# Smart Walker Project

Jackson Farley, Ellie Langley, and Erik Leinen

May 9, 2019

## Contents



## List of Figures

# 1 Project Summary

This device is implemented on a smart walker and determines whether the walker is being used and at what time. When not in use, the device gently reminders the user to use the walker, and when the gentle reminder fails, the harsh reminder is enacted. As more and more elderly use walking aids, and as the elderly

population is more susceptible to injurious falls, it is difficult for an elderly person to achieve independence safely. This device will allow the elderly who require walking aids to live by themselves safely.

This device utilizes sensors to detect human presence on the upper and lower part of the walker. Depending on whether a human presence has been detected from the lower sensor, the upper sensor will then take a measurement. If the upper sensor has also taken a measurement with a proximal signal, then the device will output a "gentle reminder" by lighting the LED modules on the walker a solid yellow. If the walker has still not been used, then the device will output a "strong" reminder by flashing the LEDs red. When the buttons on the walker are pressed, the walker will go to an "in use state". A time stamp will be taken using a clock module and output to the SD card module.

Most of the electrical parts for the smart walker have been placed within a plastic, red Sparkfun box, with holes drilled for wires which need to go outside the box. The two push buttons have been placed on the handlebars of the walker. The device is powered by one 9V battery.

### 1.1 Objectives

The end objectives of this project are listed below. Some of the objectives are outside the scope of the immediate project, but the design process took place with all of these in mind, so they are all listed here. The final item on the list was specific to our implementation, and is therefore marked separately.

- **Accurately Analyze Human Interaction with the Walker**: Use two sensors to detect the presence of the user effectively, as well as have two buttons that detect whether the user is gripping the handles.
- **Accurate Time Recording**: Have a real-time clock able to keep time even when the system is powered off
- **Notification**: Succesfully notify users when they are not using the walker and should. To do this we must have sufficient display capability and accurate analysis as explained above. The item should also be able to notify others if there is an emergency.
- **Tracking**: The item should be able to record when the user has begun using the walker and keep that information for later analysis. Potentially an end objective would be creating a use-profile, so a doctor could be notified if there were significant deviations from this
- **Portability**: An older person should be able to transport this as easily as his/her standard walker. It should easily fit in the back of a small car and not add to the bulk of the system
- **Low Cost**: This system, when fully deployed, should be inexpensive (under $100, ideally under $50)
- **Modularity**: The system should attach to the user's existing walker with very little setup and no power tools.
- **Low Power**: The system should continue to work for at least 24 hours of normal operation on a single charge.

∗ **Relative Hardware Independence**: The code should work with potential changes in sensor or code with as little alteration as possible, therefore the functions and the objects to be interchangeable with a small amount of work where possible.

## 2 Hardware Design

### 2.1 Hardware Platform

The Arduino Mega 2560 was selected as the hardware platform, as the sensors purchased for the project best interface with the Arduino IDE, and this is what the previous developers had been using and was readily

available for low cost. Additionally, the added capability over the Arduino Uno gave greater flexibility and resources for computing once our system became much larger.

Other useful features include:

- Quick Up-Time: The Mega, once turned on, is ready in a matter of seconds, which allows it to record initial use and reduce potential annoyance to users. While it is designed to be on all the time, it is also good to know that the device won't break or fry if there is a sudden loss of power.
- Lower Power: The Beaglebone, while much more capable than the Mega, consumed much more power than the mega, meaning it would require a heavy and expensive battery to continue operation for the objective amount of time.
- Configurable Interrupts: the buttons implemented in this project must be configured as interrupts for the purpose of the device to be achieved.
- $I^2C$: many sensors communicate via this protocol, including some of the QWIIC enabled components by Sparkfun like the Real-Time Clock and the RF sensor.

## 2.2 Schematic and Parts Used

# 3 Software Design

The software for this project was all written in the arduino IDE. Much of it was taken from example sketches, specifically the setup of sensors and peripherals. The libraries for all of the main functions are included in the libraries/ folder in the GitRepository. The software design main loop was built with the idea of being compatible with an app later on for monitoring. This app was furnished with Arduino compatibilty by Blynk, an Internet of Things (IoT) company based on creating solutions that the user can control from his/her phone. The main blynk documentation can be found by clicking here.

To go along with the modular approach to this design, the discrete components of the system are broken up into different files, unified by a common header *definitions.h*.

### 3.0.1 Blynk Disclaimer

Currently, the Blynk website uploading and writing to the phone breaks the system and freezes it, so that functionality is currently under investigation. In the meantime, that Blynk functionality is commented out.

## 3.1 Timing

Before getting too deep into the code, it is important to bring up the issue of timing. Rather than using delays that could tie down the microcontroller, it would be far better to use timers that run specific functions at certain intervals. For this reason we use a **BlynkTimer**, something very similar to a simple timer created to be compatible with the Blynk system. This timer is an object that can have up to 10 different intervals and callbacks that can be set with the **setInterval()** command. The developer has the option of setting the interval, which should be long enough to not tie down the microcontroller, but short enough to allow for the accurate polling of the sensors.

It's also important to note that in the code, not every sequence has to be complete or run every time the function is called. We add static integers and case statements to have the desired functionality without over-burdening the system. These are explained in detail as they arise.

## 3.2 Header file definitions.h

The header file contains all of the information that applies to multiple files. This allows for the functions to be more readable and less cluttered if things are moved around. It also includes all other file inclusions that

are necessary, globals, and pin declarations. Ideally, if one person was looking to hook up all the wires in the system correctly, they would only need to look at the header file.

In addition to this information, it includes many of the module or object declarations that are used later on in the programs. These include things like the ledChip declaration. It is important to note that the ledChip and ledChip2 objects that are declared have numbers associated with them. These are the specific $I^2C$ addresses that allow them to operate separately from one another. If one wants more information on this specifically, they can look at the led chip driver hookup guide here.

Finally, there are several global constants or other important information located in the header. One of the most important is the global **CARD_OK**. This global can toggle wether an SD card is detected and allow the system to continue operation regardless of whether one is detected or not. While this is not necessary, If an SD card is not inserted, it is recommended to set this value to 0. Also the names of the states for the state machine are enumerated here, as well as the current state and next state. This allows functions from the button interrupts, for example, to change the value of next state even while it is in a separate file. The entire header file can be found in Listing 5.

## 3.3 Main

Based on a Blynk Example sketch for the ESP8266 wifi chip on the wifi shield, the main loop of the software function is designed to be as simple as possible. As with all Arduino main functions, there is a **setup** function and a **loop** function. The full main code is found in Listing 6 on Page 11.

### 3.3.1 The Setup

The setup function is executed once at the beginning, and then the loop function is run indefinitely. The setup is shown below in Listing 1. Since the main loop hasn't been entered, delays are allowed here to make sure that Serial ports are set and any protocol or setup is taken care of correctly. All of the delays are kept small, however, to keep the Up-Time as short as possible, because there is a fair bit of handshaking that has to happen. Additionally, a lot of the specifics for the setup are kept out of this loop and are done in the individual files. Each of these functions in their entirety are in the Appendices in Unaltered Code.

Listing 1: The Main Setup

```
void setup()
{
        // Debug console
        Serial.begin(9600);

        delay(10);

        // Set ESP8266 baud rate
        EspSerial.begin(ESP8266_BAUD);
        delay(10);

        // setting up function calls via the timer
        // first argument is the duration between events in milliseconds
        // up to 16 timers per timer object.
        // max data sending rate of 10 values per second
        timer.setInterval(STATE_MACH_INTERVAL, StateMachine); // this is the timer
            ↪ that calls the state machine every STATE_MACH_INTERVAL miliseconds
        timer.setInterval(LED_INTERVAL,LEDMain); // can be on a separate time from
            ↪ the state machine if flashes must occur slower or faster

        // WIFI Manager Attempts
        // one-time thing:
        //wifiManager.resetSettings();
```

```
        // setting up the auto connnect using WiFiManager Library
        //wifiManager.setConfigPortalTimeout(180); // waits 3 minutes and then will
            ↪ shut off.
        //wifiManager.autoConnect("RemindME Walker","remindme");

        SetupLEDs();
        SetupRTCandSD();
        SetupSensors();
        SetupButtonInterrupts();
        //Blynk.begin(auth, wifi, ssid, pass);
        SetupLEDDriveCurrents();
        ClearLED();
        DEBUG.println("Setup Complete!");
}
```

### 3.3.2 The Loop

The loop, as specified, is shown below in Listing 2. As one can see, it has been kept as simple as possible. Ideally, only the timer would be run in the main loop, however, there were some functions that had to occur frequently: the updates from the real-time clock module (RTC) and monitoring if the SD card is currently inserted.

Listing 2: The Main Loop

```
void loop()
{
        //Blynk.run();
        timer.run();
        if (rtc.updateTime() == false) //Updates the time variables from RTC
        {
                Serial.print("RTC failed to update");
        }
        // periodically checking
        // currently disabled to prevent the SD card from preventing other operation
            ↪ of the device.

        // FIXME: maybe add a parameter that enables easier checking and
            ↪ initialization if possible
        if (!digitalRead(cardDetect))
        {
                initializeCard();
        }

}
```

## 3.4 State Machine

This state machine is based off of the Activity Diagram shown in Figure The state machine function, called every 100 ms as specified in *definitions.h*, is shown in Listing 3. Like most state machines, this is implemented using a case statement. Each state has a dedicated function that carries out the operations of that state, whether it is monitoring, displaying LED output, or logging data. The full *State_Functions.ino* file can be found in Listing 7 on Page 13.

Listing 3: The State Machine Callback Function

```
void StateMachine()
{
        static int counter = 0;
        static unsigned long time_elapsed = 0;
        counter = counter + 1;
        time_elapsed = counter * STATE_MACH_INTERVAL;

        if(next_state != current_state) {
                //DEBUG.println("State change! Counter Reset");
                DEBUG.print("Current State: "); DEBUG.println(next_state);
                counter = 0;
        }
        // to account for if next_state was changed during the time interval, we
            ↪ switch right here
        current_state = next_state;

        switch(current_state) {
                case waiting:
                next_state = WaitingState(time_elapsed,&counter);
                break;
                case iAmHere:
                next_state = IAmHereState(time_elapsed,&counter);
                break;
                case gentleReminder:
                next_state = GentleReminderState(time_elapsed,&counter);
                break;
                case strongReminder:
                next_state = StrongReminderState(time_elapsed,&counter);
                break;
                case thankYou:
                next_state = ThankYouState(time_elapsed,&counter);
                break;
                case inUse:
                next_state = InUseState(time_elapsed,&counter);
                break;
                default:
                DEBUG.println("Error occured in the state diagram.");
                break;
        }

}
```

Each of the states is elaborated a little further here below:

- **waiting**: the initial state, this is where the walker is scanning for human presence on either sensor. It checks the lower sensor every time, since it is a constant digital output on the PIR. The upper sensor is only checked every 2 seconds by comparison.
- **iAmHere**: this state is gone to when either sensor detects some movement in front of it. The output is a white LED powerup sequence that reminds the user that it is there to be used. If a "proximal" signal, meaning the user could reach out and grab the walker, is detected from the upper sensor, the system moves to gentle reminder.
- **gentleReminder**: this state is a sequence of repeating yellow lights in an effort to have the user to grab the walker without a sterner reminder. If this state isn't observed, the state can either go to strongReminder if we believe the user is ignoring the walker or still near, or the state can go to waiting if the user hasn't moved.

- **strongReminder**: this state is a stronger flashing of red lights, eventually to be accompanied with an alarm that strongly suggests the user use the walker. If this warning is not observed after some amount of time, or the user has moved out of the way, then the state machine goes back to waiting.
- **thankYou**: this state is triggered when one of the handles has been grabbed. It can only be reached from one of the interrupt handlers from the button external interrupts. It goes through a pleasant green light sequence and records the time of the user grabbing the walker to the SD card. After this has been completed and the sequence ends, the state machine always goes to inUse.
- **inUse**: this state is used when either of the buttons is actively being pressed. If the person takes both hands off of the walker, the system goes to gentleReminder to remind the user to put their hands on the walker.

#### 3.4.1 Timing and Sequence Management in States

Since we use the timer to call the state machine function, the scope of the state machine is not passed any information about what time it is, or what state is was in earlier. Due to reccomendations about avoiding the millis() function, static integers, which retain their value when the function returns, were used to remedy the problem. There is a static counter variable *counter* that increments by 1 each time the function is called. And since the timer interval is known, one can calculate how much time has elapsed since we entered the state. From there, the *time_elapsed* is passed to the corresponding state functions. The counter is also passed in case the function resets the counter at any point. This is used in waitingState and inUse specifically to avoid getting massive counter integers that could wrap around. Otherwise, it's not currently muched used, but gives a lot of flexibility about how to manage time. The same basic time management scheme is used in the LED state to allow for sequences to occur in the same state. If interested to see the exact implementation, it is provided in the *LED_Functions.ino* file in Listing 9 with the counter and function calls in the *LED_Driver.ino* file in Listing 8.

It is important to note at this point that whenever the state changes, **the counter is reset to 0**.

### 3.5 Sensors

The sensor setup and functionality is largely taken care of the libraries that have been included. For the software, the sensors are simply black boxes, with functionality shown below:

- Lower Sensor: detects motion and outputs a 1 if detected. outputs a 0 otherwise upon measurement.
- Upper Sensor: detects distance. Outputs 2 if the user could grab the walker (about 1 meter away), outputs a 1 if the person is nearby (about 1-2.5 meters), and outputs 0 otherwise.

The RF sensor and OpenPIR sensor are configured via the functions to do exactly this. In addition, any setup necessary should happen in the **SetupSensors()** function called in the setup loop.

In addition, the buttons are technically sensors, so they have been included in this file. The interrupts are enabled in the setup as well, and then in the callback they set the next state to *thankYou*. The entire file is found in Listing 11 on Page 35

### 3.6 LEDs

The LEDs are broken up into two files: one that is entirely modular and independent of the LEDs used called *LED_Drivers.ino* (Listing 8)and one that is hardware dependent and called *LED_Functions.ino* (Listing 9). The independent file is formatted very similarly to the State Machine Callback, as it requires the same basic functionality. It calls state-specific functions that will do the sequences described above. Then the hardware dependent file goes and implements these. Depending on the time elapsed, the LEDs could be turning on or off, so there are sequence steps that are iterated through using a case statement. A simple example of one is reproduced below in Listing 4. As one can see, depending on the counter value which is static like earlier,

the LEDs will either turn on (by specifying a duty cycle of 50 out of 255) or turn off (by specifying a duty cycle of 0) and reset the counter to begin the cycle again. The duty cycle is much less than the maximum to save power, and since that brightness was found to be sufficient in indoor situations.

Listing 4: The Strong Reminder LED function

```
void StrongReminderLED(int * counter)
{

        switch(*counter){// this switch statement allows the same function to be
            ↪ called at a timer instance but have different results.
                case 1:
                ledChip.SetChannelPWM(redLED1,50);  // turn on
                ledChip.SetChannelPWM(redLED2,50);  // turn on
                ledChip.SetChannelPWM(redLED3,50);  // turn on
                ledChip2.SetChannelPWM(redLED1,50);  // turn on
                ledChip2.SetChannelPWM(redLED2,50);  // turn on
                ledChip2.SetChannelPWM(redLED3,50);  // turn on
                break;
                case 2:
                ledChip.SetChannelPWM(redLED1,0);  // turn off
                ledChip.SetChannelPWM(redLED2,0);  // turn off
                ledChip.SetChannelPWM(redLED3,0);  // turn off
                ledChip2.SetChannelPWM(redLED1,0);  // turn off
                ledChip2.SetChannelPWM(redLED2,0);  // turn off
                ledChip2.SetChannelPWM(redLED3,0);  // turn off
                *counter = 0; // by setting it to 0 we get 1 the next time it comes
                    ↪ around
                break;
        }

}
```

## 3.7 Real Time Clock Module

The Real Time Clock Module (RTC Module) is a battery powered oscillator that can continue to hold an accurate time after the system has turned off. It also has a very nice library that works alongside it. The real-time clock has a straightforward setup and then can get a Date string as well as a Time string. The specific functions and how they are used in **PrintDateandTime()** are found in Listing 10 on Page 25.

## 3.8 Micro SD Card and Data Logging

The micro SD card used uses a shifting $\mu$SD sparkfun board that uses Serial Peripheral Interface (SPI) protocols unlike the rest of the boards $I^2C$. It is additionally able to continually check for the presence of an SD card, but also operate without one should the need arise. There is a nice library for this as well available, that was cannibalized to fit the needs of this project. The micro SD card should work directly out of the box, but if it does not then there is searchable documentation to format it appropriately.

Additionally, since there was not one perfect way to save files, and one years worth of files would have been excessive, we decided to break files into week intervals. Each text file would start on Sunday of a given week of the year and be named the year, the word week, and then the week of the year: *19week12.txt*. This is implemented using the **PrintDateandTime()** function, which proceeds to put this onto the SD card. This card can then be plugged into most computers using a micro-SD to SD card adapter and read easily.

One can also use **PrintDateandTimeVerbose()** For additional information about the specific day, although this is unnecessary. In future iterations, the specific logging and methods used may need to change,

so this should act more as a framework than a final solution. The full code is located with the RTC Module code in Listing 10 on Page 25.

### 3.9 Future Software Directions

As aforementioned in the Blynk Disclaimer is the wifi, while implemented in hardware, was not fully able to be utilized. The ability for the wifi to be connected via a soft access point so that the end user doesn't have to do any coding is also a final product must-have. Additionally, the wifi had data transfer problems, and would freeze up the state machine. Fixing that is a top priority in future endeavors, and there are a couple possible solutions to help this problem:

- Take other functions out of the main loop
- Extend timing periods for the state machine
- Look into disabling the SPI (SD card) and see if that makes the transfer easier
- Change the buttons from interrupt to polling
- Look into other alternatives to Blynk, which may require more work or an html page

Another issue is the button triggers. Even with a debouncing circuit, there is still a significant amount of noise from the interrupts. Potentially this response could be better debounced by simply polling the levels, since that is the quickest that the information could be acted on. Additionally, freeing up external interrupts, as mentioned, could solve some of the wifi connection problems.

## 4 System Setup Instructions

## 5 Future Improvements

## 6 Appendices

### 6.1 Unaltered Code

Listing 5: Header File

```
#ifndef SMART_WALKER_DEF_H
#define SMART_WALKER_DEF_H

/************* FILE Inclusions ***************/
//#include <queue.h>
//#include <ESP8266WiFi.h>

#include <ESP8266_Lib.h>
#include <BlynkSimpleShieldEsp8266.h>

// LEDS

#include <lp55231.h>

// for the wifi manager
//#include <WiFiManager.h> // Wifi configuration manager used thanks to tazpu
//#include <DNSServer.h>
//#include <ESP8266WebServer.h>
```

```
// used in SD_SAVE_FILE
#include <SPI.h>
#include <SD.h>
#include <SparkFun_RV1805.h>

// used in Sensor_Functions
#include <SparkFun_RFD77402_Arduino_Library.h> //Use Library Manager or download
    ↪ here: https://github.com/sparkfun/SparkFun_RFD77402_Arduino_Library
#include <Wire.h>

/************ COMMUNICATIONS ***********/
/* Comment this out to disable prints and save space */
#define BLYNK_PRINT Serial
#define DEBUG Serial

// FOR ESP WIFI
// Hardware Serial on Mega, Leonardo, Micro...
#define EspSerial Serial1

// or Software Serial on Uno, Nano...
//#include <SoftwareSerial.h>
//SoftwareSerial EspSerial(10, 11); // RX, TX


/************ PINOUT **************/

// used in Sensor Functions
#define LEFT_HANDLE_BUTTON 2
#define RIGHT_HANDLE_BUTTON 3


/*
* While these button inputs are true for the system, the buttons are actually in
    ↪ different
* pin headers to allow for the debouncing circuit to route to these pins. These pins
    ↪  are:
* A0 -> To Left Handle Button
* A2 -> From Left Handle Button
* A1 -> To Right Handle Button
* A3 -> From Right Handle Button.
*/


#define PIR_DOUT 6 // PIR digital output on D2

// used In SD_SAVE_FILE
const uint8_t chipSelect = 8;
const uint8_t cardDetect = 9;


// FOR SD CARD
/*
* MEGA_P51 -> SD_DI
* MEGA_P50 -> SD_DO
* MEGA_P52 -> SD_SCK
```

```
*/

/************ MODULES AND CLASSS DEFINITIONS **********/
// module stuff
RV1805 rtc;
File fd;

// LEDS
static Lp55231 ledChip(0x32);
static Lp55231 ledChip2(0x33);

RFD77402 myDistance_upper; // hook object to library for upper sensor

/******************** CONSTANTS ******************/
// used in SD_SAVE_FILE
#define str_len 3

// Timing
#define STATE_MACH_INTERVAL 100L
#define LED_INTERVAL 500L


// Your ESP8266 baud rate:
#define ESP8266_BAUD 9600


bool CARD_OK = 1;

/***************** GLOBALS *********************/
enum State{waiting, iAmHere, gentleReminder, strongReminder, thankYou, inUse};
enum State current_state;
enum State volatile next_state; // so that it can be changed in the interrupt
    ↪ callbacks

/***************** LED CHANNELS *****************/
int greenLED1 = 0;
int blueLED1 = 1;
int greenLED2 = 2;
int blueLED2 = 3;
int greenLED3 = 4;
int blueLED3 = 5;
int redLED1 = 6;
int redLED2 = 7;
int redLED3 = 8;

#endif // SMART_WALKER_DEF_H
```

Listing 6: Main Arduino Code

```
/*************************************************************
 Download latest Blynk library here:
  https://github.com/blynkkk/blynk-library/releases/latest

 Blynk is a platform with iOS and Android apps to control
 Arduino, Raspberry Pi and the likes over the Internet.
 You can easily build graphic interfaces for all your
```

```
  projects by simply dragging and dropping widgets.

   Downloads, docs, tutorials: http://www.blynk.cc
   Sketch generator:           http://examples.blynk.cc
   Blynk community:            http://community.blynk.cc
   Follow us:                  http://www.fb.com/blynkapp
                               http://twitter.com/blynk_app

  Blynk library is licensed under MIT license
  This example code is in public domain.

  **************************************************************

  This example shows how to use ESP8266 Shield (with AT commands)
  to connect your project to Blynk.

  WARNING!
   It's very tricky to get it working. Please read this article:
   http://help.blynk.cc/hardware-and-libraries/arduino/esp8266-with-at-firmware

  Change WiFi ssid, pass, and Blynk auth token to run :)
  Feel free to apply it to any other example. It's simple!
  **************************************************************/

#include "definitions.h"

// define pinouts, define interrupts



// You should get Auth Token in the Blynk App.
// Go to the Project Settings (nut icon).
char auth[] = "a2f791ce584c4b91a59bdc40e6e9b4d9";

// including the timer necessary instead of delays (so that blink can still run
    ↪ compatibly)
BlynkTimer timer;

// adding the wifi Manager
//WifiManager wifiManager;

// Your WiFi credentials.
// Set password to "" for open networks.
char ssid[] = "iPhone";
char pass[] = "morecoffee";


ESP8266 wifi(&EspSerial);

void setup()
{
 // Debug console
 Serial.begin(9600);

 delay(10);
```

```
 // Set ESP8266 baud rate
 EspSerial.begin(ESP8266_BAUD);
 delay(10);

 // setting up function calls via the timer
 // first argument is the duration between events in milliseconds
 // up to 16 timers per timer object.
 // max data sending rate of 10 values per second
 timer.setInterval(STATE_MACH_INTERVAL, StateMachine); // this is the timer that
    ↪ calls the state machine every STATE_MACH_INTERVAL miliseconds
 timer.setInterval(LED_INTERVAL, LEDMain); // can be on a separate time from the
    ↪ state machine if flashes must occur slower or faster

 // WIFI Manager Attempts
 // one-time thing:
 //wifiManager.resetSettings();
 // setting up the auto connnect using WiFiManager Library
 //wifiManager.setConfigPortalTimeout(180); // waits 3 minutes and then will shut
    ↪ off.
 //wifiManager.autoConnect("RemindME Walker","remindme");

 SetupLEDs();
 SetupRTCandSD();
 SetupSensors();
 SetupButtonInterrupts();
 //Blynk.begin(auth, wifi, ssid, pass);
 SetupLEDDriveCurrents();
 ClearLED();
 DEBUG.println("Setup Complete!");
}

void loop()
{
 //Blynk.run();
 timer.run();
 if (rtc.updateTime() == false) //Updates the time variables from RTC
 {
  Serial.print("RTC failed to update");
 }
 // periodically checking
 // currently disabled to prevent the SD card from preventing other operation of the
    ↪  device.

 // FIXME: maybe add a parameter that enables easier checking and initialization if
    ↪ possible
 if (!digitalRead(cardDetect))
 {
   initializeCard();
 }

}
```

Listing 7: The State Machine and Associated Functions

```
#include "definitions.h"
```

```
//enum State{waiting, iAmHere, gentleReminder, strongReminder, thankYou, inUse};

void StateMachine()
{
 static int counter = 0;
 static unsigned long time_elapsed = 0;
 counter = counter + 1;
 time_elapsed = counter * STATE_MACH_INTERVAL;


 if(next_state != current_state) {
  //DEBUG.println("State change! Counter Reset");
  DEBUG.print("Current State: "); DEBUG.println(next_state);
  counter = 0;
 }
 // to account for if next_state was changed during the time interval, we switch
    ↪ right here
 current_state = next_state;

 switch(current_state) {
  case waiting:
    next_state = WaitingState(time_elapsed,&counter);
   break;
  case iAmHere:
    next_state = IAmHereState(time_elapsed,&counter);
   break;
  case gentleReminder:
    next_state = GentleReminderState(time_elapsed,&counter);
   break;
  case strongReminder:
    next_state = StrongReminderState(time_elapsed,&counter);
   break;
  case thankYou:
    next_state = ThankYouState(time_elapsed,&counter);
   break;
  case inUse:
    next_state = InUseState(time_elapsed,&counter);
   break;
  default:
    DEBUG.println("Error occured in the state diagram.");
   break;
 }

}



enum State WaitingState(unsigned long time_elapsed, int * counter) {
  if(LowerSensorTakeMeasurement() == 1){
   // DEBUG.println("going to I AM Here");
   return iAmHere;
  }
  if(time_elapsed >= 2000L) { // every two seconds
   //DEBUG.println(time_elapsed);
   if(UpperSensorTakeMeasurement() > 0) {
    // detected motion, go to i am here
```

```
    //DEBUG.println("going to I AM Here");
    return iAmHere;
   }
   *counter = 0; // reset the counter in order to continue to monitor.
  }

  return waiting;
}

enum State IAmHereState(unsigned long time_elapsed, int * counter) {
  if(time_elapsed % 500 == 0) { // every second FIXME: make more frequent in release
   if(UpperSensorTakeMeasurement() == 2){
    // moving to gentle reminder state due to a new proximal signal
    //DEBUG.println("Moving to gentle reminder");
    return gentleReminder;
   }
  }
  if(time_elapsed >= 10000) { // every ten seconds
   if(LowerSensorTakeMeasurement() == 0) { // no further signal detected
    //DEBUG.println("Going back to waiting");
    return waiting;
   }
  }
  return iAmHere;
}

enum State GentleReminderState(unsigned long time_elapsed, int * counter) {
  // do gentle reminder things
  if(time_elapsed % 500 == 0) { // every half second
   if(UpperSensorTakeMeasurement() < 1){
    // no signal
    //DEBUG.println("Moving to strong reminder");
    return strongReminder;
   }
  }
  if(time_elapsed >= 1000L * 5 && time_elapsed % 1000 == 0) { // after ten seconds,
   ↪ every second
   if(LowerSensorTakeMeasurement() == 0 && UpperSensorTakeMeasurement() == 0){ //
   ↪ nobody home
    //DEBUG.println("Going to waiting");
    return waiting;
   }
   else if (UpperSensorTakeMeasurement() == 2){
     // detected proximal signal. What is the user up to? come on and grab me
     return strongReminder;

   }
  }

  if(time_elapsed >= 1000L * 15) { // after 15 seconds, it doesn't matter, we're
   ↪ going to waiting
   return waiting;
  }
  // if none of these things happen, we stay at the same state.
  return gentleReminder;
}
```

```
enum State StrongReminderState(unsigned long time_elapsed, int * counter) {
  // do strong reminder things
  if(time_elapsed > 3000 && time_elapsed % 1000L == 0)
  {
   if(LowerSensorTakeMeasurement()==0 && UpperSensorTakeMeasurement() == 0) {
    // nobody home! both have to be 0 for it to be effective
    return waiting;
   }
  }
  if(time_elapsed >= 1000L * 15){ // be loud for 15 seconds, then timeout
   // timeout feature to go back to waiting
   return waiting;
  }
  return strongReminder;
}

enum State ThankYouState(unsigned long time_elapsed, int * counter) {
  // always gone to when the walker is grabbed.
  //DEBUG.println("THANK YOU!!!");
  if(time_elapsed >= 6500L) { // the amount of time
    // after this, the LED sequence should be done
    if(CARD_OK == 1) // that means the card is ok to use and we can successfully
   ↪ print
    {
     printDateAndTime();
    }
    //DEBUG.println("Going to In Use");
    return inUse;
   }
  return thankYou;
}

enum State InUseState(unsigned long time_elapsed, int * counter) {
  while(digitalRead(LEFT_HANDLE_BUTTON)==1 || digitalRead(RIGHT_HANDLE_BUTTON) == 1)
  {
   // if the person is using the buttons, then we are absolutely in use and don't
   ↪ need to check the other criterion.
   return inUse;
  }

  if(time_elapsed >= 1000L) { // every second
   if(UpperSensorTakeMeasurement() > 1){
    // proximal signal. User still standing there.
    //DEBUG.println("Moving back to Waiting");
    //EnableButtonInterrupts();
    *counter = 0;
    return gentleReminder;
   }
   else{
    return waiting;
   }
   *counter = 0;
  }
  return inUse;
}
```

```
/********** REDIRECTION FUNCTIONS **********/
void GoToThankYouLeft(){
 if(current_state != inUse) {
  // we don't want a continual thank you state if the person is currently using the
    ↪ walker.
  DEBUG.println("Redirect Left");
  //DisableButtonInterrupts();
  //delayMicroseconds(1000);
  next_state = thankYou;
 }
}

void GoToThankYouRight() {
 if(current_state != inUse) {
  DEBUG.println("Redirect Right");
  //delayMicroseconds(1000);
  next_state = thankYou;
 }
}
```

Listing 8: The LED Drivers

```
/******************************************************************************
simple-two-chips.ino
simple demo of using two LP5s5231s in parallel to control 18 LEDs.
Byron Jacquot @ SparkFun Electronics
October 21, 2016
https://github.com/sparkfun/SparkFun_LP55231_Arduino_Library

Demonstration of two LP55231s on an I2C bus.

Two Lp55231 objects are declared, at unique I2C addresses. The program then
writes different values to each chip, resulting an a chase pattern in which one
chip appears to be a step behind the other.

Resources:
Written using SparkFun Pro Micro controller, with two LP55231 breakout boards.

Hardware Configuration:
The second Lp55231 board is configured for address 0x33, by changing
 Address Jumper A0. The first board has
the I2C pullup resistors disconnected.

Development environment specifics:
Written using Arduino 1.6.5


This code is released under the [MIT License](http://opensource.org/licenses/MIT).

Please review the LICENSE.md file included with this example. If you have any
    ↪ questions
or concerns with licensing, please contact techsupport@sparkfun.com.

Distributed as-is; no warranty is given.
******************************************************************************/
```

```
#include "definitions.h"

// don't need the enable
//static const int32_t enable_pin = 10; // Apparently active high?

// A quick example demonstrating two LP55231's on the same I2C bus.
//
// One of them will need to be reconfigured to address 0x33, by changing
// Address Jumper A0.
//
// to interface them, two Lp55231 objects are declared, each with their own address.
//
// It will simply chase among the LED outputs. The second chip will be one LED
// ahead of the other, and the chip are set to opposite ends of the range of
// output current, making the first chip brighter than the second..

// not used with this timer
//static uint32_t next;


// NOTE Must call Setup LED Drive currents to complete.
// rather than keeping it one function I made it two and have other setups acting as
    ↪  the delay(1000) to reduce setup time
void SetupLEDs ()
{
 //Serial.begin(9600);

 //delay(2000);
 //Serial.println("### Setup entry");

 // the enable pin doesn't acutally make any difference.
 //pinMode(enable_pin, OUTPUT);
 //digitalWrite(enable_pin, LOW);
 //digitalWrite(enable_pin, HIGH);

 ledChip.Begin();
 ledChip.Enable();

 ledChip2.Begin();
 ledChip2.Enable();


 //delay(1000);
}

void SetupLEDDriveCurrents() {
 for(uint8_t i = 0; i < 9; i++)
 {
  ledChip.SetDriveCurrent(i, 0xff);
  ledChip2.SetDriveCurrent(i, 0xff);
 }

 Serial.println("LED Setup Complete");

}
```

```
void LEDMain() {
  // based off the state, do various LED functions

  // The LED main function has one master counter that goes to all of the sub
    ↪ functions.
  // The sub functions have LED cycles they do based on the counter value, and can
    ↪ reset the counter to 0 if necessary
  static int counter = 0;
  // we want the LED sequence to start fresh (counter = 0) each time the state is
    ↪ switched to it.
  // an easy way to do this is to have a static state, previous state, that we can
    ↪ check the current state against.
  // If they don't match, then we know that we should reset the counter to 0
  static enum State previous_state = waiting; // initialized to the idle state of
    ↪ the system.
  if(current_state != previous_state){
   //DEBUG.println("LED detected a change in state");
   ClearLED();
   counter = 1;
   previous_state = current_state; // only changing here since we don't need to
    ↪ assign it otherwise
  }

  //DEBUG.println(counter);

  switch(current_state) {
  case waiting:

   break;
  case iAmHere:
    StartUpLED(&counter);
   break;
  case gentleReminder:
    GentleReminderLED(&counter);
   break;
  case strongReminder:
    StrongReminderLED(&counter);
   break;
  case thankYou:
    ThankYouLED(&counter);
   break;
  case inUse:
    InUseLED(&counter);
   break;
  default:
    DEBUG.println("Error occured in the LED diagram.");
   break;
 }

 // update the counter
 counter++;


}
```

```
void Flash(int led_channel) {
 // can be up to 9 different channels.
 // if channel
 static int counter = 0;
 switch(counter){ // this switch statement allows the same function to be called at
    ↪ a timer instance but have different results.
  case 0:
    ledChip.SetChannelPWM(led_channel,0xff); // turn on
    ledChip2.SetChannelPWM(led_channel,0xff); // turn on
   break;
  case 1:
    ledChip.SetChannelPWM(led_channel,0); // turn off
    ledChip2.SetChannelPWM(led_channel,0); // turn on
   break;
 }
 counter ++;
 counter = counter%2; // right now only goes between 0 and 1.
}


/* Example code from the fader example that I might want to use later


#include <Wire.h>
#include <lp55231.h>

Lp55231 ledChip;

void setup() {
 // put your setup code here, to run once:

 Serial.begin(9600);
 delay(5000);
 Serial.println("-- Starting Setup() --");

 ledChip.Begin();
 ledChip.Enable();

 ledChip.AssignChannelToMasterFader(0, 0);
 ledChip.AssignChannelToMasterFader(1, 0);
 ledChip.AssignChannelToMasterFader(6, 0);

 ledChip.SetLogBrightness(0, true);
 ledChip.SetLogBrightness(1, true);
 ledChip.SetLogBrightness(6, true);

 ledChip.SetDriveCurrent(0, 0xff);
 ledChip.SetDriveCurrent(1, 0xff);
 ledChip.SetDriveCurrent(6, 0xff);

 // can adjust color here easily
 ledChip.SetChannelPWM(0,0x00); // og 80
 ledChip.SetChannelPWM(1,0xff); // og 40
 ledChip.SetChannelPWM(6,0xd0); // og ff

 delay(500);
```

```
 Serial.println("-- Setup() Complete --");

}

void loop() {
 // put your main code here, to run repeatedly:

 // current will track the LED we're turning on
 // previous will keep track of the last one we turned on to turn it off again

 static uint8_t current = 0, previous = 0;
 static uint32_t next = millis()+1000;

 if(millis() >= next)
 {
  next += 50;

  Serial.print("Illuminating: ");
  Serial.println(current);

  ledChip.SetMasterFader(0, current);

  current++;
 }
}


*/
```

Listing 9: The LED Functions

```
#include "definitions.h"

// DECLARE THE COUNTER OUTSIDE OF THE LED FUNCTIONS

void ThankYouLED(int * counter)
{

 switch(*counter){ // this switch statement allows the same function to be called at
    ↪ a timer instance but have different results.
  case 1:
    ledChip.SetChannelPWM(greenLED1,50); // turn on
    ledChip.SetChannelPWM(greenLED2,0); // turn on
    ledChip.SetChannelPWM(greenLED3,0); // turn on
    ledChip2.SetChannelPWM(greenLED1,0); // turn on
    ledChip2.SetChannelPWM(greenLED2,0); // turn on
    ledChip2.SetChannelPWM(greenLED3,50); // turn on
   break;
  case 2:
    ledChip.SetChannelPWM(greenLED1,0); // turn off
    ledChip.SetChannelPWM(greenLED2,50); // turn on
                  ledChip2.SetChannelPWM(greenLED3,0); // turn off
    ledChip2.SetChannelPWM(greenLED2,50); // turn on
   break;
  case 3:
```

```
    ledChip.SetChannelPWM(greenLED2,0); // turn off
    ledChip.SetChannelPWM(greenLED3,50); // turn on
                    ledChip2.SetChannelPWM(greenLED2,0); // turn off
    ledChip2.SetChannelPWM(greenLED1,50); // turn on
    *counter = 0; // by setting it to 0 we get 1 the next time it comes around, to
    ↪ restart the cycle
   break;
 }

}

void StrongReminderLED(int * counter)
{

 switch(*counter){// this switch statement allows the same function to be called at
    ↪ a timer instance but have different results.
  case 1:
    ledChip.SetChannelPWM(redLED1,50); // turn on
    ledChip.SetChannelPWM(redLED2,50); // turn on
    ledChip.SetChannelPWM(redLED3,50); // turn on
                    ledChip2.SetChannelPWM(redLED1,50); // turn on
    ledChip2.SetChannelPWM(redLED2,50); // turn on
    ledChip2.SetChannelPWM(redLED3,50); // turn on
   break;
  case 2:
    ledChip.SetChannelPWM(redLED1,0); // turn off
    ledChip.SetChannelPWM(redLED2,0); // turn off
    ledChip.SetChannelPWM(redLED3,0); // turn off
                    ledChip2.SetChannelPWM(redLED1,0); // turn off
    ledChip2.SetChannelPWM(redLED2,0); // turn off
    ledChip2.SetChannelPWM(redLED3,0); // turn off
    *counter = 0; // by setting it to 0 we get -1 the next time it comes around
   break;
 }

}

void StartUpLED(int * counter)
{
 // this routine only happens once, so we refrain from resetting the counter in the
    ↪ last step.
 switch(*counter){ // this switch statement allows the same function to be called at
    ↪  a timer instance but have different results.
  case 1:
    ledChip.SetChannelPWM(blueLED1,50);
    ledChip.SetChannelPWM(greenLED1,50);
    ledChip.SetChannelPWM(redLED1,50);
    ledChip.SetChannelPWM(blueLED2,0);
    ledChip.SetChannelPWM(greenLED2,0);
    ledChip.SetChannelPWM(redLED2,0);
    ledChip.SetChannelPWM(blueLED3,0);
    ledChip.SetChannelPWM(greenLED3,0);
    ledChip.SetChannelPWM(redLED3,0);
                    ledChip2.SetChannelPWM(blueLED1,0);
    ledChip2.SetChannelPWM(greenLED1,0);
    ledChip2.SetChannelPWM(redLED1,0);
```

```
    ledChip2.SetChannelPWM(blueLED2,0);
    ledChip2.SetChannelPWM(greenLED2,0);
    ledChip2.SetChannelPWM(redLED2,0);
    ledChip2.SetChannelPWM(blueLED3,50);
    ledChip2.SetChannelPWM(greenLED3,50);
    ledChip2.SetChannelPWM(redLED3,50);
   break;
  case 2:
    ledChip.SetChannelPWM(blueLED1,0);
    ledChip.SetChannelPWM(greenLED1,0);
    ledChip.SetChannelPWM(redLED1,0);
    ledChip.SetChannelPWM(blueLED2,50);
    ledChip.SetChannelPWM(greenLED2,50);
    ledChip.SetChannelPWM(redLED2,50);
                    ledChip2.SetChannelPWM(blueLED3,0);
    ledChip2.SetChannelPWM(greenLED3,0);
    ledChip2.SetChannelPWM(redLED3,0);
    ledChip2.SetChannelPWM(blueLED2,50);
    ledChip2.SetChannelPWM(greenLED2,50);
    ledChip2.SetChannelPWM(redLED2,50);
   break;
  case 3:
    ledChip.SetChannelPWM(blueLED2,0);
    ledChip.SetChannelPWM(greenLED2,0);
    ledChip.SetChannelPWM(redLED2,0);
    ledChip.SetChannelPWM(blueLED3,50);
    ledChip.SetChannelPWM(greenLED3,50);
    ledChip.SetChannelPWM(redLED3,50);
                    ledChip2.SetChannelPWM(blueLED2,0);
    ledChip2.SetChannelPWM(greenLED2,0);
    ledChip2.SetChannelPWM(redLED2,0);
    ledChip2.SetChannelPWM(blueLED1,50);
    ledChip2.SetChannelPWM(greenLED1,50);
    ledChip2.SetChannelPWM(redLED1,50);
   break;
   case 4:
    ledChip.SetChannelPWM(blueLED1,50);
    ledChip.SetChannelPWM(greenLED1,50);
    ledChip.SetChannelPWM(redLED1,50);
    ledChip.SetChannelPWM(blueLED2,50);
    ledChip.SetChannelPWM(greenLED2,50);
    ledChip.SetChannelPWM(redLED2,50);
                    ledChip2.SetChannelPWM(blueLED3,50);
    ledChip2.SetChannelPWM(greenLED3,50);
    ledChip2.SetChannelPWM(redLED3,50);
    ledChip2.SetChannelPWM(blueLED2,50);
    ledChip2.SetChannelPWM(greenLED2,50);
    ledChip2.SetChannelPWM(redLED2,50);
   break;
  case 10:
    ClearLED(); // keep the lights on for a bit before the setup sequence turns off
   break;
  default:
   break;
}
```

```
}

void GentleReminderLED(int * counter)
{
 switch(*counter){ // this switch statement allows the same function to be called at
    ↪ a timer instance but have different results.
  case 1:
    ledChip.SetChannelPWM(greenLED1,50); // turn on
    ledChip.SetChannelPWM(redLED1,50); // turn on
    ledChip.SetChannelPWM(greenLED3,0); // turn on
    ledChip.SetChannelPWM(redLED3,0); // turn on
                  ledChip2.SetChannelPWM(greenLED3,50); // turn on
    ledChip2.SetChannelPWM(redLED3,50); // turn on
    ledChip2.SetChannelPWM(greenLED1,0); // turn on
    ledChip2.SetChannelPWM(redLED1,0); // turn on

   break;
  case 2:
    ledChip.SetChannelPWM(greenLED2,50); // turn on
    ledChip.SetChannelPWM(redLED2,50); // turn on
    ledChip.SetChannelPWM(greenLED1,0); // turn on
    ledChip.SetChannelPWM(redLED1,0); // turn on
                  ledChip2.SetChannelPWM(greenLED2,50); // turn on
    ledChip2.SetChannelPWM(redLED2,50); // turn on
    ledChip2.SetChannelPWM(greenLED3,0); // turn on
    ledChip2.SetChannelPWM(redLED3,0); // turn on
   break;
  case 3:
    ledChip.SetChannelPWM(greenLED3,50); // turn on
    ledChip.SetChannelPWM(redLED3,50); // turn on
    ledChip.SetChannelPWM(greenLED2,0); // turn on
    ledChip.SetChannelPWM(redLED2,0); // turn on
                  ledChip2.SetChannelPWM(greenLED1,50); // turn on
    ledChip2.SetChannelPWM(redLED1,50); // turn on
    ledChip2.SetChannelPWM(greenLED2,0); // turn on
    ledChip2.SetChannelPWM(redLED2,0); // turn on
    *counter = 0;
   break;
 }

}

void InUseLED(int * counter)
{

 int slow_counter = *counter/5 + 1; // slow down the counter to change functionality
    ↪ every 5th counter cycle.
 switch(slow_counter){// this switch statement allows the same function to be called
    ↪ at a timer instance but have different results.
  case 1:
   ledChip.SetChannelPWM(redLED2,20); // turn on
   ledChip.SetChannelPWM(greenLED2,20); // turn on
   ledChip.SetChannelPWM(blueLED2,20); // turn on
          ledChip2.SetChannelPWM(redLED2,20); // turn on
   ledChip2.SetChannelPWM(greenLED2,20); // turn on
   ledChip2.SetChannelPWM(blueLED2,20); // turn on
```

```
    break;
   case 2:
    ledChip.SetChannelPWM(redLED2,0); // turn off
    ledChip.SetChannelPWM(greenLED2,0); // turn off
    ledChip.SetChannelPWM(blueLED2,0); // turn on
            ledChip2.SetChannelPWM(redLED2,0); // turn off
    ledChip2.SetChannelPWM(greenLED2,0); // turn off
    ledChip2.SetChannelPWM(blueLED2,0); // turn on
    break;
 }
 // resets counter after every 10 cycles.
 *counter = *counter % 10;
}

void ClearLED(void)
{
 //delay(10);
 ledChip.SetChannelPWM(blueLED1,0);
 ledChip.SetChannelPWM(greenLED1,0);
 ledChip.SetChannelPWM(redLED1,0);
 ledChip.SetChannelPWM(blueLED2,0);
 ledChip.SetChannelPWM(greenLED2,0);
 ledChip.SetChannelPWM(redLED2,0);
 ledChip.SetChannelPWM(blueLED3,0);
 ledChip.SetChannelPWM(greenLED3,0);
 ledChip.SetChannelPWM(redLED3,0);
 ledChip2.SetChannelPWM(blueLED1,0);
 ledChip2.SetChannelPWM(greenLED1,0);
 ledChip2.SetChannelPWM(redLED1,0);
 ledChip2.SetChannelPWM(blueLED2,0);
 ledChip2.SetChannelPWM(greenLED2,0);
 ledChip2.SetChannelPWM(redLED2,0);
 ledChip2.SetChannelPWM(blueLED3,0);
 ledChip2.SetChannelPWM(greenLED3,0);
 ledChip2.SetChannelPWM(redLED3,0);
}
```

Listing 10: The SD card and RTC functionality

```
#include "definitions.h"


const uint8_t BUFFER_SIZE = 20;
char fileName[15] = "19Week00.txt"; // SD library only supports up to 8.3 names
char buff[BUFFER_SIZE+2] = ""; // Added two to allow a 2 char peek for EOF state
uint8_t index = 0;


enum states: uint8_t { NORMAL, E, EO };
uint8_t state = NORMAL;

bool alreadyBegan = false; // SD.begin() misbehaves if not first call

String currentMonth;
String currentDay;
String currentYear;
```

```
String currentCentury;

int month;
int day;
int year;
int century;

 String monthName;
 String dayName;
 String firstDay;

 char curMon[str_len];
 char curDay[str_len];
 char curYear[str_len];
 char curCent[str_len];

 int dayOfWeek;
 int firstOfYear;
 int leapYear;
 int addyear;
 int firstYear;
 int addmonth;
 int dayOfYear;
 int weekNumber;

 int previousWeek;
 int currentWeek;
 int previousDay = 10; // this is not a real day (1-6). Guarantees that

void SetupRTCandSD()
{

 //Wire.begin();

 //Serial.begin(57600); // hopefully this doesn't wreak havok.


 if (rtc.begin() == false)
 {
  Serial.println("Something went wrong with the RTC, check wiring");
 }
 //String currentDate = rtc.stringDateUSA(); //Get the current date in mm/dd/yyyy
    ↪ format (we're weird)
 //String currentTime = rtc.stringTime(); //Get the time
 //calculateDayOfYear(month,day,year,century);
 //calculateWeekday(dayOfWeek,firstOfYear);

 DEBUG.println("RTC online!");
 while (!Serial); // Wait for serial port to connect (ATmega32U4 type PCBAs)

 // Note: To satisfy the AVR SPI gods the SD library takes care of setting
 // SS_PIN as an output. We don't need to.
 //pinMode(chipSelect, OUTPUT);
 pinMode(cardDetect, INPUT);

 initializeCard();
```

```
}


int initializeCard(void)
{
 static int tries = 1;
 Serial.print(F("Initializing SD card..."));
 //CARD_OK = 0;
 // Is there even a card?
 if (!digitalRead(cardDetect))
 {
  Serial.println(F("No card detected. Waiting for card."));
  if(!digitalRead(cardDetect) && tries <= 10)
  {
   CARD_OK = 0;
   tries++;
  }
  else if(tries <= 10)
  { // this means that a card was read since this condition has not been satisfied
   tries = 1;
   CARD_OK = 1;
  }
  else {
   // proceed with no card by setting global CARD_OK to 0, aka it's not ok use;
   CARD_OK = 0;
  }
  delay(250); // 'Debounce insertion'
 }

 // Card seems to exist. begin() returns failure
 // even if it worked if it's not the first call.
 if (!SD.begin(chipSelect) && !alreadyBegan) // begin uses half-speed...
 {
  Serial.println(F("Initialization failed!"));
  if(tries <= 10){
   initializeCard(); // Possible infinite retry loop is as valid as anything
   tries++;
   return -1;
  }
  else {
   Serial.println(F("Giving up on using SD"));
   CARD_OK = 0;
   return -1;
  }
 }
 else
 {
  alreadyBegan = true;
  tries = 1; // success! reset tries
 CARD_OK = 1;
 }
 Serial.println(F("Initialization done."));

 Serial.print(fileName);
 if (SD.exists(fileName))
 {
```

```
  Serial.println(F(" exists."));
 }
 else
 {
  Serial.println(F(" doesn't exist. Creating."));
 }

 Serial.print("Opening file: ");
 Serial.println(fileName);

 Serial.println(F("Enter text to be written to file. 'EOF' will terminate writing.")
    ↪ );

 return 0;
}

void eof(void)
{
 printDateAndTime();

 // Re-open the file for reading:
 fd = SD.open(fileName);
 if (fd)
 {
  Serial.println("");
  Serial.print(fileName);
  Serial.println(":");

  while (fd.available())
  {
   Serial.write(fd.read());
  }
 }
 else
 {
  Serial.print("Error opening ");
  Serial.println(fileName);
 }
 fd.close();
}

void printDateAndTime(void)
{
 if(CARD_OK == 1)
 {
 String currentDate = rtc.stringDateUSA(); //Get the current date in mm/dd/yyyy
    ↪ format (we're weird)
 //String currentDate = rtc.stringDate(); //Get the current date in dd/mm/yyyy
    ↪ format
 String currentTime = rtc.stringTime(); //Get the time
 //currentDate.toCharArray(curDate, str_len);

 previousDay = day;

 currentMonth = currentDate.substring(0,2);
 currentDay = currentDate.substring(3,5);
```

```
 currentCentury = currentDate.substring(6,8);
 currentYear = currentDate.substring(8,10);

 currentMonth.toCharArray(curMon, str_len);
 currentDay.toCharArray(curDay, str_len);
 currentYear.toCharArray(curYear, str_len);
 currentCentury.toCharArray(curCent, str_len);

 month = atoi(curMon);
 day = atoi(curDay);
 year = atoi(curYear);
 century = atoi(curCent);

// Use these to test random dates for debugging
// month = 04;
// day = 27;
// century = 20;
// year = 19;

 calculateDayOfYear(month,day,year,century);

 calculateWeekday(dayOfWeek,firstOfYear);

 previousWeek = currentWeek;
 currentWeek = weekNumber;
 if(currentWeek != previousWeek)
 {
  sprintf(fileName, "%dweek%d.txt", year, weekNumber);
 }

 // DEBUG Step
 DEBUG.println("printing to "); DEBUG.println(fileName);

 fd = SD.open(fileName, FILE_WRITE);
 if (fd)
 {
  if(previousDay != day)
  {
   fd.print(dayName);
   fd.println(":");
  }
  fd.print(F(" The chair was used on "));
  fd.print(currentDate);
  fd.print(F(" at "));
  fd.println(currentTime);
  fd.println();
  fd.flush();
  index = 0;
  fd.close();
 }

 }
 else{
  DEBUG.println("Card not detected, skipping PrintDateAndTime call.");
 }
}
```

```
void readByte(void)
{
 byte byteRead = Serial.read();
 Serial.write(byteRead); // Echo
 buff[index++] = byteRead;

 // Must be 'EOF' to not get confused with words such as 'takeoff' or 'writeoff'
 if (byteRead == 'E' && state == NORMAL)
 {
  state = E;
 }
 else if (byteRead == 'O' && state == E)
 {
  state = EO;
 }
 else if (byteRead == 'F' && state == EO)
 {
  eof();
  state = NORMAL;
 }
}

void calculateDate(String currentDate, String currentTime)
{
 //String currentDate = rtc.stringDateUSA(); //Get the current date in mm/dd/yyyy
    ↪ format (we're weird)
 //String currentDate = rtc.stringDate()); //Get the current date in dd/mm/yyyy
    ↪ format
 //String currentTime = rtc.stringTime(); //Get the time

}

void calculateDayOfYear(int month,int day, int year, int century)
{
  if(month > 2)
 {
  addmonth = month - 2;
  addyear = year;
 }
 else
 {
  addmonth = month + 10;
  addyear = year - 1;
 }

 firstYear = year - 1;
 leapYear = year % 4;

 firstOfYear = (1 + ((13 * 11 - 1) / 5) + firstYear + (firstYear / 4) + (century /
    ↪ 4) - 2 * century) % 7;
 dayOfWeek = (day + ((13 * addmonth - 1) / 5) + addyear + (addyear / 4) + (century /
    ↪  4) - 2 * century) % 7;


 switch(month)
```

```
{
 case 1:
 {
  monthName = "January";
  dayOfYear = day;
  break;
 }
 case 2:
 {
  monthName = "February";
  dayOfYear = 31 + day;
  break;
 }
 case 3:
 {
  monthName = "March";
  dayOfYear = 59 + day;
  break;
 }
 case 4:
 {
  monthName = "April";
  dayOfYear = 90 + day;
  break;
 }
 case 5:
 {
  monthName = "May";
  dayOfYear = 120 + day;
  break;
 }
 case 6:
 {
  monthName = "June";
  dayOfYear = 151 + day;
  break;
 }
 case 7:
 {
  monthName = "July";
  dayOfYear = 181 + day;
  break;
 }
 case 8:
 {
  monthName = "August";
  dayOfYear = 212 + day;
  break;
 }
 case 9:
 {
  monthName = "September";
  dayOfYear = 243 + day;
  break;
 }
 case 10:
```

```
  {
   monthName = "October";
   dayOfYear = 273 + day;
   break;
  }
  case 11:
  {
   monthName = "November";
   dayOfYear = 304 + day;
   break;
  }
  case 12:
  {
   monthName = "December";
   dayOfYear = 334 + day;
   break;
  }
  default:
  {
   monthName = "Something";
   dayOfYear = 366;
   break;
  }
 }

 if(leapYear == 0)
 {
  dayOfYear += 1;
 }
}

void calculateWeekday(int dayOfWeek, int firstOfYear)
{
 weekNumber = (dayOfYear / 7) + 1;

 switch(dayOfWeek)
 {
  case 0:
  {
   dayName = "Sunday";
   break;
  }
  case 1:
  {
   dayName = "Monday";
   break;
  }
  case 2:
  {
   dayName = "Tuesday";
   break;
  }
  case 3:
  {
   dayName = "Wednesday";
   break;
```

```
 }
 case 4:
 {
  dayName = "Thursday";
  break;
 }
 case 5:
 {
  dayName = "Friday";
  break;
 }
 case 6:
 {
  dayName = "Saturday";
  break;
 }
}



switch(firstOfYear)
{
 case 0:
 {
  firstDay = "Sunday";
  break;
 }
 case 1:
 {
  firstDay = "Monday";
  break;
 }
 case 2:
 {
  firstDay = "Tuesday";
  break;
 }
 case 3:
 {
  firstDay = "Wednesday";
  break;
 }
 case 4:
 {
  firstDay = "Thursday";
  break;
 }
 case 5:
 {
  firstDay = "Friday";
  break;
 }
 case 6:
 {
  firstDay = "Saturday";
  break;
```

```
  }
 }
}

void printDateAndTimeVerbose(void)
{
 String currentDate = rtc.stringDateUSA(); //Get the current date in mm/dd/yyyy
    ↪ format (we're weird)
 //String currentDate = rtc.stringDate(); //Get the current date in dd/mm/yyyy
    ↪ format
 String currentTime = rtc.stringTime(); //Get the time
 //currentDate.toCharArray(curDate, str_len);

 previousDay = day;

 currentMonth = currentDate.substring(0,2);
 currentDay = currentDate.substring(3,5);
 currentCentury = currentDate.substring(6,8);
 currentYear = currentDate.substring(8,10);

 currentMonth.toCharArray(curMon, str_len);
 currentDay.toCharArray(curDay, str_len);
 currentYear.toCharArray(curYear, str_len);
 currentCentury.toCharArray(curCent, str_len);

 month = atoi(curMon);
 day = atoi(curDay);
 year = atoi(curYear);
 century = atoi(curCent);

// Use these to test random dates for debugging
// month = 04;
// day = 27;
// century = 20;
// year = 19;

 calculateDayOfYear(month,day,year,century);

 calculateWeekday(dayOfWeek,firstOfYear);

 previousWeek = currentWeek;
 currentWeek = weekNumber;
 if(currentWeek != previousWeek)
 {
  sprintf(fileName, "%dweek%d.txt", year, weekNumber);
 }


 fd = SD.open(fileName, FILE_WRITE);
 if (fd)
 {
  if(previousDay != day)
  {
   fd.print(dayName);
   fd.println(":");
  }
```

```
  fd.print(F(" The chair was used on "));
  fd.print(currentDate);
  fd.print(F(" at "));
  fd.println(currentTime);
// this is what is added in the verbose
  fd.print(F("Month: "));
  //fd.print(currentMonth);
  //fd.print(month);
  fd.print(monthName);
  fd.print(F(" Day: "));
  //fd.print(currentDay);
  fd.print(day);
  fd.print(F(" Year: "));
  //fd.println(currentYear);
  fd.print(century);
  fd.println(year);
  fd.print(F("The day of week is: "));
  fd.print(dayName);
  fd.print(F(" The first day of the year was a: "));
  //fd.print(firstOfYear);
  fd.print(firstDay);
  if(leapYear == 0)
  {
   fd.println(F(" It is a leap year"));
  }
  else
  {
   fd.println(F(" It is not a leap year"));
  }
  fd.print(F("Day of Year: "));
  fd.print(dayOfYear);
  fd.print(F(" Week of Year: "));
  fd.println(weekNumber);
  fd.println();
  fd.flush();
  index = 0;
  fd.close();
 }
}
```

Listing 11: The Sensor Functions

```
#include "definitions.h"

/*******Sensor notes********/
/*
 The PIR sensor is highly adjustable on the back. It will also, if the LED is
    ↪ enabled, blink upon startup and take a moment to initialize before use.




*/
int SetupButtonInterrupts(){
 pinMode(LEFT_HANDLE_BUTTON, INPUT); // need to configure for button interrupt
```

```
 pinMode(RIGHT_HANDLE_BUTTON, INPUT);
 attachInterrupt(digitalPinToInterrupt(LEFT_HANDLE_BUTTON), GoToThankYouLeft, RISING
    ↪ );
 attachInterrupt(digitalPinToInterrupt(RIGHT_HANDLE_BUTTON), GoToThankYouRight,
    ↪ RISING);
 return 0;
}

int EnableButtonInterrupts(){
 attachInterrupt(digitalPinToInterrupt(LEFT_HANDLE_BUTTON), GoToThankYouLeft, RISING
    ↪ );
 attachInterrupt(digitalPinToInterrupt(RIGHT_HANDLE_BUTTON), GoToThankYouRight,
    ↪ RISING);
}

int DisableButtonInterrupts(){
 detachInterrupt(digitalPinToInterrupt(LEFT_HANDLE_BUTTON));
 detachInterrupt(digitalPinToInterrupt(RIGHT_HANDLE_BUTTON));
}


int SetupSensors() {

 // for lowerSensor
 pinMode(PIR_DOUT,INPUT);



 if (myDistance_upper.begin() == false){
  Serial.println("Upper RF Sensor failed to initialize. Check wiring.");
  while(1); // freezes
 }

 DEBUG.println("Sensors Up!");
 return 0;
}


int LowerSensorTakeMeasurement() {

 int motion_sense;
 motion_sense = digitalRead(PIR_DOUT);
 // maybe have some sort of filtering/window in this part of the function. Hold off
    ↪ for the moment
 return motion_sense;
 // return decision
}

int UpperSensorTakeMeasurement() {
 // using RFD77402
 int distance;
 myDistance_upper.takeMeasurement();
 distance = myDistance_upper.getDistance();
 // DEBUG.println(distance);
 // distance returned on a scale of 0 to 2047
 if(distance < 650) { // empirically determined that it's a proximal signal.
```

```
  return 2;
 }
 else if(distance < 1250) { // this is what we are considering someone close by
  return 1;
 }
 else {
  return 0;
  // no signal detected, or the signal is sufficiently far away
 }


}
```

## 6.2 URL References

- BLYNK Main Document: `https://docs.blynk.cc/`
- LED Hookup Guide: `https://learn.sparkfun.com/tutorials/lp55231-breakout-board-hookup-guide/all`