Adventure Kit 2 (AI Apocalypse) Guidebook

Evan Robinson
2023

Version of 2023-10-15

Table of Contents

Circuit Diagrams

Parts Lists

Code Blocks

Prologue

How did I get here?

I joined a couple of Facebook groups, in particular Adventure Kit 2: Day 0. That's how. I thought I might be able to gently mentor some people in the writing of code for Arduino systems.

And in turn be mentored in the creation of circuitry to for which to write code.

So, when I started the inventr.io Adventure Kit 2 (AI Apocalypse) (hereinafter "Kit 2"), I thought I'd post my progress periodically.

When nobody objected, I continued. Right now, I'm working on Chapter 01, section[1] 4: Solar Simulation Shenanigans[2].

And I'm starting to collect the posts in this document, which I will (probably) eventually post to the Facebook group. I seriously doubt it's going to be worth actually publishing it.

To a large extent, the contents of this document will be the same as the posts I have put in the Facebook group

[1] Unlike Adventure Kit 1: 30 Days Lost in Space, each project in Kit 2 isn't conveniently identifiable by a Globally Unique Identifier (GUID), so I have referred to each individual post in the Course Content as a "section". I hope that's clear enough.

[2] Yes, in the Course Content it's spelled Shinanigans. They're wrong.

To Do List

Chapter 01: Moving In

- Do these antiques still work:
    - Install System
    - Run Blink successfully
- Bad Wiring Systems (Fixing the Lights)
    - Control an LED with the HERO (on/off)
- Easy Light Toggles (adding button inputs)
    - Set light state using momentary on/off button (down on/up off)
    - Use button to signal Hero to turn light off/on – press changes light from on to off to on
- Solar Simulation Shenanigans (power grid issues)
    - Use photoresistor to simulate solar charging (more light, more charge)
    - Maintain "battery level"
        - Increase with solar input
        - Decrease with time light is on
        - Only charge if level is below a certain level
        - Turn light off when battery power level drops below a different level
        - *Extend to Add Green LED indicating 95%+ charge*
        - *Extend to Add Red LED indicating 25%- charge*
        - *Extend to Add always-on display to show current battery level and solar power*
- 404 Error: Alarms not found (buzzer)
    - Use buzzer to announce battery low power level
    - *Provide different alert sounds for low, critical, and general*
- Dim the Lights (potentiometer & PWM)
    - Use a potentiometer to set power level
    - Set light intensity (via PWM) according to current desired light power level

Chapter 02: Base Security 101

- Motion Sensor Security System
    - Hook up PIR Motion Sensor
    - Connect PIR Motion Sensor to White LED (simulated floodlight)
    - Button to Turn Floodlight off
    - Connect PIR Motion Sensor to Red Alert LED
    - After delay, Flash LED
    - After delay, Audible Alarm
    - Button to Turn Alert Off
- Keypad Door Lock
    - Connect KeyPad for Input
    - Connect LCD display for Text Output
    - Add Brightness Pot to control LCD display brightness
    - Set PIN in code
    - Detect PIN input by keypad
    - Identify correct PIN and display success on LCD Display

    - Identify incorrect PIN and display failure on LCD Display
    - Add Audible tones for success and failure
    - Add Red/Green LED for success failure indicator
    - After 3 failed attempts, do not take additional input for set time
- NFC Badges
    - 3.3V!!!
    - Add RFID reader
    - Distinguish between multiple users
        - Add multiple fobs
        - Different levels of access
        - Different responses (name?)
    - Audio access/access denied tone
    - Red/Green LED access denied/access permitted
- RTTTL Alarm
    - Switch Alarms to use tone "music"
        - Low power
        - Critical power
        - Access permitted
        - Access denied
        - Fob vs keypad?
    - Get "super secret message" for later

Chapter 03: Greenhouse

- Forgetting to water the garden again (Dry Plant Warning System)
    - Add Greenhouse object
        - Status Light & Audio
        - Water Level Measure
        - Status Light color AND brightness (PWM)
- Heat Management Pt. 1 – Fan Ventilation System Simulation
    - Greenhouse ventilation
    - Temperature sensor
    - Fan speed (PWM)
    - *Extend to Display temperature and humidity*
- Heat Management Pt. 2 – Automatic Fan System Failure (Power draw too high – Relays)
    - Add Relay
    - Add external power supply (OLED?)

Chapter 04: Daily Life Essentials

- Accurate Alarm Clock
    - Add RTC
        - 7 segment display
        - Audible alert
        - Clock board
    - Display current time
    - Display/set alarm time
    - *Extend to add Multiple Alarms*

    - *Extend to Time Zones? UI to select*
    - *Extend to Brightness Level?*
- Infrared Smart Lights
    - IR Receiver
    - IR Remote
    - Add light flicker indicator for receiving remote signal?
    - Display code received
    - Add RGB LED
    - Attach R,G,B on/off to remote buttons
    - *Add Remote buttons:*
        - *turn interior lights on/off*
        - *turn floodlight (exterior light) on/off*
        - *turn greenhouse fan on/off*
        - *silence audible alarms*
        - *Status Display: battery, solar, motion sensor status, most recent entry, current time, greenhouse water status, greenhouse fan status, next alarm time,*
- Clap Lights
    - Add audible sensor
    - Detect clap
    - Interior Lights on/off with clap
    - Detect multiple claps
    - Interior/exterior lights on/off with different claps
        - 1 clap: interior Lights
        - 2 claps: exterior lights

Chapter 05: Phoenix Restoration (Resistance Group for Humanity)

- There are other survivors. Getting Started T-Display (and discovering others exist!)
    - Hook up and test T Display
- The other survivors share their knowledge – Time to fight back! (Advanced T-Display Networking/Communication)
    - Set up WiFi Access Point: BanzaiInstitute
    - Display IP address for TDisplay
    - Connect device to Wifi: BanzaiInstitute
    - Access IP address from above
    - Turn Light on/off
    - *Extend to Red/Green LED: on/off/color*
    - *Extend to include Password for WiFi: BBQS@uce*

Chapter 06: Base Security++ (Radar System)

- Automatic 180 Degree Sweep Radar Upgrade
    - Re-wire other systems to accommodate Touch Screen Board
    - Wire up Touch Screen & Ultrasonic Sensor
    - Add Servo
    - Set Servo to scan 180°

    - Display detected objects
    - *Color code changes*
    - *Add UI to override servo position*
        - *Single point*
        - *Scan endpoints*
    - *Add Alarm indicator for detected item of concern*
- False Signals – RGB Turret w/LCD Touch Screen
    - 9V battery for power
    - Joystick wired for X only
    - Detect joystick button
    - Stepper Motor with RGB LED “gun”
    - Random color
    - *Test separate button for “Fire”*
- False Signals 2 – RGB Turret w/T-Display
    - Duplicate above using T-Display
        - 9V battery for power
        - Joystick wired for X only
        - Detect joystick button
        - Stepper Motor with RGB LED “gun”
        - Random color
        - *Test separate button for “Fire”*

Chapter 07: Showdown Against The AI

- Official Victory Signal Flare (Finale!)
    - SSID
    - SSID_PASSWORD
    - USER_EMAIL
    - API_KEY
    - Make the animation happen
    - Check on the map

Chapter 01: Moving In

Section 01: Do these antiques still work?

I'm leaving this one for the inventr.io people to manage. It is how you get all the software downloaded and connect up your HERO board and make sure you know how to compile code, upload it to the board, and make the built in LED light blink to your command.

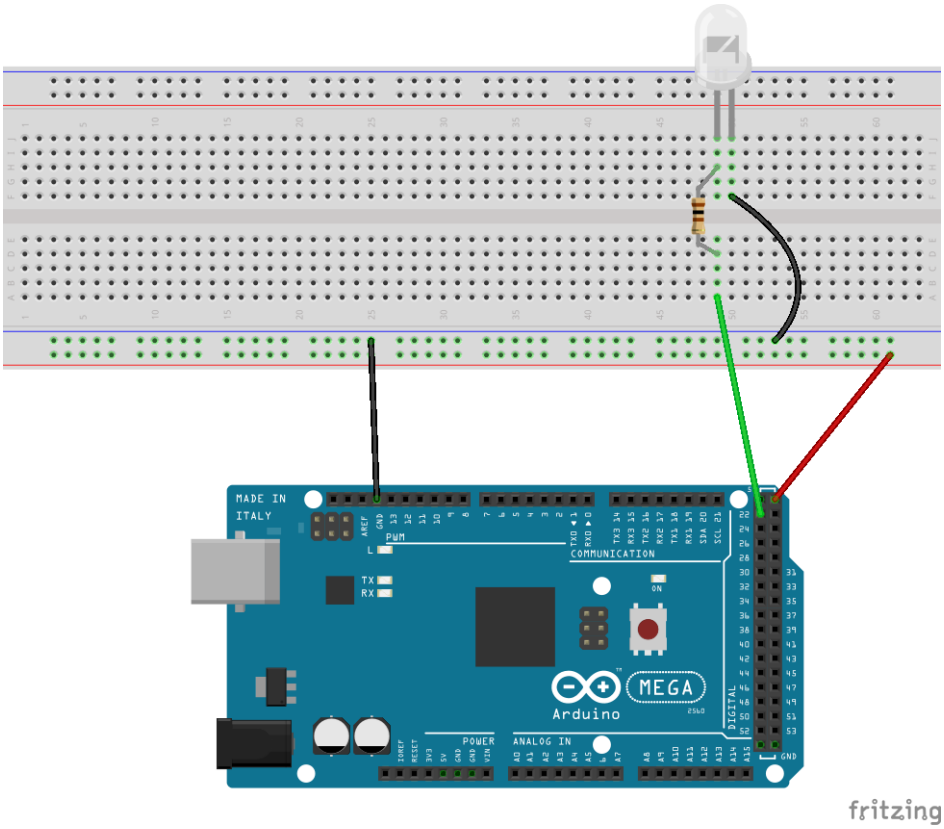
Section 02: Bad Wiring Systems

ADDED: I forgot to say that I'm posting these because I hope they will be helpful to people with less experience than I have coding. It's entirely possible I'm wrong, in which case somebody tell me and I'll stop.

I'm also posting these because I'm almost a complete noob at the electronics, and I'm trying to work them from scratch here (still using searches to find things out, but I'm not just copying any circuit that is provided by inventr). So I imagine that at some point somebody will say (roughly) "WTF are you doing? Do it this way!" -- and I'm very happy with that idea.

Feel free to disagree with me. To quote Bull Durham, "I believe what I believe", and I'm not going to insist that y'all agree.

My spec: Control an LED with the HERO (on and off).



Circuit 1: Section 2: Bad Wiring Systems

White LED
Arduino Mega 2560 (Rev3)
100Ω Resistor

Table 1: Circuit 1 Parts List

Code:

```
#include <Arduino.h>

const uint8_t whiteLEDControlPin = 22;
const int ledOnIndefintely = 0;

// put function forward declarations here:
void turnLEDOnFor(int millis = 1000, uint8_t pin = whiteLEDControlPin);
void turnLEDOff(uint8_t pin = whiteLEDControlPin);

void setup() {
  pinMode(whiteLEDControlPin, OUTPUT);
}

#define THROW_AWAY_CODE[3]

void loop() {
#ifdef THROW_AWAY_CODE
  digitalWrite(whiteLEDControlPin, HIGH);
  delay(1000);
  digitalWrite(whiteLEDControlPin, LOW);
  delay(1000);
#else
  turnLEDOnFor(300);
  delay(300);
#endif
}

// put function definitions here:
void turnLEDOnFor(int millis, uint8_t pin) {
  digitalWrite(pin, HIGH);
  if (millis != ledOnIndefintely) {
    delay(millis);
    turnLEDOff();
  }
}
```

[3] An explanation of the #ifdef THROW_AWAY_CODE ... #else ... #endif:
The four lines of code between #ifdef THROW_AWAY_CODE and #else are only compiled if the macro THROW_AWAY_CODE has been defined earlier in the source file.
As the code is, THROW_AWAY_CODE has been defined, and those four lines will be executed. So far as I'm concerned, that is the simplest solution to this particular spec.
HOWEVER, as I plan to use this code as the basis for future code, I have already refactored the code into what I consider a more obvious, understandable, usable, and extendable form.
I may be wrong about any one or more of those adjectives: I may find in the future that this code (the code outside the THROW_AWAY_CODE macro region) is less obvious, less understandable, less usable, and/or less extendable. But it's my current judgement that I'm doing future me a favor here.

```
void turnLEDOff(uint8_t pin) {
  digitalWrite(pin, LOW);
}
```

Code Block 1: Section 2: Bad Wiring Systems

Commentary:

1. Use of 'const' instead of #define

   I prefer using 'const int constantName = XX' to '#define CONSTANT_NAME XX primarily because I've been programming for long enough that I've seen other people make a lot of mistakes, I've made a lot of mistakes, and I've used enough tool sets that had bugs in them that I dislike #define.

   Why?

   Because #define is a *textual substitution*, not a syntactic or semantic substitution. Those are big words that means #define is like doing a search/replace on your code. If you've ever tried to, say, replace each use of 'end' in a file with 'END', you will have discovered that you have oddities like 'sEND' and 'trEND' which you did not intEND[4].

   Furthermore, #define can be used to create more complex substitutions, such as '#define MAX(x, y) x > y x : y' which replaces 'MAX(x,y) with a complex expression (the c++ 'ternary operator') which almost certainly needs to be fully parenthesized[5] to deliver what the author intended.

   Because such substitutions happen in the scanning and tokenizing section of the compiler (I know, way more than you want to hear), the code as compiled is not easily available to the programmer, which means the bug is **hard to find**.

   Not Good.

2. Use of default parameters

   The 'int millis = 1000, uint8_t pin = whiteLEDControlPin' portion of the forward declarations uses something called 'default parameters'. Essentially, if I call the function turnLEDOnFor() without any parameters, it uses these. If I call it with a single parameter, it is as though I gave it *only the first* parameter (millis). Thus by passing whiteLEDControlPin as a default parameter, I preserve my option to control any LED with turnLEDOnFor() and turnLEDOff(), but I don't need to provide the pin parameter every time I call them.

---

[4] This is a common enough error that Word repeatedly replaced 'intEND' with 'intend' while I was writing this.

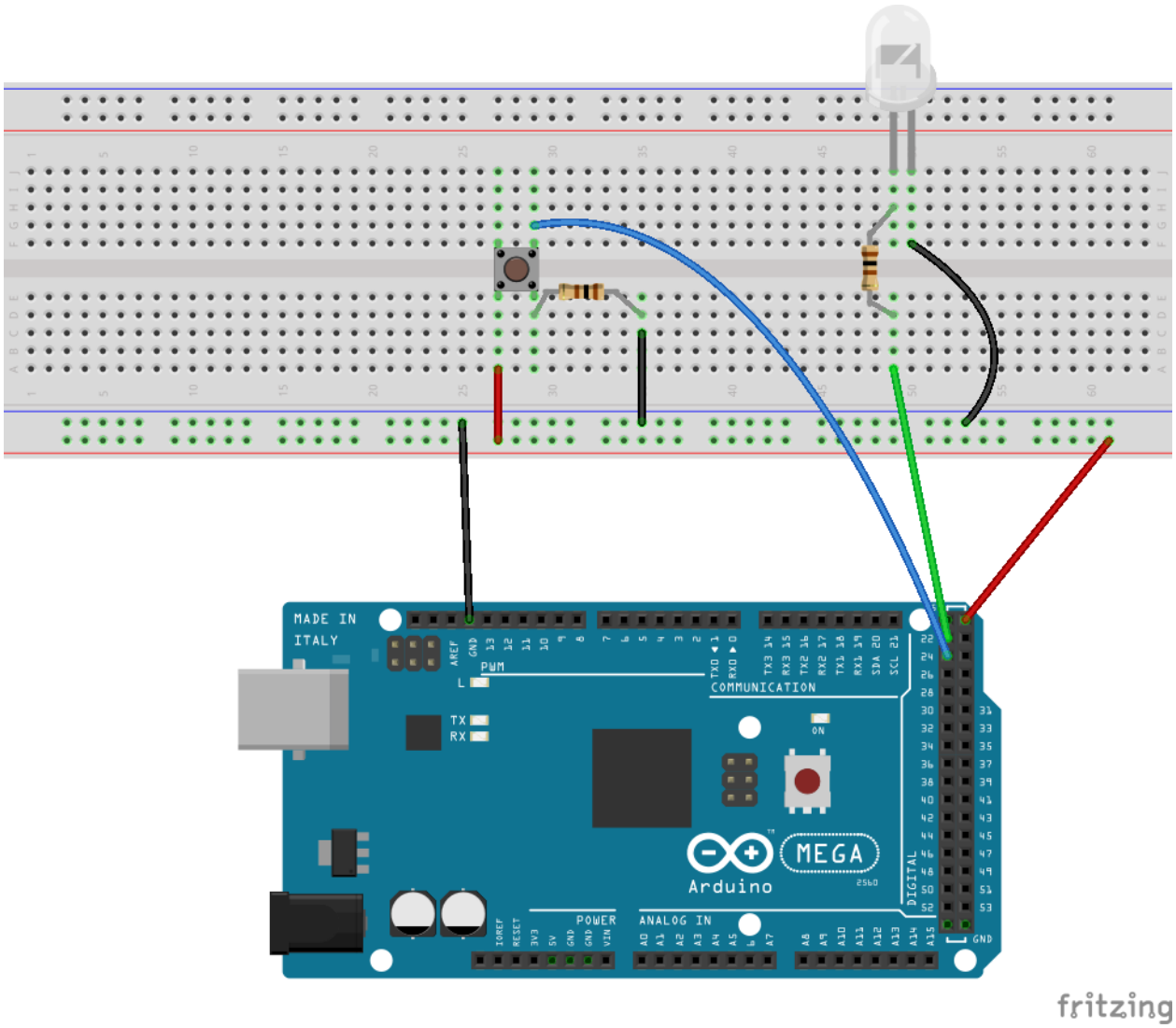
[5] #define MAX(x, y) (((x) > (y)) (x) : (y)): which is hardly the most readable thing in the world – although I've seen worse.

And if either 'x' or 'y' are not simple identifiers or numeric constants but instead are *statements*, it gets worse fast. Consider, just for a moment, what happens in this MAX macro if x is 'counter++', which may be textually substituted in twice, so you *might* get two increments instead of one. But that's not the real problem. The real problem is that you didn't intend for the values 'passed' to MAX(x, y) to be *effected at all*, much less twice.

Section 03: Easy Light Toggles

My spec:

1) toggle light state using momentary on/off button (directly);
2) use button to signal Hero to toggle light from off to on and back



Circuit 2: Section 3: Easy Light Toggles

2	100Ω Resistor
1	Arduino Mega 2560 (Rev3)
1	LEDs
1	Pushbutton

Table 2: Circuit 2 Parts List

Code:

```
#include <Arduino.h>

const uint8_t whiteLEDControlPin = 22;
```

```
const uint8_t buttonInputPin = 24;
const int ledOnIndefintely = 0;
const long debounceInterval = 50L;
const int buttonPressed = 1;

// put function forward declarations here:
void turnLEDOnFor(int millis = 1000, uint8_t pin = whiteLEDControlPin);
void turnLEDOff(uint8_t pin = whiteLEDControlPin);
void toggleInteriorLights(void);

void setup() {
  pinMode(whiteLEDControlPin, OUTPUT);
  turnLEDOff();
  pinMode(buttonInputPin, INPUT);

  Serial.begin(9600);
  Serial.println("setup complete");
}

void loop() {
  static int previousPinValue = 0;
  static long previousDebounceTime = millis();
  int pinValue = digitalRead(buttonInputPin);

  if ((millis() - previousDebounceTime) < debounceInterval) {
    // debouncing: if the transition was less than debounceInterval ms ago, ignore it
    return;
  }

  if (previousPinValue != pinValue) {
    previousPinValue = pinValue;
    if (pinValue == buttonPressed) {
      toggleInteriorLights();
    }
  }
}

// put function definitions here:
void toggleInteriorLights(void) {
  static bool lightIsOn = false;

  if (lightIsOn) {
    turnLEDOff();
    lightIsOn = false;
  }
  else {
    turnLEDOnFor(ledOnIndefintely);
    lightIsOn = true;
```

```
  }
}

void turnLEDOnFor(int millis, uint8_t pin) {
  digitalWrite(pin, HIGH);
  if (millis != ledOnIndefintely) {
    delay(millis);
    turnLEDOff();
  }
}

void turnLEDOff(uint8_t pin) {
  digitalWrite(pin, LOW);
}
```

Code Block 2: Section 3: Easy Light Toggles

Commentary:

1. 'toggleInteriorLights()'

   This is the first indication that the code is going to become more Literate. Everything else in the code here and that we've seen so far is very hardware specific. Very much in the Arduino/electronics domain, if you will. The code that runs the lighting refers to LED and lots of variable names reference 'pin' of one sort or another.
   What I mean by "in the Arduino … domain" is that the code is not talking about the fantasy inventr.io is providing us. Instead, its talking about the microcontroller and the things on the breadboard.
   The next Section, Solar Simulation Shenanigans, will change this in a big way by *refactoring the code into the problem domain*.

2. Debouncing

   "Debouncing" a key or button refers to managing the moments where it is so close to closing or opening that the value presented by the electronics is indeterminate or changing back and forth so quickly that it may be 0 one millisecond, 1 the next, and 0 again the one after that.
   Humans do not operate at that speed.
   So we introduce code to make sure that the state of the switch is stable for some period of time that humans can operate at. In this code, the simplest possible debouncing: ignoring any state change in the switch return value for at least 50ms. It's not perfect, but for the learning experience it is good enough.

Interlude: Refactoring

Refactoring is restructuring your code to make it better.

What does "better" mean?

For my purposes, it means "easier to understand". Because that means "easier to change", "easier to reuse", "easier to fix", and "easier to build upon".

You will find a variety of definitions online for the word "refactoring". Common among them is the idea that this change does not change the underlying function, and that the changes are made systematically – sometimes to the extent of following a bunch of rules.

I use the word somewhat loosely.

I was introduced to the word "refactoring" in 1999, when I worked at Adobe Systems in the core technology group. Jon Reid, one of my team, got very interested in the book when it came out. As I was no longer primarily a programmer, I got the gist but haven't ever done the formal process.

In this case, there are going to be two major things happening to the code: I'm going to introduce some classes (real Object Oriented Programming stuff!); and I'm going to rename and re-organize the code from the hardware/microcontroller oriented names and organization into what I call the "*problem domain*[6]" – instead of building circuits and writing code to make electronic components do certain things, I'm going to start creating a "post-apocalyptic dwelling".

So no more referring to the lights as "LEDLights" or anything like that. These lights are the *interior lights* of our house. The button isn't constantly referred to using the buttonInputPin, now it's called the *interiorLightsButton*.

---

[6] The first time I remember using this specific term, I was a Technical Director at Electronic Arts providing advice to the team that was designing and building Madden Football for the 3DO console. We were discussing how the developers should handle kickoffs, and the designer was talking about force applied to the football and vectors and directions.

I stopped him and said (approximately) "you want the code to mirror the problem domain. How does a kicker think about kicking the football? They don't say to themselves 'I'm gonna hit this ball with 72 Newtons of force in a horizontal vector of 22° to the right of center at an upward angle of 42.5°'. They say to themselves 'kick away to the right side', 'onside straight ahead looking for a pooch up', or 'I want the ball to bounce straight up near the right sideline and goal line".

These last three descriptions exist in the world of *football*, not the world of *physics*. Yes, they will eventually get translated to the world of physics, but that's not how they start.

Classes:

led.h / led.cpp:

The LED class encapsulates the basic functioning of an LED on the breadboard, with a pulldown resistor (as opposed to a pullup resistor – made evident by the use of HIGH as on and LOW as off). It provides a *constructor* (used to create the object, bind it to the relevant pin, and set the necessary pinMode), *methods* (functions) to turn the LED on and off, and a *method* to determine whether the LED is currently on.

Classes in c++ (opinions differ as to whether the Arduino language is a full implementation of c++ or not. I have no opinion) are generally *declared* (described from an external point of view) in a .h file and *defined* (described sufficiently to create executable code) in a .cpp file.

Classes are the basic unit of *inheritance* (the re-use of code by creating *subclasses* or derivative classes which extend their *superclass* or ancestor class). Effectively that means that I can create a subclass of LED which would have the same public interface but run different code where necessary (if creating a class to control an RGB LED, or a BiColor LED). The public interface can be extended to include additional necessary functionality (as to set the color of an RGB LED).

```
/*
    led.h
    Evan Robinson, 2023-09-30

    Class to manage LED on arduino
    Presumes HIGH is on, LOW is off
*/

#ifndef led_h
#define led_h

#include <Arduino.h>

class LED {
    public:
        LED(uint8_t pin);
        void turnOn();
        void turnOff();
        bool isOn();
    private:
        uint8_t _pin;
        bool _isOn;
};

#endif
```

Code Block 3: Interlude: led.h (the declaration of the class LED)

```
/*
    led.cpp
    Evan Robinson, 2023-09-30
```

```
    Class to manage LED on arduino
    Presumes HIGH is on, LOW is off
*/

#include "led.h"
#include <Arduino.h>

LED::LED(uint8_t pin) {
    _pin = pin;
    pinMode(_pin, OUTPUT);
    turnOff();
}

void LED::turnOn() {
    digitalWrite(_pin, HIGH);
    _isOn = true;
}

void LED::turnOff() {
    digitalWrite(_pin, LOW);
    _isOn = false;
}

bool LED::isOn() {
    return _isOn;
}
```

Code Block 4: Interlude: led.cpp (the definition of the class LED)

button.h / button.cpp:
The Button class encapsulates the basic function of a momentary on push button (or switch), again with a pulldown resistor (so HIGH is pressed, LOW is not pressed).

```
/*
    button.h
    Evan Robinson, 2023-09-30

    Class to manage momentary button on arduino
    Presumes HIGH is pressed, LOW is not pressed
    Press occurs on down press of button
*/

#ifndef button_h
#define button_h

#include <Arduino.h>

class Button {
    public:
```

```
        Button(uint8_t pin);
        bool isPressed();
        bool hasChanged();
        int value();
    protected:

    private:
        uint8_t _pin;
        int _value;
        int _valueHasChanged;
};

#endif
```

Code Block 5: Interlude: button.h (the declaration of the class Button)

Note that there is only one method where the value of the push button is read from the hardware. This guarantees that any necessary housekeeping will be done no matter how the user programmer accesses the value.

```
/*
    Button.cpp
    Evan Robinson, 2023-09-30

    Class to manage momentary button on arduino
    Presumes HIGH is pressed, LOW is not pressed
    Press occurs on down press of button
*/

#include "button.h"
#include <Arduino.h>

Button::Button(uint8_t pin) {
    _pin = pin;
    pinMode(_pin, INPUT);
    _value = value();
    _valueHasChanged = false;
}

bool Button::isPressed() {
    _value = value();
    return (_value == HIGH);
}

int Button::value() {
    // TBD: Debounce
    int currentValue = digitalRead(_pin);
    _valueHasChanged = (_value != currentValue);
    // if (_valueHasChanged) {
    //     Serial.println(currentValue);
```

```
    // }
    return digitalRead(_pin);
}

bool Button::hasChanged() {
    return _valueHasChanged;
}
```

Code Block 6: Interlude: button.cpp

```
/*
  Simulated post-apocalyptic dwelling

  Supports simulated interior lighting (interiorLights)
    controlled by a momentary switch (interiorLightsButton)
    which toggles lights on and off
*/
#include <Arduino.h>
#include "button.h"
#include "led.h"

// Hardware values
const uint8_t whiteLEDControlPin = 22;
const uint8_t buttonInputPin = 24;

// Dwelling Contents
Button interiorLightsButton = Button(buttonInputPin);
LED interiorLights = LED(whiteLEDControlPin);

// Forward Declarations
void interiorLighting(void);

// Arduino Setup
void setup() {
}

// Arduino Loop
void loop() {
  // Loop every 1/10th seconds
  delay(100);

  interiorLighting();
}

// Local Functions
void interiorLighting() {
  // Turn interiorLights on and off
  if (interiorLightsButton.isPressed()) {
    if (interiorLightsButton.hasChanged()) {
```

```
        if (interiorLights.isOn()) {
          Serial.println("Lights Off");
          interiorLights.turnOff();
        }
        else {
          Serial.println("Lights On");
          interiorLights.turnOn();
        }
    }
  }
}
```

Code Block 7: Interlude: main.cpp

Although the file is still called main.cpp, it bears little resemblance to its predecessor. The pin definitions are the same, but no setup happens in setup() anymore because it's all handled by the constructors under the // Dwelling Contents comment.

The interior lights are now fully managed by a single function call in loop(): interiorLighting(); interiorLighting() asks interiorLightsButton if it 'isPressed'. If it is, and it 'hasChanged', then it's time for us to toggle the interior lights from on to off or off to on.

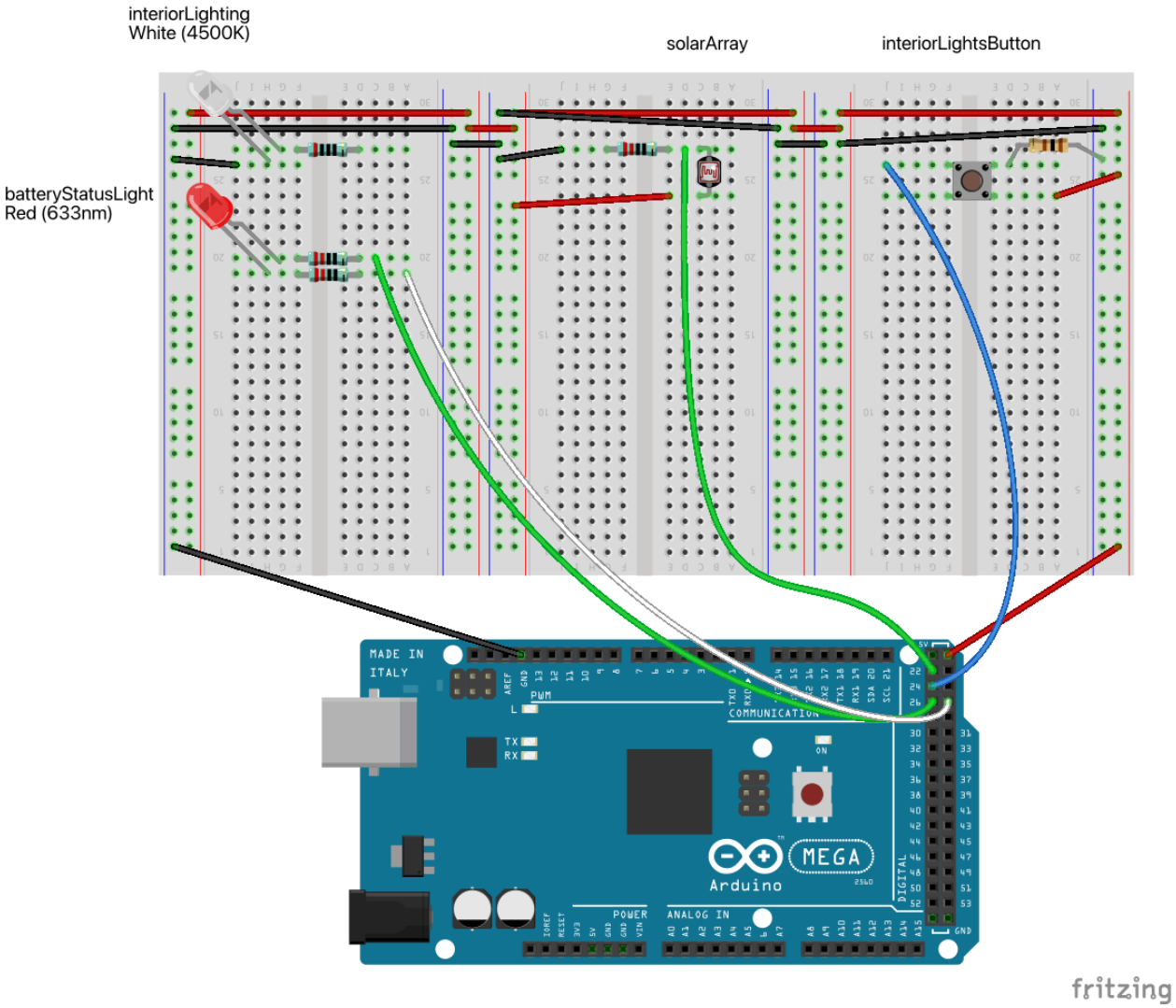
We are now in the problem domain, doing things to our simulated dwelling instead of our real physical circuitry.

Now it's time to add the simulated battery and solar charging systems.

Section 04: Solar Simulation Shenanigans

The Spec:

- Use photoresistor to simulate solar charging (more light, more charge)
- Maintain "battery level"
    - Increase with solar input
    - Decrease with time light is on
    - Only charge if level is below a certain level
    - Turn light off when battery power level drops below a different level
    - Add Green LED indicating 90%+ charge
        - Blink LED from 80%-90%
    - Add Red LED indicating 25%- charge
        - Blink LED from 10%-25%



Circuit 3: Section 4: Solar Simulation Shenanigans

Amount	Part Type

1	100Ω Resistor
1	200Ω Resistor
3	220Ω Resistor
1	Arduino Mega 2560 (Rev3)
1	PHOTOCELL
1	Pushbutton
1	Red/Green BiColor LED
1	White (4500K) LED

Table 3: Circuit 3 Parts List

Code:

```
/*
  Simulated post-apocalyptic dwelling

  Supports simulated interior lighting (interiorLights)
    controlled by a momentary switch which toggles lights
    on and off
  Simulated electrical storage (electricalStorage) and
    solar charger (power level supplied by photoresistor)
  Lighting now simulates a power drop when on
*/
#include <Arduino.h>
#include <math.h>
#include "button.h"
#include "led.h"
#include "power.h"
#include "photoresistor.h"

// Hardware values
const uint8_t whiteLEDControlPin = 22;
const uint8_t buttonInputPin = 24;
const uint8_t photoResistorInputPin = 0;

// Timing constants
const unsigned long oneTenthOfASecond = 100L; // one 'tick'
const int ticksPerLighting = 1;               // lighting input happens every tick
const int ticksPerCharging = 10;              // charging happens every second

// Dwelling Contents
Button interiorLightsButton = Button(buttonInputPin);
LED interiorLights = LED(whiteLEDControlPin);
Power electricalStorage = Power(photoResistorInputPin);

const double interiorLightsPowerUsage = 3.0;

// Forward Declarations
```

```
void interiorLighting(void);
void batteryChargingAndUsage(void);

// Arduino Setup
void setup() {
  Serial.begin(9600);
  Serial.println("setup complete");
}

// Arduino Loop
// Instead of using delay(), millis() is used to enforce a timing 'tick' of
// 1/10 of a second (oneTenthOfASecond).
void loop() {
  static int tickCount = 0;
  static unsigned long previousMillis = 0L;
  unsigned long currentMillis = millis();
  if ((currentMillis - previousMillis) < oneTenthOfASecond) {
    return;
  }
  previousMillis = currentMillis;
  tickCount++;
  electricalStorage.tick();

  if ((tickCount % ticksPerLighting) == 0) {
    interiorLighting();
  }

  if ((tickCount % ticksPerCharging) == 0) {
    batteryChargingAndUsage();
  }
}

// Local Functions
void interiorLighting() {
  // Turn interiorLights on and off using button
  if (interiorLightsButton.isPressed()) {
    if (interiorLightsButton.hasChanged()) {
        if (interiorLights.isOn()) {
          Serial.println("Lights Off");
          interiorLights.turnOff();
        }
        else if (!electricalStorage.isCritical()) {
          if (electricalStorage.isLow()) {
            Serial.println("Lights On But DIMMED");
            // TBD: turn them on dim if batteryLevel
             //  is between critical and warning levels
             interiorLights.turnOn();
          }
```

```
          else {
            Serial.println("Lights On");
            interiorLights.turnOn();
          }
        }
    }
  }

  // manage lights depending upon current power levels
  if (interiorLights.isOn()) {
    if (electricalStorage.isCritical()) {
      interiorLights.turnOff();
      Serial.println("Power Critical, Lights Off");
    }
    else if (electricalStorage.isLow()) {
      // TBD: dim interiorLights
      Serial.println("Lights are on, but dimmed");
    }
  }
}

void batteryChargingAndUsage() {
  static double previousBatteryLevel = electricalStorage.batteryLevel();

  electricalStorage.chargeBattery();

  // account for interiorLights power usage
  if (interiorLights.isOn()) {
    electricalStorage.usePower(interiorLightsPowerUsage);
  }

  // send battery power level to serial
  // TBD: Remove when this info is sent to hardware display instead of Serial
  if (previousBatteryLevel != electricalStorage.batteryLevel()) {
      electricalStorage.showStatus();
  }
  previousBatteryLevel = electricalStorage.batteryLevel();
}
```

Code Block 8: Section 4: Refactored main.cpp showing object use instead of working directly with the hardware

Commentary:

1. In this image, I used a RED LED to indicate a BiColor Red/Green LED (with two leads) because fritzing does not have a BiColor Red/Green LED in the standard parts bin. Such LEDs are cheaply available (without a data sheet) from Amazon.[7]
2. I've gone a little far afield of the instructions here. There are a LOT of TBDs (To Be Dones) in comments (you can search for them if you have a helpful IDE), which mostly have to do with adding hardware I haven't included here: a buzzer for audible alerts; a supplementary display to give you continuous information on your electrical use and solar collection; dimming the simulated interior lights by using PWM (Pulse Width Modulation) on the white LED. Not only did I add a completely new electrical system status light, I bought extra hardware to make it use only one LED – but it wouldn't be hard to wire it up using two: a red and a green – it wouldn't even use any more resistors.
3. In line with the Interlude of Refactoring, I now have objects for hardware like LEDs (both single and bicolor), the button[8], and the photoresistor, but *also* for objects within the simulation like the Power subsystem, the photoresistor which appears in the code as the _solarArray within the Power subsystem, and the LED which is now the interiorLights object.[9] More and more things are appearing in the problem domain, which is, IMO, good for understanding the system.
4. I have switched my development IDE from Arduino to PlatformIO (using Visual Studio Code). I'm pretty happy with it. Immediately after writing this section (or perhaps during it), I will be moving the code from separate directories on my desktop for each section into git and probably into GitHub, since I move around a certain amount and it will be inconvenient to try and keep multiple systems synced without using tools to do so.

---

[7] I bought these for about 8¢ (US) apiece. I sacrificed two to figure out how they worked and what size resistor would work (220Ω, IIRC – I tried to make the fritzing image correct). The trick is that you apply HIGH to one pin and LOW to the other to get one color, then swap the current flow to get the alternate color. In this context RED means bad, GREEN means good, and blinking means not quite so bad/good as solid.

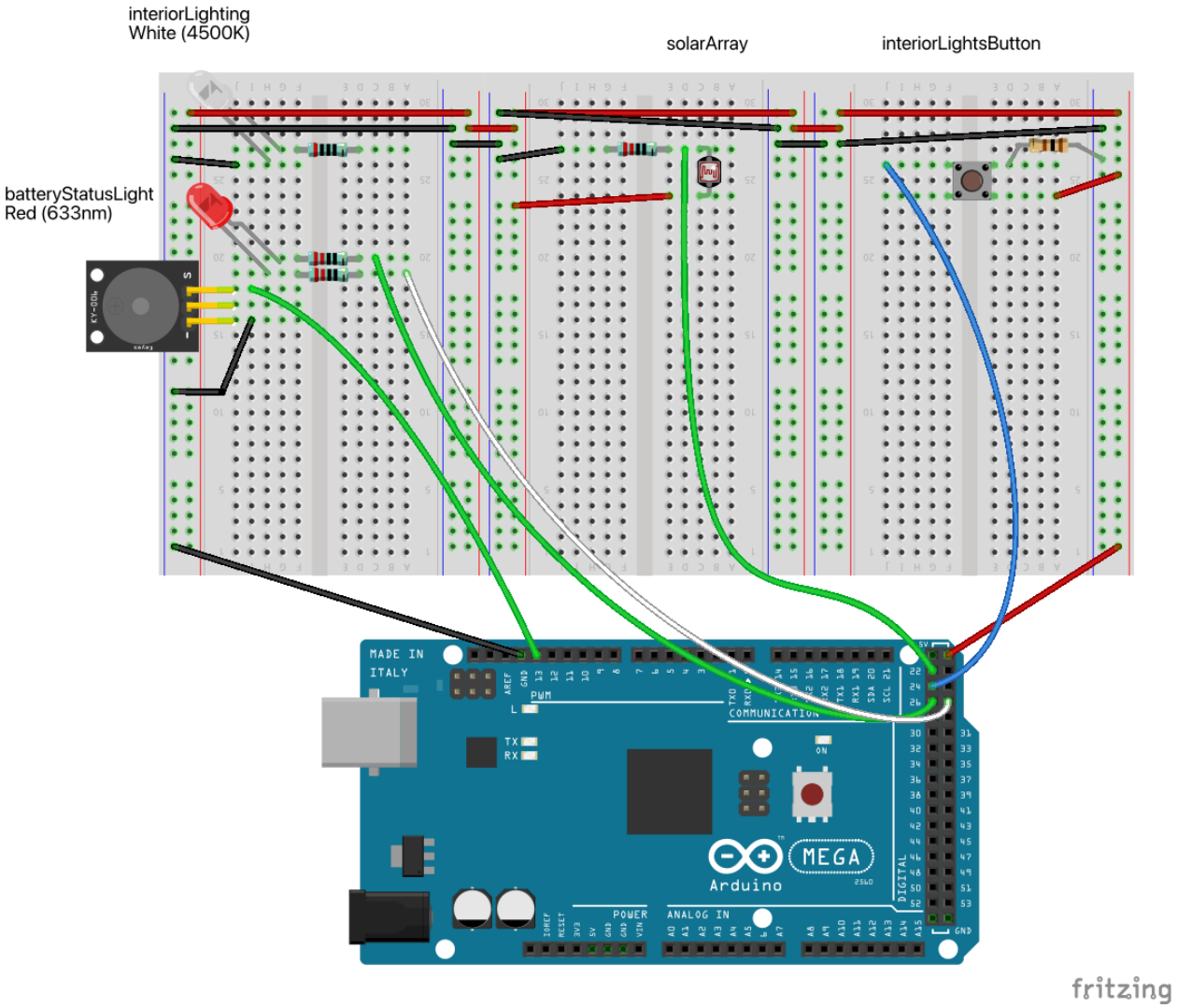
[8] Oh, there are stories about the button hardware. I spent about 20 minutes on Amazon, Adafruit, and several other sites and have ordered perhaps 20 different kinds of button hardware in hopes of finding several which are easier to wire and control. I will spare you the details, bu the best way I've found to use the included tactile switch button is to use pliers to straighten each leg and then push it *really, really hard* into the breadboard – and then it *might* stop providing floating signals which turn the lights on and off like gremlins (I had to try this with several of them before I got one that was reasonably stable).

[9] Having the _solarArray (photoresistor) as a *subobject* of the Power object requires a special technique for constructing the Power object: using a *constructor initializer list* because the _solarArray object can't use a default constructor – it requires a pin number input from the Power object constructor.

Section 05: 404 Error: Alarms not found

The Spec:

- Use buzzer to announce battery low power level
- *Provide different alert sounds for low, critical, and general*



Circuit 4: Section 5: 404 Error (Alarms Not Found)

Amount	Part Type
1	100Ω Resistor
1	200Ω Resistor
3	220Ω Resistor
1	Arduino Mega 2560 (Rev3)
1	PHOTOCELL
1	Passive Buzzer
1	Pushbutton

1	Red/Green BiColor LED
1	White (4500K) LED

Table 4: Circuit 4 Parts List

Code:

Starting with this day, I'm no longer listing all the code. Instead, you can find today's code (and supporting Fritzing diagram, images, and BOM (bill of materials) on GitHub at https://github.com/EvanRobinson/AdventureKit2/tree/Chapter01Day05.

Additions to the code include `passive_buzzer.h/.cpp`, which provide a lightweight object to play alarm noises (specified by the type `AlarmSignals`) and turn them off. As sounds may (in the future) be modulated over time, the object includes a `tick()` function to be called periodically.

In main.cpp, the primary changes are the creation of an `Buzzer` object called `AlarmSystem`, which is called in appropriate places to create different alarm sounds selected from `power_low`, `power_critical`, and a generic `alert`.

Commentary

1. There's not much to say about this. It's a simple addition. So instead I'll talk about my process.
2. As you can see from the lede paragraph in the Code: section, I've added source control (git) to my toolbox[10]. Git is hugely the most popular source control system in existence, used by 93% of developers according to StackOverflow[11]. Since I haven't been an active developer in over a decade, that didn't include me. But getting started with it is relatively low friction, it provides a command line interface (CLI), multiple graphical interfaces (including git-gui and Git Desktop), and integrates easily with Visual Studio Code/PlatformIO.
3. As a single engineer working on a simple series of projects, source control might be considered overkill. But it makes it easy for any readers I may someday have to snag my code and for me to go back and revisit previous code states easily – and to move the project from machine to machine, as I'm somewhat peripatetic these days, and I don't see that changing anytime soon. In any case, using source control is a good habit to git into (git it?).
4. When I start a new day's worth of work, the first thing I do is check my To Do List for the basic spec. I really hate watching the videos, so I made that list by watching most of them at 1.75 – 2x and making notes, then adding things I thought would be good additions. Those additions are in *italics* in the To Do List.
5. The basic spec tells me what new components (if any) I'm going to add: in this case, it was a buzzer. If I need to, I go look online (sometimes in the inventr.io Course, sometimes elsewhere) for information on how that component works, what libraries might be available for it (if I haven't already used them in 30 Days Lost in Space). Then

[10] Previous code sections can be found on GitHub at:
Day 02: Bad Wiring Systems: https://github.com/EvanRobinson/AdventureKit2/tree/Chapter01Day02
Day 03: Easy Light: https://github.com/EvanRobinson/AdventureKit2/releases/tag/Chapter01Day03
Day 03.5: Refactoring: https://github.com/EvanRobinson/AdventureKit2/tree/Chapter01Day03.5
Day 04: Solar Simulation Shenanigans: https://github.com/EvanRobinson/AdventureKit2/tree/Chapter01Day04

[11] https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/

I wire the new stuff up and figure out how to test it. Since I'd already used a buzzer, I just needed to pull one out of the box and figure out if it was active or passive, then add a ground wire[12] and a connecting wire to an output pin – either a digital pin or a PWM pin.

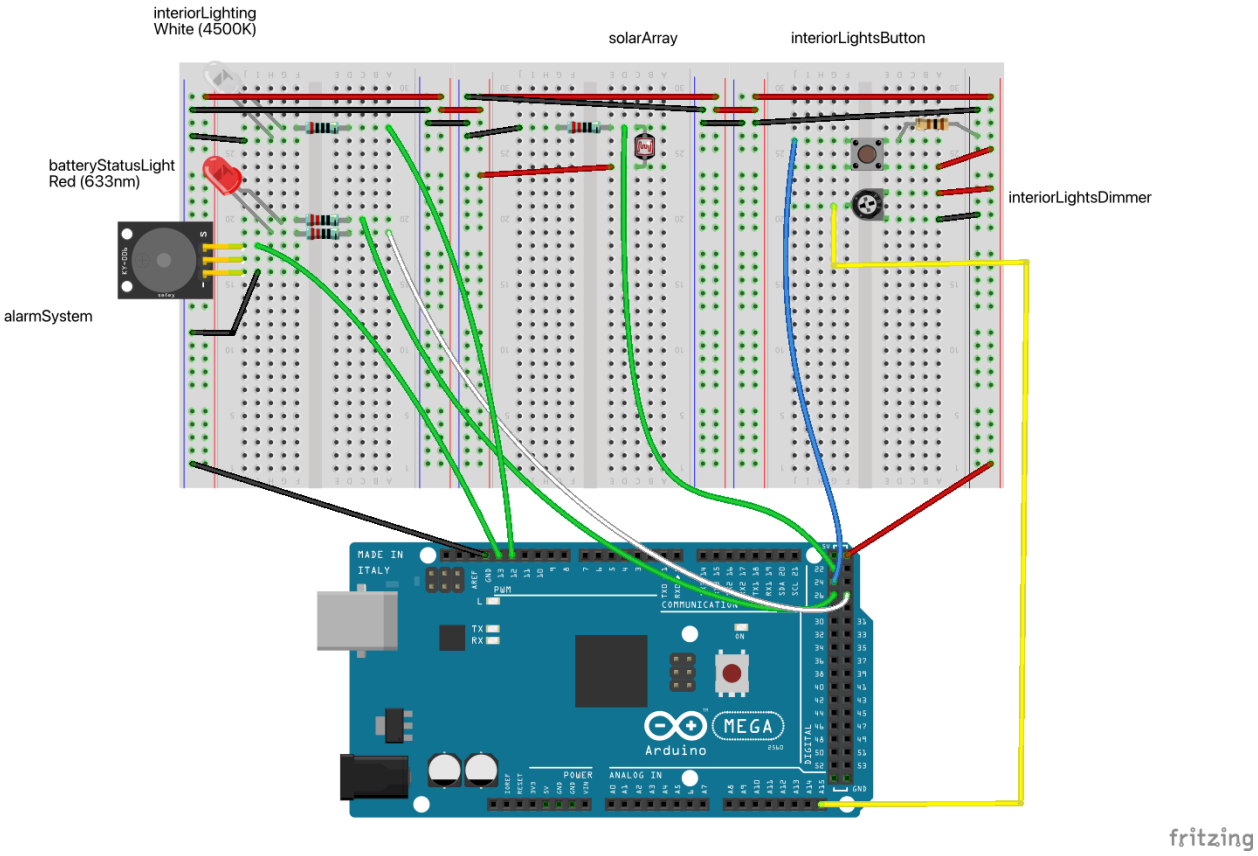
6. Wherever possible, I try to use black or dark grey wire for ground and red or orange wire for power. I break control wires into green, blue, and yellow so that each component is relatively distinct – because I haven't added any components yet that require a lot of wire, like a 7 segment display or a keypad. When I get to those, I'll do the best I can.
7. After I've experimented enough to believe I have the wiring right, I update the Fritzing file with the current layout, generate an image and a BOM (which I will further edit down after importing the .CSV file [which isn't a proper CSV file, using ';' instead of ',' as a delimiter] into Excel.
8. Then I write throwaway code (in my *dev* branch) to make sure the system works the way I want it to. In the case of the buzzer, I just put a call to `tone()` in the `loop()` function with necessary `delay()` calls to turn it on and off.
9. Once I have throwaway code, I package it up into an object and replace calls to the throwaway code with the slightly more refined version. This often involves backing and filling in the object definition and sometimes in how I interpret the spec. I really wanted a repeating beep announcing critical power levels, but it was just too annoying, so I settled for different frequencies.
10. When I'm happy with the code, I re-test it with a clean build, commit changes to the *dev* branch, push them to GitHub, merge them with the *main* branch, tag the *main* branch so it should be easily retrievable, and go write these notes.

[12] I have the jumper wires provided by inventr.io in the two course kits I have, plus some from other kits, plus a set of cut and bent jumpers of various lengths from Amazon, plus spools of colored wire (also from Amazon) that I can cut to length, strip, and bend myself. I tend to accumulate such things when I get interested in something, because I want to make sure I get the good tools. In particular, I recommend this wire stripper from Amazon. It's amazing. I don't get anything from people buying stuff. I just like it.

Section 06: Dim the Lights!

The Spec:

- Dim the Lights (potentiometer & PWM)
    - Use a potentiometer to set power level
    - Set light intensity (via PWM) according to current desired light power level[13]



Circuit 5: Section 6: Dim the Lights

1	100Ω Resistor
1	200Ω Resistor
3	220Ω Resistor
1	Arduino Mega 2560 (Rev3)
1	PHOTOCELL
1	Passive Buzzer
1	Pushbutton
1	Red/Green BiColor LED

[13] Note that, although the potentiometer (`interiorLightsDimmer`) is included in this diagram, it's not actually necessary for the Spec. It was needed to make sure that I understood how to read the pot and how to set the PWM value for the LED, but after that it's surplus, and will probably be removed from future wiring diagrams. But it's useful to see how both the `interiorLightsDimmer` and `interiorLighting` are wired up, so I've left it in for now.

1	Trimmer Potentiometer
1	White (4500K) LED

Table 5: Circuit 5 Parts List

Code:

Code, layout diagram, BOM are all on GitHub at https://github.com/EvanRobinson/AdventureKit2/releases/tag/Chapter01Day06. There are layouts and BOMs for the intermediate work with the potentiometer on the board providing direct dimming control and for the final work using the dimmed LED.

This is the basic code to read the potentiometer (which returns a value from 0-1023) and set the PWM value for the LED (which wants a value from 0-255, so we crudely divide the pot value by 4 for testing purposes).

```
  int potValue = analogRead(potentiometerAnalogInputPin);   // 0 – 1023
  analogWrite(whiteLEDPWMControlPin, potValue/4);
```

Code Block 9: Section 6: Example code to read Pot and set PWM LED

Putting this code into the loop() allows us to control the brightness of the LED in real time. This is throwaway code, even though I haven't marked it as such. This code will become the basis of the `Potentiometer` or "dimmer" class.

But there's one other thing that's worth noting. We did a crude scaling of the potentiometer value, and I will build into the `Potentiometer` class a more robust version, capable of scaling the input value to any arbitrary range:

```
double Potentiometer::readScaledTo(double minValue, double maxValue) {
    read();
    double rangeSize = maxValue - minValue;
    double readingSize = 1023.0;

    double scaled = minValue + (_currentValue * rangeSize / readingSize);
    scaled = min(scaled, maxValue);
    scaled = max(minValue, scaled);
    return scaled;
}
```

Code Block 10: Section 6: Code to scale input values to an arbitrary output scale.

This code is probably overdone, but it's a good source of potential off by one errors, so I want it to be bulletproof. It *does* have a magic value in it (1023) which is derived by observation from working with the code in Code Block 9. That should probably be corrected to read something like "`maximumInputValueFromPotentiometer`", and I'll add a TBD to the code to remind me to fix that – and probably a bunch of others.

Never a bad idea to do a little cleanup.

Commentary

1. The addition of a `DimmableLED` is very simple: just add a value that indicates the brightness the LED should be turned on to, and change the pin from a standard digital output pin to

a PWM pin. Now every time the LED is turned on, it's turned on to the previously known brightness.

2. This doesn't allow for real time altering of the brightness of the interior lights. That may be something to add later, but I really want the visible indication that power levels are low – I think that's more important than dimmable lighting in general. Any model is a limited representation of "reality" (since the dwelling exists only in the fantasy world of the Adventure Kit 2, I don't know that "reality" is the right word, but it's the best I've got), and I value the representation of low power levels more than the representation of overall dimmable lighting. At least for now.
3. I've quietly introduced polymorphism in a very limited way. `LED` and `DimmableLED` have nearly identical interfaces (it's a bit grandiose to call the object interface an API, but it's definitely an interface), and except for calls to `dimmerLevel()` the two classes are interchangeable. I would have to put considerably more thought into the class tree design to make `LED`, `DimmableLED`, and `RedGreenLED` into a coherent class tree with overlapping interface that made sense, so I'm not going to do it.

Interlude: Internal Pullup Resistors and Polymorphism

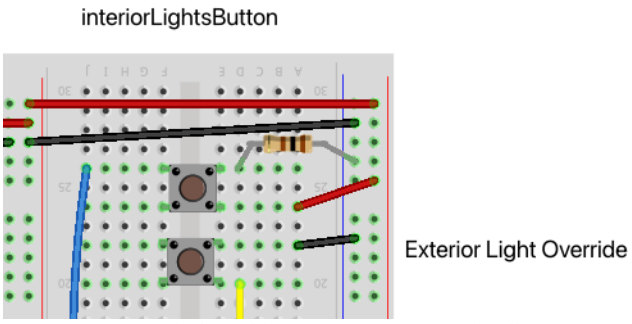
So I discovered, doing something else, that Arduinos have *internal* pullup resistors for all their digital input pins. That way, you can wire a switch without adding a resistor on the board, so long as you're willing to put up with the switch providing a LOW digital value when it's pressed and a HIGH digital value when it isn't.

Since I was adding a couple of switches to the board for Chapter 02, Section 01 (ExteriorLightOverride and AlarmOverride, if you care), I figured I'd take the opportunity to reduce my component count and add the first one using the internal pullup resistor.

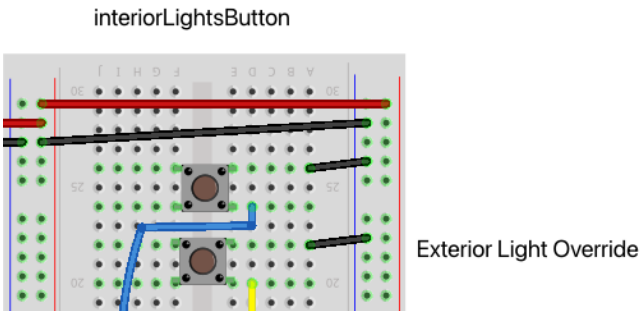
It worked, so I pulled the external pulldown resistor from the other switch and rewired it as well.

Which meant I had to change the code that reads buttons.

Except…



Circuit 6: Interlude: Add Button using internal pullup resistor



Circuit 7: Interlude: Rewrite Button using external pulldown resistor with one using internal pullup resistor.

(posted to the Adventure Kit 2 Facebook group)

This is what I love about OOP:

I had wired my switch to turn the interior lights off and on with a pulldown resistor -- because that's the way I first learned to wire a switch. And so when I created a simple Button class, it

assumed a pulldown resistor.

But then I found out that the Mega 2560 has INPUT_PULLUP, which means that I can take one resistor off the board for each switch if I change the Button code.

So I subclassed my Button class to add a ButtonPullUp class. I had to change the following lines in the original class:

Yes, that's right. Not a thing. OK, I did need to move two class variables from private: to protected: so that the subclass could access them.

I had to write this code for the subclass:

```
ButtonPullUp::ButtonPullUp:(uint8_t pin) : Button(pin) {
  pinMode(_pin, INPUT_PULLUP);
  _value = value();
}

bool ButtonPullUp::isPressed() {
  _value = value();
  return (_value == LOW);
}
```

Code Block 11: Interlude: Internal Pullup Resistors and Polymorphism:
Code to add support for buttons using pullup resistors instead of pulldown resistors.

That was it. Write the new constructor to access the original constructor, change the pinMode, and re-set the current value of the pin, then write a new version of isPressed that compared with LOW instead of HIGH.

Now to rewire any other buttons and change classes.

(end of Facebook post)

Careful or lucky decomposition of code into objects can make adding this sort of functionality relatively easy.

I won’t claim that I knew this was going to happen, but I knew that encapsulating the button code into the Button class might provide benefits down the road, and it just did. I have to admit that my decomposition of the LED code into a class has not been nearly as successful, as the change from one-color LED to bi-color LED to dimmable LED isn’t nearly so clean.

So it doesn’t always work, but it did right here.

I rewired the interiorLights button, changed the class of the variable from Button to ButtonPullUp, recompiled, and it worked. That’s one less component on the breadboard for every switch going forward – an investment in reduced complexity.

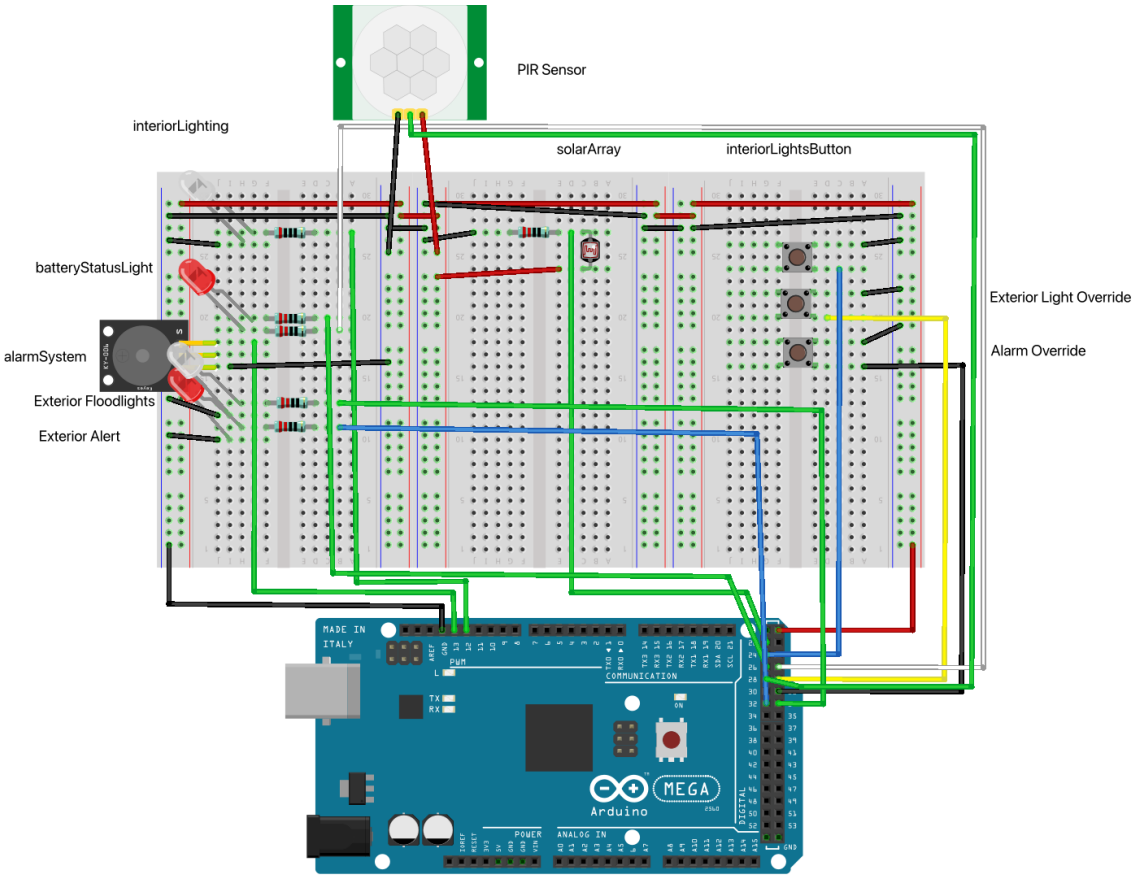
This code IS in the repository, but it's not tagged. You'll see it when you pull down the Chapter 02 Section 07 version, or you can figure out which commit it's in.

Chapter 02: Base Security 101

Section 07: Motion Sensor Security System

The Spec:

- Hook up PIR Motion Sensor
- Connect PIR Motion Sensor to White LED (simulated floodlight)
- Button to Turn Floodlight off
- Connect PIR Motion Sensor to Red Alert LED
- After delay, Flash LED
- After delay, Audible Alarm
- Button to Turn Alert Off



Circuit 8: Section 7: Motion Sensor Security System

1	200Ω Resistor
5	220Ω Resistor
1	Arduino Mega 2560 (Rev3)
1	HC-SR501 Body Sensor Module
1	PHOTOCELL
1	Passive Buzzer
3	Pushbutton
1	Red (633nm) LED

1	Red/Green BiColor LED
2	White (4500K) LED

Table 6: Circuit 8 Parts List

Code: https://github.com/EvanRobinson/AdventureKit2/releases/tag/Chapter02Day07.

I just can't help myself. I got to thinking about a generic DigitalIO class, and decided to try and build it. I actually ended up with three: `DigitalPinIn`, `DigitalPinOut`, and `DigitalPinIO`. Here's the .h file:

```
/*
    DigitalPinIn.h
    Evan Robinson, 2023-10-19

    Class to encapsulate simple digital I/O via standard pin
*/

#include <Arduino.h>

#ifndef DigitalPinIn_h
#define DigitalPinIn_h

class DigitalPinIn {
public:
    DigitalPinIn(uint8_t pin, bool pullup, bool highOn);
    bool isOn(void);
    bool isOff(void);
    bool hasChanged(void);

protected:
    int value(void);
    int _lastReadValue;

private:
    uint8_t _pin;
    bool _highIsOn;

    bool _valueHasChanged;
};

class DigitalPinOut {
public:
    DigitalPinOut(uint8_t pin, bool highIsOn);
    bool isOn(void);
    bool isOff(void);
    void turnOn(void);
```

```
    void turnOff(void);
    bool hasChanged(void);
    void toggle(void);

protected:
    int value(void);
private:
    uint8_t _pin;
    bool _highIsOn;
    int _lastSetValue;
};

class DigitalPinIO {
public:
    static const bool withPullup = true;
    static const bool withoutPullup = false;
    static const bool highOn = true;
    static const bool lowOn = false;
};

#endif
```

Code Block 12: Section 7: DigitalPinIO.h

`DigitalPinIn` and `DigitalPinOut` manage single pin two level digital input and output respectively. `DigitalPinIO` is for necessary constants and definitions that will be used across both (and possibly future) classes. The package handles plain INPUT and INPUT_PULLUP as well as objects which are "on" when HIGH and "on" when LOW. In addition, the output class maintains a current state in case a client wants to test it.

In this section, `DigitalPinIn` serves as a Button and a Motion Sensor while `DigitalPinOut` serves only as a single color LED. It's my hope that I will find other uses for them. I attempted to bring `Button` and `ButtonPullUp` into the object tree under `DigitalPinIn` and failed miserably. I expect I'll get it next time.

The system is rather straightforward. The PIR sensor (`intruderAlarm` is either on or off). When it's on, an intruder has been detected and the LED serving as the `exteriorFloodlights` turns on as well as the LED for the `exteriorAlertLight` on the main panel. After a few seconds, the `exteriorAlertLight` starts to blink. There is a switch to turn off the `exteriorFloodlights` and another to turn off the `exteriorAlertLight.`

Commentary:

1. It's become obvious that the program is developing into something vaguely akin to the MVC paradigm: Model, View, Controller. In this case, the Model should be an object representing our dwelling, so I'll be refactoring that in the near future. The various hardware input devices are our Controller, and the various output devices are our View. While I won't start renaming files in a traditional MVC way, gradually the internal state will go one way, the output UI another, and the input UI yet another. Off the top of my

head, the Controller (main.cpp, basically, but it may eventually be a named object) will contain the Dwelling object (including the interior and exterior lights, solar array, motion sensor, etc.) and the Control Panel object (including the switches and indicator lights showing light and battery status).

2. I haven't shown you this, but I cannibalized a 16x2 LED I2C display from somewhere and I'm using it as sort of a status display. Currently it shows Battery and Solar Panel levels as well as indicators for several of the buttons being pressed, and whether the motion sensor is currently triggered. I'll add it to the layout sometime in the future and it will become part of the Control Panel.
3. Possibly I'll try to lay out a physical control panel. Easiest to do would be to get a sheet of acrylic or something similar and use standoffs to mount subsystems. LEDs could just be fitted in drilled holes. Coolest to do would be to 3D print a panel like the ones for 30 Days Lost in Space. Of course, that would involve learning modeling and probably acquiring a 3D printer[14].

[14] I did use a 3D printing service to get one of the 30DLiS panels printed, but it seemed pretty expensive. Of course, it WAS a lot cheaper than buying a 3D printer and figuring out how it works. I'll have to check if there's a Maker Space convenient to home.