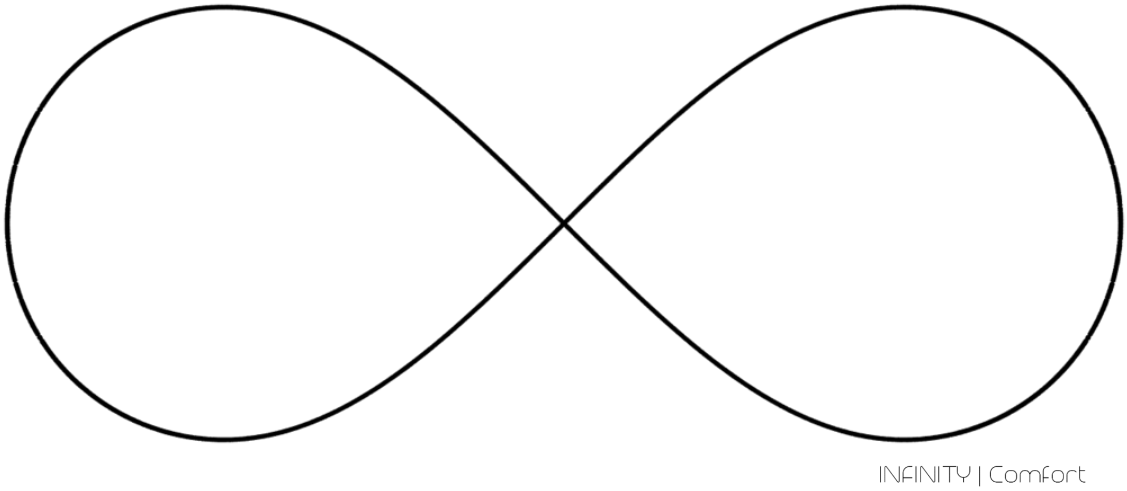
INFINITY | Comfort

OFFICIAL PROJECT REPORT

ECE 150 – 001



TEAM NAME:
Infinity Comfort

TEAM MEMBERS:

Jason Antao
Aditya Arora
Julian Parkin

TEAM NUMBER:
#39 - Team Number

PROJECT OVERVIEW

What is Infinity Cushion?

The 'Infinity Cushion' embedded project is a device that provides valuable, meaningful analytical data to users based on input over periods of time. The device, implemented as a connected cushion, such that it is used when the user sits down over a given interval of time, uses a combination of sensors over various sensitivities (piezoelectric and force resistors), to analyze motion and deviations in vibrations over the study period for which the person is working. The results from the sensors in the system will be stored and analyzed against functions that are developed to match the greatest regression factor(ex: linear regression plot), and at a given period, when the user desires, providing a well-formed summary and report of their studying pattern and focus over their selected interval, enabling the user to optimize their focus. Such a summary is deliverable in the form of a html document, readable on any web browser. The intended user is a student/studying individual, and the inspiration behind the design's model is a university student optimizing their studying abilities so that they are able to achieve the greatest level of focus while working at their desk.

From a software perspective, the data provided from the sensors in the cushion as well as the simple input from the button to start/stop studying at the screen will be written to a file, which is then processed immediately by polling it along a predetermined interval that matches the average study period for the particular time. All sensor data will serve as a functional layer of files in our software structure. The data written to a file will then, according to our predetermined interval, be run against the aforementioned functions to signify when the greatest motion over a period of time occurred, periods at which no motion was detected, as well as other minor parameters and oddities based on reasonable assumptions (ex: small motions not considered, as within certain percentage error for general human motion). Such statistical analysis will be ported from the functional layer to the analytical layer of our software structure, and further be transferred as a last step, to our output layer (the report file). The statistical data will then be processed into an output file, which the user can access via I/O through either the web or USB and view their respective report, while being able to save their data for future reference.

Concept Goal, for Doing Everything

If the Infinity Cushion could go beyond its current state of development for our project, we imagine the product being implemented with more advanced software algorithms developed through proven health-based research to evaluate any individual's sitting and focus conditions over a period of time. With the current hardware installed in our device, it is fully capable of having its data run through advanced software that would evaluate the aforementioned conditions. Furthermore, as with the Internet of Things, we foresee this product being connected to the web in order to have its application monitored over multi-use systems, such as in universities and/or health centres. Such data could be collected for statistical and analytical purposes, without the need for lengthy examinations to be conducted by physicians and/or professionals evaluating sitting/comfort levels while focusing. In retrospect, given increased time and resources, we would further enable the Infinity Cushion to have our data run against more advanced software algorithms, as well as connect it to the web for further statistical collaboration and monitoring.

What Subset does Infinity Cushion Fulfill?

The Infinity Cushion fulfills the subset of a device used for optimization purposes, both for personal use and for use in medical settings. The primary system I/O of the device takes input from the user as they work, essentially functioning as a monitor, while simultaneously processing data over that same period of time. The only major change from what we originally had planned to have in the product is a completely inclusive circuit packaging, whereas with our soldering and circuits, some components remained outside the shell of the cushion. Furthermore, filling such subset would enable the cushion to be run against real-life healthcare data for practical use in the respective industry.

SYSTEM DESIGN

Overview

The Infinity Cushion's overall system design can be broken down into the hardware and software level, each of which contain further layers of functions that work together to enable the unit to function as a whole. While designing our layers of hardware, we closely evaluated how the user would interact with the product, and what stress barriers were most susceptible to user interaction, hence identifying weak points in our initial design, which we were able to resolve. At the software level, our system is partitioned across multiple files, each of which serves as a vital component of the product's effectiveness. At the core of our software is our data-processing, further which we move onto data output, sensor aggregation, as well as the hardware and user interface. While details of the Infinity Cushion's software design will be analyzed below, an in-depth analysis of our software is included in the 'Software' section of the report.

Hardware

- Infinity Cushion
- Piezoelectric Sensors
- Force Sensor
- Display
- Buttons
- Omega Connection Shell
- External I/O Shell

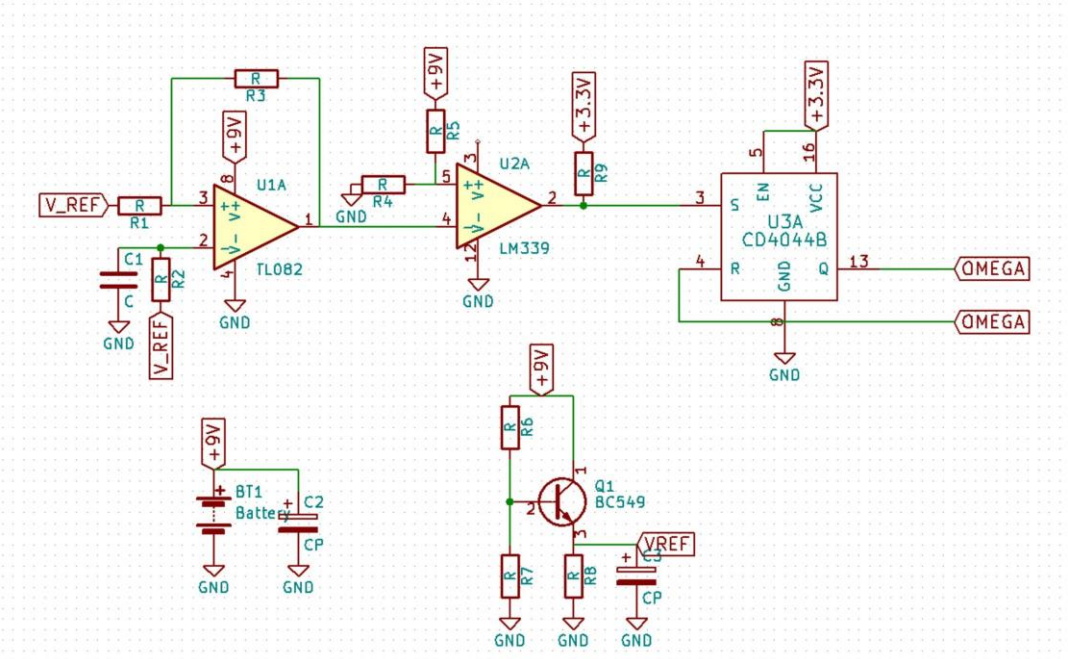


Figure #1: Piezoelectric Sensor Individual & Force Sensor Schematic for Placement within Bottom-Layer of Cushion

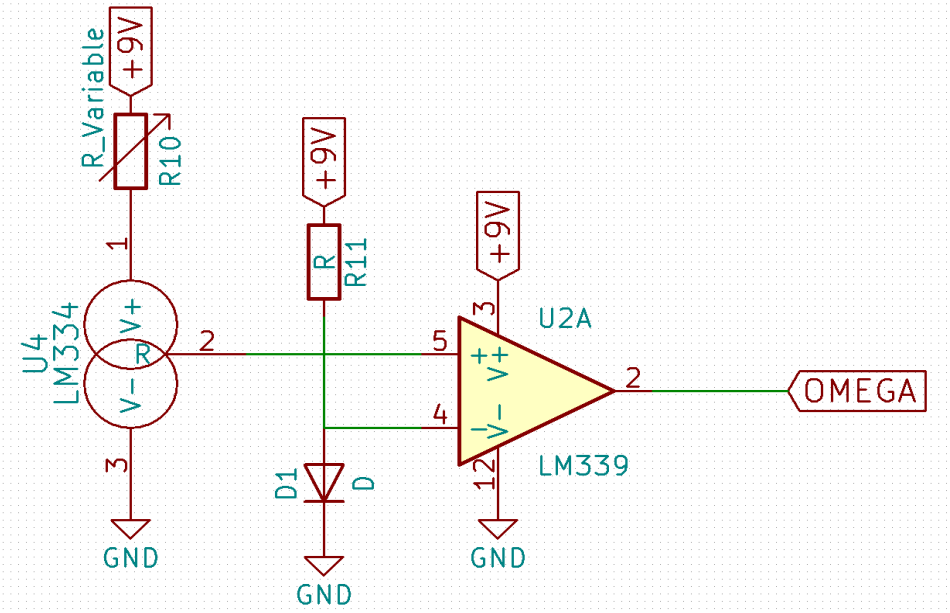


Figure #2: Force Sensitive Resistor Circuit

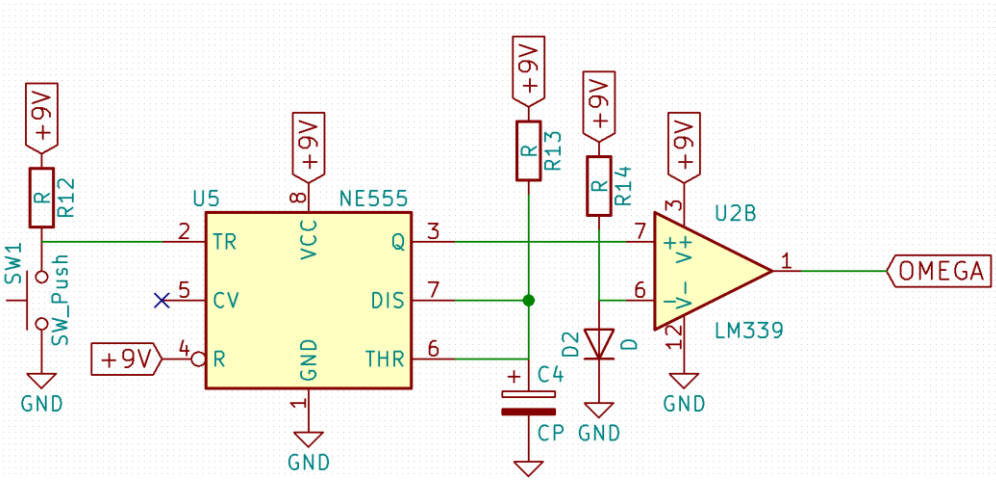


Figure #3: Button Input Circuit

As seen in the schematic above, the Infinity Cushion's interior shell contains a series of sensor components that are all implemented together within the bottom layer of the cushion's interior padding, enabling the sensors to function accurately while inside the cushion, which proved to be difficult to maintain without sufficient static protection.

Figure #4: Upper Echelon Products Specialty Cushion

The cushion itself is a high-quality cushion developed by Upper Echelon Products, and was custom ordered to ensure proper access to the cushion's interior. Furthermore, the cushion's ergonomic design makes it comfortable to use, while also serving its technical functionality.

The piezoelectric circuitry within the Infinity Cushion is interfaced using an amplifier, comparator, and latch, to the Omega board and requires its own custom soldering and power source, provided through a 9V battery. Three sensors are present within the cushion itself, positioned at symmetric positions relative to the surface area in order to result in the most accurate approximation of the motion of the user. A high-sensitivity force sensor is positioned at the center of the cushion to differentiate between a user positioned on the cushion, and an arbitrary object which may have been placed on the seat – a realistic assumption for the use of a cushion on a seat.

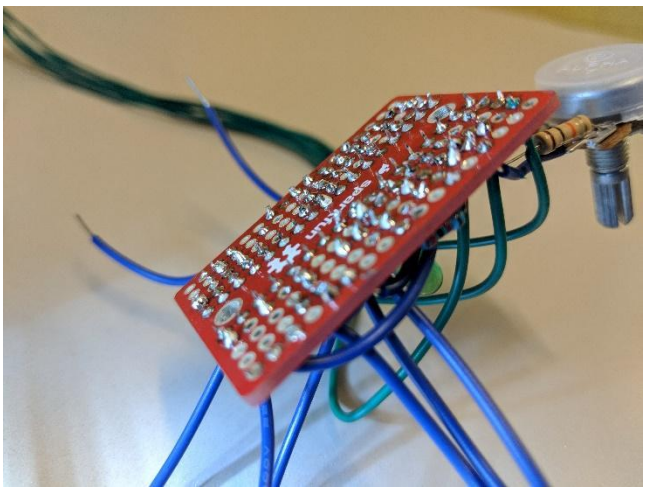


Figure #5: Soldering Detail on Sensor Interface Board

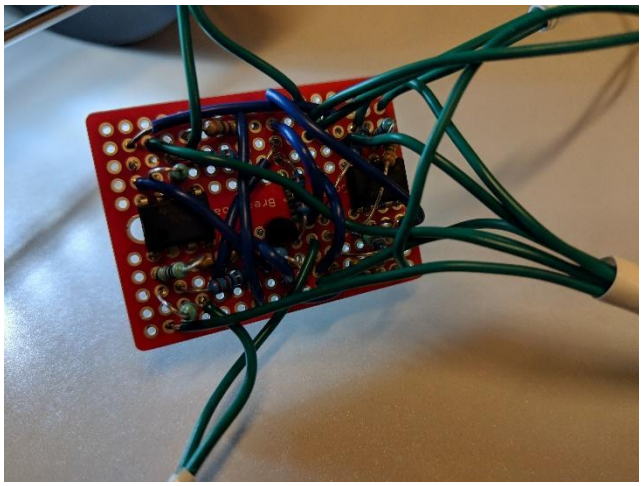
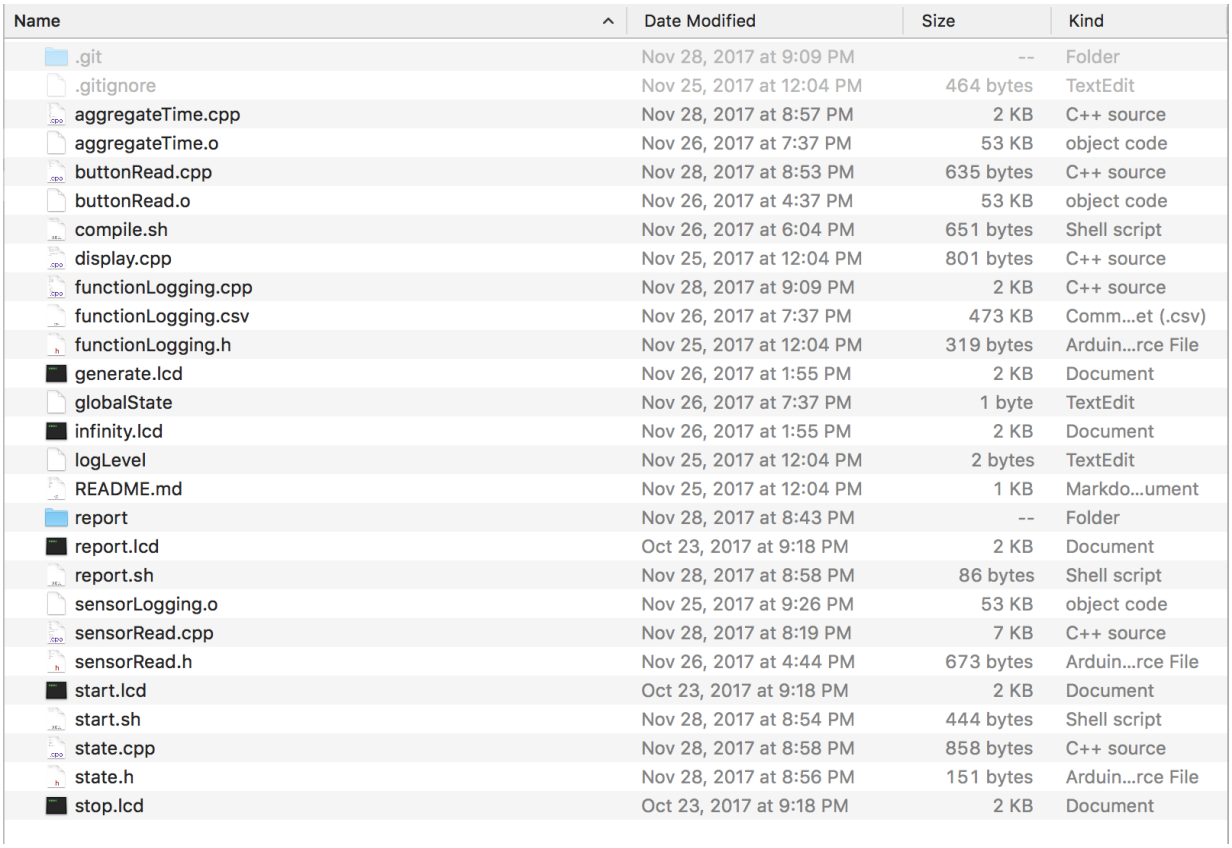


Figure #6: Circuitry on Amplifier Board

The Infinity Cushion further features an external shell, which holds both the Omega board as well as the input/output hardware for the product. The external shell is a plastic enclosure which is connected to the cushion through external wiring. Within the enclosure is the Omega board, as well as the Omega OLED Display and button, which is used to take input from the user to press the button at the point when they wish to begin studying. This display is used as a source of input confirmation for the user, indicating to them when the cushion is recording their performance, and when they wish to output their overall report.

Software

Similar to the hardware structural design, the software design for the Infinity Cushion follows a multi-shell model as seen in Figure #2 in the 'Software Design' section. All of the code used for both data processing and I/O for the system was worked on collaboratively through the pull/merge process on GitHub, (with over 65 unique commits) so that we were able to review and edit each other's work in real-time. Further details regarding the specific software components and the structure of code within each respective program is included in the 'Software Design' section.



Name	Date Modified	Size	Kind
.git	Nov 28, 2017 at 9:09 PM	--	Folder
.gitignore	Nov 25, 2017 at 12:04 PM	464 bytes	TextEdit
aggregateTime.cpp	Nov 28, 2017 at 8:57 PM	2 KB	C++ source
aggregateTime.o	Nov 26, 2017 at 7:37 PM	53 KB	object code
buttonRead.cpp	Nov 28, 2017 at 8:53 PM	635 bytes	C++ source
buttonRead.o	Nov 26, 2017 at 4:37 PM	53 KB	object code
compile.sh	Nov 26, 2017 at 6:04 PM	651 bytes	Shell script
display.cpp	Nov 25, 2017 at 12:04 PM	801 bytes	C++ source
functionLogging.cpp	Nov 28, 2017 at 9:09 PM	2 KB	C++ source
functionLogging.csv	Nov 26, 2017 at 7:37 PM	473 KB	Comm...et (.csv)
functionLogging.h	Nov 25, 2017 at 12:04 PM	319 bytes	Arduin...rce File
generate.lcd	Nov 26, 2017 at 1:55 PM	2 KB	Document
globalState	Nov 26, 2017 at 7:37 PM	1 byte	TextEdit
infinity.lcd	Nov 26, 2017 at 1:55 PM	2 KB	Document
logLevel	Nov 25, 2017 at 12:04 PM	2 bytes	TextEdit
README.md	Nov 25, 2017 at 12:04 PM	1 KB	Markdo...ument
report	Nov 28, 2017 at 8:43 PM	--	Folder
report.lcd	Oct 23, 2017 at 9:18 PM	2 KB	Document
report.sh	Nov 28, 2017 at 8:58 PM	86 bytes	Shell script
sensorLogging.o	Nov 25, 2017 at 9:26 PM	53 KB	object code
sensorRead.cpp	Nov 28, 2017 at 8:19 PM	7 KB	C++ source
sensorRead.h	Nov 26, 2017 at 4:44 PM	673 bytes	Arduin...rce File
start.lcd	Oct 23, 2017 at 9:18 PM	2 KB	Document
start.sh	Nov 28, 2017 at 8:54 PM	444 bytes	Shell script
state.cpp	Nov 28, 2017 at 8:58 PM	858 bytes	C++ source
state.h	Nov 28, 2017 at 8:56 PM	151 bytes	Arduin...rce File
stop.lcd	Oct 23, 2017 at 9:18 PM	2 KB	Document

Figure #7: File Map for Logging Files

The code is divided into three major sections:

- Sensor Input
- Logging
- Data Processing

The sensor input file is primarily responsible for taking sensor input, as it connects to the Omega board and saving such information over a buffer period – an application specific to our product's functionality. Furthermore, in order to process our data as fast and simply as possible, the sensor input taken in from the four connected sensors is converted into a custom 'char', through which we have encoded our HIGH/LOW values for each sensor to be logged and processed. The pin configuration for the Omega software is as follows: Pins 0-2 are receiving input from the three piezoelectric sensors, pin 3 is the force sensitive resistor, pin 18 is connected to the signal from the button and pin 19 is the reset line for the latch.

The software for the Infinity Cushion includes two separate logging files, for sensor logging and function logging, respectively. The purpose of the sensor logging is so that in case of error, or in that of a period of time, the status of each sensor is displayed so that it can be referenced later for reviewing and/or debugging purposes. The function logging file is responsible for taking our custom ‘char’ from the sensor input, and processing it over a selected buffer period, which we have maximized in order to allow for the greatest rate of data processing.

The data processing files for the Infinity Cushion serve the purpose of taking our input values and passing them to our statistical functions, which develop historical records for the sensor values over time, as well as process a report for the data. We implement graphics through the use of data-displaying and use integration to find the area between curves for the average sensor values over time, which correlate to differences between left and right movement in the cushion. This correlates to unsettlement for an individual at the seat, which outlines their individual movement based on the left and right side of the cushion, hence identifying minute movements.

SOFTWARE DESIGN

Function Call Tree

The flow of data from the point of input from the sensor to output in the form of report generation is passed through a series of functions as outlined below. Through the process, data is encoded from its raw form, as sensor input, to a ‘char’, which is then passed through a series of functions which process, log, and record the data for the purpose of quality assurance and system testing.

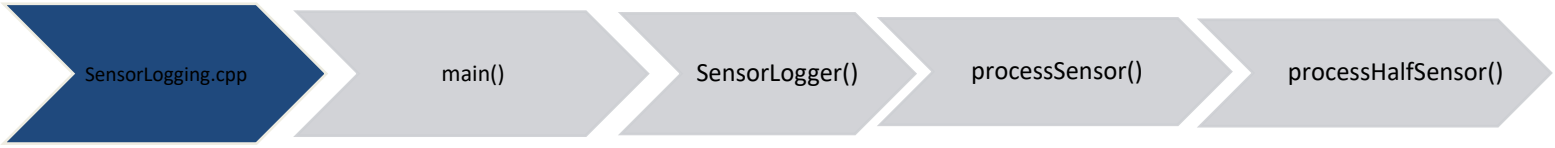


Figure #6: Function Call Tree for Software Implementation

Within the sensorLogging.cpp file, which servers the purpose of logging data from the sensors when input is received, five key functions are included:

- main()
- sensorLogger()
- processSensor()
- processHalfSensorData()
- summation()

The control of the entire project is handled by a bash script, start.sh. The bash script initializes and controls the display. At the start it waits inside buttonRead() until a button is pressed. Then the control is passed over to sensorLogging.cpp The main() function inside sensorLogging.cpp calls the sensorLogger() function which reads the GPIO values through a gpioReader class in the sensorRead.h header file. The sensor values are read until the button is pressed at which point sensor logging stops. The control is then passed on to aggregateTime.cpp which converts movement.csv to a human readable hours.csv which consists of two columns that display the date and the number of hours spent studying that day. These files are then read by our files in the report folder which visually plot the report as a colorful graph for the user to review. The report can be printed or can be saved in a pdf file for a later date.

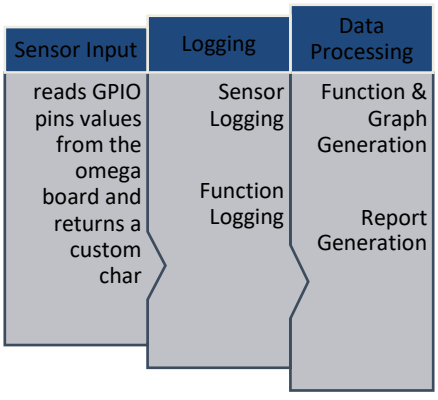


Figure #7: Multi-Layered Software Platform

System Dependent Components & Data / Method Flow

Classes & Interface:

The system dependent components are implemented in SensorRead.cpp and SenorRead.h. SensorRead.h provides a class gpioReader than handles all of the system dependent components in order to provide a clean interface for the system independent code. After a new gpioReader object is created, the init method can be called to allocate all the required pins for interfacing with the hardware. Then, the getSensorData method is called whenever the state of the hardware is needed. It also provides a setLogLevel method to aid with control over logging verbosity.

In retrospect, our individual 'char' is the data element which is taken and passed through the various fore mentioned functions and class, so that it is able to be decoded, and then processed for use in the final report generation.

System Independent Components

The system independent components include all the programs in the analytical and data preprocessing layers: sensorLogging.cpp and aggregateTime.cpp. sensorLogging.cpp decodes the custom char returned by the gpioReader class and writes it to a file after completing preprocessing. aggregateTime.cpp parses over this movement.csv file and accumulates the data in the format of number of hours studied per day and writes it to hours.csv.
The system independent components further includes logging, which is done with the help of functionLogging.h and functionLogging.cpp. More information regarding our function logging sequences and standards are included in the 'Logging Infrastructure' section below.

Logging Infrastructure

FunctionLogging - Sample Log

[2017-11-25.20:43:40] (INFO): StateChanged State changed to 0
[2017-11-25.20:44:46] (FATAL): gpioReader: failed to open gpio file, open() returned error code: 2
[2017-11-25.20:44:46] (INFO): gpioReader: assigning new log level
[2017-11-25.20:44:46] (DEBUG): sensorLogger entered loop of sensor reading and writing to file after processing
[2017-11-25.20:44:46] (INFO): StateChanged State changed to 1
[2017-11-25.20:44:46] (DEBUG): gpioReader: getting sensor data

Our function logging infrastructure follows a standard and consistent approach to what is displayed, in order to allow for the greatest effectiveness when used for testing and/or debugging purposes. The initial time-stamp of the log prints out the exact date, in the Year-Month-Day.Hour:Minute:Second format. We then identify the type of issue through the use of five levels of severity identifiers, which allow for higher readability and classification. These identifiers range from DEBUG to FATAL (only used if the program had to stop execution because of an unrecoverable error). Lastly, our preset errors and actions are displayed as necessary under the type-identifier. In retrospect, our logging infrastructure allows for easy time, type, and data identification so that any person is able to debug, and test our software and hardware implementation.

TESTING

Quality Assurance & Data Accuracy

Testing sensor input and especially calibrating output to have meaningful results proves to be an extremely challenging task for any physical ‘smart’ product. Likewise, the Infinity Cushion required extensive calibration and sensor testing for each individual component and facet of its use in order to result in accurate, meaningful data being outputted to the user. From using our own logging functions to debug issues within our software to physical testing of the report generation through simply sitting on the cushion for extended periods of time, with various loads, we were able to test and refine our report generation to allow for the highest level of accuracy in our calibrated and processed input. Specific details on our file testing assurance process are included below.

3-Way Communication Compatibility Assurance

Following along the experience of our team lead, we employed a 3-Way Communication Compatibility Assurance technique for all aspects of our project design. By deploying all software and schematics to all members of the group to see, and requiring confirmation from all members towards the accuracy of each component, we effectively prevented compatibility issues between our individual work from occurring later on in the build-stage. Our software files and logging files are all written to the same template and structure, following the same process for data-output, in order to allow for the greatest representation of data across all components of the program for the product.

File Testing

In order to test our program and product for the greatest level of accuracy, we conducted multiple tests over the development and final testing-phase that ranged from individual sensor I/O logging, to full-system tests over conducted periods of time. For our full-system tests, each individual sat on the cushion and ran through a full cycle for an individual product-use cycle. Furthermore, various live-loads were also placed on the cushion at varying positions to test for accuracy of force calibration, and differentiation between moving users and an inert, arbitrary object. After each testing phase we reevaluated issues within our product and software design, using our ‘3-Way Communication Compatibility Assurance’ method to confirm our understanding of the solution to each individual issue, up until the point that the product satisfied its intended purpose.

LIMITATIONS

Corner Cases & Physical Barriers

While the Infinity Cushion, by its original standard of design, fulfills all components, there still exist corner cases where the product may not function as originally intended. From a software perspective, over extenuated periods of time, the file size for our report generation and data processing reaches very large file sizes, into the hundreds of megabytes range. In order to reduce the impact of the slower data processing as a result of larger memory allocation, we took measures to reduce our data in the report generation process, effectively reducing our file sizes by significant factors.

From a physical perspective, the cushion is most accurately able to process data when positioned in its intended configuration, with the hump at the rear of the seat. However, in the case that the user places the cushion in an unintended configuration, (reverse-hump or upside down), any sensor logging occurring over that period of time may result in incorrect data collection, all under the assumption that data recording is still running, a function that the user can switch off.

LESSONS LEARNED

Overview

Having all worked on prior projects throughout the past years, we understand that reflecting on our work is a vital component of our success both as individuals, and as a team. Throughout the build of the Infinity Cushion, we were able to work effectively through, using effective communication mediums and time-management techniques to ensure that we always stayed on task and met our checkpoints for each week. Nonetheless, there exist a few key technical components through which we feel we could improve, if given the opportunity….

Data Encoding & Acquisition

When working on an embedded system, extended cycles for the processor caused due to inefficient code leads to significant issues over the time of use of the product for a user. Similarly, in our case, the constant processing and acquisition of sensor data multiple times every second from each of our five sensors lead to an initial dump of data that we were not able to process efficiently. However, by working to encode such data into a single 'char', and then decode the 'char' when writing our report, we were able to reduce our data processes, resulting in minor delays over extenuated periods of time for which the product was being used. If we could complete our software component again, we would like to spend time to create an even more efficient solution to our data processing and buffering, allowing faster report-generation and smaller file sizes over extended periods of time.

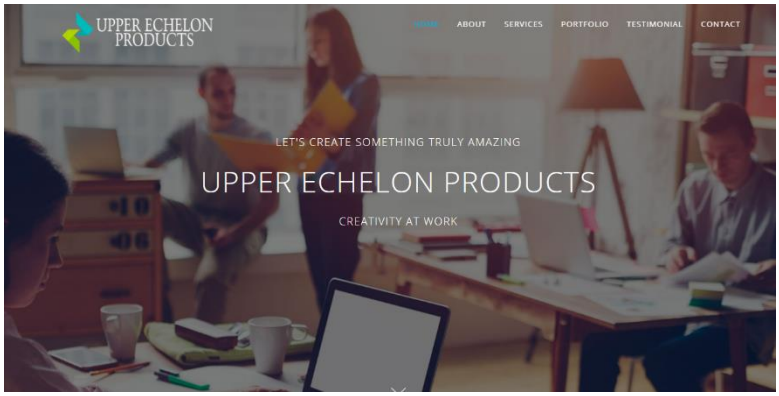


Figure #8: Upper Echelon Products, Austin TX. Through direct communication with the manufacturer, a custom cushion was acquired

Custom Cushion

When searching for an appropriate apparatus for our sensors in the form of a cushion, we created a 'Standards-Check' for which we evaluated each prospective cushion before finally making a purchase. However, initially we neglected to consider the significant buildup of static charge that certain cushions could cause over time. Nonetheless, we contacted the specific manufacturer and received a custom-made cushion with anti-static foam material for us to lay our circuitry inside. While in this case our communication with the direct manufacturer resulted in a novel solution to our issue, analyzing the stress-bounds of each material in use over time would enable us to design and build an even-more robust product.

SOURCE CODE

start.sh

```
#!/bin/sh

oled-exp -i # initialize the display
oled-exp draw infinity.lcd # draw text to display
sleep 3 # sleep before clearing the display
oled-exp -c # clear the display
oled-exp draw start.lcd
./buttonRead.o # detect button press
oled-exp -c
oled-exp draw generate.lcd
sleep 1
./sensorLogging.o # start logging sensorValues to movement.csv
./buttonRead.o
source report.sh # aggregate hours over time
oled-exp -c
oled-exp draw infinity.lcd
```

sensorRead.h

```
#ifndef SENSORREAD_H
#define SENSORREAD_H

/* Sensor Reading GPIO Class

Usage:

Create a new gpioReader object.

Call init method when ready to use GPIO (note that this does a lot of file io at once)

Call getSensorData to receive the data in a char:

LSB     Sensor 0
bit 1   Sensor 1
bit 2   Sensor 2
bit 3   Force Sensor
bit 4   button
bits 5-7  not used (always set)

The desctructor will unallocate the GPIO pins once the object is deleted.*/

#define BUF_LEN 128

//pin definitions in string form to make filename manipulation easier
#define PIN_0 "0"
#define PIN_1 "1"
#define PIN_2 "2"
#define FSR_PIN "3"
#define BUTN_PIN "18"
#define RST_PIN "19"
#define GPIO_PATH "/sys/class/gpio/"
#define GPIO_PIN_PATH "/sys/class/gpio/gpio"

#define INVARIANT_TEST (_invariant == 0xDEADBEEF)

#include "functionLogging.h" //needed to define logLevel enum

class gpioReader{
  public:
        int init();
        void setLogLevel(logLevel level);

        char getSensorData();

        gpioReader();
        gpioReader(logLevel level);
        ~gpioReader();
  private:
        int _gpioFiles[6];
        bool _init;
        logLevel _logLevel;
        int _invariant;
};

#endif
```

sensorRead.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#include "sensorRead.h"

#include "functionLogging.h"

//quick copy function that concatenates two const char* into a char* buffer, useful when working with the
#defined string constants
const char* _cat(char* dest, const char* const a, const char* const b){
  strncpy(dest, a, BUF_LEN);
  strcat(dest, b);
  return dest;
}


//initializes GPIO
int gpioReader::init(){
  if(_logLevel >= DEBUG)
        logger("DEBUG", "gpioReader:", "entered init function", 0);
  const char* gpioPins[6]; //most of the following file code is repetitive, so storing pin names in a array
lets us loop
  gpioPins[0] = PIN_0;
  gpioPins[1] = PIN_1;
  gpioPins[2] = PIN_2;
  gpioPins[3] = FSR_PIN;
  gpioPins[4] = BUTN_PIN;
  gpioPins[5] = RST_PIN;

  int file;
  char buffer[BUF_LEN];
  if((file = open(_cat(buffer, GPIO_PATH, "export"), O_WRONLY)) == -1){ //this file is written to to request
the GPIO
        logger("FATAL", "gpioReader:", "failed to open gpio file, open() returned error code:", errno);
        return -1;
  }
  for(int i = 0; i < 6; ++i){ //GPIO pins are requested in this loop
        strcpy(buffer, gpioPins[i]);
        if(write(file, buffer, strlen(buffer)) == -1){
        logger("FATAL", "gpioReader:", "failed to allocate GPIO, write() returned error code:", errno);
        return -2;
        }
  }

  close(file);
  if(_logLevel >= INFO)
        logger("INFO", "gpioReader:", "successfully alloacted GPIOs", 0);

  for(int i = 0; i < 6; ++i){//now we loop through and set if pins are input or output
        _cat(buffer, GPIO_PIN_PATH, gpioPins[i]);
        strcat(buffer, "/direction"); //we can use regular strcat here because buffer isn't const
        if((file = open(buffer, O_WRONLY)) == -1){
        logger("FATAL", "gpioReader:", "failed to open GPIO direction file, open() returned error code:",
errno);
        return -3;
        }
        strcpy(buffer, i == 5 ? "out" : "in"); //the reset latch pin is the only output
        if(write(file, buffer, strlen(buffer)) == -1){
```

```
        logger("FATAL", "gpioReader:", "failed to set GPIO direction, write() returned error code:", errno);
        return -4;
        }
        close(file);
  }

  if(_logLevel >= INFO)
        logger("INFO", "gpioReader:", "successfully set direction of GPIOs", 0);

  for(int i = 0; i < 6; ++i){ //now we open the actual files to read and write the state of each pin, and
store them so we can access them later
        _cat(buffer, GPIO_PIN_PATH, gpioPins[i]);
        strcat(buffer, "/value");
        if((file = open(buffer, O_RDWR)) == -1){
        logger("FATAL", "gpioReader:", "failed to open GPIO value file, open() returned error code:", errno);
        return -1;
        }
        _gpioFiles[i] = file;
  }

  if(_logLevel >= INFO)
        logger("INFO", "gpioReader:", "successfully initialized GPIOs", 0);

  _init = true;

  return 0;
}

void gpioReader::setLogLevel(logLevel level){
  logger("INFO", "gpioReader:", "assigning new log level", 0);
  _logLevel = level;
  return;
}

gpioReader::gpioReader(){
  _invariant = 0xDEADBEEF;
  _logLevel = INFO;
  _init = false;
}

gpioReader::gpioReader(logLevel level){
  _logLevel = level;
  if(_logLevel >= DEBUG)
        logger("DEBUG", "gpioReader:", "creating new gpioreader object", 0);
  _invariant = 0xDEADBEEF;
  _init = false;
}

gpioReader::~gpioReader(){
  if(_logLevel >= DEBUG){
        logger("DEBUG", "gpioReader:", "destroying gpioreader object", 0);
        if(!INVARIANT_TEST){
        logger("ERROR", "gpioReader:", "failed invariant test, something is very wrong here...", 0);
        }
  }
  if(_init){
        if(_logLevel >= INFO)
        logger("INFO", "gpioReader:", "closing GPIO files", 0);
        for(int i = 0; i < 6; ++i){
        close(_gpioFiles[i]);
        }
        if(_logLevel >= INFO)
        logger("INFO", "gpioReader:", "freeing GPIO", 0);

        const char* gpioPins[6];
        gpioPins[0] = PIN_0;
```

```
        gpioPins[1] = PIN_1;
        gpioPins[2] = PIN_2;
        gpioPins[3] = FSR_PIN;
        gpioPins[4] = BUTN_PIN;
        gpioPins[5] = RST_PIN;
        bool error = false;

        int file;
        char buffer[BUF_LEN];
        //now we have to do the reverse of init and write to unexport in order to free the allocated pins
        if((file = open(_cat(buffer, GPIO_PATH, "unexport"), O_WRONLY)) == -1){
        logger("FATAL", "gpioReader:", "failed to open gpio file, open() returned error code:", errno);
        error = true;
        }
        for(int i = 0; i < 6 && !error; ++i){
        strcpy(buffer, gpioPins[i]);
        if(write(file, buffer, strlen(buffer)) == -1){
        logger("FATAL", "gpioReader:", "failed to free GPIO, write() returned error code:", errno);
        error = true;
        }
        }
        if(_logLevel >= INFO && !error)
        logger("INFO", "gpioReader:", "successfully freed GPIO", 0);
        _init = false;
  }
}

char gpioReader::getSensorData(){
  if(!_init){
        if(_logLevel >= DEBUG)
        if(!INVARIANT_TEST)
        logger("ERROR", "gpioReader:", "failed invariant test, something is very wrong here...", 0);
        if(_logLevel >= WARNING)
        logger("WARNING", "gpioReader:", "tried to read without initializing, will return 0", 0);
        return 0;
  }
  if(_logLevel >= DEBUG){ //we only bother logging in DEBUG mode, to avoid incessant file writing
        //this code is exactly the same as below, only with a lot of extra log messages
        if(!INVARIANT_TEST){
        logger("ERROR", "gpioReader:", "failed invariant test, something is very wrong here...", 0);
        }
        logger("DEBUG", "gpioReader:", "getting sensor data", 0);
        char* buffer = new char;
        char output = 0;
        for(int i = 0; i < 5; ++i){
        if(lseek(_gpioFiles[i], 0, SEEK_SET) == -1)
        logger("ERROR", "gpioReader:", "failed to read data, lseek() returned error code", errno);
        if(read(_gpioFiles[i], buffer, 1) == -1)
        logger("ERROR", "gpioReader:", "failed to read data, read() returned error code", errno);
        else
        output += (*buffer == '1') << i;
        }
        if(lseek(_gpioFiles[5], 0, SEEK_SET) == -1)
        logger("ERROR", "gpioReader:", "failed to write data, lseek() returned error code", errno);
        if(write(_gpioFiles[5], "0", 1) == -1)
        logger("ERROR", "gpioReader:", "failed to write data, write() returned error code", errno);
        if(lseek(_gpioFiles[5], 0, SEEK_SET) == -1)
        logger("ERROR", "gpioReader:", "failed to write data, lseek() returned error code", errno);
        if(write(_gpioFiles[5], "1", 1) == -1)
        logger("ERROR", "gpioReader:", "failed to write data, write() returned error code", errno);
        output |= 0XE0;
        return output;
  }
  char* buffer = new char; //we only ever need to store '0' or '1', single char is fine
  char output = 0;
  for(int i = 0; i < 5; ++i){
```

```
        lseek(_gpioFiles[i], 0, SEEK_SET);
        read(_gpioFiles[i], buffer, 1);
        //quick way to set individual bits based on character:
        output += (*buffer == '1') << i; //comparison will return int 1 or 0 for true/false, which will get
shifted by current i and then added to char
  }
  //now we just need to strobe the latch pin low to reset
  lseek(_gpioFiles[5], 0, SEEK_SET);
  write(_gpioFiles[5], "0", 1);
  //circuitry is rated in the Mhz, delay caused by file writing is more than enough of a pulse width
  lseek(_gpioFiles[5], 0, SEEK_SET);
  write(_gpioFiles[5], "1", 1);
  output |= 0XE0; //mask the upper bits to 1 so they are defined
  return output;
}

//quick ifdef so this file can be compiled and run on its own (after including logging) to test sensors
#ifdef SENSOR_TEST
int main(){
    gpioReader gpio;
  gpio.setLogLevel(DEBUG);
    gpio.init();
  while(1){
        char a = gpio.getSensorData();
        printf("%d\n", a);
        usleep(1000000);
  }
    return 0;
}
#endif
```

buttonRead.cpp

```
#include <stdlib.h>
#include "functionLogging.h"
#include "state.h"
#include "sensorRead.h"

void buttonRead(){
  gpioReader gpio;
  gpio.setLogLevel(DEBUG);
  gpio.init();
  int output[5] = {0,0,0,0,0};
  logger("INFO", "buttonRead", "reading button values");
  while(output[4] != 1){
    char val = gpio.getSensorData();
    for(int i = 0; i < 5; i++){
      output[i] = 0;
    }
    for (int i = 0; i < 5; ++i) {
      output[i] = (val >> i) & 1;
    }
  }
}

int main(const int argc, const char* const argv[]){
  buttonRead();
  return 0;
}
```

sensorLogging.cpp

```
/*
logs sensor values to movement.csv
*/
#include <iostream>
#include <string>
#include <stdio.h>
#include <time.h>
#include <fstream>
#include "functionLogging.h"
#include "state.h"
#include "sensorRead.h"
using namespace std;

// void logger(char errorTag[], char functionName[], char message[], int errorCode);
float processSensor(int output[5]){
  /*
    pre-process sensor data
  */
  logger("DEBUG","processSensor", "entered pre-process sensor data");
  float val = 0.4 * output[3] + 0.25 * (output[0]+output[1]) + 0.3 * (output[2]);
  if(val > 0.8){
    return 1;
  }
  logger("DEBUG","processSensor", "exited pre-process sensor data");
  return 0;
}

float processHalfSensor(int output[1]){
  /*
  process data for left and right halves seperately
  */
  if(readState() >= 4){
    logger("DEBUG","processHalfSensor", "entered process data for left and right halves seperately");
  }
  else{
    logger("INFO","processHalfSensor", "processing halves");
  }
  int sum = output[0];
  if(readState() >= 4){
    logger("DEBUG","processHalfSensor", "exited process data for left and right halves seperately");
  }
  return sum;
}

float summation(int buffer[]){
  /*
  sum over buffer
  */
  logger("DEBUG","summation", "entered sum over buffer");
  float sum = 0;
  for(int i = 0; i < 10; i++){
    sum += buffer[i];
  }
  logger("DEBUG","summation", "exited sum over buffer");
  return sum;
}

void sensorLogger(){
  /*
  loop of sensor reading and writing to file after processing
  */
  gpioReader gpio;
  gpio.setLogLevel(DEBUG);
  gpio.init();
```

```
  logger("DEBUG", "sensorLogger", "entered loop of sensor reading and writing to file after processing");
  ofstream ofs;
  int buffer[10] = {0,0,0,0,0,0,0,0,0,0};
  int leftBuffer[10] = {0,0,0,0,0,0,0,0,0,0};
  int rightBuffer[10] = {0,0,0,0,0,0,0,0,0,0};
  int j = 0;
  ofs.open ("report/movement.csv", ofstream::out | ofstream::app);
  if (!ofs.is_open()) {
    logger("FATAL", "OpenFile", "Unable to open file movement.csv", 3);
    return;
  }
  int output[5] = {0,0,0,0,0};
  ofs<<"date,movement,left,right,fsr\n";
  writeState(1); //state: START
  while(output[4] != 1){
    char val = gpio.getSensorData();
    for(int i = 0; i < 5; i++){
      output[i] = 0;
    }
    /*
      output[5] = {sensor1, sensor2, sensor3, FSR, button1}
      value     = {button1, FSR, sensor3, sensor2, sensor1}
    */
    for (int i = 0; i < 5; ++i) {
      output[i] = (val >> i) & 1;
      cout<<output[i];
    }
    cout<<'\n';
    int leftOutput[1] = {output[0]};
    int rightOutput[1] = {output[1]};
    leftBuffer[j] = processHalfSensor(leftOutput);
    rightBuffer[j] = processHalfSensor(rightOutput);
    buffer[j] = processSensor(output);
    j++;
    if(j > 9){
      j = 0;
      float sumTotal = summation(buffer);
      float sumLeft = summation(leftBuffer);
      float sumRight = summation(rightBuffer);
      ofs<<currentDateTime()<<","<<sumTotal<<","<<sumLeft<<","<<sumRight<<","<<output[3]<<"\n";
      ofs.flush();
      logger("DEBUG", "fileWrite", "written to files and flushed");
    }
  }
  if(output[4] == 1){
    writeState(0); //state: REPORT_GENERATION
  }
  logger("DEBUG", "sensorLogger", "closed all files and exited");
  ofs.close();
}

int main()
{
       sensorLogger();

       return 0;
}
```

aggregateTime.cpp

```
/*
  accumulates the fsr values from movement.csv to hours.csv
*/

#include <iostream>
#include <string>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <fstream>
#include "state.h"
#include "functionLogging.h"

using namespace std;

bool parseLine(char line[]){
  // parse line to read only the fsr
  int commaCount = 0;
  int i = 0;
  while(commaCount != 4){
    if(line[i] == ','){
      commaCount++;
    }
    i++;
  }
  return line[i] == '1';
}

void hours(){
  // design hours.csv using movement.csv
  logger("INFO", "hours", "logging hours to hours.csv");
  ifstream ifs;
  ofstream ofs;
  ifs.open("report/movement.csv");
  if (!ifs.is_open()) {
    //checking if file is open
    logger("FATAL", "OpenFile", "Unable to open file movement.csv", 3);
    return;
  }
  ofs.open("report/hours.csv");
  if (!ofs.is_open()) {
    //checking if file is open
    logger("FATAL", "OpenFile", "Unable to open file movement.csv", 3);
    return;
  }
  int seconds = 0;
  char date[11];
  char previousDate[11];
  bool dateSet = false;
  char line[100];
  bool done = false;
  int fileLineNumber = 0;
  while (!done) {             // Read from file
      ++fileLineNumber;
      if (!ifs.getline(line, 99)) { // Get next line
          if (ifs.eof()) {                      // End of file check
              done = true;
              break;
          }
      }
      if(fileLineNumber-1){
        for(int i = 0; i < 10; i++){
          date[i] = line[i];
        }
```

```
        date[10] = '\0';
        if(!dateSet){
          strcpy(previousDate, date);
          dateSet = true;
        }
        if(strcmp(date, previousDate) != 0){
          // only write when the date changes
          for(int i = 0; i < 10; i++){
            ofs<<previousDate[i];
          }
          ofs<<','<<(float)seconds/3600<<endl;
          seconds=0;
          if(parseLine(line)){
            seconds++;
          }
          strcpy(previousDate, date);
        }
        else{
          if(parseLine(line)){
            seconds++;
          }
        }
      }
      else{
        ofs<<"date,hoursStudying\n";
      }
    }
    for(int i = 0; i < 10; i++){
      ofs<<date[i];
    }
    ofs<<','<<(float) seconds/3600<<endl;
    ifs.close();
    ofs.close();
    logger("INFO", "hours", "finished logging hours");
}


int main(){
  hours();
  // change global state to 0
  writeState(0);
}
```

functionLogging.h

```
#ifndef FUNCTIONTEST_H
#define FUNCTIONTEST_H
#include <string>
enum logLevel{FATAL, ERROR, WARNING, INFO, DEBUG};
int returnLevel(const char errorTag[]);
const std::string currentDateTime();
void logger(const char errorTag[], const char functionName[], const char message[], const int errorCode = 0);

#endif
```

functionLogging.cpp

```
#include <string>
#include <string.h>
#include <fstream>

#include "functionLogging.h"

using namespace std;

int returnLevel(const char errorTag[]){
  // return log level from global file log level
  if(!strcmp(errorTag, "FATAL")){
    return 0;
  }
  else if(!strcmp(errorTag, "ERROR")){
    return 1;
  }
  else if(!strcmp(errorTag, "WARNING")){
    return 2;
  }
  else if(!strcmp(errorTag, "INFO")){
    return 3;
  }
  else if(!strcmp(errorTag, "DEBUG")){
    return 4;
  }
  else{
    return 4;
  }
}

const std::string currentDateTime() {
    /*
    return currentDate and Time in string format
    format is YYYY-MM-DD.HH:mm:ss
    */
    time_t     now = time(0);
    struct tm  tstruct;
    char       buf[80];
    tstruct = *localtime(&now);
    strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);

    return buf;
}

void logger(const char errorTag[], const char functionName[], const char message[], const int errorCode){
      /*
      function[]: function names
      exitFlag: whether a function is being entered into or exited from
      */
      ifstream ift;
      ift.open("logLevel");
```

```
        int logLevel;
        ift>>logLevel;
        ift.close();
        ofstream oft;
        if(logLevel > 4){
          logLevel = 4;
        }
        if(returnLevel(errorTag) > logLevel){
          return;
        }
        oft.open("functionLogging.csv", ofstream::out | ofstream::app);
        oft<<'['<<currentDateTime()<<"] ";
        oft<<'(';
        for(int i = 0; i < strlen(errorTag); i++){
            oft<<errorTag[i];
        }
        oft<<"): ";
        for(int i = 0; i < strlen(functionName); i++){
            oft<<functionName[i];
        }
        oft<<" ";
        for(int i = 0; i < strlen(message); i++){
            oft<<message[i];
        }
        if(errorCode != 0){
          oft<<" ";
          oft<<errorCode;
        }
        oft<<"\n";
        oft.close();
}
```

report.sh

```
oled-exp -c # clear the display
oled-exp draw report.lcd
./aggregateTime.o
```

PEER CONTRIBUTION

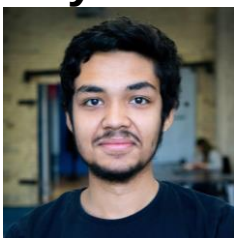
Jason Antao

My role in this project was project facilitation as well as report and structure implementation. From the software perspective, starting with a first-principles approach, Aditya and myself were able to create a critical-flow path for the software components for the course of the software development. Moreover, aside from soldering connections to the display board to meet the physical packaging design of the hardware, I fitted the completed sensors into the custom-built cushion to ensure robustness. I enjoyed working with my team members, both of which were experts in their respective divisions, and facilitating a working product over a limited period of time, with limited resources. I especially enjoyed my communication with external suppliers and being able to effectively communicate each individual step of the development into a concise, yet extremely effective product and report. I will cherish the time spent working with Julian Parkin and Aditya Arora, as we were all able to learn from one another, both technically, and socially, all while enjoying what we do best.

Julian Parkin

In this project I mostly dealt with the hardware aspect of things. This included designing and building the three major circuits to interface the piezos, the force sensor, and the button. As well, I also wrote the custom GPIO driver and interface for the hardware so that the rest of the program could receive the sensor data. What I most enjoyed in this project was being able to design custom hardware for a specific purpose, something that I don't usually get a chance to do in class. Working with Aditya and Jason was great. I found it an effective team dynamic to have group members each with our own area of expertise to bring to the project.

Aditya Arora



In this project I handled the software aspect, which included the scripts that controlled the state of the software which include all the individual cpp files that fulfill the purpose of button reading, sensor logging, aggregating time as well as generating the final report in HTML. Being a software developer at heart, I really relished writing code in the middle of the night. I loved seeing how the project grew step by step to become what it is today. I genuinely enjoyed learning from my peers in their area of expertise especially the sensor reading and GPIO having myself never played with such hardware. Both my teammates kept me on my toes and I was always learning while still being a valuable contributor to the team.

END OF OFFICIAL PROJECT REPORT

Please note that this document is the Official Project Report for the ECE150-001 Final Project. All details and sources for parts have been cross-checked for greatest accuracy and detail. All changes from the Initial Project Report to our Final Project Report reflect a combination of the feedback we received via Dropbox as well as discussions with our group and Prof. Ward. Thank you.