userInterface, (void *)Opts);
    pthread_create(&thread4, NULL, (void *)&buttonPress, (void *)Opts);

    //initialize real time task and make periodic with specified period
    RT_TASK* rtwrite1 = rt_task_init(nam2num("thrd0"), 0, 512, 256);
    rt_task_make_periodic(rtwrite1, rt_get_time() + nano2count(500000000), BaseP);

    int i;
    gettimeofday(&Opts->StartTime, NULL);
    //data collection loop
    for(i = 0; i < iterations; i++){
        if(Opts->StopChoice == 1){
            pthread_exit(0); //if stop has been selected, abort data collection
        }
    }
}

```



```

        if(Opts->PauseChoice != 1){
            getTemp(); //get temperature measurement
        }
        else{
            if(Opts->PrintChoice == 1){
                printf("\nPaused...\n"); //shows pause if terminal output enabled and paused
            }
        }

        if(Opts->Reset == 1){
            i = 0; //if reset as has be selected, reset data collection
        }
        rt_task_wait_period(); //wait until next period
    }

    //alert user that data collection completed
    printf("\nData Collection Complete!\n");
    usleep(5000);
    pthread_exit(0);
}

//gets a temperature reading from channel 0 of the MAX 197 ADC on the TS-7250
void getTemp(){
    volatile unsigned char *option, *controlValue, *busy; //registers for ADC
    TempPoint *TempMeasure; //structure for temperature measurement
    int fd = open("/dev/mem", O_RDWR|O_SYNC); //file descriptor for mapping
    int named_pipe;
    unsigned short *result;
    struct timeval time;
    unsigned char *busyValue;
    unsigned char *optionValue;

    TempMeasure = (TempPoint*)malloc(sizeof(TempPoint)); //allocate memory

    //map addresses of registers
    option = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x22400000);
    controlValue = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x10F00000);
    busy = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x10800000);

    //open named pipe for writing
    if((named_pipe = open("named_pipe", O_WRONLY)) < 0){
        printf("error while creating pipe\n");
        pthread_exit(0);
    }

    //get time of temperature measurement
    gettimeofday(&TempMeasure->time_a, NULL);

    optionValue = (unsigned char *)option;

    //get sample from ADC Channel 0, Unipolar, 5V Range
    *controlValue = 0x40;

    busyValue = (unsigned char *)busy;

    //wait until conversion complete
    while(*busyValue & 0x80){ //bit 7 is 1 when complete
        busyValue = (unsigned char *)busy;
        printf("");
    }

```

```

    }
    result = (unsigned short *)controlValue;

    //store result in structure as a double
    TempMeasure->value_a = (double)*result;

    //write to pipe to get temperature structure to fileWrite thread
    sem_wait(&sem1);
    if(write(named_pipe, TempMeasure, sizeof(*TempMeasure)) != sizeof(*TempMeasure)){
        printf("error while writing to pipe\n");
        pthread_exit(0);
    }
    sem_post(&sem2);
}

//writes to .csv and .txt (if selected) and displays to terminal (if selected)
void fileWrite(void *ptr){
    Options *Opts = (Options *)ptr;
    TempPoint *TempMeasure;
    TempMeasure = (TempPoint*)malloc(sizeof(TempPoint));
    int named_pipe;
    double tempCel, tempFehr, tempKelv;
    FILE *fp1, *fp2;
    double timeMS;

    //if .csv option selected, open/create .csv file for writing
    if(Opts->FileChoice == 1 || Opts->FileChoice == 3){
        fp1 = fopen("data.csv", "w+");
        fprintf(fp1, "Time, Celsius, Fehrenheit, Kelvin, Flag");
    }

    //if .txt option selected, open/create .txt file for writing
    if(Opts->FileChoice == 2 || Opts->FileChoice == 3){
        fp2 = fopen("data.txt", "w+");
        fprintf(fp2, "Temperature Data\n");
    }

    //open named pipe for reading
    if((named_pipe = open("named_pipe", O_RDONLY)) < 0){
        printf("error while creating pipe\n");
        pthread_exit(0);
    }

    while(1){
        sem_wait(&sem2);
        //if reset selected, then reopen output files erasing pevious contents
        if(Opts->Reset == 1){
            if(Opts->FileChoice == 1 || Opts->FileChoice == 3){
                fclose(fp1);
                fp1 = fopen("data.csv", "w+");
                fprintf(fp1, "Time, Celsius, Fehrenheit, Kelvin, Flag");
            }

            if(Opts->FileChoice == 2 || Opts->FileChoice == 3){
                fclose(fp2);
                fp2 = fopen("data.txt", "w+");
                fprintf(fp2, "Temperature Data\n");
            }

```

```

        Opts->TimeSet = 0; //reset start time
        Opts->Reset = 0; //deselect reset
    }

    //read from named pipe
    if((read(named_pipe, TempMeasure, sizeof(*TempMeasure))) < 0){
        printf("error while reading from pipe\n");
        pthread_exit(0);
    }

    //set start time if timeset is 0 meaning start time has not been set
    if(Opts->TimeSet == 0){
        Opts->StartTime = TempMeasure->time_a;
        if(Opts->timeToMeasure != 0){
            //start monitoring for time specified by user
            pthread_t thread2;
            pthread_create(&thread2, NULL, (void *)&monitorTime, (void *)Opts);
        }
        Opts->TimeSet = 1; //set timeset to show start time has been set
    }

    //convert time to milliseconds
    timeMS = ((TempMeasure->time_a.tv_sec * 1000.0) - (Opts->StartTime.tv_sec * 1000.0));
    timeMS += ((TempMeasure->time_a.tv_usec / 1000.0) - (Opts->StartTime.tv_usec / 1000.0));

    //convert ADC result to Celsius, Fahrenheit, and Kelvin
    tempCel = TempMeasure->value_a;
    tempCel = tempCel * 5000 / 4096;
    tempCel = (tempCel - 500) / 10;
    tempKelv = tempCel + 273.15;
    tempFehr = tempCel * 1.8 + 32;

    //if .csv is selected, perform writing of temperature measurement to data.csv
    if(Opts->FileChoice == 1 || Opts->FileChoice == 3){
        if(Opts->Flag == 1){ //if flag selected, flag measurement
            fprintf(fp1, ", Flag! Comment...");
        }
        fprintf(fp1, "\n%.3f, %.1f, %.1f, %.1f", timeMS, tempCel, tempFehr, tempKelv);
    }

    //if .txt is selected, perform writing of temperature measurement to data.txt
    if(Opts->FileChoice == 2 || Opts->FileChoice == 3){
        if(Opts->Flag == 1){ //if flag selected, flag measurement
            fprintf(fp2, "Flag! Comment...\n");
        }
        fprintf(fp2, "\nTime: %.3f\n", timeMS);
        fprintf(fp2, "Temp (Celsius): %.1f\n", tempCel);
        fprintf(fp2, "Temp (Fahrenheit): %.1f\n", tempFehr);
        fprintf(fp2, "Temp (Kelvin): %.1f\n", tempKelv);
    }

    //if print to terminal enabled, display temperature measurement to terminal
    if(Opts->PrintChoice == 1){
        printf("\nTime: %.3f\n", timeMS);
        printf("Temp (Celsius): %.1f\n", tempCel);
        printf("Temp (Fahrenheit): %.1f\n", tempFehr);
        printf("Temp (Kelvin): %.1f\n", tempKelv);
    }
}

```

```

        //deselect flag
        Opts->Flag = 0;
        sem_post(&sem1);
    }
    usleep(5000);
    free(TempMeasure);
    pthread_exit(0);
}

//displays user interface to terminal and performs actions
void userInterface(void *ptr){
    Options *Opts = (Options *)ptr;
    int menuChoice;
    printf("Collection Started...\n"); //tell user data collection has started
    while(1){ //display options to user during data collection
        printf("\n1 - Pause/Start Collection\n");
        printf("\n2 - Stop Data Collection\n");
        printf("\n3 - Flag Previous Measurement\n");
        printf("\n4 - View Data Collection\n");
        printf("\n5 - Change Specified Notification Time\n");
        scanf("%d", &menuChoice);

        //if option 1 selected, start/pause data collection
        if(menuChoice == 1){
            Opts->PauseChoice = Opts->PauseChoice ^ (1 << 0);
            if(Opts->PauseChoice == 0){
                printf("\nCollection Started...\n");
            }
            else if(Opts->PauseChoice == 1){
                printf("\nCollection Paused...\n");
            }
        }
        //if option 2 selected, abort data collection
        else if(menuChoice == 2){
            Opts->StopChoice = 1;
            printf("\nData Collection Aborted!\n");
            pthread_exit(0);
        }
        //if option 3 selected, flag previous measurement
        else if(menuChoice == 3){
            Opts->Flag = 1;
            printf("\nPrevious Measurement Flagged!\n");
        }
        //if option 4 selected, print temperature measurements to terminal
        else if(menuChoice == 4){
            Opts->PrintChoice = 1;
            while(!kbhit()){ //wait for keyboard press and then end viewing
                printf("");
            }
            menuChoice = 0;
            Opts->PrintChoice = 0;
        }
        //if option 5 selected, prompt user for new time to be specified
        else if(menuChoice == 5){
            printf("\nWould you like to take a measurement at a specific time?\n"
                "Specify time in milliseconds (0 = No): ");
            scanf("%lf", &Opts->timeToMeasure);
        }
    }
}

```

```

}

//performs actions of button presses
void buttonPress(void *ptr){
    Options *Opts = (Options *)ptr;
    int fifo, button;

    //open fifo to get button press from kernel module
    fifo = open("/dev/rtf/1", O_RDWR);

    while(1){
        read(fifo, &button, sizeof(int)); //read from fifo
        switch(button){
            case 1: //if B0 pressed, start/pause data collection
                Opts->PauseChoice = Opts->PauseChoice ^ (1 << 0);
                break;
            case 2: //if B1 pressed, abort data collection
                Opts->StopChoice = 1;
                printf("\nData Collection Aborted!\n");
                usleep(100000000);
                break;
            case 3: //if B2 pressed, get temperature measurement
                getTemp();
                break;
            case 4: //if B3 pressed, flag previous measurement
                printf("\nPrevious Measurement Flagged!\n");
                Opts->Flag = 1;
                break;
            case 5: //if B4 pressed, reset data collection
                Opts->Reset = 1;
                break;
        }
    }
}

void monitorTime(void *ptr){
    Options *Opts = (Options *)ptr;
    struct timeval time;
    double timeDiff;

    while(1){ //continually get time of day and compare to user specified time
        gettimeofday(&time, NULL);
        timeDiff = ((time.tv_sec * 1000.0) - (Opts->StartTime.tv_sec * 1000.0));
        timeDiff += ((time.tv_usec / 1000.0) - (Opts->StartTime.tv_usec / 1000.0));
        timeDiff = abs(Opts->timeToMeasure - timeDiff);

        if(timeDiff < 1){ //take temperature measurement if near time specified
            getTemp();
        }
    }
}

//function used to monitor for a keyboard press
//this is a function that is widely available on the internet for this purpose
//I did not create this function myself, I got it from the web address below
//http://cboard.cprogramming.com/c-programming/63166-kbhit-linux.html
int kbhit(void)
{
    struct timeval tv;

```

```
    fd_set rdfs;

    tv.tv_sec = 0;
    tv.tv_usec = 0;

    FD_ZERO(&rdfs);
    FD_SET(STDIN_FILENO, &rdfs);

    select(STDIN_FILENO+1, &rdfs, NULL, NULL, &tv);
    return FD_ISSET(STDIN_FILENO, &rdfs);
}
```

```

/*
=====
Name      : ProjectKernel.c
Author    :
Version   :
Copyright :
Description : Detects button presses and sends the number of the button
               to the user space program.
=====
*/

#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

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

MODULE_LICENSE("GPL");

unsigned long *ptr1;

static void my_handler(unsigned irq_num, void *cookie){
    unsigned long *bEOI, *bRawSts, *pbdr;
    static int button[5] = { 1, 2, 3, 4, 5 };

    pbdr = (unsigned long *)((char *)ptr1 + 0x04); //Port B Data Register
    bEOI = (unsigned long *)((char *)ptr1 + 0xB4); //GPIOBEOI
    bRawSts = (unsigned long *)((char *)ptr1 + 0xC0); //RawIntStsB

    //disable interrupt
    rt_disable_irq(59);

    //if B0 pressed, writes 1 to fifo
    if((*bRawSts & (1 << 0)) == 0x01){
        printk("button 1 interrupt\n");
        *pbdr = *pbdr & 0xFFFFFFF1F;
        *pbdr = *pbdr | 0x80;

        rtf_put(1, &button[0], sizeof(int));
    }
    //if B1 pressed, writes 2 to fifo
    else if((*bRawSts & (1 << 1)) == 0x02){
        printk("button 2 interrupt\n");
        *pbdr = *pbdr & 0xFFFFFFF1F;
        *pbdr = *pbdr | 0x40;

        rtf_put(1, &button[1], sizeof(int));
    }
    //if B2 pressed, writes 3 to fifo
    else if((*bRawSts & (1 << 2)) == 0x04){

```

```

        printk("button 3 interrupt\n");
        *pbdr = *pbdr & 0xFFFFF1F;
        *pbdr = *pbdr | 0xC0;

        rtf_put(1, &button[2], sizeof(int));
    }
    //if B3 pressed, writes 4 to fifo
    else if((*bRawSts & (1 << 3)) == 0x08){
        printk("button 4 interrupt\n");
        *pbdr = *pbdr & 0xFFFFF1F;
        *pbdr = *pbdr | 0x20;

        rtf_put(1, &button[3], sizeof(int));
    }
    //if B4 pressed, writes 5 to fifo
    else if((*bRawSts & (1 << 4)) == 0x10){
        printk("button 5 interrupt\n");
        *pbdr = *pbdr & 0xFFFFF1F;
        *pbdr = *pbdr | 0xA0;

        rtf_put(1, &button[4], sizeof(int));
    }

    *bEOI = *bEOI | 0x1F; //clear interrupt

    rt_enable_irq(59); //enable interrupt
}

int init_module(void){
    unsigned long *pbdr, *pbddr;
    unsigned long *bEnable, *bType1, *bType2, *bDebounce, *bEOI;

    ptr1 = (unsigned long *)__ioremap(0x80840000, 4096, 0);

    pbdr = (unsigned long *)((char *)ptr1 + 0x04); //Port B Data
    pbddr = (unsigned long *)((char *)ptr1 + 0x14); //Port B Data Direction
    bType1 = (unsigned long *)((char *)ptr1 + 0xAC); //GPIOBIntType1
    bType2 = (unsigned long *)((char *)ptr1 + 0xB0); //GPIOBIntType2
    bEnable = (unsigned long *)((char *)ptr1 + 0xB8); //GPIOBIntEn
    bDebounce = (unsigned long *)((char *)ptr1 + 0xC4); //GPIOBDB
    bEOI = (unsigned long *)((char *)ptr1 + 0xB4); //GPIOBEOI

    *pbddr = *pbddr | 0xE0; //set buttons as inputs, LEDs as outputs
    *pbdr = *pbdr & 0xFFFFF1F; //turn off lights

    rtf_create(1, sizeof(int)); //create fifo

    rt_request_irq(59, my_handler, 0, 1); //attach handler to interrupt line

    *bEnable = *bEnable & 0xFFFFFE0;
    *bType1 = *bType1 | 0x1F; //edge sensitive
    *bType2 = *bType2 & 0xFFFFFE0; //falling edge
    *bDebounce = *bDebounce | 0x1F; //enable debounce
    *bEOI = *bEOI | 0x1F; //clear interrupts
    *bEnable = *bEnable | 0x1F;

    rt_enable_irq(59); //enable interrupt
}

```



```

        return 0;
    }

void cleanup_module(void){
    unsigned long *pbdr, *pbddr;
    unsigned long *bEnable, *bType1, *bType2, *bDebounce, *bEOI;

    pbdr = (unsigned long *)((char *)ptr1 + 0x04);          //Port B Data
    pbddr = (unsigned long *)((char *)ptr1 + 0x14);          //Port B Data Direction
    bType1 = (unsigned long *)((char *)ptr1 + 0xAC);         //GPIOBIntType1
    bType2 = (unsigned long *)((char *)ptr1 + 0xB0);         //GPIOBIntType2
    bEnable = (unsigned long *)((char *)ptr1 + 0xB8);        //GPIOBIntEn
    bDebounce = (unsigned long *)((char *)ptr1 + 0xC4);      //GPIOBDB
    bEOI = (unsigned long *)((char *)ptr1 + 0xB4);           //GPIOBEOI

    *pbddr = *pbddr | 0xE0; //set buttons as inputs, LEDs as outputs
    *pbdr = 0xFFFFFFFFF; //turn off lights

    rt_release_irq(59);
    rtf_destroy(1);
}

```