Electrical Contact Monitoring
GHSP

Zac Lynn - Lynnz@jsjcorp.com
4/29/2022

Table of Contents

Arduino Setup:

Requirements:

1.) The input voltage range to the analog inputs(ADC) should be within 0-5 volts. Voltage dividers can be used to increase the maximum voltage input at the cost of voltage resolution.

2.) The SD card file system should be FAT32/FAT16. 32Gb and smaller SD cards should either arrive formatted as FAT32/FAT16 or can easily be reformatted with windows file explorer. 64Gb SD cards by default are usually formatted as EXFAT which is not compatible with the system. 64Gb cards cannot be reformatted as FAT32 with built in windows tools, so to reformat as FAT32 free disk partitioning software similar to this tool should be used. NOTE: The partition type must be MBR for the Arduino to be able to interface with the SD card.

3.) The Arduino source code depends on the yxml, and SD_LIB libraries to function. In order for the code to compile all files need to be in the same directory. If the yxml library is not present it can be downloaded from here. The other files can be found in appendix B.

4.) Arduino IDE may be required to upload the Contact_Monitoring.ino code to the Arduino. Arduino IDE can be downloaded here.

Limitations:

1.) The 10-bit ADC has a resolution of 4.9mV when using the 0-5V input range. Thus, at most only 3 significant figures from the data output should be used. If a voltage divider is used then the voltage resolution follows the equation below.

$$V_{resolution} = V_{max}/ADC_{bits} = V_{max}/1024$$
$$V_{resolution} = 5.0V/1024 = 0.00488\text{V}$$

2.) The sample period should never be less than 2ms. As more input channels are used the contact period will have to be raised. A 5ms sample period is sufficiently long for reading 16 contacts.

3.) There will be some variation from the desired sample period in the data, however the average change in time between measurements will be the sample period. A few larger deviations in the change in time may exist as a result of the internal buffer being full and having to stop data acquisition in order to save data.

4.) The FAT32 file system limits files to 4Gb so the output data will be broken up into sequential files for long tests.

Directions:

1.) Some configuration variables in the Arduino code may need to be modified. Figure 1 below shows the configuration variables in Contact_Monitoring.ino.

```
/******************** Configurations ********************************/
#define FILE_NAME_TEMPLATE "test"
#define FILE_LIMIT 75
#define WRITE_LIMIT 1040000
#define V_SCALE 5.00//14.7282
#define SAMPLE_PERIOD 5
uint32_t BYTE_LIMIT = 1000000 * 75;
/*********************************************************************/
```

Figure 1: Arduino Code Configuration Variables

The FILE_NAME_TEMPLATE variable determines the naming convention of the output data file(s). When files are saved, the file name template and a number appended to the end are used to create unique names.

The FILE_LIMIT configuration should be set based on the storage size of the SD card in use. The maximum file size in the FAT32 files system is 4Gb, so for a 64Gb SD card using 4Gb files the FILE_LIMIT should be 16.

The WRITE_LIMIT variable is used to limit the size of a file based on how many rows it will take up in an excel file. The value of 1040000 is close to the maximum excel matrix height. Limiting the file size in this way should be done if the user plans to analyze the data in any way using excel.

If a voltage divider is used on the analog inputs to increase the input voltage range, the V_SCALE configuration will need to be updated. V_SCALE should be set to the maximum allowable input voltage to the voltage divider. For example, if a ⅖ voltage divider is used then V_SCALE should be $(\frac{2}{5})^{-1} * 5V = 5V * 2.5 = 12.5V$. This means that up to 12.5V can be applied to the input of the voltage divider without exceeding the 5v max input to the ADC. The equation below shows how the V_SCALE config should be calculated.

$$V_{Scale} = (\frac{R_2}{R_1+R_2})^{-1} * 5V$$

The SAMPLE_PERIOD can be set to control how frequently data is sampled. This works if the sample period is longer than the periods discussed in the limitations section.

The BYTE_LIMIT along with the WRITE_LIMIT control how large any single output file can be. The BYTE_LIMIT can be used if excel is not needed for analysis, otherwise it is better to use the WRITE_LIMIT. If the resulting .csv will be used directly by analysis tools such as Python or MATLAB then the BYTE_LIMIT can be set to 4GB, the FAT32 file size limit.

If the configuration variables have been changed the code will need to be reuploaded to the Arduino. Before trying to upload the code, the correct COM port and Arduino model need to be selected. The Arduino board should be set to “Arduino Mega or Mega 2560” as shown in figure 2 on the next page.

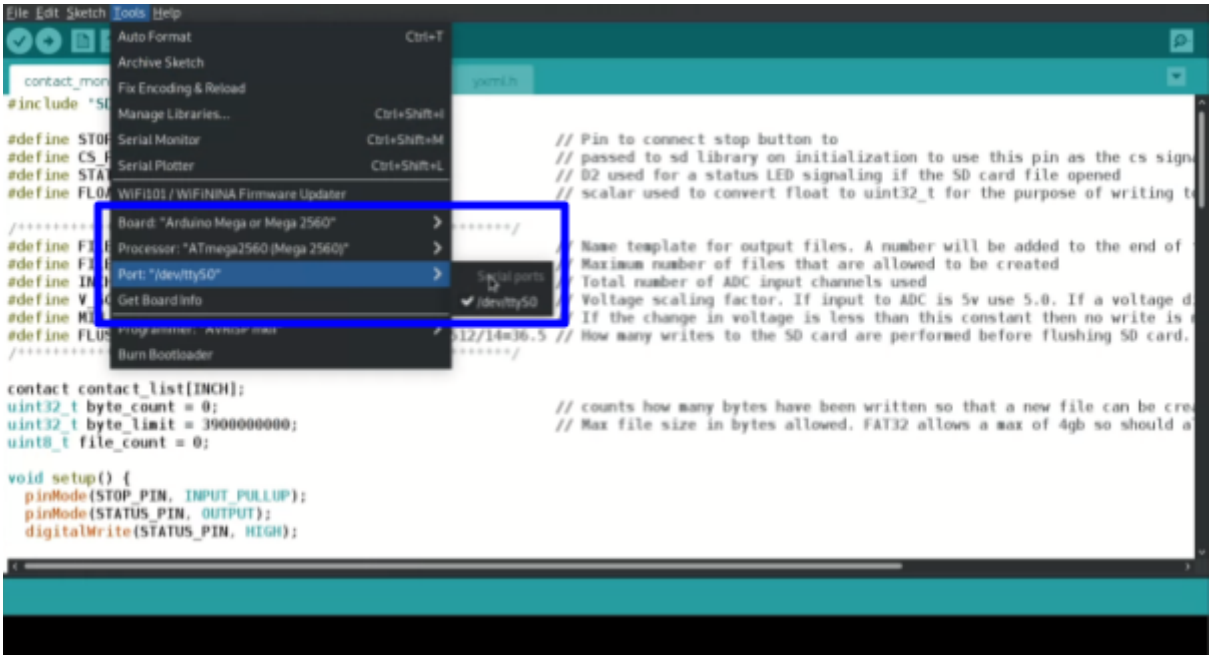


Figure 2: Selecting COM port and Arduino Model

After setting the port and board, click the upload button in the top left corner of the Arduino IDE window as highlighted in figure 3 below.

Figure 3: Uploading Arduino Code

2.) Write an XML configuration file and save to SD card. Figure 4 below shows the required structure of the configuration file and a text version is available in appendix A. The file should be named “config.xml”. Note that on the Arduino mega the first analog pin, A0, is represented in the configuration file as 54. The valid range for pin numbers is A0-A15 or as they should appear in the configuration file 54-69.

```
<config>
    <contact>
        <group> 10 </group>

        <voltageLow> 0.000 </voltageLow>
        <voltageHigh> 2.100 </voltageHigh>
        <voltageLow> 2.100 </voltageLow>
        <voltageHigh> 2.400 </voltageHigh>
        <voltageLow> 2.400 </voltageLow>
        <voltageHigh> 3.700 </voltageHigh>
        <voltageLow> 3.700 </voltageLow>
        <voltageHigh> 3.900 </voltageHigh>
        <voltageLow> 3.900 </voltageLow>
        <voltageHigh> 5.000 </voltageHigh>

        <pin> 54 </pin>
    </contact>

    <contact>
        <group> 11 </group>

        <voltageLow> 0.000 </voltageLow>
        <voltageHigh> 2.100 </voltageHigh>
        <voltageLow> 2.100 </voltageLow>
        <voltageHigh> 2.400 </voltageHigh>
        <voltageLow> 2.400 </voltageLow>
        <voltageHigh> 3.700 </voltageHigh>
        <voltageLow> 3.700 </voltageLow>
        <voltageHigh> 3.900 </voltageHigh>
        <voltageLow> 3.900 </voltageLow>
        <voltageHigh> 5.000 </voltageHigh>

        <pin> 55 </pin>
    </contact>
</config>
```

Figure 4: XML Configuration File

The group identifier can be between 0-255 and is used in decoding the output to sort the voltage data based on contacts. Using the voltageLow and voltageHigh configuration the system can assign a state to each contact in the output file. Data saving does not depend on voltage levels so if the state is not needed the voltage high and low values can be anything. Additionally, if multiple voltage zones should be tested, the voltageLow and voltageHigh elements can be repeated to configure more zones. The state assigned for each zone is based on the order they appear in the configuration file starting with a state of 1. State 0 is reserved for any “undefined” voltage measured outside of all voltage zones. So, reordering the voltage zones in the config file will change what state number is assigned to a zone. Make sure that voltageLow precedes

voltageHigh in the config file. Only 10 voltage zones are allowed, 11 if the undefined “one” is counted as well.

3.) Connect electrical contact test points to Arduino analog pins using the screw terminals. Make sure to tie the ground of the test part to the Arduino GND pin. The analog input screw terminals are highlighted in the figure below.

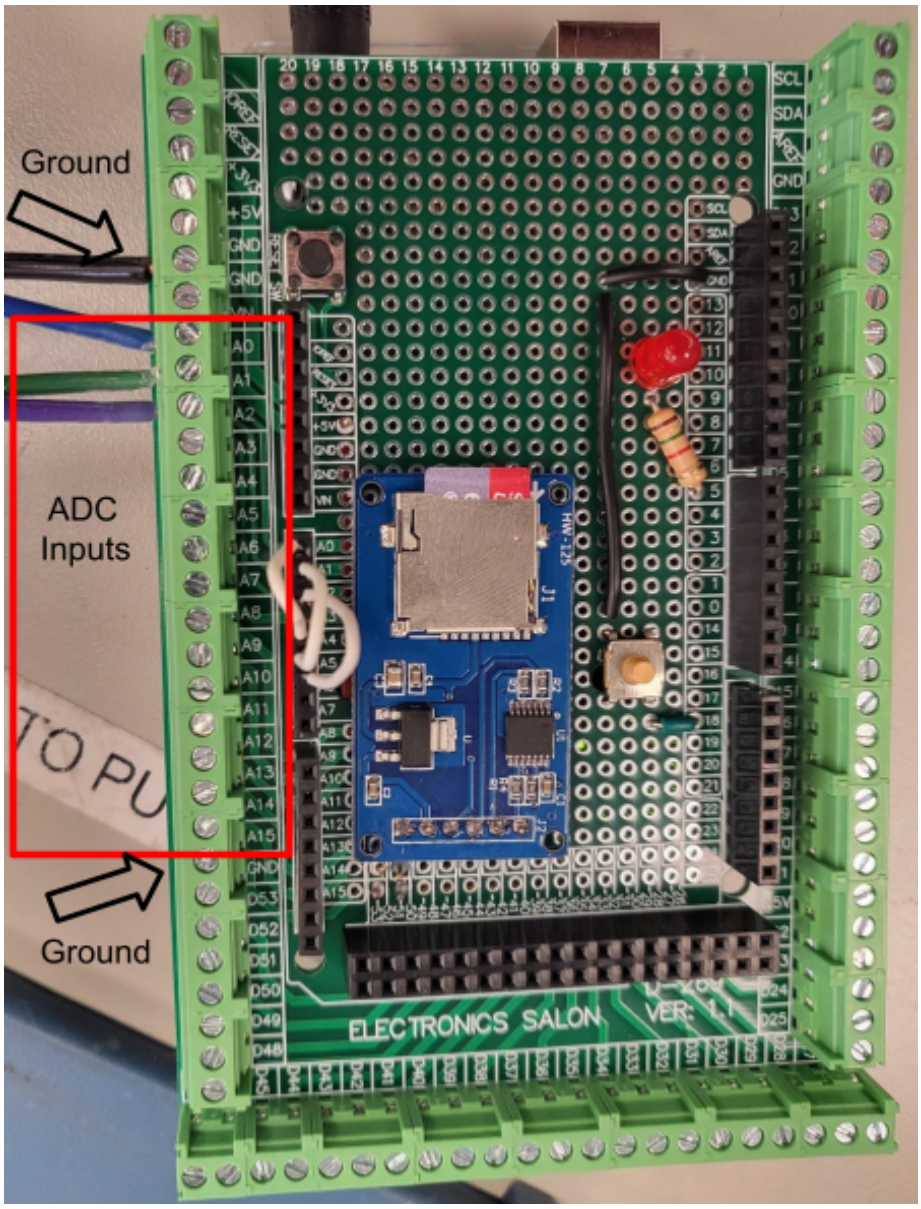


Figure 5: Arduino Analog Inputs

4.) Power up test part and Arduino to check for the following conditions:

- If the LED begins blinking, then the Arduino failed to read the config file
- If the LED remains on, then the Arduino failed to create a new data file
- If the LED turns off, then data collection has begun

A USB serial monitor can be used to see additional debugging output while initializing. Do not connect and or reconnect the USB serial monitor during a test as it causes a reset of the Arduino. It is okay to remove the serial monitor though. For example, the Arduino can be started with the serial monitor connected then after successful initialization the usb cable can be removed from the arduino without causing a reset.

5.) Before removing power from the Arduino press the button and wait for the LED to turn back on. Pushing the button saves and closes the output file on the SD card and ends data acquisition. Additionally, if the system's maximum data limit set by the FILE_LIMIT and BYTE/WRITE_LIMIT variables is reached the last file will be saved and closed and the LED will turn on after data acquisition has ended. Removing power from the Arduino before the SD file has been saved and closed can cause the file to be corrupted which usually requires the SD card to be erased and reformatted.

6.) Before beginning a test it is recommended to run a short test, around 1 minute, and confirm the system has been configured correctly using the csv output described in the PC Setup section.

PC Setup:

Requirements:

1.) Must have Python installed. Python can be downloaded here.
2.) The Numpy library must also be installed. The figure below shows how to install Numpy after python has been installed. If the commands that follow do not work, try replacing "python3" with either "py" or "py3".

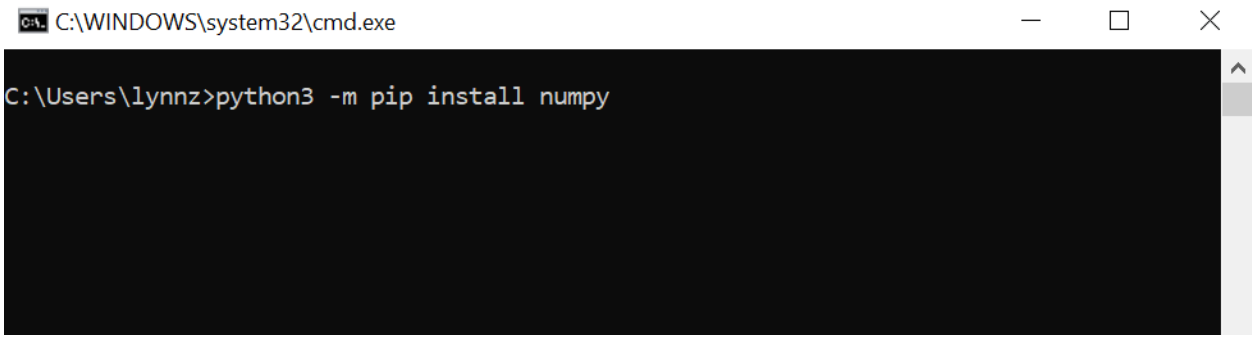


Figure 6: Installing Numpy

Directions:

1.) Copy the binary data file(s) from the SD card into the same directory as the Contact_Monitoring.py file.

2.) The binary to csv conversion can do one file or multiple files using the filename stem and number standard. The filename stem is used to check a directory for files with the same name in a numbered sequence. All files in sequence up to 255 will be converted. For example if the filename stem is “test”, all files named “test1”, “test2”, “test3” all the way up to “test255” will be processed if they exist. To convert the files, open a command terminal in the same directory and run the command shown below.

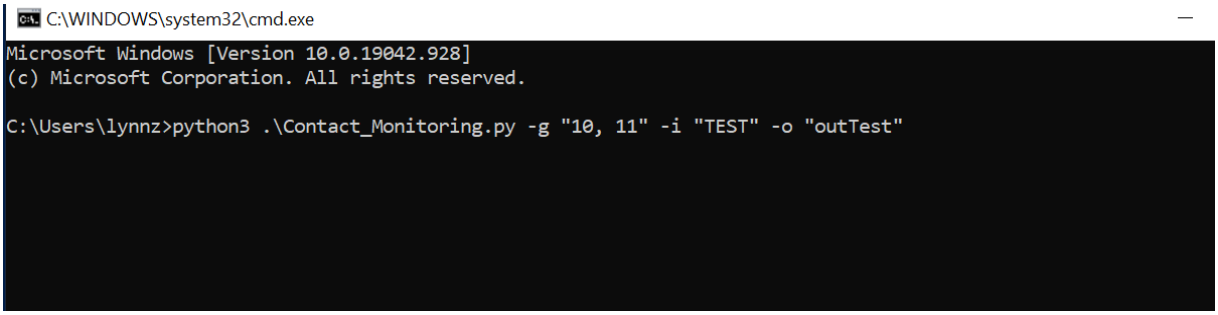


Figure 7: Decoding Data Command

The only required option is -g or –group. The group argument should be a list of contact groups separated by semicolons containing one or more contacts separated by commas.For example if a setup has two buttons being tested each with 3 redundant contacts you may use something similar to “10, 11, 12; 20, 21, 22”. Only data points with an ID that matches the supplied list IDs will be saved to the output CSV file. For example, if data was collected from 6 contacts named: “1, 2, 3; 4, 5, 6” and the -g argument was “1, 2, 3” then the output csv file will only contain data from the first three contacts. The available options and arguments are described on the next page in table 1.

Table 1: Data Decoding Options

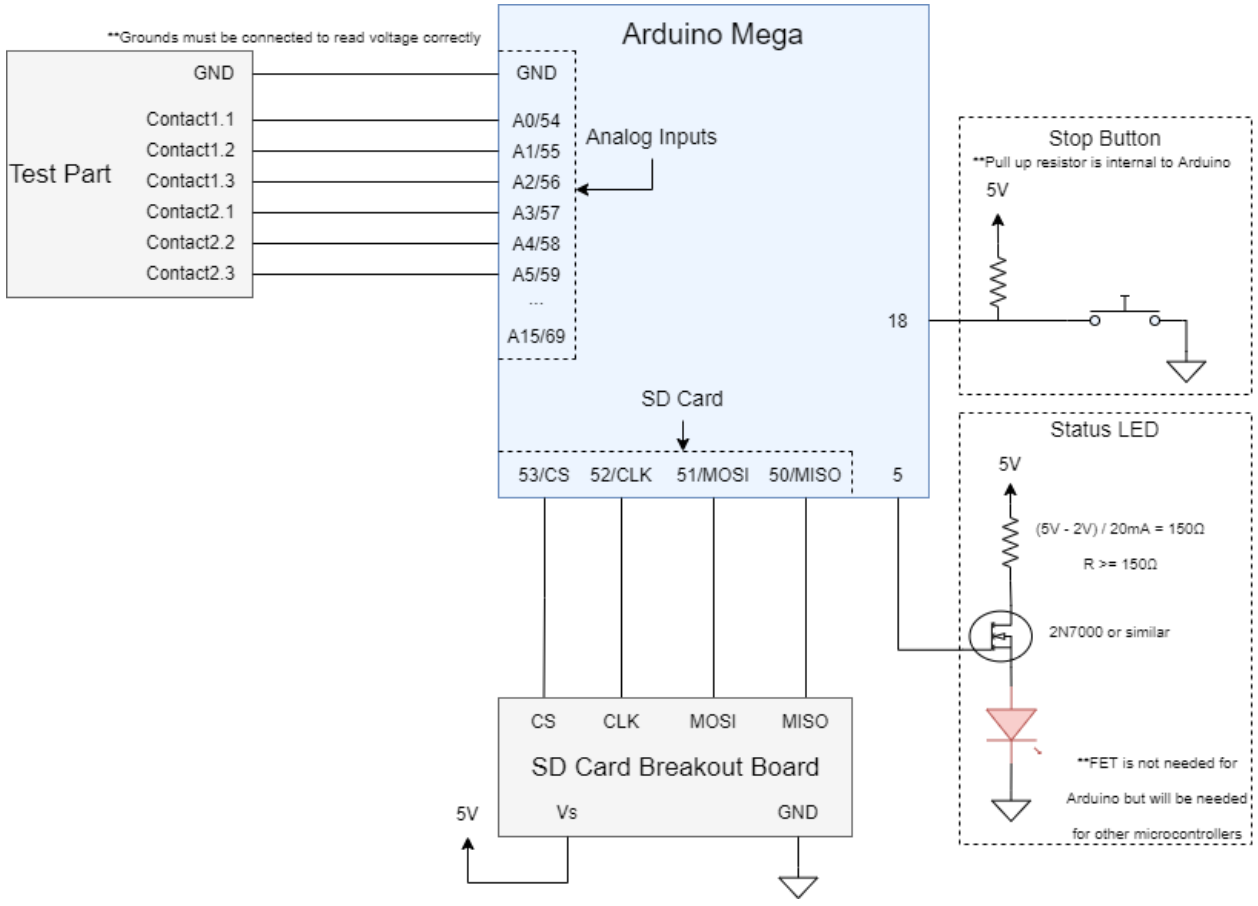
Option	Symbol	Arguments	Description
Help	-h or –help	NONE	Provides a summary of the available options
Input File Name	-i or –input	String representing input file name stem. Ex: -i “TEST”	Defines the input filename stem. If this option is not used the filename defaults to “TEST#.bin”
Output File Name	-o or –output	String representing output filename stem. Ex: -o “output”	Defines the output filename. If this option is not used the filename defaults to “output#.csv”
Group IDs	-g or –group	List of contact groups. Each “;” marks the end of a contact group and each “,” delimits the individual contacts. Ex: -g “1,2,3;4,5,6;7,8,9”	The group argument is required for program execution. A single ID or list of comma separated IDs may be given. Defines which contacts should be decoded and written to the output file.
Process Limit	-p or –pLimit	Positive integer. For a 4 core CPU should not exceed p=4 unless file size is very small. Ex: -p 3	Number of files to be processed in parallel. If p is not given the default is 2.
Timing Analysis	-t or –time	Arg1: check_time - any delta time larger than this will be flagged in the summary output Arg2: press_debounce - number of consecutive “press” states required to debounce Arg3: unpress_debounce - number of consecutive “unpress” states required to debounce Arg4: timeout - if all contacts in a group do not press or unpress within this many rows the action is flagged in the summary file. Ex: -t “10, 5, 5, 30”	Enables timing analysis which produces a summary csv. Timing analysis was designed for pushbuttons. To be considered a good press/unpress, all contacts in a contact group must close/open within the specified timeout argument.

Appendix A:

A-1: Table of Required Software Description and Links

Name	Description	Link
AOMEI	Disk partitioning software. Used to format 64Gb SD cards as FAT32.	https://www.diskpart.com/free-partition-manager.html
Arduino IDE	Used to write and upload code to Arduino boards.	https://www.arduino.cc/en/software
Python 3	Used to decode the binary output data.	https://www.python.org/downloads/release/python-3103/
YXML Library	Arduino library used to parse xml files. Used to read contact configuration in.	https://github.com/JulStrat/LibYxml

A-2: Schematic



Appendix B:

B-1: Configuration File Example

```
<config>
  <contact>
    <group> 10 </group>

    <voltageLow> 0.000 </voltageLow>
    <voltageHigh> 2.100 </voltageHigh>
    <voltageLow> 2.100 </voltageLow>
    <voltageHigh> 2.400 </voltageHigh>
    <voltageLow> 2.400 </voltageLow>
    <voltageHigh> 3.700 </voltageHigh>
    <voltageLow> 3.700 </voltageLow>
    <voltageHigh> 3.900 </voltageHigh>
    <voltageLow> 3.900 </voltageLow>
    <voltageHigh> 5.000 </voltageHigh>

    <pin> 54 </pin>
  </contact>

  <contact>
    <group> 11 </group>

    <voltageLow> 0.000 </voltageLow>
    <voltageHigh> 2.100 </voltageHigh>
    <voltageLow> 2.100 </voltageLow>
    <voltageHigh> 2.400 </voltageHigh>
    <voltageLow> 2.400 </voltageLow>
    <voltageHigh> 3.700 </voltageHigh>
    <voltageLow> 3.700 </voltageLow>
    <voltageHigh> 3.900 </voltageHigh>
    <voltageLow> 3.900 </voltageLow>
    <voltageHigh> 5.000 </voltageHigh>

    <pin> 55 </pin>
  </contact>
</config>
```

B-2: Arduino Main Code

```
/*****************************************************************
 * Project: Electrical Contact Monitoring
 * Author: Zac Lynn
 * Date: 4/26/2022
 * Description: This program reads contact configurations from an xml
 *              file and records voltage and timing data as a
 *              binary output file saved on an SD card.
 *****************************************************************/
#include "SD_Lib.h"
#include "timer_interrupt.h"

// Do not change
#define STOP_PIN 18                                     // Pin to connect stop button to
#define CS_PIN 53                                       // Passed to sd library on initialization to
use this pin as the cs signal
#define STATUS_PIN 5                                    // D5 used for a status LED signaling
if the SD card file opened
#define FLOAT_TO_LONG 10000000                          // Scalar used to convert float
to uint32_t for the purpose of writing to bin file more efficiently
#define INCH_MAX 16                                     // Maximum number of ADC input
channels

/******************** Configurations *******************************/
#define FILE_NAME_TEMPLATE "test"                       // Name stem for output
files. A number will be added to the end of the name to create uniue filenames
#define FILE_LIMIT 75                                   // Maximum number of files that are
allowed to be created
#define WRITE_LIMIT 1040000                             // Determines the maximum
number of rows the output excel file will be. 10485760 is the excel max number of rows
#define V_SCALE 13.13                                   // Voltage scaling factor. If input to
ADC is 5v use 5.0. If a voltage divider is used put the scaling factor here
#define SAMPLE_PERIOD 5                                 // Desired time in ms between
samples
uint32_t BYTE_LIMIT = 1000000 * 150;                    // Max file size in bytes.
FAT32 allows a max of 4gb so should always be less than that.
/*****************************************************************/

// Do not change
```

```
#define packet_size 11                                  // Number of bytes required to save each
data point.
uint8_t file_count = 0;                                 // Current number of files
uint16_t flush_count = 0;                               // Index last written to in the data buffer
int last_flush_count = 0;                               // Index in data buffer where last
complete row finished. Saving and closing a file should always use this count
SD_Lib writer = SD_Lib();                               // Initializes SD_Lib object
SD_Lib::contact contact_list[INCH_MAX];                 // Creates a list of contact
structs defined in SD_Lib

/*********************************************************************
 * Function: setup()
 * Description: This function initialies the STOP_PIN as an input with
 *              a pullup resistor and interrupts enabled, the
 *              STATUS_PIN as an output, and SPI pins. This function
 *              also calls SD_Lib to read in the configuration file.
 *
 * Param: NONE
 * Return: NONE
 *********************************************************************/
void setup() {
  Serial.begin(9600);
  pinMode(STOP_PIN, INPUT_PULLUP);                      // Use internal pullup
resistor for stop button
  pinMode(STATUS_PIN, OUTPUT);
  digitalWrite(STATUS_PIN, HIGH);
  delay(1000);

  writer.SD_read_config(CS_PIN, STATUS_PIN, contact_list);      // Reads the
config.xml file and sets user configurations
  writer.SD_open(create_filename());                    // Creates\Opens the first data
output file
  file_count++;

  writer.SD_allocate_buffer();                          // Allocating buffers is done outside of
constructor so that there is more free memory during the reading of the config file

  Serial.print("----------------- ");                  // Print out the contact configurations for
debugging
  Serial.print(writer.INCH);
```

```
  Serial.println(" configs ----------------");
  for (int i = 0; i < writer.INCH; i++) {
    to_string(contact_list[i]);
  }

  attachInterrupt(digitalPinToInterrupt(STOP_PIN), close_sd, LOW);          // Initialize the
STOP_PIN with interrupts enabled when line is low
  digitalWrite(STATUS_PIN, LOW);                                    // If SD card initializes turn off
status LED
  delay(100);
  init_timer5_interrupt(SAMPLE_PERIOD);                                 // Initializes interrupts on
timer/counter every SAMPLE_PERIOD ms
}

/*****************************************************************
 * Function: loop()
 * Description: This function calls to SD_Lib to save data to the SD
 *              card. This function also checks for the file limit
 *              being exceeded then saves the current file and
 *              starts a new file.
 *
 * Param: NONE
 * Return: NONE
 *****************************************************************/
void loop() {

  if (writer.write_count >= WRITE_LIMIT ||
      writer.byte_count >= BYTE_LIMIT) {                            // Checks if a new file needs to
be created

    if (writer.SD_close(&flush_count, last_flush_count)) {
      Serial.print("File");
      Serial.print(file_count, DEC);
      Serial.println(" size limit reached");

      if (file_count >= FILE_LIMIT) {
        close_sd();                                         // If file count passes file limit stop to avoid
writing more than SD card size
      }
```

```
    writer.SD_open(create_filename());                    // Create and open new data file
    file_count++;
    writer.byte_count = 0;
    writer.write_count = 0;
   }
  }

  if (writer.SD_save_bin(contact_list, &flush_count)) {
   last_flush_count -= 512;
  }
}


/*********************************************************************
 * Function: loop()
 * Description: This function is called by the Timer/Counter
 *              interrupts. Interrupts are disabled, read ADC is
 *              executed, then interrupts are reenabled.
 *
 * Param: NONE
 * Return: NONE
 *********************************************************************/
ISR(TIMER5_COMPA_vect) {
  TIMSK5 &=~ (1 << OCIE5A);                          // Disable timer interrupts during
ADC reading
  read_ADC();
  TIMSK5 |= (1 << OCIE5A);                           // Enable interrupts again
}


/*********************************************************************
 * Function: read_ADC
 * Description: This function iterates through the contact
 *              configurations set by the config file and reads the
 *              voltage channel associated with he contacts. There
 *              is no return value but, the values of the contact
 *              structures in the contact_list array are updated
 *              here.
 *
 * Param: NONE
```

```
 * Return: NONE
 ***********************************************************************/
void read_ADC(void) {
  /*
   *  To begin:
   *  analogRead has output from 0 - 1023 (10-bit ADC)
   *  analogRead / 1023 * 5v = ADC voltage reading scaled to represent 0-5 v
   *
   *  So, the line shown below reads the ADC value and converts to a voltage
   *  between 0-5. Then FLOAT_TO_LONG is multipled by the result so that it
   *  may be stored as integer rather than a float/double. This is done to
   *  imporve SD writing efficiency. When decoding the data the same large
   *  number as FLOAT_TO_LONG is divided from the data to return it to a float.
   *
   *  analogRead(contact_list[index].pin) / 1023.0 * V_SCALE * FLOAT_TO_LONG
   *
   */
  uint8_t index;

  if (flush_count > 4220 - packet_size * writer.INCH) {                  // If the buffer size will be
exceeded by a write then return
    return;
  }

  for (index = 0; index < writer.INCH; index++) {
    contact_list[index].voltage = analogRead(contact_list[index].pin) /
                                  1023.0 * V_SCALE * FLOAT_TO_LONG;        // Read voltage and store
in the contact struct

    contact_list[index].timestamp = millis();                              // Read and store timestamp in ms

    contact_list[index].state = 0;                                         // Set state to 0. if voltage is inside a
valid zone state will be updated
    for (int zone = 0; zone < contact_list[index].voltage_zones; zone++) {  // Check which if any
voltage range the contact is in. updates the state according to voltage zone
      if (contact_list[index].voltage <= contact_list[index].voltage_range[zone][1]) {
        if (contact_list[index].voltage >= contact_list[index].voltage_range[zone][0]) {
          contact_list[index].state = zone + 1;
        }
      }
```

```
    }

    // saving data to buffer
    writer.buffer[flush_count++] = 0xEE;                    // Header to mark the begining of
each entry is 0xEE

    writer.buffer[flush_count++] = (contact_list[index].timestamp & 0xFF000000) >> 24;
    writer.buffer[flush_count++] = (contact_list[index].timestamp & 0x00FF0000) >> 16;
    writer.buffer[flush_count++] = (contact_list[index].timestamp & 0x0000FF00) >> 8;
    writer.buffer[flush_count++] = (contact_list[index].timestamp & 0x000000FF);

    writer.buffer[flush_count++] = (byte)contact_list[index].group;

    writer.buffer[flush_count++] = (contact_list[index].voltage & 0xFF000000) >> 24;
    writer.buffer[flush_count++] = (contact_list[index].voltage & 0x00FF0000) >> 16;
    writer.buffer[flush_count++] = (contact_list[index].voltage & 0x0000FF00) >> 8;
    writer.buffer[flush_count++] = (contact_list[index].voltage & 0x000000FF);

    writer.buffer[flush_count++] = (byte)contact_list[index].state;

   }
   writer.write_count += 1;                                 // Limits file length by the number of
rows it will take up in a n excel sheet
   last_flush_count = flush_count;
}


/*****************************************************************
 * Function: close_sd
 * Description: This function saves and closes the current SD file,
 *              turns the status LED back on and enters an infinite
 *              loop. This function ends data collection and makes
 *              it safe to remove power from the Arduino.
 * Param: NONE
 * Return: NONE
 *****************************************************************/
void close_sd(void) {
  do {
    if (writer.SD_save_bin(contact_list, &flush_count)) {
      last_flush_count -= 512;
```

```
    }
  }while(!writer.SD_close(&flush_count, last_flush_count));
  TIMSK5 &=~ (1 << OCIE5A);

  digitalWrite(STATUS_PIN, HIGH);
  Serial.println("END");
  while(1) delay(1000);
}


/*****************************************************************
 * Function: create_filename
 * Description: This function uses the FILE_NAME_TEMPLATE
 *              configuration value and the file count variable to
 *              create a unique file name for consecutive files.
 *
 * Param: NONE
 * Return: String (ret) - a unique filename
 *****************************************************************/
String create_filename(void) {
  String ret = FILE_NAME_TEMPLATE;
  String ext = ".bin";
  return ret + (file_count+1) + ext;
}


/*****************************************************************
 * Function: to_string
 * Description: This function prints a formatted string of values
 *              held in the supplied contact struct.
 *
 * Param: contact (obj) - a contact struct to print
 * Return: NONE
 *****************************************************************/
void to_string(SD_Lib::contact obj) {
  int index = 0;

  Serial.print("Group: ");
  Serial.println(obj.group);
  for (int i = 0; i < obj.voltage_zones; i++) {
```

```
    Serial.print("VLOW_");
    Serial.print(i);
    Serial.print(": ");
    Serial.println(obj.voltage_range[i][0]);
    Serial.print("VHIGH_");
    Serial.print(i);
    Serial.print(": ");
    Serial.println(obj.voltage_range[i][1]);

  }
  Serial.print("pin: ");
  Serial.println(obj.pin);
  Serial.print("Zones: ");
  Serial.println(obj.voltage_zones);
  Serial.println("----------------------------------------");
}
```

B-3: Arduino SD Lib Source File

```
/*****************************************************************
 * Project: Electrical Contact Monitoring
 * Author: Zac Lynn
 * Date: 4/26/2022
 * Description: This code uses the yxml and SD libraries to read and
 *              save data on an SD card.
 *****************************************************************/
#include "SD_Lib.h"

/*****************************************************************
 * Function: SD_Lib
 * Description: This function initializes the member variables used
 *              for configuration.
 *
 * Param: NONE
 * Return: NONE
 *****************************************************************/
SD_Lib::SD_Lib() {
  this->INCH = 0;
  this->write_count = 0;
  this->byte_count = 0;
}


/*****************************************************************
 * Function: SD_allocate_buffer
 * Description: This function allocates the main data buffer. Done
 *              seperately from constructor so that more free memory
 *              exists during setup and buffer size can be maximized.
 *
 * Param: NONE
 * Return: NONE
 *****************************************************************/
void SD_Lib::SD_allocate_buffer() {
  buffer = (byte*)malloc(4224);
}


/*****************************************************************
```

```
 * Function: SD_open
 * Description: This function tries to open a file with the given name
 *              on an SD card. Enters an infinite loop if fails to
 *              open file.
 *
 * Param: String (filename) - name of file to open
 * Return: NONE
 ***********************************************************************/
void SD_Lib::SD_open(String filename) {
  data_out = SD.open(filename, O_WRITE | O_TRUNC | O_CREAT);
  if (data_out == 0) {
    Serial.println("Failed to open data output file");
    while(1);                                          // If the file failed to open stay in loop forever
(status LED on)
  }
}


/***********************************************************************
 * Function: SD_close
 * Description: This closes the open SD file. The file should always
 *              be closed before removing power from the Arduino in
 *              order to prevent data corruption.
 *
 * Param: NONE
 * Return: NONE
 ***********************************************************************/
bool SD_Lib::SD_close(uint16_t *flush_count, int last_flush_count) {

  if (*flush_count < last_flush_count ||                       // If the flush_count <
ladt_flush_count then a save was just made and read_ADC needs to be called again first
     last_flush_count > 512 ||                                 // If last_flush_count > 512 there is a
memory error
     last_flush_count <= 0) return false;                      // If last flush_count <= 0 then a
save was just made and read_ADC needs to be called again first

  byte *temp = (byte*)malloc(last_flush_count);                // Moved data to save to a
temporary buffer
  memcpy(temp, buffer, last_flush_count);
```

```
  TIMSK5 &=~ (1 << OCIE5A);                                    // Disable interrupts so that
flush_counts cannot chnage here
  *flush_count -= last_flush_count;
  memmove(buffer, buffer + last_flush_count, *flush_count);          // Shift main buffer so
that the first index is the next data point needed to be saved
  TIMSK5 |= (1 << OCIE5A);                                   // re-enable interrupts

  data_out.write(temp, last_flush_count);                      // Write buffer to SD card
  data_out.flush();
  data_out.close();
  byte_count += last_flush_count;
  free(temp);                                          // Release temporary buffer

  return true;
}


/*****************************************************************
 * Function: SD_save_bin
 * Description: This function checks if the main buffer has at least
 *              512 bytes in it, and writes a 512 byte block to
 *              the SD card if it can.
 *
 * Param: contact* (contact_list) - list of contact structs
 *       uint16_t* (flush_count) - current index in main buffer
 * Return: NONE
 *****************************************************************/
bool SD_Lib::SD_save_bin(contact *contact_list, uint16_t *flush_count) {
  uint8_t index;

  if (*flush_count >= 512) {
    byte *temp = (byte*)malloc(512);                           // Copy data that needs to be saved
to temporary buffer
    memcpy(temp, buffer, 512);

    TIMSK5 &=~ (1 << OCIE5A);                                  // Disable interrupts so
flush_count does not change
    *flush_count -= 512;
    memmove(buffer, buffer + 512, *flush_count);                    // Shift main buffer left by
512 bytes
```

```
    TIMSK5 |= (1 << OCIE5A);

    data_out.write(temp, 512);                                  // Writes the temporary buffer to the
SD card
    free(temp);
    byte_count += 512;
    return true;
   }
  return false;
}


/*****************************************************************
 * Function: SD_read_config
 * Description: This function reads the config file and copies the
 *              configurations to the contact_list array.
 *
 * Param: int (cs) - Pin to use as chip select for SPI SD card
 * Param: uint8_t (status_pin) - LED pin number
 * Param: contact* (contact_list) - Array of contact structs
 * Return: NONE
 *****************************************************************/
void SD_Lib::SD_read_config(int cs, uint8_t status_pin, contact *contact_list) {
  // xml parsing variables
  int c;
  yxml_ret_t r;
  yxml_t x[1];
  char stack[128];
  int len;

  bool flag = 0;
  File conf;
  String filename = "config.xml";                               // Config file name is always
expected to be "config.xml"

  SD.begin(cs);                                                 // Start SD card library with CS_pin

  if (SD.exists(filename)) {                                    // Check that the file exists before trying
to open
    conf = SD.open(filename, O_READ);                           // Opens in override write mode
```

```
  if (conf == 0) {
    Serial.println("Failed to open config file");
    while(1) {                                          // If the file failed to open stay in loop forever
      delay(500);
      flag ^= 1;
      digitalWrite(status_pin, flag);
    }
  }
 }
 else {                                                 // If the file does not exists stay in loop forever
  Serial.println("Error: config file does not exist");
  while(1) {                                            // If the file failed to open stay in loop forever
    delay(500);
    flag ^= 1;
    digitalWrite(status_pin, flag);
  }
 }

 // len of xml file
 len = conf.size();

 // init xml parsing object
 yxml_init(x, stack, sizeof(stack));

 for (int i = 0; i < len; i++) {
   c = conf.read();                                     // Reads a single character from xml file
   r = yxml_parse(x, c);                                // Adds the character to the yxml
interpreter object
   decode_config(x, r, contact_list);                   // Uses flags in yxml class to load
contact configurations
 }
 conf.close();

 if (INCH < 1 || INCH > 16) {                           // Check that INCH has been set to a
valid number
  Serial.print("Error: ");
  Serial.print(INCH);
  Serial.println(" input channels found in config file");
  while(1) {
    delay(500);
```

```
      flag ^= 1;
      digitalWrite(status_pin, flag);
    }
  }
}


/*****************************************************************
 * Function: decode_config
 * Description: This function parses the xml file using the yxml
 *              library.
 *
 * Param: yxml_t* (x) - main yxml struct. See yxml.h for info
 * Param: yxml_ret_t (r) - yxml return token. See yxml.h for info
 * Param: contact* (contact_list) - list of contact structs
 * Return: NONE
 *****************************************************************/
void SD_Lib::decode_config(yxml_t *x, yxml_ret_t r, contact *contact_list) {
  static char buf[10];
  static int index;
  static uint8_t cpy_flag = 1;                          // Stores whether or not the current
contact entry has be saved already
  static char last[16];
  static int contact_count = 0;
  static contact temp = {0, 0, 0, 0, 0, 0, 0};
  double sum;

  /*    Contact entries in the config file should be ordered the same as the if statements here.
   * It is possible that the order can be changed and the config will still be loaded correctly
   * but counts increment on the voltageHigh and pin entries so the order should match.
   */
  if (r == YXML_ELEMSTART) {                                  // Is true at the start of each
element
    index = 0;
    memcpy(last, x->elem, 16);
  }

  if (r == YXML_ELEMEND) {                                    //  Is true at the end of each
element
    sum = char_to_double(buf, index);
```

```
    if (strcmp(last, "group") == 0) {
      cpy_flag = 1;
      temp.group = (int) sum;
    }
    else if (strcmp(last, "voltageLow") == 0) {
      temp.voltage_range[temp.voltage_zones][0] = sum * FLOAT_TO_LONG;
    }
    else if (strcmp(last, "voltageHigh") == 0) {
      temp.voltage_range[temp.voltage_zones++][1] = sum * FLOAT_TO_LONG;    //
Voltage_zones increments each time here so that multiple voltage ranges can be supplied
    }
    else if (strcmp(last, "pin") == 0) {                          // PIN mUst be the last entry for a
contact
      if (cpy_flag == 1) {                                        // Ensures that each structure is copied to
the contact list only once
        cpy_flag = 0;
        temp.pin = (int) sum;
        contact_list[contact_count++] = temp;
        temp = {0, 0, 0, 0, 0, 0, 0};
        INCH++;                                                   // Increment at the end of each contact
      }
    }
  }

  if (r == YXML_CONTENT) {
    if (x->data[0] == 32 || x->data[0] == 10) {                   // If the char is a space or newline
ignore it
    }
    else {                                                        // If the char is not a space or newline save it to
the buffer
      buf[index] = x->data[0];
      index++;
    }
  }
}


/*****************************************************************
 * Function: char_to_double
```

```
 * Description: This function converts a string to a double. Used by
 *              decode_config to read in voltage range variables.
 *
 * Param: char* (buf) - string to convert to double
 * Param: int (index) - length of char buffer
 * Return: double - the double equivelant of the provided str
 ***************************************************************/
double SD_Lib::char_to_double(char *buf, int index) {
  bool fractional_flag = false;
  int fractional_weight = 0;
  int max_fractional_weight;
  int integer_weight = 0;
  double sum = 0;

  for (int i = 0; i < index; i++) {
    if (buf[i] == '.') {                                  // If a decimal point is reached start counting
fractional weights instead of integer weights
      fractional_flag = 1;
    }
    else {
      // keeps track of the weighting of each number
      if (fractional_flag) {
        fractional_weight++;
      }
      else {
        integer_weight++;
      }
    }
  }

  max_fractional_weight = fractional_weight;
  fractional_flag = 0;

  for (int i = 0; i < index; i++) {
    if (buf[i] == '.') {
      fractional_flag = 1;
    }
    else {
      if (fractional_flag) {
        sum += ((int)buf[i] - 48) * pow(10, -1 * (max_fractional_weight - --fractional_weight));
```

```
      }
      else {
        sum += ((int)buf[i] - 48) * pow(10,(--integer_weight));
      }
    }
  }
  return sum;
}
```

B-4: Arduino SD Lib Header File

```
/*****************************************************************
 * Project: Electrical Contact Monitoring
 * Author: Zac Lynn
 * Date: 4/26/2022
 * Description: This code uses the yxml and SD libraries to read and
 *              save data on an SD card.
 *****************************************************************/
#ifndef MY_LIBRARY_H
#define MY_LIBRARY_H

#include <Arduino.h>
#include <SPI.h>
#include <SD.h>
#include "yxml.h"


#define FLOAT_TO_LONG 10000000                                  // Scalar used to convert float to uint32_t for the purpose of writing to bin file more efficiently
/***************** Configuration Variable *****************/
#define ZONES 10                                                // Sets the maximum number of voltage zones allowed in a contact config. Example sliding contacts may have one contact with 5 distinct zones
/**********************************************************/

class SD_Lib {
  private:
    File data_out;                                              // File pointer to output binary file

  public:
    byte *buffer;
    uint32_t write_count;                                       // If MIN_VOLTAGE_CHANGE <= 4.8mv this represents number of rows in an excel file. Used to limit files based on row count
    uint32_t byte_count;                                        // Counts how many bytes have been written to the current file. Used to limit files based on size
    uint8_t INCH;                                               // Number of ADC input channels

    // Contact structure to hold contact configuration and data to save
    struct contact{
      // Variables to write to sd card
```

```
    uint8_t group;                                  // Unique group ID for each contact
    uint32_t timestamp;
    uint32_t voltage;
    uint8_t state;                                  // State is pressed(1) or unpressed(0) based
on voltage_range[2]

    // Config Variables
    uint32_t voltage_range[ZONES][2];
    uint8_t pin;

    // other
    uint8_t voltage_zones;                          // Stores how many voltage zones, and
thus states, are used
  };

  /*****************************************************************
   * Function: SD_Lib
   * Description: This function initializes the member variables used
   *              for configuration.
   *
   * Param: NONE
   * Return: NONE
   *****************************************************************/
  SD_Lib();

  /*****************************************************************
   * Function: SD_allocate_buffer
   * Description: This function allocates the main data buffer. Done
   *              seperately from constructor so that more free memory
   *              exists during setup and buffer size can be maximized.
   *
   * Param: NONE
   * Return: NONE
   *****************************************************************/
  void SD_allocate_buffer();

  /*****************************************************************
   * Function: SD_open
   * Description: This function tries to open a file with the given name
   *              on an SD card. Enters an infinite loop if fails to
```

```
 *              open file.
 *
 * Param: String (filename) - name of file to open
 * Return: NONE
 ***********************************************************************/
void SD_open(String filename);

/***********************************************************************
 * Function: SD_close
 * Description: This closes the open SD file. The file should always
 *              be closed before removing power from the Arduino in
 *              order to prevent data corruption.
 *
 * Param: NONE
 * Return: NONE
 ***********************************************************************/
bool SD_close(uint16_t *flush_count, int last_flush_count);

/***********************************************************************
 * Function: SD_save_bin
 * Description: This function checks if the main buffer has at least
 *               512 bytes in it, and writes a 512 byte block to
 *               the SD card if it can.
 *
 * Param: contact* (contact_list) - list of contact structs
 *       uint16_t* (flush_count) - current index in main buffer
 * Return: NONE
 ***********************************************************************/
bool SD_save_bin(contact *contact_list, uint16_t *flush_count);

/***********************************************************************
 * Function: SD_read_config
 * Description: This function reads the config file and copies the
 *              configurations to the contact_list array.
 *
 * Param: int (cs) - Pin to use as chip select for SPI SD card
 * Param: uint8_t (status_pin) - LED pin number
 * Param: contact* (contact_list) - Array of contact structs
 * Return: NONE
 ***********************************************************************/
```

```
    void SD_read_config(int cs, uint8_t status_pin, contact *contact_list);

    /*********************************************************************
     * Function: decode_config
     * Description: This function parses the xml file using the yxml
     *              library.
     *
     * Param: yxml_t* (x) - main yxml struct. See yxml.h for info
     * Param: yxml_ret_t (r) - yxml return token. See yxml.h for info
     * Param: contact* (contact_list) - list of contact structs
     * Return: NONE
     *********************************************************************/
    void decode_config(yxml_t *x, yxml_ret_t r, contact *contact_list);

    /*********************************************************************
     * Function: char_to_double
     * Description: This function converts a string to a double. Used by
     *              decode_config to read in voltage range variables.
     *
     * Param: char* (buf) - string to convert to double
     * Param: int (index) - length of char buffer
     * Return: double - the double equivelant of the provided str
     *********************************************************************/
    double char_to_double(char *buf, int index);
};
#endif
```

B-5: Arduino Timer Interrupt Source File

```
/*********************************************************************
 * Project: Electrical Contact Monitoring
 * Author: Zac Lynn
 * Date: 3/24/2022
 * Description: This code initializes interrupts on timer 5 using
 *              compareA. No pin output is generated.
 *********************************************************************/
#include "timer_interrupt.h"

/*********************************************************************
 * Function: init_timer5_interrupt
 * Description: This function initializes timer 5 interrupts given a
 *              period in ms. Interrupts to not genereate any pin
 *              output.
 *
 * Param: uint16_t (period) - period for interrupts in ms
 * Return: NONE
 *********************************************************************/
void init_timer5_interrupt(uint16_t period) {
  TCCR5A = 0x00;                                          // Clear TCCR1A register
  TCCR5B = 0x00;                                          // Clear TCCR1B
  TCNT5 = 0x00;                                          // Initialize count register to 0


  // Set compareA register with the interrupt period in ms
  OCR5A = (int)(period * 15.625);

  // Turn on CTC mode
  TCCR5B |= (1 << WGM12);

  // Set clock divider to /1024. 16MHz /1024 = 15.625KHz
  TCCR5B |= (1 << CS12) | (1 << CS10);

  // Enable timer compareA interrupt
  TIMSK5 |= (1 << OCIE5A);
}
```

B-6: Arduino Timer Interrupt Header File

```
/**********************************************************************
 * Project: Electrical Contact Monitoring
 * Author: Zac Lynn
 * Date: 4/26/2022
 * Description: This code initializes interrupts on timer 5 using
 *              compareA. No pin output is generated.
 **********************************************************************/

#include <Arduino.h>


/**********************************************************************
 * Function: init_timer5_interrupt
 * Description: This function initializes timer 5 interrupts given a
 *              period in ms. Interrupts to not genereate any pin
 *              output.
 *
 * Param: uint16_t (period) - period for interrupts in ms
 * Return: NONE
 **********************************************************************/
void init_timer5_interrupt(uint16_t period);
```

B-5: Python Data Output Decoding

```
#-----------------------------------------------------------------------
# Project: Electrical Contact Monitoring
# Author: Zac Lynn
# Date: 4/26/2022
# Description: This program decodes the binary output file from the
#              Arduino based data acquisition system. This program
#              should be executed from a terminal window so that
#              options may be used (-g is required, -h for help).
#-----------------------------------------------------------------------
from hashlib import new
import numpy as np
import time
import multiprocessing as mp
import csv
import getopt
import sys

FLOAT_TO_LONG = 10000000                                    # Comes from the arduino
code. doubles were stored as uint32_t * 10000000 instead of a double
IN_FILENAME = "TEST"                                        # Name of input file stem to
read from
OUT_FILENAME = "output"                                     # Name stem to give to the
output csv file
GROUPS = None                                               # List of group IDs to decode data
for

refined_data = []                                           # Stores the data as a 1D list of
dictionaries
data = []                                                   # Stores the data seperated by contact ID
in_file_count = 1


#-----------------------------------------------------------------------
# Function: read_file()
# Description: This function reads the entire contents of the
#   binary data file and stores it in a numpy array.
#
# param: void
```

```
# return: void
#--------------------------------------------------------------------
def read_file(filename):
    global raw_data, in_file_count
    raw_data = None                                    # Stores the raw binary data from the
input file
    dtype = np.dtype('B')
    try:
        with open(filename, "rb") as fp:
            raw_data = np.fromfile(fp, dtype)
    except Exception as e:
        print("**Error reading file:\t" + str(e))
        return

    fp.close()


#--------------------------------------------------------------------
# Function: count_files()
# Description: This function checks how many input files exist with the
#                given stem name. Only checks for files with numbers
#                between 0 and 255.
#
# param: void
# return: void
#--------------------------------------------------------------------
def count_files():
    global in_file_count, in_file_list
    in_file_count = 0
    in_file_list = []
    print("\nCounting files...")

    for i in range(255):
        try:
            filename = IN_FILENAME + str(i+1) + ".bin"
            with open(filename, "rb") as fp:
                pass
        except Exception as e:
            continue
```

```
        fp.close()
        in_file_list.append(i+1)
    in_file_count = len(in_file_list)
    print("File count: " + str(in_file_count) + "\n")


#---------------------------------------------------------------------
# Function: convert_data()
# Description: This function converts the raw binary data back
#   into its original type. 0xEE and 0xCE are delimiters
#
# param: void
# return: void
#---------------------------------------------------------------------
def convert_data(filename):
    global raw_data
    global refined_data
    refined_data = []
    i = 0
    temp_timestamp = 0
    temp_group = 0
    temp_voltage = 0
    temp_state = 0

    while (i < len(raw_data)):                                  # A while loop is used instead of a
for loop so that i can be incremented inside of the loop
        if (raw_data[i] == 0xEE):                               # If a new observation is starting
save the last one and get ready for next
            if (i > 0):                                         # Don't try to save the temp data on the first
iteration. Files always start with "EE" so skip first
                temp = {"timestamp": temp_timestamp,
                        "group": temp_group,
                        "voltage": temp_voltage,
                        "state": temp_state}
                refined_data.append(temp)

            i += 1
        else:                                                   # If the data is not a delimiter reformat it
back to its original type
            try:
```

```
            temp_timestamp = (raw_data[i] << 24 |                    # Timestamp was saved as
uint32_t
                              raw_data[i + 1] << 16 |
                              raw_data[i + 2] << 8 |
                              raw_data[i + 3])

            temp_group = raw_data[i + 4]                             # Group is a uint8_t

            temp_voltage = (raw_data[i + 5] << 24 |                  # Voltage was saved as
(uint32_t)(double * FLOAT_TO_LONG)
                            raw_data[i + 6] << 16 |
                            raw_data[i + 7] << 8 |
                            raw_data[i + 8]) / FLOAT_TO_LONG

            temp_state = raw_data[i + 9]                             # State is a uint8_t
            i += 10
          except IndexError as e:
            print(str(filename) + ": may be missing data in last row")      # If this happens
tinmestamps may not be alined in the next file
            break
    try:                                                    # Read the last entry if it exists
        temp = {"timestamp": temp_timestamp,
                "group": temp_group,
                "voltage": temp_voltage,
                "state": temp_state}
        refined_data.append(temp)
    except Exception as e:
        print("At end of decoding: " + str(e))


#----------------------------------------------------------------------
# Function: separate_data()
# Description: This function uses the refined_data array and
#              separates the data into a 2d array by the group
#              identifiers
#
# param: void
# return: void
#----------------------------------------------------------------------
def separate_data():
```

```
    global data
    data = []
    for group in GROUPS:
        for contact in group:
            temp = []
            for datum in refined_data:
                if (datum['group'] == contact):
                    temp.append([datum['group'], datum['timestamp'],
                            datum['voltage'], datum['state']])

            data.append(temp)


#---------------------------------------------------------------------
# Function: write_to_csv()
# Description: This function writes the contents of the binary data
#                 file into a csv file organized by group identifiers.
#
# param: void
# return: void
#---------------------------------------------------------------------
def write_to_csv(filename):
    index = 0
    temp = []
    shift = 0

    max_len = len(data[0])
    for group in GROUPS:                                    # Find the length of the longest data
set
        for contact in range(len(group)):
            if (len(data[contact + shift]) > max_len):
                max_len = len(data[contact + shift])
        shift += len(group)


    with open(filename, 'w', newline='\n') as csvfile:
        csv_writer = csv.writer(csvfile, delimiter=',')

        for group in range(shift):                          # Sets up the headers at the top of
excel file
```

```
            temp.append("Group")
            temp.append("Time(ms)")
            temp.append("Voltage(V)")
            temp.append("State")
            temp.append("")
        csv_writer.writerow(temp)

        while (index < max_len):
            temp = []                                   # Temp stores each row of the csv file
before it is written

            for group in range(shift):                      # Assemble next row to write to csv
                if (index < len(data[group])):
                    temp.append(data[group][index][0])
                    temp.append(data[group][index][1])
                    temp.append(data[group][index][2])
                    temp.append(data[group][index][3])
                    temp.append('')

            csv_writer.writerow(temp)

            index += 1


#---------------------------------------------------------------------
# Function: save_timing_summary()
# Description: This function is a helper to timing_analysis() that,
#                 creates an additional CSV that gives the results of
#                 the analysis for each .bin file.
#
# param: All parameters are defined in timing_analysis function
# return: void
#---------------------------------------------------------------------
def save_timing_summary(filename, undefined_count_total, low_out_count_total,
        pressed_count_total, transition_count_total, unpressed_count_total,
        high_out_count_total, good_press, bad_press, good_unpress, bad_unpress,
        bad_press_locations, bad_unpress_locations, check_time_locations):

    temp = []
```

```
    newline = []                                    # Stores a csv file row that uses "-" and "|"
to separate the data visually

    # Generate the CSV newline based on how many contacts are being analyzed
    for group in range(len(GROUPS)):
        for contact in range(len(GROUPS[group]) + 1):
            newline.append("---------------------------")
        newline.append("|")

    # Creates a file using same naming convention as the output csv files but with "_summary"
added to the end
    with open(filename[0:-4] + "_summary.csv" , 'w', newline='\n') as csvfile:
        csv_writer = csv.writer(csvfile, delimiter=',')

        # Write contact group headers
        for group in range(len(GROUPS)):
            temp.append("Group: " + str(group))
            for contact in range(len(GROUPS[group])):
                temp.append("")
            temp.append("|")
        csv_writer.writerow(temp)
        temp = []

        # Write contact ID headers
        for group in range(len(GROUPS)):
            temp.append("Contact: ")
            for contact in range(len(GROUPS[group])):
                temp.append(GROUPS[group][contact])
            temp.append("|")

        csv_writer.writerow(temp)
        csv_writer.writerow(newline)                        # Write newline in csv
        temp = []
        shift = 0

        # Write zone 0 count for all groups
        for group in range(len(GROUPS)):
            temp.append("Zone: 0")
            for contact in range(len(GROUPS[group])):
                temp.append(undefined_count_total[contact + shift])
```

```
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
temp = []
shift = 0

# Write zone 1 count for all groups
for group in range(len(GROUPS)):
    temp.append("Zone: 1")
    for contact in range(len(GROUPS[group])):
        temp.append(low_out_count_total[contact + shift])
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
temp = []
shift = 0

# Write zone 2 count for all groups
for group in range(len(GROUPS)):
    temp.append("Zone: 2")
    for contact in range(len(GROUPS[group])):
        temp.append(pressed_count_total[contact + shift])
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
temp = []
shift = 0

# Write zone 3 count for all groups
for group in range(len(GROUPS)):
    temp.append("Zone: 3")
    for contact in range(len(GROUPS[group])):
        temp.append(transition_count_total[contact + shift])
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
```

```
temp = []
shift = 0

# Write zone 4 count for all groups
for group in range(len(GROUPS)):
    temp.append("Zone: 4")
    for contact in range(len(GROUPS[group])):
        temp.append(unpressed_count_total[contact + shift])
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
temp = []
shift = 0

# Write zone 5 count for all groups
for group in range(len(GROUPS)):
    temp.append("Zone: 5")
    for contact in range(len(GROUPS[group])):
        temp.append(high_out_count_total[contact + shift])
    temp.append("|")
    shift += len(GROUPS[group])

csv_writer.writerow(temp)
csv_writer.writerow(newline)                    # Write newline in csv
temp = []

# Write how many valid presses there were for each group
for group in range(len(GROUPS)):
    temp.append("Good press: ")
    temp.append(good_press[group])

    for contact in range(len(GROUPS[group]) - 1):
        temp.append("")
    temp.append("|")

csv_writer.writerow(temp)
temp = []

# Write how many valid unpresses there were for each group
```

```
for group in range(len(GROUPS)):
    temp.append("Good unpress: ")
    temp.append(good_unpress[group])

    for contact in range(len(GROUPS[group]) - 1):
        temp.append("")
    temp.append("|")

csv_writer.writerow(temp)
temp = []

# Write how many bad presses
for group in range(len(GROUPS)):
    temp.append("Bad press: ")
    temp.append(bad_press[group])

    for contact in range(len(GROUPS[group]) - 1):
        temp.append("")
    temp.append("|")

csv_writer.writerow(temp)
temp = []

# Write how many bad unpresses
for group in range(len(GROUPS)):
    temp.append("Bad unpress: ")
    temp.append(bad_unpress[group])

    for contact in range(len(GROUPS[group]) - 1):
        temp.append("")
    temp.append("|")

csv_writer.writerow(temp)
csv_writer.writerow(newline)                    # Write newline in csv
temp = []

# Write header and first bad press/unpress locations
for group in range(len(GROUPS)):
    temp.append("Bad press rows:")
    try:
```

```
            temp.append(bad_press_locations[group][0])
        except:
            temp.append("")
        temp.append("Bad unpress rows:")
        try:
            temp.append(bad_unpress_locations[group][0])
        except:
            temp.append("")

        for contact in range(len(GROUPS[group]) - 3):
            temp.append("")
        temp.append("|")

    csv_writer.writerow(temp)
    temp = []

    maximum_prints = max(bad_press + bad_unpress)                    # Finds the max number
of bad presses or unpresses so that every check location is printed

    # Write the rest of the bad press/unpress locations
    for i in range(1, maximum_prints):
        for group in range(len(GROUPS)):
            temp.append("")
            try:
                temp.append(bad_press_locations[group][i])
            except:
                temp.append("")
            temp.append("")
            try:
                temp.append(bad_unpress_locations[group][i])
            except:
                temp.append("")

            for contact in range(len(GROUPS[group]) - 3):
                temp.append("")
            temp.append("|")

        csv_writer.writerow(temp)
        temp = []
```

```
csv_writer.writerow(newline)
shift = 0
temp = []

# Find max number of check times to know how long to iterate for
maximum = 1
for i in range(len(check_time_locations)):
    if (len(check_time_locations[i]) > maximum):
        maximum = len(check_time_locations[i])

# Write bad delta time locations
for i in range(maximum):
    shift = 0
    for group in range(len(GROUPS)):
        if (i == 0):
            temp.append("Check time rows:")
            try:
                temp.append(check_time_locations[shift][i]["row"])
            except Exception as e:
                temp.append("")
            temp.append("Delta(ms):")
            try:
                temp.append(check_time_locations[shift][i]["delta"])
            except Exception as e:
                temp.append("")
        else:
            temp.append("")
            try:
                temp.append(check_time_locations[shift - 1][i]["row"])
            except Exception as e:
                temp.append("")
            temp.append("")
            try:
                temp.append(check_time_locations[shift - 1][i]["delta"])
            except Exception as e:
                temp.append("")

        for contact in range(len(GROUPS[group]) - 3):
            temp.append("")
        temp.append("|")
```

```
            shift += len(GROUPS[group])

        csv_writer.writerow(temp)
        temp = []
    csv_writer.writerow(newline)


#---------------------------------------------------------------------
# Function: timing_analysis()
# Description: This function analyzes the input data for good and bad
#              presses using state 2 as pressed and state 4 as
#              unpressed. Good and bad presses are defined by the
#              user by setting the press/unpress_debounce_limit and
#              the timeout_limit. This function also reports how
#              many sample periods were greater than 10ms.
#
# param:(str) out_filename     - name of output csv file. used for print
#                                statements that are traceable to a file
#     (int) check_time       - any delta time(ms) greater than check
#                                time is reported in the summary file
#     (int) press_debounce   - number of consecutive measurements
#                                needed to debounce press state
#     (int) unpress_debounce - number of consecutive measurements
#                                needed to debounce unpress state
#     (int) timeout          - If all of the contacts in a group do
#                                not enter the press or unpress state
#                                within timeout rows it is marked as bad
# return: void
#---------------------------------------------------------------------
def timing_analysis(out_filename, check_time=7, press_debounce=5, unpress_debounce=5,
timeout=30):
    # Button States
    undefined = 0
    low_out = 1
    pressed = 2
    transition = 3
    unpressed = 4
    high_out = 5

    # debounce conditions
```

```
    # press_debounce_limit = 5                                  # How many consecutive samples
required to debounce the "pressed" state
    # unpress_debounce_limit = 5                                 # How many consecutive
samples required to debounce the "unpressed" state
    # timeout_limit = 30                                        # The first and last contact must be
debounced in this period or else invalid press. ~5ms sample period * 30 = 150ms

    maximum = len(max(data))

    total = 0
    for group in GROUPS:                                        # Iterates through the contact groups
        total += len(group)

    undefined_count_cur = [0] * total                           # Stores how many consecutive
iterations a contcat has been in state 0
    low_out_count_cur = [0] * total                             # Stores how many consecutive
iterations a contcat has been in state 1
    pressed_count_cur = [0] * total                             # Stores how many consecutive
iterations a contcat has been in state 2
    transition_count_cur = [0] * total                          # Stores how many consecutive
iterations a contcat has been in state 3
    unpressed_count_cur = [0] * total                           # Stores how many consecutive
iterations a contcat has been in state 4
    high_out_count_cur = [0] * total                            # Stores how many consecutive
iterations a contcat has been in state 5

    undefined_count_total = [0] * total                         # Stores how many total iterations
a contcat was in state 0
    low_out_count_total = [0] * total                           # Stores how many total iterations
a contcat was in state 1
    pressed_count_total = [0] * total                           # Stores how many total iterations
a contcat was in state 2
    transition_count_total = [0] * total                        # Stores how many total iterations
a contcat was in state 3
    unpressed_count_total = [0] * total                         # Stores how many total iterations
a contcat was in state 4
    high_out_count_total = [0] * total                          # Stores how many total iterations
a contcat was in state 5
```

```
    delta_time = [0] * total                                    # Stores the difference between current
timestamp and the last timestamp of a contact
    check_time_locations = [[] for i in range(total)]            # Stores a list of delta times
greater than 10ms and what row in the CSV file they occur
    timing_offset_shift = [0] * total                           # If timestamps are not aligned at
the beginning of a file this will shift the indices so that they align

    timeout_press_counter = [0] * len(GROUPS)                   # Stores the count used to
determine if all contacts closed within enough time of eachother
    timeout_unpress_counter = [0] * len(GROUPS)                 # Stores the count used to
determine if all contacts opened within enough time of eachother
    timeout_press_flag = [True] * len(GROUPS)
    timeout_unpress_flag = [True] * len(GROUPS)

    good_press = [0] * len(GROUPS)                              # Stores how many total good
presses there were for each contact group
    bad_press = [0] * len(GROUPS)                               # Stores how many total bad
presses there were for each contact group
    bad_press_locations = [[] for i in range(len(GROUPS))]      # Sores the location of
every recorded bad press

    good_unpress = [0] * len(GROUPS)                            # Stores how many total good
unpresses there were for each contact group
    bad_unpress = [0] * len(GROUPS)                             # Stores how many total bad
unpresses there were for each contact group
    bad_unpress_locations = [[] for i in range(len(GROUPS))]    # Sores the location of
every recorded bad press

    new_press_flag = [True] * len(GROUPS)
    new_unpress_flag = [True] * len(GROUPS)

    adjusted_index = 0

    for index in range(maximum):                                # Iterates through the rows of the
csv file
        shift = 0

        # These two loops iterate through all of the contacts and update state counts
        for group in range(len(GROUPS)):                        # Iterates through the contact
groups
```

```
        for contact in range(len(GROUPS[group])):                    # Iterates through the
individual contact test points that make up a single dome pad or other contact.
            adjusted_index = index + timing_offset_shift[contact + shift]

            if (adjusted_index > 0 and adjusted_index < len(data[contact + shift]) - 1):
                delta_time[contact + shift] = (data[contact + shift][adjusted_index][1] -
                                    data[contact + shift][adjusted_index - 1][1])

                if (delta_time[contact + shift] > check_time):
                    check_time_locations[contact + shift].append(
                        {"row": index + 2, "delta": delta_time[contact + shift]})

            if (index == 0):
                delta_time[contact + shift] = data[contact + shift][adjusted_index][1]       # on first
iteration this ensures that timestamps are aligned

            if (len(data[contact + shift]) - 1 < adjusted_index):                    # Handles index
errors from the contacts having different numbers of samples recorded
                continue

            if (data[contact + shift][adjusted_index][3] == low_out):
                low_out_count_cur[contact + shift] += 1

                # update total counts and reset current counts
                undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
                pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
                transition_count_total[contact + shift] += transition_count_cur[contact + shift]
                unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]
                high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]

                undefined_count_cur[contact + shift] = 0
                pressed_count_cur[contact + shift] = 0
                transition_count_cur[contact + shift] = 0
                unpressed_count_cur[contact + shift] = 0
                high_out_count_cur[contact + shift] = 0

            elif (data[contact + shift][adjusted_index][3] == pressed):
                pressed_count_cur[contact + shift] += 1

                # update total counts and reset current counts
```

```
        undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
        low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
        transition_count_total[contact + shift] += transition_count_cur[contact + shift]
        unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]
        high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]

        undefined_count_cur[contact + shift] = 0
        low_out_count_cur[contact + shift] = 0
        transition_count_cur[contact + shift] = 0
        unpressed_count_cur[contact + shift] = 0
        high_out_count_cur[contact + shift] = 0

    elif (data[contact + shift][adjusted_index][3] == transition):
        transition_count_cur[contact + shift] += 1

        # update total counts and reset current counts
        undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
        low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
        pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
        unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]
        high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]

        undefined_count_cur[contact + shift] = 0
        low_out_count_cur[contact + shift] = 0
        pressed_count_cur[contact + shift] = 0
        unpressed_count_cur[contact + shift] = 0
        high_out_count_cur[contact + shift] = 0

    elif (data[contact + shift][adjusted_index][3] == unpressed):
        unpressed_count_cur[contact + shift] += 1

        # update total counts and reset current counts
        undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
        low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
        pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
        transition_count_total[contact + shift] += transition_count_cur[contact + shift]
        high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]

        undefined_count_cur[contact + shift] = 0
        low_out_count_cur[contact + shift] = 0
```

```
            pressed_count_cur[contact + shift] = 0
            transition_count_cur[contact + shift] = 0
            high_out_count_cur[contact + shift] = 0

        elif (data[contact + shift][adjusted_index][3] == high_out):
            high_out_count_cur[contact + shift] += 1

            # update total counts and reset current counts
            undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
            low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
            pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
            transition_count_total[contact + shift] += transition_count_cur[contact + shift]
            unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]

            undefined_count_cur[contact + shift] = 0
            low_out_count_cur[contact + shift] = 0
            pressed_count_cur[contact + shift] = 0
            transition_count_cur[contact + shift] = 0
            unpressed_count_cur[contact + shift] = 0

        elif (data[contact + shift][adjusted_index][3] == undefined):
            undefined_count_cur[contact + shift] += 1

            # update total counts and reset current counts
            low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
            pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
            transition_count_total[contact + shift] += transition_count_cur[contact + shift]
            unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]
            high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]

            low_out_count_cur[contact + shift] = 0
            pressed_count_cur[contact + shift] = 0
            transition_count_cur[contact + shift] = 0
            unpressed_count_cur[contact + shift] = 0
            high_out_count_cur[contact + shift] = 0

    pressed_flag = 0
    unpressed_flag = 0
    group_len = len(GROUPS[group])
```

```
        # Check state_count_cur lists to debounce pressed and unpressed states
        for i in range(group_len):
            if (pressed_count_cur[i + shift] >= press_debounce):                    # If the contact has been in the pressed sate consecutively for "press_debounce_limit" iterations
                pressed_flag += 1
                if (timeout_press_flag[group] == True):                            # Save the index at which a contact was first press debounced to check that all contacts close within "timeout_limit" iterations
                    timeout_press_counter[group] = adjusted_index
                    timeout_press_flag[group] = False

            elif (unpressed_count_cur[i + shift] >= unpress_debounce):              # If the contact has been in the unpressed sate consecutively for "press_debounce_limit" iterations
                unpressed_flag += 1
                if (timeout_unpress_flag[group] == True):                          # Save the index at which a contact was first unpress debounced to check that all contacts open within "timeout_limit" iterations
                    timeout_unpress_counter[group] = adjusted_index
                    timeout_unpress_flag[group] = False
                                                                                   # If not all of the contacts close within the timeout limit then a bad press is recorded
        if (pressed_flag > 0 and (adjusted_index - timeout_press_counter[group]) > timeout and new_press_flag[group]):
            bad_press[group] += 1
            bad_press_locations[group].append(adjusted_index + 2)
            timeout_unpress_flag[group] = True
            new_press_flag[group] = False
            new_unpress_flag[group] = True
                                                                                   # If not all of the contacts open within the timeout limit then a bad unpress is recorded
        elif (unpressed_flag > 0 and (adjusted_index - timeout_unpress_counter[group]) > timeout and new_unpress_flag[group]):
            bad_unpress[group] += 1
            bad_unpress_locations[group].append(adjusted_index + 2)
            timeout_press_flag[group] = True
            new_unpress_flag[group] = False
            new_press_flag[group] = True

        elif (pressed_flag == group_len and new_press_flag[group]):                # If all of the contacts in a group were debounced in the pressed state successfully, record a good press
```

```
                good_press[group] += 1
                timeout_unpress_flag[group] = True
                new_press_flag[group] = False
                new_unpress_flag[group] = True

            elif (unpressed_flag == group_len and new_unpress_flag[group]):                    # If all
of the contacts in a group were debounced in the unpressed state successfully, record a good
press
                good_unpress[group] += 1
                timeout_press_flag[group] = True
                new_unpress_flag[group] = False
                new_press_flag[group] = True

            shift += group_len
        if (index == 0):                                                    # On the first iteration check
that the first timestamp for every contact is aligned
            for i in range(len(delta_time)):
                if (max(delta_time) - delta_time[i] > 3):
                    timing_offset_shift[i] += 1
                    print(out_filename + " Timing offset applied to contact: " +
                            str(i) + " to align timestamps during analysis")

    # At the end of the data, add any current state counts to totals
    shift = 0
    for group in range(len(GROUPS)):                                                # Iterates through
the contact groups
        for contact in range(len(GROUPS[group])):
            undefined_count_total[contact + shift] += undefined_count_cur[contact + shift]
            low_out_count_total[contact + shift] += low_out_count_cur[contact + shift]
            pressed_count_total[contact + shift] += pressed_count_cur[contact + shift]
            transition_count_total[contact + shift] += transition_count_cur[contact + shift]
            unpressed_count_total[contact + shift] += unpressed_count_cur[contact + shift]
            high_out_count_total[contact + shift] += high_out_count_cur[contact + shift]
        shift += len(GROUPS[group])

    save_timing_summary(out_filename, undefined_count_total, low_out_count_total,
pressed_count_total,
                        transition_count_total, unpressed_count_total, high_out_count_total,
                        good_press, bad_press, good_unpress, bad_unpress, bad_press_locations,
                        bad_unpress_locations, check_time_locations)
```

```
#---------------------------------------------------------------------
# Function: usage()
# Description: This function prints a help message to the cosole if
#              the help option was selected or an option was used
#              incorrectly.
#
# param: void
# return: void
#---------------------------------------------------------------------
def usage():

   print("\n-------------------------------------------------------  HELP
-------------------------------------------------------\n")

   print("-o\t--output\tOutput filename stem. If this option is omitted the default name of
\"output\" will be used.")
   print("\t\t\tExample: -o \"output\"\n")

   print("-i\t--input\t\tInput filename stem. If this option is omitted the default name of \"TEST\"
will be used.")
   print("\t\t\tExample: -i \"input\"\n")

   print("-g\t--group\t\tGroup IDs to process. This option must be given or else the program will
not execute.")
   print("\t\t\tWhen entering this option use \",\" to delimit redundant contacts for a single
button,\n\t\t\tand ; to delimit button groups.")
   print("\t\t\tExample: \"10, 11, 12; 20, 21, 22; 30, 31, 32\" or \"10, 11, 12\"\n")

   print("-p\t--pLimit\tControls how many files can be worked on in parallel. If no value is given
the default is 2.")
   print("\t\t\tThe value of the pLimit should be between 2-6 for a 4 core CPU. Larger file sizes
should use smaller pLimit.")
   print("\t\t\tExample: -p 2\n")

   print("-t\t--time\t\tEnable timing analysis. Timing analysis requires 0 parameter or 4 to
function. The parameters are:")
   print("\t\t\tcheck_time - any delta time between measurements larger than this is reported in
the summary,")
```

```
    print("\t\t\tpress_debounce - the number of consecutive measurements in press state required for debounce,")
    print("\t\t\tunpress_debounce - the number of consecutive measurements in unpress state required for debounce,")
    print("\t\t\ttimeout - if all contacts in a group do not enter the press or unpress state together within this time then it is marked as bad")
    print("\t\t\tThe function parameters should be entered as a string wit commas seperating the parameters.")
    print("\t\t\tExample: -t \"7, 5, 5, 30\"\tor -t")
    exit()


#---------------------------------------------------------------------
# Function: convert_file()
# Description: This function calls all of the other necessary functions
#              to convert a binary input file to a CSV output. This
#              function is where new processes are sent and eventually
#              die.
#
# param: (str) in_filename - Input filename stem
#        (str) out_filename - Output filename stem
#        (int[][]) groups - 2D array of contact groups and contact IDs
#        (queue) q - Shared memory queue used to signal to parent when
#                    a child is ready to be terminated
#        (bool) flag - True perform analysis, False do not analyze
#        (int[]) args - List of arguments to pass to timing_analysis()
# return: void
#---------------------------------------------------------------------
def convert_file(in_filename, out_filename, groups, q, flag, args):
    global GROUPS
    GROUPS = groups
    read_file(in_filename)

    convert_data(out_filename)

    separate_data()

    write_to_csv(out_filename)

    if (flag):
```

```
        if (len(args) == 4):
            timing_analysis(out_filename, args[0], args[1], args[2], args[3])
        else:
            timing_analysis(out_filename)
    while True:
        q.put(1)
        time.sleep(0.5)


#---------------------------------------------------------------------
# Function: main()
# Description: This function interpets the options used and manages the
#                 starting and terminating of child processes.
#
# param: void
# return: void
#---------------------------------------------------------------------
def main():
    global OUT_FILENAME, IN_FILENAME, GROUPS
    global raw_data, refined_data, data, in_file_count, in_file_list
    timing_analysis_flag = False
    process_limit = 2                                   # default limit for how many files can
be parallelized at a time. letting limit go to inf slows execution drastically due to memory and
cpu usage
    timing_args = None
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:i:g:p:t:",
                                ["help", "output=", "input=", "groups=", "pLimit=", "time="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)                                      # Will print something like "option -a not
recognized"
        usage()

    for o, a in opts:
        if o in ("-h", "--help"):
            usage()
        elif o in ("-t", "--time"):
            timing_analysis_flag = True
            timing_args = list(map(int, a.split(",")))
```

```
            if (len(timing_args) == 1):                                    # If the user passed -t with no
arguments use timing_analysis function defaults
                print("Using default timing analysis settings")
                continue
            elif (len(timing_args) != 4):                                   # Timing analysis function
takes 4 user parameters
                print("Timing analysis expects 4 arguments: check_time, " +
                    "press_debounce, unpress_debounce, timeout")
                exit()
            for a in timing_args:
                if (a == None or a < 0):                                   # if any of the arguments are invalid
raise exception
                    print("Timing analysis parameters must be greater than 0")
                    exit()

        elif o in ("-o", "--output"):
            OUT_FILENAME = a
        elif o in ("-i", "--input"):
            IN_FILENAME = a
        elif o in ("-g", "--groups"):
            GROUPS = list(map(str, a.split(";")))
            GROUPS = [list(map(int, i.split(","))) for i in GROUPS]

        elif o in ("-p", "--pLimit"):
            try:
                process_limit = int(a)
                if (process_limit < 1): raise Exception
                elif (process_limit > 20): raise Exception
            except:
                print("**Error: Invalid process limit of " + str(a) + ". Limit must fall within 1-20.\n")
                exit()
        else:
            assert False, "unhandled option"

    if (len(opts) == 0): usage()

    # GROUPS = [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
    # IN_FILENAME = "./TEST"
    # OUT_FILENAME = "outTest"
```

```
    # timing_analysis_flag = True

    if (GROUPS == None):
        print("**Error: Must include list of group IDs. Try -g \"10, 11, 12\"\n")
        exit()

    count_files()
    file_num = 0
    p = [None] * in_file_count
    q = [None] * in_file_count
    process_count = 0

    while (file_num <= in_file_count + 1):                                  # While there are still files to
convert
        if (process_count < process_limit and file_num < in_file_count):          # Start converting
the next file if more processes are allowed to be started
            q[file_num] = mp.Queue()                                        # Queue is shared and protected
memory for multiprocessing to use. the queue will be written to so that the parent thread can
terminate child processes
            in_filename = IN_FILENAME + str(in_file_list[file_num]) + ".bin"
            out_filename = OUT_FILENAME + str(in_file_list[file_num]) + ".csv"
            p[file_num] = mp.Process(target=convert_file,                   # Create a new process to
convert an input file
                        args=(in_filename, out_filename, GROUPS, q[file_num],
                        timing_analysis_flag, timing_args,), daemon=True, name=in_filename)
            p[file_num].start()

            print("Starting:\t" + str(p[file_num]))
            process_count += 1
            file_num += 1
            time.sleep(0.5)
        else:                                                               # If the maximum number of allowed
processes already exists
            for i in range(in_file_count):
                if (p[i] == None): continue                                 # If none the processes has already
finished and been released or has yet to be initialized
                try:
                    q[i].get(timeout=1)                                     # If the queue is empty it will
timeout and raise an exception
                    print("Killing:\t" + str(p[i]))
```

```
                p[i].kill()                                    # If a rocess queue returns any value then
terminate the process
                time.sleep(1.0)                                # Sleep 1 second after terminating a
process so that the .close() function will work
            except Exception as e:
                continue                                       # If the queue times out then the process
has not finished yet

            try:
                p[i].close()                                   # If a process is terminated call .close()
to free process resources
                del p[i]
                del q[i]
                p.append(None)                                 # None is append to the end of the
list so that size of the list does not change during a for loop
                q.append(None)

            except:
                print("Failed to release process")
            process_count -= 1
        if (file_num == in_file_count and process_count == 0):          # If all files have been
converted an all child process terminated, then break out of main loop and end program
            break

    print("\nEND\n")
    exit()


if __name__ == "__main__":
    main()
```