**Protecting Pet Safety using Temperature-Sensitive Collars**

ECE 4220 Final Project Report

May 15, 2015

Stephanie Cahail

**Abstract**

When it’s summer or winter, it can be easy for outdoor pets to get hyperthermia or hypothermia from extreme temperatures. This project provides a solution for this issue by designing and implementing pet collars that measure body heat of pets and relay the temperature status of the pets to the pet owner’s central device. There was a slight lag in temperature readings during experiments, but besides that the project accomplished what it needed to.

**Introduction**

The problem stated in the abstract was chosen for which to find a solution because pets suffer from extreme weather conditions more often than they should due to owners just not being aware that their pets may be too cold or too hot. Although owners can tell if their pet is overheated or too cold based on how the animal is acting, it can be hard for owners to monitor the temperature of an animal because they’re not outside all the time or they’re not at their home. A solution for owners to be more aware of the condition of their pet(s) in extreme weather temperatures is to make a collar that can measure the temperature of the pet wearing it. The temperature status of the pet(s) wearing the collar, ranging from low, regular, and high, would be sent to a user interface on a central device for the pet owner to view. Additionally, when the temperature of a pet is too low or too high, a sound would be emitted from the central device to alert the owner. To make this communication between the collars and the central device possible, UDP sockets were used. In order for the sound to emit when needed, a fifo was used between the central device’s user space and kernel module.

The first goal for the project solution is to have a thermistor successfully measure temperatures. The second goal is to be able to have two microcontrollers communicate with each other. An additional goal is to successfully trigger alerts on a microcontroller that display various warning message pertaining to a pet’s temperature. The final goal is to be able to use multiple microcontrollers with thermistor circuits to represent multiple pet collars being used. Although the thermistor readings aren’t perfect in terms of speed, the first goal as a whole was mostly achieved. The second and third goals have both been achieved. The last goal was not achieved due to not having enough wiring to create a second thermistor circuit.

**Background**

When considering a solution for this problem, it was pretty trivial that measuring the pet’s body temperature was necessary. How to go about obtaining temperature measurements, however, was not so trivial. The main inspiration for recording the temperature came from Biotrack’s method of sensing temperature differences in animals to determine hibernation patterns and signs of death. They track body temperature of animals by placing a thermistor under the animal’s tracking tag in a fashion so that the thermistor is touching the animal’s skin or fur [1]. Although this implementation is used to solve a problem that is different from the problem being addressed in this project, it is a method that could very easily be applied to the desired solution to the project’s problem.

Another factor needed for the solution is converting the output voltage of the thermistor to a temperature. An approach that was used to do so was built upon from some articles on Adafruit, the Arduino website, and GarageLab. The Adafruit article article explained how to set up a circuit to measure a thermistor with an Arduino microcontroller [2]. The Arduino website and the GarageLab website both provided methods for converting the output voltage of the thermistor to temperature using the Steinhart-Hart thermistor equation [3, 4]. The circuit design from the Adafruit project was applied to the project, and a combination of both methods of applying the Steinhart-Hart thermistor equation provided by the Arduino and GarageLab websites were implemented in the project as well. The fact that this project uses a TS-7250 board instead of an Arduino was taken into consideration when applying methods from these sources.

**System Design and Implementation**

This project consists of two TS-7250 boards. One board serves as the pet collar/server board, and the other board serves as the central device/client board. The first step of completing this project was obtaining temperature readings from the thermistor. To do so, a 10KOhm NTC thermistor was placed in a voltage divider with two 20KOhm thermistors put in parallel to give off a 10KOhm resistance. The circuit diagram is in Figure 1.

TS-7250 Microcontroller

10kOhm

Thermistor

10kOhm Resistor

Figure 1: Circuit Diagram for Thermistor

The server board powers the circuit and reads in the voltage of the thermistor. Then an analog to digital conversion is applied to that voltage so it can be used in the server program to convert to temperature in Fahrenheit. As mentioned earlier, the Steinhart-Hart thermistor equation is used, which is:

Temperature (in Kelvin) = 1/(A+B(ln[R]) + C(ln[R])\*3),

where A = 0.001129148, B = 0.000234125, and C = 0.0000000876741 [3].

R is the resistance value, which can be found by this equation:

10000.0/(1024.0/RawADC – 1)

Since an NTC thermistor is being used, the temperature and resistance of the thermistor is inversely proportional, so this equation takes that into account [4]. After getting the temperature, it needs to be converted to Fahrenheit, which can be done using the following equations:

temp = temp -273.15

temp = (temp \* 9.0)/5.0 + 32.0

The first equation converts the temperature from Kelvin to Celsius, and the second equation converts the temperature from Celsius to Fahrenheit [3].

The next step of project completion is initializing socket communication between the client board and the server board. A UDP socket was used. Once the socket is initialized in the client and server boards, the client initializes the communication by first accepting an input message and then sending that message to the server. Once that communication is established, the server sends a message back to the client based on the temperature readings. If the temperature is too low (below 100.5 degrees Fahrenheit) then a message stating the pet’s temperature is too low is sent to the client board. If the temperature is too high (above 102.5 degrees Fahrenheit) then a message stating the pet’s temperature is too high is sent to the client board. Otherwise, a message saying the pet’s temperature is normal is sent to the client board. The reason for choosing this specific temperature range is because this is the normal temperature range for pets [5]. Once the client board receives a temperature status message it displays the message. This communication keeps occurring as long as an initial communication message continues to be sent from the client board to the server board.

The final step in the project is using the kernel module to emit a sound when a temperature too low or too high warning message is sent to the client board. To do this, a fifo is established between the client board user space and the client board kernel module. When a message is sent to the client board from the server board, the words “low” and “high” are searched for in the message. If those words are found then a flag is set to one and sent to the kernel module via the fifo. After that occurs the program delays for three seconds and the flag is set back to zero and sent to the kernel module via the fifo. This allows the sound to play for three seconds and turn back off. From the kernel module, a real time task continuously receives flag values from the fifo. If the input flag value is one then a sound emits. Otherwise, the sound does not occur. The flowchart for the socket communication and fifo communication can be found in Figure 2.

= Thermistor Measurement = Socket Communication

TS-7250 Server Board

User Space

TS-7250 Client Board User Space

Thermistor Circuit

TS-7250 Client Board Kernel Module

= Fifo Communication

Figure 2: Process Communication Flowchart

**Experiments and Results**

Some experiments performed to ensure the program works involve using different methods of heating up and cooling down the thermistor. By doing so the variety of temperatures the thermistor can read is determined, the rate at which the thermistor can heat up and cool down is discovered, and the temperature status messages can be tested. When the server program is run the digital voltage value the board reads as well as the converted voltage value is printed to ensure that the temperature reading is accurate. When the client board initializes communication with the server board the server board prints the message received from the client board to ensure communication was successful. An example of output messages from the server board is shown in Figure 3.

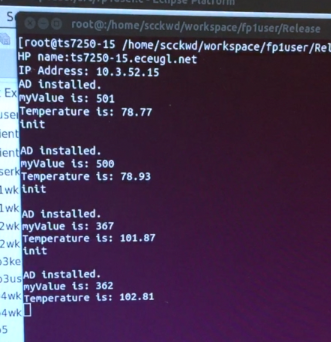


Figure 3: A Screenshot of Temperature Readings

In order to ensure that the server board is successfully sending temperature status messages to the client board the message sent is printed. Additionally, if the message is a warning message, a flag needs to be set to one and sent to the kernel module via a fifo so the kernel module will allow the board to emit a sound. To test that the flag is being set, a “Reached.” statement is printed when the condition for the flag to be set is true. A screenshot of some output messages from the client board can be seen in Figure 4.

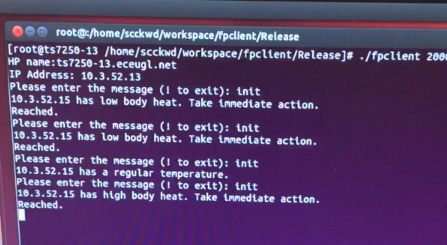


Figure 4: A Screenshot of Temperature Status Messages

Some methods used when applying different levels of heat to the thermistor involved using my hand to warm up the thermistor, using a hair dryer to warm up the thermistor, and touching something cool with my hand and then using that hand to cool down the thermistor. While using a hair dryer was a good way to keep the thermistor warm, it threw off the temperature readings because the breadboard would stay warm even if no heat or cool heat were applied to the thermistor. Using my hand after holding something cold to cool down the thermistor was a really good way to cool down the thermistor. The best method found for testing the temperature ranges was using room temperature to test the hypothermia case and using my hand to warm up the thermistor to test the normal case and the hyperthermia case. By doing so the thermistor was able to warm up and cool down rather quickly, which created an optimal testing environment.

**Discussion and Conclusions**

Overall the project was pretty successful. One major flaw of the project is that the temperature readings have some slight lag, so that threw off the warning messages a little bit. For the most part, however, the temperature status messages displayed when they were expected to. Additionally, the temperature readings were fairly accurate. The readings seemed to be about five to ten degrees higher than they should have been, but they seemed consistent. The alert sounds behaved as expected. The sound turned on for three seconds and then turned off when the temperature was too low or too high. When the temperature was within range there was no sound emission.

Some issues that came up during project development were issues with consistent socket communication and issues playing the sound. Initially the only board that was going to send messages through the socket was the server board. However, when trying to do so the messages would get lost and nothing would display. I discovered that when using UDP socket communication the client board has to initiate communication with a server board before the server board can send anything back. Therefore, the implementation now has the client board initializing communication. Another problem that occurred is that the boards would communicate for a little bit and then stop communicating. In order to keep communication consistent I decided to stick with the sample UDP client broadcast program format and enter a message to be sent to the server board every time, with the option to enter “!” to exit the program. This fixed the problem. The final problem was with playing the sound. Initially the plan was to press a button on the client board and use two fifos and the socket to have the server board emit a sound to simulate a pet finder feature. Due to issues with implementing the interrupt and using the socket to communicate when the button was pressed this feature could not be implemented. I wanted to implement the kernel module in the project somehow, so I decided to have the client board emit a sound notification when a pet temperature status is low or high. Having the sound constantly play would be annoying, so I tried to use a three second delay in the real time task in charge of reading from the fifo and playing the sound based on the flag value. Nothing would play on the board doing this. I tried using the method of sending the set flag to the kernel module, putting the three-second delay in the client user space program and sending the zero value flag to the kernel module and having a condition statement to either play the sound or not in the kernel module. This method proved successful.

I learned a lot from this project. I learned how to write code for a client board. In the labs there was a client program provided, so writing my own client program was definitely a great learning experience. I also learned how to convert analog voltage to digital voltage.

One limitation of the project is that a user running the client program has to enter a message to send to the server each time. One improvement I would love to make is implementing a way to continuously initialize communication from the client to the server without any user input, all while keeping the socket communication consistent. Another improvement that would be great is actually being able to implement the pet finder feature that was discussed earlier.

This project proved to be challenging, but implementing the project was a great learning experience it was overall quite successful.

**Appendices**

**Works-Cited**

[1] Biotrack, 'Temperature Sensing'. [Online]. Available: http://www.biotrack.co.uk/pdf/sensetemp.pdf.

[2] Learn.adafruit.com, 'Using a Thermistor | Thermistor | Adafruit Learning System'. [Online]. Available: https://learn.adafruit.com/thermistor/using-a-thermistor.

[3] Playground.arduino.cc, 'Arduino Playground - Thermistor2', 2015. [Online]. Available: http://playground.arduino.cc/ComponentLib/Thermistor2.

[4] Garagelab.com, 'Tutorial: Using NTC Thermistors With Arduino', 2015. [Online]. Available: http://garagelab.com/profiles/blogs/tutorial-using-ntc-thermistors-with-arduino.

[5] Pet Health Network, 'Hypothermia and Your Pet', 2012. [Online]. Available: http://www.pethealthnetwork.com/dog-health/dog-diseases-conditions-a-z/hypothermia-and-your-pet.

**Code**

/\*

============================================================================

Name : fpclient.c

Author : Stephanie Cahail

Version :

Copyright : Your copyright notice

Description : User space for final project that represents central unit/client

============================================================================

\*/

#include <stdio.h>

#include <stdlib.h>

#include <rtai.h>

#include <rtai\_lxrt.h>

#include <pthread.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <time.h>

#include <sys/mman.h>

#include <fcntl.h> //for O\_RDWR

#include <rtai\_fifos.h>

#include <semaphore.h>

#include <pthread.h>

#define MSG\_SIZE 70 // message size

int flag = 0; // end on flag = 1

int sock, length, n;

int sockExit = 0; // holds exit request

void error(const char \*msg)

{

perror(msg);

exit(0);

}

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

int permission = 1; // broadcast permission value

socklen\_t fromlen;

struct hostent \*hp;

struct sockaddr\_in from;

struct sockaddr\_in addr;

char hostname[64], IP[16];

char\* warningMessage;

char buffer[MSG\_SIZE]; // to store received messages or messages to be sent.

sock = socket(AF\_INET, SOCK\_DGRAM, 0); // Creates socket. Connectionless.

if (sock < 0)

error("socket");

// change socket permissions to allow broadcast

if (setsockopt(sock, SOL\_SOCKET, SO\_BROADCAST, &permission, sizeof(permission)) < 0)

{

printf("error setting socket options\n");

exit(-1);

}

length = sizeof(addr); // length of structure

//bzero((char\*)&addr,length); // sets all values to zero. memset() could be used

addr.sin\_family = AF\_INET; // symbol constant for Internet domain

addr.sin\_port = htons(2000);

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255"); // IP address of the machine on which

//the server is running

gethostname(hostname, 64); //get IP address

hp = gethostbyname(hostname);

//bcopy((char\*)hp->h\_addr, (char\*)&addr.sin\_addr, hp->h\_length);

strcpy(IP, inet\_ntoa(\*((struct in\_addr \*)hp->h\_addr)));

printf("HP name:%s\n", hp->h\_name);

printf("IP Address: %s\n", IP);

fromlen = sizeof(struct sockaddr\_in); // size of structure

//fifo initialization

int fifo1 = open("/dev/rtf/0", O\_RDWR);

if(fifo1 == -1){

printf("error.\n");

exit(-1);

}

do{

// bzero: to "clean up" the buffer. The messages aren't always the same length...

bzero(buffer,MSG\_SIZE); // sets all values to zero.

printf("Please enter the message (! to exit): ");

fgets(buffer, MSG\_SIZE-1, stdin);

if(buffer[0] != '!'){

//sprintf(buffer, "Initializing communication with collars.");

n = sendto(sock, buffer, strlen(buffer), 0, (const struct sockaddr \*)&addr, length);

if (n < 0)

error("sendto");

// receive temperature status message (general format: "1 has certain temperature status")

bzero(buffer, MSG\_SIZE);

n = recvfrom(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, &fromlen);

if (n < 0)

error("recvfrom");

printf("%s \n", buffer); //print warning message concerning pet that has temperature issue

if(strstr(buffer, "low") || strstr(buffer, "high")){

printf("Reached.\n");

flag = 1;

//write set flag value to kernel via fifo so sound can play

if(write(fifo1, &flag, sizeof(flag)) != sizeof(flag)){

printf("Fifo pipe write error\n");

exit(-1);

}

usleep(3000000); //sleep for 3 seconds

//write zero flag value to kernel via fifo so sound can stop

flag = 0;

if(write(fifo1, &flag, sizeof(flag)) != sizeof(flag)){

printf("Fifo pipe write error\n");

exit(-1);

}

}

}

}while(buffer[0] != '!');

//close socket

close(sock);

return 0;

}

/\*

============================================================================

Name : fpclientkernel.c

Author : Stephanie Cahail

Version :

Copyright : Your copyright notice

Description : The kernel module to be used for the client board.

============================================================================

\*/

#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\_lxrt.h>

#include <rtai\_sched.h>

#include <rtai\_fifos.h>

#include <linux/time.h>

MODULE\_LICENSE("GPL");

static RT\_TASK myTask;

RTIME period, delay;

int flag = 0;

unsigned long \*ptrA, \*PFDR, \*PFDDR;

static void rt\_process(int t){

while(1){

//read flag from pipe

rtf\_get(0, (int\*)&flag, sizeof(int));

if (flag == 1){

\*PFDR ^= 0x02; //speaker plays

}

else{

\*PFDR &= ~0x02; //disables speaker

}

rt\_task\_wait\_period();//waits for other thread to complete its task

}

}

int init\_module(void)

{

printk("MODULE INSTALLED.\n");

//starting pointer at port A

ptrA = (unsigned long \*) \_\_ioremap(0x80840000, 4096, 0);

//port F and port F direction register

PFDR = (unsigned long \*) ((char\*)ptrA + 0x30);

PFDDR = (unsigned long\*) ((char\*)ptrA + 0x34);

//upper nibble set for port F direction register, since only output will be used

\*PFDDR |= 0x02;

rt\_set\_periodic\_mode();

period = start\_rt\_timer(nano2count(1000000)); //start timer

delay = period \* 1000;//delay 1 second

rt\_task\_init(&myTask, rt\_process, 0, 256, 0, 0, 0);

rt\_task\_make\_periodic(&myTask, rt\_get\_time(), period);

//create fifo 1

rtf\_create(0, sizeof(double));

return 0;

}

void cleanup\_module(void)

{

rt\_task\_delete(&myTask);

stop\_rt\_timer();//stop timer

rtf\_destroy(0);//delete fifo 0

printk("MODULE REMOVED \n");

}

/\*

============================================================================

Name : fp1user.c

Author : Stephanie Cahail

Version :

Copyright : Your copyright notice

Description : User space of final project that resembles pet collar entity

============================================================================

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/mman.h>

#include <fcntl.h> //for O\_RDWR

#include <math.h>

#include <string.h>

#define MSG\_SIZE 70 // message size

#define R2VAL 10000 // constant resistor value

#define VIN 5 // input voltage value

int flag = 0; // when flag is set write flag to kernel via fifo

double thermVal = 0; // thermistor value

double temperature = 0; // thermistor temperature

double oldTemperature = 0; // previous thermistor temperature

char hostname[64], IP[16];

char warningMessage[MSG\_SIZE];

char petNames[5];

char buffer[MSG\_SIZE]; // to store received messages or messages to be sent.

int n;

struct sockaddr\_in addr, from;

socklen\_t fromlen;

//prints error message and exits when socket errors occur

void error(const char \*msg)

{

perror(msg);

exit(0);

}

//converts analog voltage to digital voltage and converts digital voltage to temperature in Fahrenheit

double ADCr(void){

double temp = 0;

/\*define variables \*/

int fd = open("/dev/mem", O\_RDWR|O\_SYNC);

char\* control\_register = (char\*)mmap(NULL, getpagesize(), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0x10F00000);

char\* reg2240 = (char\*)mmap(NULL, getpagesize(), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0x22400000);

char\* complete = (char\*)mmap(0, getpagesize(), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0x10800000);

volatile int AD\_complete;

int bit\_mask\_complete = 0x80, AD\_installed;

short \*read\_value;

short myValue;

read\_value = (short \*)control\_register;

AD\_installed = \*(reg2240) & 0x01;

if(AD\_installed == 0x01){

printf("AD installed.\n");

fflush(stdout);

//perform ADC

while(flag == 0){

\*control\_register = 0x40; //0100 0000

AD\_complete = bit\_mask\_complete & \*complete;

while(AD\_complete != 0x00 && flag == 0){ //wait, bit 7 = 0 on complete.

//prevent\_optimizer();

AD\_complete = bit\_mask\_complete & \*complete;

}

/\*conversion complete \*/

myValue = \*read\_value;

printf("myValue is: %d \n", myValue);

/\*convert digital voltage value to temperature in Fahrenheit via Steinhart-Hart equation\*/

//the resistance and temperature values are inversely proportional w/the thermistor

//so that needs to be accounted for.

temp = log(10000.0/(1024.0/myValue - 1));

temp = 1/(0.001129148 + (0.000234125 + (0.0000000876741 \* temp \* temp))\* temp);

temp = temp - 273.15;//convert K to C

temp = (temp \* 9.0)/5.0 + 32.0;//convert C to F

printf("Temperature is: %.2f\n", temp);

return temp;

usleep(1000000); //1s sleep

}

}

else{

printf("AD not installed.\n");

fflush(stdout);

}

}

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

int sock, newsock, length, i;

int permisson = 1; // broadcast permission value

struct hostent \*hp;

sock = socket(AF\_INET, SOCK\_DGRAM, 0); // Creates socket. Connectionless.

if (sock < 0)

error("socket");

length = sizeof(addr); // length of structure

bzero(&addr,length); // sets all values to zero.

addr.sin\_family = AF\_INET; // symbol constant for Internet domain

addr.sin\_addr.s\_addr = INADDR\_ANY; // IP address of the machine on which

//the server is running

addr.sin\_port = htons(2000);

gethostname(hostname, 64); //get IP address

hp = gethostbyname(hostname);

strcpy(IP, inet\_ntoa(\*((struct in\_addr \*)hp->h\_addr)));

printf("HP name:%s\n", hp->h\_name);

printf("IP Address: %s\n", IP);

//binds the socket to the address of the host and the port number

if (bind(sock, (struct sockaddr \*)&addr, length) < 0)

error("binding");

// change socket permissions to allow broadcast

if (setsockopt(sock, SOL\_SOCKET, SO\_BROADCAST, &permisson, sizeof(permisson)) < 0)

{

printf("error setting socket options\n");

exit(-1);

}

fromlen = sizeof(struct sockaddr\_in); // size of structure

//fifo initialization

int fifo1 = open("/dev/rtf/1", O\_RDWR);

if(fifo1 == -1){

printf("error.\n");

exit(-1);

}

while (1)

{

bzero(buffer, MSG\_SIZE); //clear buffer

temperature = ADCr(); //obtain temperature

//obtain initialization message from client

n = recvfrom(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, &fromlen);

if (n < 0)

error("recvfrom");

printf("%s\n", buffer);

}

//if temperature is below 100.5 degrees send low temp warning message to client

if(temperature < 100.5){

bzero(buffer, MSG\_SIZE);

sprintf(buffer, "%s %s", IP, "has low body heat. Take immediate action. ");

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255"); // broadcast address

n = sendto(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if (n < 0)

error("sendto");

}

//if temperature is above 102.5 degrees send high temp warning message to client

else if(temperature > 102.5){

bzero(buffer, MSG\_SIZE);

sprintf(buffer, "%s %s", IP, "has high body heat. Take immediate action. ");

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255"); // broadcast address

n = sendto(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if (n < 0)

error("sendto");

}

//if temperature is within range send regular temp message to client

else{

bzero(buffer, MSG\_SIZE);

sprintf(buffer, "%s %s", IP, "has a regular temperature. ");

n = sendto(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if (n < 0)

error("sendto");

}

}

return 0;

}