Final Report - ENG EC 535 Spring 2019

Team EmKG

Benjamin Wong (bcjwong@bu.edu) || Apollo Lo (apollol@bu.edu) || Gennifer Norman (gnorman@bu.edu)

Abstract: Our project utilizes embedded systems to create a wearable and lightweight system which records and displays the user's electrocardio activity in addition to calculating and displaying the user's heart rate in beats per minute. We believe our device has the potential to improve the lives of many people with a multitude of different applications.

Introduction

With technologies such as Fitbits and Apple Watches becoming increasingly popular, the trend in accessible health measurements is clear. More people are interested in having their health statistics readily available, quite literally, at their fingertips than ever before. The motivations for such interest varies from general curiosity to athleticism to health concerns and many more. The companies catering to this rise in demand must produce technology that is not only accurate but convenient for the consumer to use. This translates to non complex user interfacing and portability. The goal of our project is to utilize embedded systems to create a wearable and lightweight system which records and displays the user's electrocardio activity in addition to calculating and displaying the user's heart rate in beats per minute (bpm).

We believe this is an important device which could be used to improve the lives of many. Users such as those living with atrial fibrillation (AFib) would benefit greatly with our device as it displays electrocardio activity in real time. This data could be used to help an AFib affected user to monitor the regularity and rate of their heartbeat. Similarly, users at risk of heart attacks would be able to monitor their heart rate and subsequently be better equipped to take preventative measures should they notice an alarming increase in heart rate. Our device also has the potential to help those with anxiety as a common symptom of panic attacks is an accelerated heart rate. If users are in an environment unsafe or unsuited for an oncoming panic attack, being alerted of their increasing heart rate would afford them valuable time to relocate and/or be more prepared to experience the attack.

Our system primarily revolves around the gumstix board and a pulse sensor. The pulse sensor attaches to the tip of the user's finger, transmits data regarding the voltage through an amplifier while also converting analog signal to digital, and then the amplified data is passed along to the gumstix board where it is processed, analyzed, and then interpreted as heart rate. We were able to implement our device with calibration capability, comprehensive data display, and all the basic functionality of a typical electrocardiogram (EKG).

Design Flow

Our project is comprised of five main components:

1. *Pulse Sensor hardware:* LED light shines through the back of hardware into the fingertip (capillary tissues) and a built in ambient light sensor reads the light bouncing back.
2. *Arduino hardware:* The arduino is only used for amplifying the signal from the pulse sensor and converting it from analog signal into digital signal. The digital signal is then passed onto the gumstix for analysis via wires that represent binary digits.
3. *Gumstix hardware:* Used to read in digital signal via binary digits, analyze the data such as configuring the voltage threshold, calculate the beats per minute (BPM), and display the graph onto the LCD display.
4. *LCD configuration for displaying pulse signal:* Using Qt, the LCD display information such as BPM and plot out electrical activity of the heart.
5. *Buttons & LEDs:* Button used for initiation of calibration sequence; LEDs used as alert for concerningly high heart rates.

Our project's data begins with the pulse sensor. The sensor records analog data from the user's fingertip which is immediately transmitted to the Arduino Uno to be amplified and converted to a digital signal. The analog to digital conversion must occur in order for the gumstix board to be able to process and manipulate the data. After being converted to a ten bit number, the eight most significant bits are transmitted bitwise from the Arduino Uno to the gumstix board via eight GPIOs. The data is brought into the main script by a timed intake of GPIO data maintained by a kernel timer. Through data analysis and calculation, the raw data becomes a heart rate in beats per minute. Both the heart rate and current sensor value are transmitted to the LCD to be

displayed via reading from and writing to the device file */dev/mypulse* and */proc/mygraph* respectively. The diagram below outlines the intersystem communication:

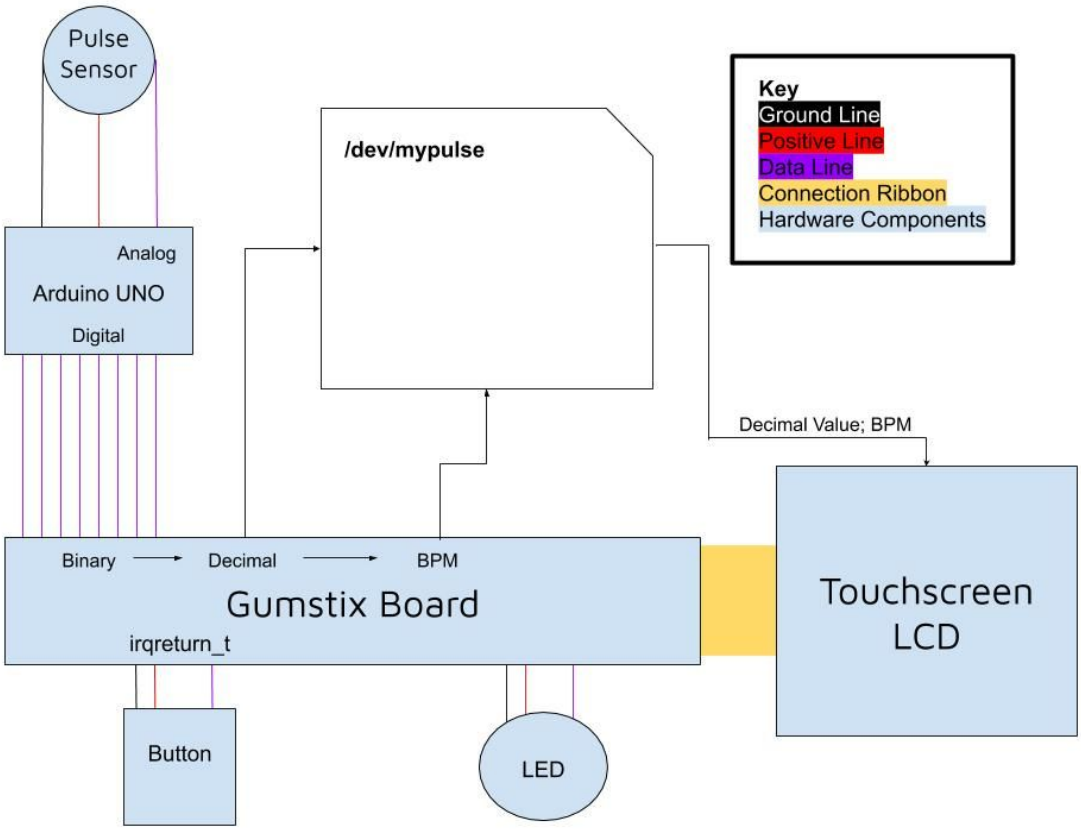

figure 1. Diagram of Device Components and Communication Flow

For the most part, we all had a hand in each aspect of the project. We worked collaboratively while roughly maintaining the structure we had laid out at the very beginning of our project process. Apollo took lead on the LCD screen and associated functionality such as data transmission to the Qt files. Benjamin took lead on data analysis which included taking in raw data from arduino and converting it to beats per minute in gumstix as well as linking the buttons and LEDs. Gennifer took lead on implementing other functionality of the data analysis such as calibration functionality and took lead in formulating the report. Overall our contributions split in the following regard:

Apollo - 37% || Benjamin - 37% || Gennifer - 26%

Project Details

a. **Pulse Sensor**

Research was done on how the pulse sensor works. Concisely, there is a LED light that emits from the sensor. The light shines through the back of hardware into the fingertip (capillary tissues) and a built in ambient light sensor reads the light bouncing back [1]. A heartbeat can be detected based on the optical reflection as light is scattered or absorbed when it passes through the blood vessels [3]. Three wires connect the pulse sensor to the arduino, ground, voltage (5V), and analog input channel (A0).

figure 2: Image of Pulse Sensor

b. **Arduino Code for converting analog signal to digital signal** *(See /EmKG_projectfinal/arduino/ADC.ino for the source code)*

With ground and voltage connected to the pulse sensor, it is able to send analog signal into the arduino via ANALOG_IN A0. The signal value can range from 0-1024 [1]. In the source file, 8 DIGITAL_OUT pins (4, 5, 6, 7, 8, 9, 10, 11) are used to send the digital signal as binary digits. After converting the signal to bits, it would output HIGH or LOW depending on if the binary value is 1 or 0.

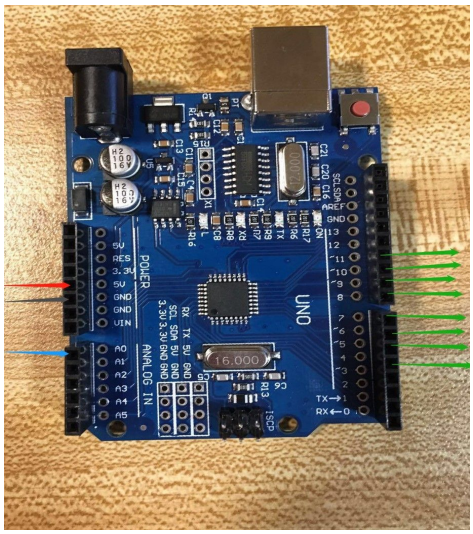

figure 3: Arduino wiring layout with input wires: red (voltage), black (ground), blue (analog in). Output wires: green 4-11 (digital out)

c. Gumstix Hardware

Our project utilized two primary components of the gumstix board hardware. The board receives the eight most significant bits of a ten bit value via GPIO pins 30, 29, 9, 28, 17, 16, 117, 118. Initially we transmitted all ten of the bits, however when transmitting data from ardiuno to gumstix, we ran into a bug regarding one of the GPIOs that controls PWM. Using GPIO 101 and GPIO 117 together causes our screen to fluctuate aggressively. In the end we compromise by removing usage of GPIO 101 in our project.

When Implementing more functionality for the project, we realized that we need more GPIO pins. This lead us to explore removing the least significant bit from the ten bit input which allowed us to use that GPIO to receive the input from the button. Later on, when integrating a tachycardia alert, we required yet another GPIO. We decided to remove the second least significant bit opting instead to power an LED array which alerts the user of a fast heart rate. Every GPIO is declared, and its direction defined, when the kernel module is initialized. We also made use of the flexible ribbon cable and serial connection to include an LCD screen in our system.

d. Device Driver/Kernel Module *(See source code /EmKG_projectfinal/km/mypulse.c)*

When the kernel module is inserted and initialized, memory space for two timers is dynamically allocated. The calibration timer (*my_calib)* completes its setup and initialization process when the attached button is pressed and the corresponding interrupt signal is handled. By contrast, the main timer (*my_timer)* continues its setup and initialization process immediately after memory space is allocated as this is what drives our data collection and subsequent calculations.

The main timer expires and resets every ten milliseconds, calling the function *display_callback* on expiration. When the callback function is called, the following sequence of events occurs. First, the data from the eight GPIO pins are collected using the standard method *pxa_get_gpio_value*. As they are collected in a bitwise fashion, we must "translate" the binary inputs to a decimal value. We do this through a simple binary to decimal conversion as shown in the code snippet in *figure 4* below, where *x* is the GPIO collected digit, *vol* is initialized as 0, *base* is initialized as 1, and this segment of code iterates ten times through a for loop.

```
if (x != 0)
{
        j = 1;
}
else
{
        j = 0;
}
vol += j*base;
base = base*2;
```

figure 4: Code to Convert Binary Data to Decimal Data

Once the data from the sensor is decimal, it is ready to be used to calculate the user's heart rate. Arguably the most important aspect of our calculation is the variable *BEAT_THRESHOLD.* This is a variable we set to a default value of 650, which can and should be calibrated to fit the user's individual readings [see part d. of this section for more information on calibration functionality]. We define a heartbeat as a sensor reading above this threshold. To find the time between each beat we count the number of readings between each beat, as we know the frequency of data sampling to be once every ten milliseconds. Counting the interim beats is made simple with the timer functionality. If

the value read after a heartbeat is less than the defined threshold, the timer is then reset using the function *mod_timer* and the data will be sampled again after ten milliseconds. This continues until a beat above the threshold is detected which activates the heart rate calculation logic.

Our function to calculate the user's heart rate in bpm is rooted in simple unit conversion. Mathematically this seemed sound, but while testing this function we noticed our results were consistently hundreds of bpm higher than expected values. We remain unsure of the reason, although our best conclusion is there was a propagation of error due to the liberties we took along the way to calculating our data such as exclusion of the two least significant bits. Through data analysis and effectively curve fitting our data, we were able to determine a proper constant to convert the received data to heart rate. Thus we produced our formula, which is shown below (*figure 5*):

```
bpm = 5000/(BEAT_COUNT+1);
```

figure 5: Formula used to Calculate Heart Rate in BPM

It is worth mentioning we are using the value *(BEAT_COUNT+1)* rather than *BEAT_COUNT*. This accounts for the beat at the end which otherwise would go unnoticed by the algorithm. Before remedying this, our tests returned results with discrepancies of about ten bpm between measured and calculated values. After fixing this bug, our results were near perfect.

e. **LCD** *(See /EmKG_projectfinal/LCD/main.cpp, /LCD/digitalclock.h, & /LCD/digitalclock.cpp for source code)*

The LCD display is created by writing a C++ program with header files called digitalclock.h. The implementation of the class goes into digitalclock.cpp. In digitalclock.cpp, a Qt timer is set up ever 0.05 seconds. This means LCD updates itself faster than the mypulse.c which will seems like the LCD is updating at real time. The cpp files reads from /dev/mypulse for the current calculated BPM and reads from /proc/mygraph for new voltage information. In the file, an array of 100 values is initialized, and whenever the new value is read from the proc file it is inserted at the end

of the array and pushes all the original values towards the front of the array. This means the newest voltage value is always at the end and the oldest voltage value is at the beginning. Every time the Qt is updated, it reads from this array and plots the information using Qpainter. In the function, the Qpainter runs in a for loop and graphs out each value in the array with 5 pixels increments in between. Everytime Qtimer reruns, Qpainter replots the entire graph. This results in the graph to appear like a rolling graph. The graph also display grid in grey dotted lines for user to see where the pulse is.

Plotting the values has to take into account a lot of limitations. The LCD has around 270 vertical pixels which means y-value cannot be directly taken from the array. The graph also has to save room on the bottom to display BPM. The LCD structured it with y-value 0 on the top and y-value 270 on the bottom of the screen. This means that we have to flip the graph so the pulse wouldn't appear upside down. The graph also has to reformat itself along with the grid so the pulse wouldn't appear too small or too big on the display. The following equation solves all the problem above:

```
y1 = 250 - (this->value[i] - min + 1)*250/(max-min);
y1 = y1 + 5;
y2 = 250 - (this->value[i+1]-min + 1)*250/(max-min);
y2 = y2 + 5;
```

f. **Buttons/LEDs** *(See /EmKG_project_final/km/mypulse.c for source code)*

When the kernel module is initialized, GPIO pin 31 is set up as an irq request line, which is connected to a button. When the button is pressed, it sends interrupt signals to the gumstix, where the interrupt will be handled in a function called *irq_BTN0*. The purpose of this button is to calibrate the user's voltage threshold. Since the pulse sensor emits a light and reads the amount of light that bounces back, different blood pressure can affect the blood flow rate through his/her fingertip. In the calibration stage, 500 data points from the arduino are read in, converted from binary digits to decimal, and then stored in an array. Then the 10 largest values and 10 smallest values are averaged, subtracted from each other, divided by 10, multiplied by 8, and then added onto the lower

average value. This formula essentially calculates 80% of the range of voltages that should not be counted as a beat. Only when the voltage exceeds this beat threshold value, will it be counted as a heart beat.

The LED light is linked up with GPIO 113. The point of having the LED is to warn the user when their BPM, which is calculated in the *display_callback* function, exceeds a certain value. While doing research, it was learnt that while resting, the normal heart rate is in the ranges of 60-100 bpm [4]. The alert system will light up when the calculated BPM exceeds a certain value. For demo purposes, the threshold was set at 90 BPM, and it was seen that the LEDs lit up when the user had increased BPM over 90.

Summary

We are ecstatic with the end product of our project. We met all of our main functionality goals with minimal sacrifices to do so. While we are proud of what we have done, there are a handful of additional functions we would have liked to include in our project had we more time. For example, the transmission of data from the sensor to the gumstix board currently uses an Arduino as an analog to digital converter (ADC) and amplifier. Ideally, we would like to decrease the amount of hardware we have in the system and would like to implement a simple ADC chip. This would reduce the hardware surface area as well as reducing power usage. Instead of powering two boards, we would rather power just one board and a small chip. We tried implementing the SSP library onto gumstix initially, but we failed to find the GPIO pins 23, 24, 25, 26 which correspond to clock, frame, transmit, and ext_clk even after looking through the provided documents. Finally, due to time constraints, we decided to opt for arduino as our ADC converter.

Another example of desired additional functionality is an idea we had throw around at the origin of our project which was quickly dismissed from the project as we started working. This function allowed the user, or the user's healthcare professional, to input the user's individual range for a healthy resting heart rate. While our data is conservative enough to safely protect the vast majority of users, personalization would be a fantastic feature and would further the device's usefulness.

References

[1] Pulse Sensor https://pulsesensor.com/

[2] Qt https://www.qt.io/

[3] Heatbeat sensor - Working and Application
https://www.elprocus.com/heartbeat-sensor-working-application/

[4] What is a Normal Heart Rate?
https://www.livescience.com/42081-normal-heart-rate.html