

# Electronics AS 91094

## Laser Tripwire

I started by hooking up an LDR to my Arduino and logging the `analogRead()` value, which returns the voltage between the output from the resistive divider created by the LDR and

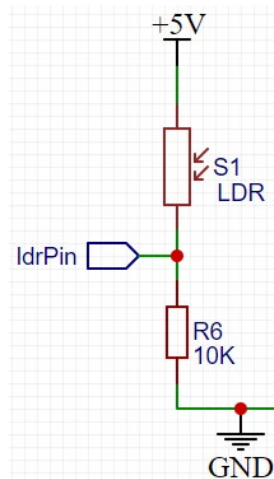


Figure 1: Testing LDR reading using Arduino

```
+int sensorPin = A0;
+int sensorValue = 0;
+
void setup() {
  // put your setup code here, to run once:
-
+ Serial.begin(9600);
-
}

void loop() {
  // put your main code here, to run repeatedly:
-
+ sensorValue = analogRead(sensorPin);
+ Serial.println(sensorValue);
-
}
```

Figure 2: Testing core functionality of LDR

accompanying resistor and ground. The circuit at this stage looked like as in Figure 1, with the LDR in series with a 10k resistor forming a resistive voltage divider. To test this circuit, I ran the code as in Figure 2, simply logging the `analogRead()` from the LDR pin.

Once this behaved exactly as I was expecting, I added a trigger functionality to the code, with a variable defining a 'trigger point' where it would trigger if the light level was below `triggerValue`. I also made it so that the software would automatically calculate an appropriate light level to trigger at by reading the ambient light and finding the mid point between the ambient light level and the maximum possible reading, such that if the ambient light environment decreased it would trigger. By using this method in calculating the trigger point the code is far more versatile than if it were hard-coded in, as it could be moved from a brighter environment to a darker environment and still function as intended. This functions by trapping the code in a 'while' loop when the `sensorValue` is 'safe', only running the rest of the code when it falls below the trigger point. Although this is a bad solution for advanced programs as it is 'blocking

```
@@ -1,13 +1,21 @@
1  int sensorPin = A0;
2  int sensorValue = 0;
3  +int initialValue = 0;
4  +int triggerValue = 0;
5  +const armedValue = 1000;
6
7  void setup() {
8    // put your setup code here, to run once:
9    Serial.begin(9600);
10 +
11 + initialValue = analogRead(sensorPin);
12 + triggerValue = armedValue - (initialValue / 2);
13 }
14
15 void loop() {
16   // put your main code here, to run repeatedly:
17 - sensorValue = analogRead(sensorPin);
18 - Serial.println(sensorValue);
19 + while (sensorValue >= triggerValue) {
20 +   sensorValue = analogRead(sensorPin);
21 +   Serial.println(sensorValue);
22 + }
23 }
```

Figure 3: Implementing a trigger system

code', meaning no other code will run while the 'while' loop is still executing, it was sufficient for the proof-of-concept purposes I was using it for.

I ran into an issue with the code in Figure 3 as sensorValue was still set equal to 0 when it first ran the conditional statement of (sensorValue >= triggerValue), meaning it would immediately return false and therefore the while would never trigger. This meant that the tripwire would immediately trip, and the trigger system failed. To fix this, I initialised sensorValue = analogRead(sensorPin) before the while loop, to ensure it had a real value.

		@@ -14,8 +14,12 @@ void setup() {
14	14	
15	15	void loop() {
16	16	// put your main code here, to run repeatedly:
	17	+ sensorValue = analogRead(sensorPin);
	18	+
17	19	while (sensorValue >= triggerValue) {
18	20	sensorValue = analogRead(sensorPin);
19	21	Serial.println(sensorValue);
20	22	}
	23	+
	24	+ Serial.println("Triggered");
21	25	}

Figure 4: Bug fix for Figure 3

The next change I made was to improve the calculation for triggerValue, by increasing the data set used to calculate it by averaging out five values. This improves the accuracy of initialValue, and in turn results in a more accurate triggerValue.

		@@ -4,11 +4,23 @@ int initialValue = 0;
4	4	int triggerValue = 0;
5	5	const armedValue = 1000;
6	6	
	7	+int initial1 = 0;
	8	+int initial2 = 0;
	9	+int initial3 = 0;
	10	+int initial4 = 0;
	11	+int initial5 = 0;
	12	+
7	13	void setup() {
8	14	// put your setup code here, to run once:
9	15	Serial.begin(9600);
	16	+
	17	+ initial1 = analogRead(sensorPin);
	18	+ initial2 = analogRead(sensorPin);
	19	+ initial3 = analogRead(sensorPin);
	20	+ initial4 = analogRead(sensorPin);
	21	+ initial5 = analogRead(sensorPin);
	22	+ initialValue = (initial1 + initial2 + initial3 + initial4 + initial5) / 5;
10	23	
11		- initialValue = analogRead(sensorPin);
12	24	triggerValue = armedValue - (initialValue / 2);
13	25	}

I ran this code to verify its function, and it performed as I expected. I then refactored it to use a for loop, running for testIterations, a variable which controls how many values to average out.

		@@ -1,27 +1,25 @@
1	1	int sensorPin = A0;
2	2	int sensorValue = 0;
3		-int initialValue = 0;
4		-int triggerValue = 0;
	3	+int disarmedValue = 0;
	4	+int triggerAt = 0;
	5	+int testTotal = 0;
	6	+const testIterations = 5;
5	7	const armedValue = 1000;
6	8	
7		-int initial1 = 0;
8		-int initial2 = 0;
9		-int initial3 = 0;
10		-int initial4 = 0;
11		-int initial5 = 0;
12		-
13	9	void setup() {
14	10	// put your setup code here, to run once:
15	11	Serial.begin(9600);
16	12	
17		- initial1 = analogRead(sensorPin);
18		- initial2 = analogRead(sensorPin);
19		- initial3 = analogRead(sensorPin);
20		- initial4 = analogRead(sensorPin);
21		- initial5 = analogRead(sensorPin);
22		- initialValue = (initial1 + initial2 + initial3 + initial4 + initial5) / 5;
23		-
24		- triggerValue = armedValue - (initialValue / 2);
	13	+ // TURN LASER MODULE OFF
	14	+
	15	+ for (int i = 0; i <= (testIterations - 1); i++) {
	16	+   testTotal += analogRead(sensorValue);
	17	+ }
	18	+
	19	+   disarmedValue = testTotal / testIterations;
	20	+   triggerAt = armedValue - (disarmedValue / 2);
	21	+
	22	+ // TURN LASER MODULE ON
25	23	}
26	24	
27	25	void loop() {

This code adds analogRead(sensorValue) to testTotal testIteration times, e.g. if analogRead(sensorValue) = 10, and testIteration is set to 5, testTotal will be equal to 50 once the for loop has finished running.

5		-const sensorPin = A0;	// LDR 'signal' pin
6		-const testIterations = 5;	// number of readings to perform when initialising
7		-const armedValue = 1000;	// maximum reading possible from LDR, therefore reading when laser is 'armed'
	5	+const int sensorPin = A0;	// LDR 'signal' pin
	6	+const int testIterations = 5;	// number of readings to perform when initialising
	7	+const int armedValue = 1000;	// maximum reading possible from LDR, therefore reading when laser is 'armed'

Figure 5: Adding type definitions to const variables

## Laser Diode

Following some further minor optimisations to the LDR trigger code I then started on getting the laser to function, which I started with just setting up a Blink program with the laser connected to pin 13.

		@@ -1,9 +1,14 @@
	1	+const int laserPin = 13;
	2	+
1	3	void setup() {
2	4	// put your setup code here, to run once:
3		-
	5	+ pinMode(laserPin, OUTPUT);
4	6	}
5	7	
6	8	void loop() {
7	9	// put your main code here, to run repeatedly:
8		-
	10	+ digitalWrite(laserPin, HIGH);
	11	+ delay(1000);
	12	+ digitalWrite(laserPin, LOW);
	13	+ delay(1000);
9	14	}

Once I verified through testing that the laser diode would behave exactly like a regular LED I then combined the two code files, adding the laser component of the tripwire system.

	6	+const int laserPin = 13;	// laser pin
6	7	const int testIterations = 5;	// number of readings to perform when initialising
7	8	const int armedValue = 1023;	// maximum reading possible from LDR, therefore read
		ing when laser is 'armed'	
8	9		
9	10	void setup() {	
10	11	pinMode(sensorPin, INPUT);	// defines (sensorPin) as an input pin
	12	+ pinMode(laserPin, OUTPUT);	// defines (laserPin) as an output pin
11	13		
12	14	Serial.begin(9600);	// initialise serial monitor at 9600 baud
13	15		
14		- // TURN LASER MODULE OFF	
	16	+ digitalWrite(laserPin, LOW);	// disengage laser for preliminary sensor reading
15	17		
16	18	for (int i = 0; i <= (testIterations - 1); i++) {	// perform the following actions (testIterations) times
17	19	testTotal += analogRead(sensorPin);	// add current analogue reading from LDR to existing (testTotal) value
		}	
		@@ -20,7 +22,7 @@ void setup() {	
20	22	disarmedValue = testTotal / testIterations;	// set (disarmedValue) equal to average of the (testIterations) tests
21	23	triggerAt = armedValue - (disarmedValue / 2);	// set (triggerAt) to mid point between 'armed' and 'disarmed' / 'triggered'
22	24		
23		- // TURN LASER MODULE ON	
	25	+ digitalWrite(laserPin, HIGH);	// engage laser tripwire

The first thing I did was to adjust the initialisation system to better represent the real trigger point, by ensuring the laser was turned off during the calculation. This means the real ambient value is taken, which can then be used to calculate an appropriate trigger value if the trigger value is broken and the laser is not in contact with the LDR. As the for loop is also 'blocking code', the laser will not be turned on until the readings have finished.

In order to test this code to ensure the calculation and tripwire is fully functioning, I added Serial.print lines to give debugging output, specifically logging the calculated value for triggerAt.

```
26 + Serial.println(triggerAt);
27 +
```

<https://youtu.be/l-XJrVRsnDY> - Testing laser tripwire

## RGB LED

Once I had my laser tripwire all testing and working, I then looked to add a red/green indicator to show whether everything was set up and good to go. To do this, I again created another proof-of-concept file, which I used to ensure the colour control would work as I was expecting it to.

```
@@ -1,9 +1,17 @@
1 +const int redPin = 8;
2 +const int greenPin = 10;
3 +const int bluePin = 12;
4 +
1 5 void setup() {
2 6 // put your setup code here, to run once:
3 -
7 + pinMode(redPin, OUTPUT);
8 + pinMode(greenPin, OUTPUT);
9 + pinMode(bluePin, OUTPUT);
4 10 }
5 11
6 12 void loop() {
7 13 // put your main code here, to run repeatedly:
8 -
14 + analogWrite(redPin, 255);
15 + analogWrite(greenPin, 0);
16 + analogWrite(bluePin, 0);
9 17 }
```

Once I ran and confirmed that this code functioned, I then decided to use digitalWrite() instead of analogWrite() for the RGB LEDs, as I did not need finer analog control, and I decided it would be easier to use a boolean set of 0/1 or LOW/HIGH instead of using 0 and 255.

```
@@ -11,7 +11,7 @@ void setup() {
11 11
12 12 void loop() {
13 13 // put your main code here, to run repeatedly:
14 - analogWrite(redPin, 255);
15 - analogWrite(greenPin, 0);
16 - analogWrite(bluePin, 0);
14 + digitalWrite(redPin, HIGH);
15 + digitalWrite(greenPin, LOW);
16 + digitalWrite(bluePin, LOW);
17 17 }
```

I then implemented this RGB LED code into the main Laser Tripwire file, and configured it such that if the tripwire failed to initialise it would display red, or it would display green if everything were set up correctly.

		@@ -4,15 +4,32 @@ int triggerAt = 0; // variable for what
4	4	int testTotal = 0; // iterative variable to record total of (testIterations) readings from LDR when initialising
5	5	const int sensorPin = A0; // LDR 'signal' pin
6	6	const int laserPin = 13; // laser pin
	7	+const int redPin = 8; // rgb module red pin
	8	+const int greenPin = 10; // rgb module blue pin
	9	+const int bluePin = 12; // rgb module blue pin
7	10	const int testIterations = 5; // number of readings to perform when initialising
8		-const int armedValue = 1023; // maximum reading possible from LDR, therefore reading when laser is 'armed'
	11	+const int armedValue = 1000; // reasonable reading for when tripwire is engaged, as (maximumValue) is infrequently reached
	12	+const int maximumValue = 1023; // maximum reading possible from LDR, therefore maximum reading when laser is 'armed'
9	13	
10	14	void setup() {
11	15	pinMode(sensorPin, INPUT); // defines (sensorPin) as an input pin
12	16	pinMode(laserPin, OUTPUT); // defines (laserPin) as an output pin
	17	+ pinMode(redPin, OUTPUT); // defines (redPin) as an output pin
	18	+ pinMode(greenPin, OUTPUT); // defines (greenPin) as an output pin
	19	+ pinMode(bluePin, OUTPUT); // defines (bluePin) as an output pin
13	20	
14	21	Serial.begin(9600); // initialise serial monitor at 9600 baud
15	22	
	23	+ digitalWrite(laserPin, HIGH); // engage laser to determine if tripwire system is functioning
	24	+
	25	+ delay(250); // pause to ensure laser is engaged prior to below reading
	26	+
	27	+ if (analogRead(sensorPin) < armedValue) { // if analogue reading from LDR indicates the tripwire is disarmed, do the following
	28	+ digitalWrite(redPin, HIGH); // turn red LED on
	29	+ digitalWrite(greenPin, LOW); // turn green LED off
	30	+ digitalWrite(bluePin, LOW); // turn blue LED off
	31	+ }
	32	+
16	33	digitalWrite(laserPin, LOW); // disengage laser for preliminary sensor reading
17	34	
18	35	for (int i = 0; i <= (testIterations - 1); i++) { // perform the following actions (testIterations) times
		@@ -20,9 +37,17 @@ void setup() {
20	37	}
21	38	
22	39	disarmedValue = testTotal / testIterations; // set (disarmedValue) equal to average of the (testIterations) tests
23		- triggerAt = armedValue - (disarmedValue / 2); // set (triggerAt) to mid point between 'armed' and 'disarmed' / 'triggered'
	40	+ triggerAt = maximumValue - (disarmedValue / 2); // set (triggerAt) to mid point between 'armed' and 'disarmed' / 'triggered'
24	41	
25	42	digitalWrite(laserPin, HIGH); // engage laser tripwire
	43	+
	44	+ delay(250); // pause to ensure laser is engaged prior to below reading
	45	+
	46	+ if (analogRead(sensorPin) > armedValue) { // if analogue reading from LDR indicates the tripwire is disarmed, do the following
	47	+ digitalWrite(redPin, LOW); // turn red LED off
	48	+ digitalWrite(greenPin, HIGH); // turn green LED on
	49	+ digitalWrite(bluePin, LOW); // turn blue LED off
	50	+ }
26	51	}
27	52	
28	53	void loop() {

<https://youtu.be/d2ljNq5b3FE> - Testing laser tripwire with RGB LED integration (LED is at bottom left of screen)

		@@ -20,16 +20,6 @@ void setup() {	
20	20		
21	21	Serial.begin(9600);	// initialise serial monitor at 9600 baud
22	22		
23		- digitalWrite(laserPin, HIGH);	// engage laser to determine if tripwire system is f
24		unctioning	
25		- delay(250);	// pause to ensure laser is engaged prior to below r
26		-	eadng
27		- if (analogRead(sensorPin) < armedValue) {	// if analogue reading from LDR indicates the tripwi
28		re is disarmed, do the following	
29		- digitalWrite(redPin, HIGH);	// turn red LED on
30		- digitalWrite(greenPin, LOW);	// turn green LED off
31		- digitalWrite(bluePin, LOW);	// turn blue LED off
32		- }	
33	23	digitalWrite(laserPin, LOW);	// disengage laser for preliminary sensor reading
34	24		
35	25	for (int i = 0; i <= (testIterations - 1); i++) {	// perform the following actions (testIterations) ti
		mes	
		@@ -42,8 +32,14 @@ void setup() {	
42	32	digitalWrite(laserPin, HIGH);	// engage laser tripwire
43	33		
44	34	delay(250);	// pause to ensure laser is engaged prior to below r
		eadng	
	35	+	
	36	+ if (analogRead(sensorPin) < armedValue) {	// if analogue reading from LDR indicates the tripwi
		re is disarmed, do the following	
	37	+ digitalWrite(redPin, HIGH);	// turn red LED on
	38	+ digitalWrite(greenPin, LOW);	// turn green LED off
	39	+ digitalWrite(bluePin, LOW);	// turn blue LED off
	40	+ }	
45	41		
46		- if (analogRead(sensorPin) > armedValue) {	// if analogue reading from LDR indicates the tripwi
		re is disarmed, do the following	
	42	+ else if (analogRead(sensorPin) > armedValue) {	// if analogue reading from LDR indicates the tripwi
		re is disarmed, do the following	
47	43	digitalWrite(redPin, LOW);	// turn red LED off
48	44	digitalWrite(greenPin, HIGH);	// turn green LED on
49	45	digitalWrite(bluePin, LOW);	// turn blue LED off

I then slightly refactored the laser tripwire code to do the ambient light level calculations before checking that the laser has turned on. This modifies the initialisation process from –

1. Turn laser on
2. Check laser is on
3. Turn laser off
4. Calculate ambient light level
5. Turn laser on
6. Check laser has turned on

To –

1. Calculate ambient light level
2. Turn laser on
3. Check laser is on



I found that I had unexpected behaviour having the laser connected to pin 13 due to the Uno's built in LED, and so I took the opportunity to re-map the pin assignments. I also used this chance to make sure the RGB LED pins were all next to each other, and so I decided to use pins 9, 10 and 11.

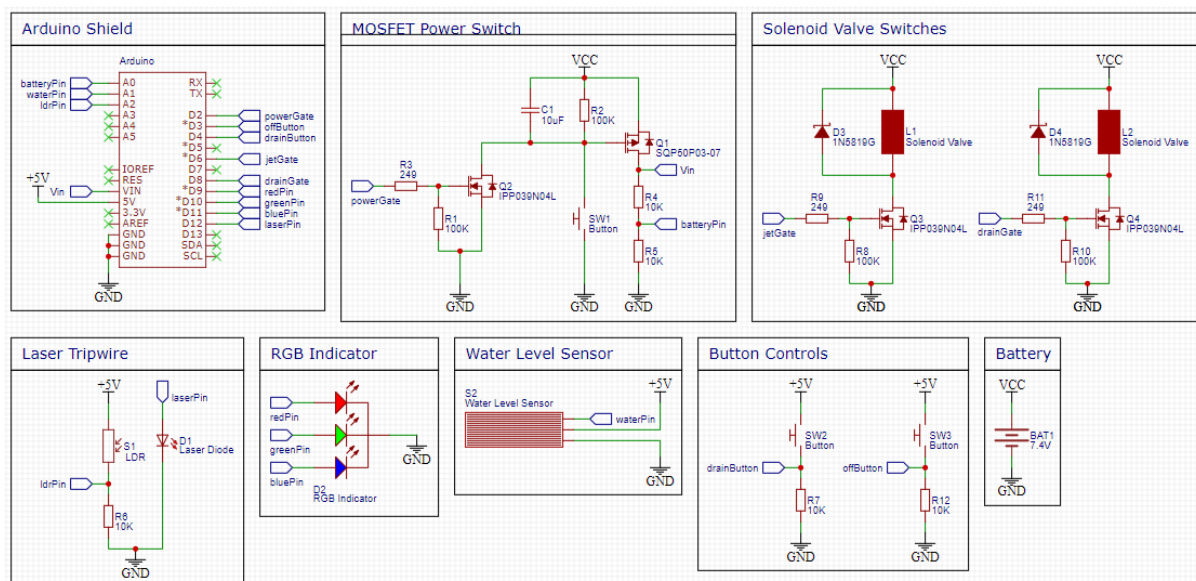
6	-const int laserPin = 13;	// laser pin
7	-const int redPin = 8;	// rgb module red pin
6	+const int laserPin = 12;	// laser pin
7	+const int redPin = 9;	// rgb module red pin
8	const int greenPin = 10;	// rgb module blue pin
9	-const int bluePin = 12;	// rgb module blue pin
9	+const int bluePin = 11;	// rgb module blue pin

## Circuit Schematic

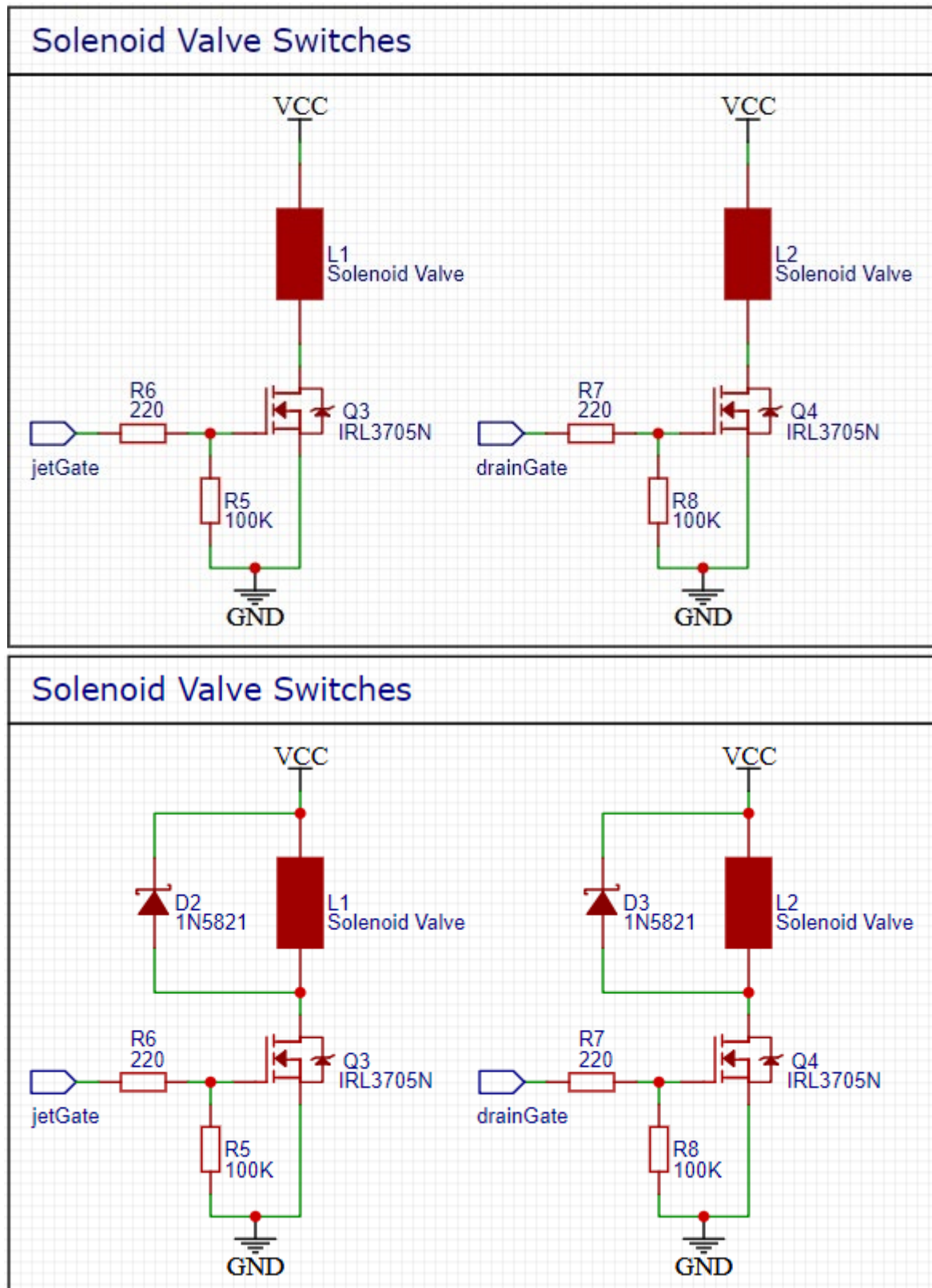
Once I had my proof of concept code files completed, I worked on developing the actual circuit for my project. To do this, I used EasyEDA to create my project schematic and later to produce the PCB design, the final results of which are shown below, with full development available alongside everything else in my GitHub repository: <https://github.com/JamesNZL/13CTE>, with development specific to the circuitry at:

<https://github.com/JamesNZL/13CTE/commits/master/Circuit/projects/Automatic%20Peg%20Cleaner>.

Commit history: <https://github.com/JamesNZL/13CTE/commits/master>



A notable design development choice I made was to add Schottky diodes D2 and D3 in reverse bias across the solenoid valves as flyback diodes, in order to prevent any unwanted voltage spikes and sparking due to the collapse of their magnetic fields.



Another addition I made later on was the addition of a pair of voltage sense resistors R4 and R5 as shown in the image below, which form a resistive voltage divider allowing the Arduino to monitor the battery's voltage. I used this to detect if the battery needed to be charged, notifying the user through blinking the indicator LED.

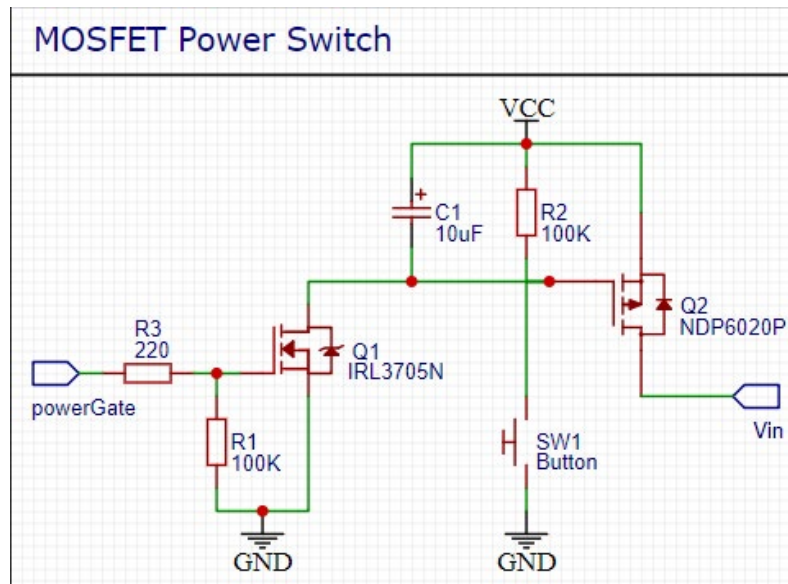
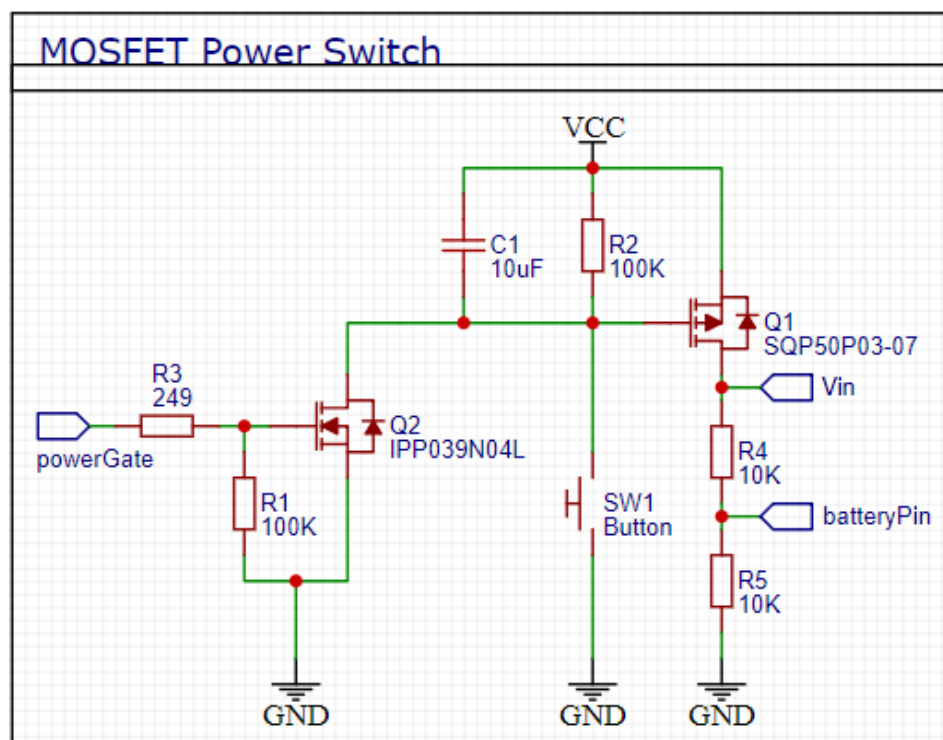


Figure 6: Without voltage sense resistors

Otherwise, my circuit schematic was reasonably straight-forward, with the only other circuit segment that was relatively advanced being the circuit for my MOSFET power switch, which I used to allow the Arduino to turn itself on when the user pushes the on button, and keeping itself on until the user pushes the off button.



The way this circuit works is that the pair of P-Channel and N-Channel MOSFETs work together to switch the output of the circuit between GND and VCC, connected to the Arduino's Vin pin, where the Arduino's GND is permanently grounded. The circuit works as when the user pushes switch SW1 the gate of the P channel MOSFET Q1 is pulled to GND, connecting VCC at the drain to the Arduino's Vin at the source. This means the Arduino begins to turn on, with a voltage output at batteryPin of half of the current voltage of the 7.4V nominal battery, which is used to check whether the battery needs to be recharged. SW1 can then be released as capacitor C1 will continue to keep the PMOS's gate low, allowing the Arduino to fully boot. Once the Arduino has fully turned on, it outputs HIGH on the powerGate pin, keeping the NMOS Q2 switched in an ON state, which maintains the GND path for the gate of Q1. If the Arduino then needs to be turned off, it simply needs to switch powerGate LOW, which in turn switches Q1 off.

## Components Selection

I chose to use MOSFETs rather than NPN and PNP transistors for a variety of reasons, such as to minimise the current draw while switched on due to the tiny  $R_{ds(on)}$  at 5V of just 0.0039 ohms and 0.007 ohms for my N and P MOS respectively. I found both using Digikey's parametric search, from which the IPP039N04LGXKSA1 N-channel MOSFET and SQP50P03-07\_GE3-ND P-channel MOSFET were the most appropriate for my needs and the most price effective in small quantities. Both MOSFETs are 'logic level', meaning they are designed for 3.3V or 5V logic circuits, with threshold voltages of gate-source (i.e. the voltage required to switch the MOSFET fully on) of 1.2V minimum and 2V maximum for the NMOS and -1.5V minimum and -2.5V maximum for the PMOS, easily manageable by my 7.4V battery / 5V Arduino.

I chose to place a gate resistor (R3) on my NMOS connected to the Arduino to ensure that the gate capacitance did not result in a high initial current draw greater than my Arduino can handle, therefore using Ohm's law I used a maximum current draw of 20mA (half of the Arduino's maximum of 40mA), which at 5V gives a resistance of 250 ohms, and so I chose a 249 ohm resistor. As MOSFETs are driven by voltage and not current (compared to a BJT transistor), there is no effect whatsoever on the speed or effectiveness of the FETs as switches. To calculate the power, I used  $P = \frac{V^2}{R}$  with  $V = 5$  and  $R = 250$ , giving me a maximum instantaneous power draw of 0.1W. I decided to use a 1/4W resistor to give some buffer over the 0.0125W tolerance for a 1/8W resistor, with next-to-nil compromise in price or size.

I am also using gate pull down and pull up resistors (R1 and R2), to prevent floating states when the Arduino is turned off or when the switch and Arduino is off respectively for the N and P channel FETs, for which I chose 100K resistors to minimise the voltage loss over the resistive divider created by R1 and R3, and to minimise current draw to a miniscule 50 microamps, which is desirable for my battery-powered project.

To calculate the appropriate capacitance of C1 to sustain a reasonable boot time for the Arduino I used  $V = V_s \cdot e^{\left(\frac{-t}{RC}\right)}$  where V is equal to the threshold voltage of the FET's gate,  $V_s$  is the supply voltage (i.e. 7.4V), R is the resistor through which C is discharging, C being the capacitance value, and t being the time taken to discharge from  $V_s$  to V.

Rearranging for C, I got  $C = \frac{-t}{R \ln\left(\frac{V}{V_s}\right)}$ , into which I plugged in  $t = 2, V = -1, V_s = -7.4V, R = 100K$  to get  $C = 9.9926 \text{ E}^{-6}$  Farads, or 10 microfarads.

By using 2 seconds I am ensuring that even with an extremely quick button press the Arduino will have enough time to fully turn on, which I am further ensuring by programming my Arduino using a second Arduino as ISP, meaning the Arduino does not need to run the bootloader and therefore is ready to go within milliseconds.

I did not require a current limiting resistor in series with my laser diode in order to protect my Arduino as it has an integrated 91 ohm resistor, which at its rated 5mW at 5V it will consume a maximum of 1mA, easily within the Arduino's 40mA limit.

I determined an appropriate resistance to use in series with my LDR to create its voltage divider experimentally, using the following table:

Light level	Sensor resistance (experimental)
Laser armed	115 ohms
Dim	10k ohms
Dark	100k ohms

Light level	Passive resistor	Voltage divider ratio	V output (5V input)
Laser armed	10k	0.9886	4.943V
Dim	10k	0.5	2.5V
Dark	10k	0.0909	0.454V

Light level	Passive resistor	Voltage divider ratio	V output (5V input)
Laser armed	47k	0.9997	4.998V
Dim	47k	0.8245	4.122V
Dark	47k	0.3197	1.598V

I started by measuring the resistance of the LDR at varying light levels using a digital multimeter, and then using a 10k resistor and a 47k resistor and trying the results. I chose the values of 10k and 47k in an attempt to concentrate the accuracy of the LDR in the range where the ambient lighting will typically be, i.e. in an enclosed 3D printed product. Therefore, I chose 10k to create a 50/50 ratio at a dim light level and 47k to experiment tending towards a darker environment, as 47k was the closest value I had to 50k. Even accounting for the expected darker light levels of the final product, I found it was overwhelmingly more reliable to stick with a 10k resistor for maximum versatility, and therefore I did so.

For the armed tripwire, a maximum voltage of 4.943V translates to an analog reading of 1011, as  $0.9889 \text{ (voltage divider ratio)} * 1023 \text{ (maximum value)} = 1011$ .

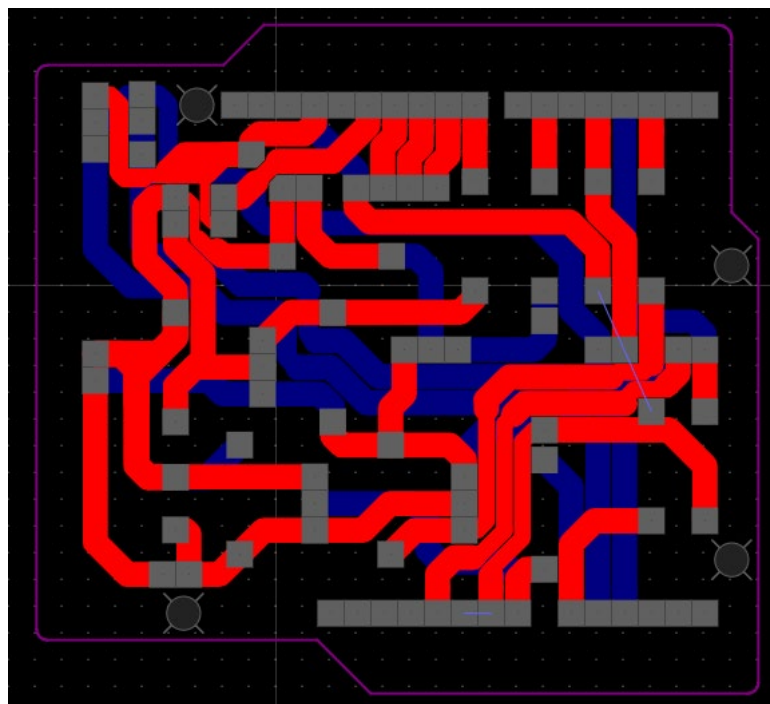
I decided it was not necessary to include reverse polarity protection in my circuitry, as I am using a keyed XT60 connector for my LiPo battery and a keyed MOLEX connector to connect my peg cleaner to the control box, therefore there is next to no risk of reverse polarity in normal operation.

To decide on a flyback diode for my solenoid valves I was looking for an average forward current of around 1A, to provide a decent buffer for the 500mA of current draw I expected from my solenoid valves. Using the DigiKey parametric search, I found the 1N5819, which has up to 25A of surge current and up to 40V of reverse voltage, meaning it was more than adequate for my application.

## PCB Development

My initial plan to produce and fabricate my PCB was to use our previous Voltera prototyping machine to validate that the circuit was fully functional on a proper board and not just on the breadboard, after which I would have it fabricated by a prototyping company such as JLCPCB. Multiple aspects of the plan changed however, and I ended up not having the time to realistically send my completed PCB design off and wait for it to arrive, and also we moved away from the Voltera printed ink circuit boards back to a CNC miller. Therefore, my final PCB being used in my project was produced using the school's CNC miller.

Various different versions of my PCB design:



*Figure 7: Top layer of my EasyEDA PCB design for the miller*

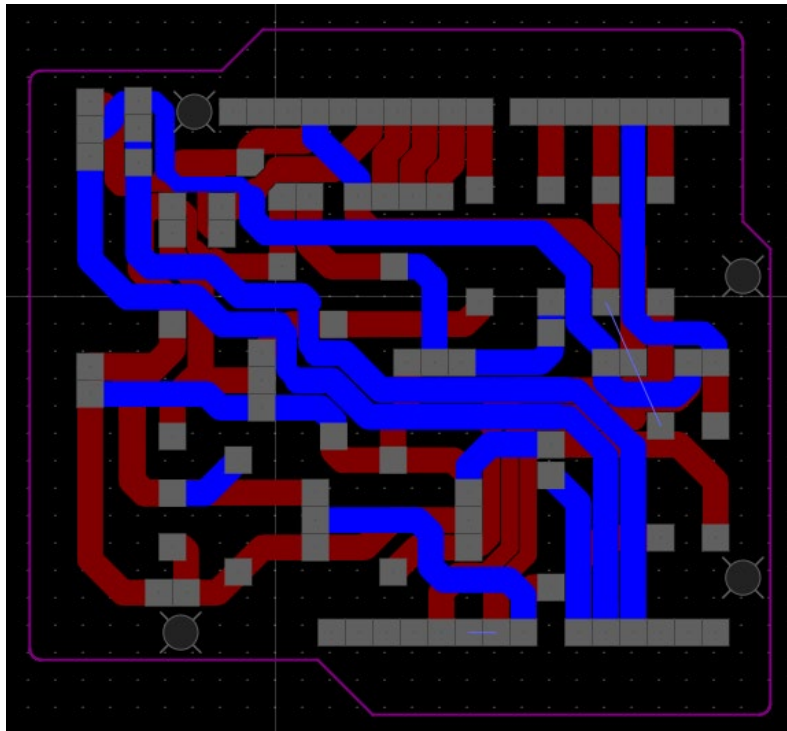


Figure 8: Bottom layer

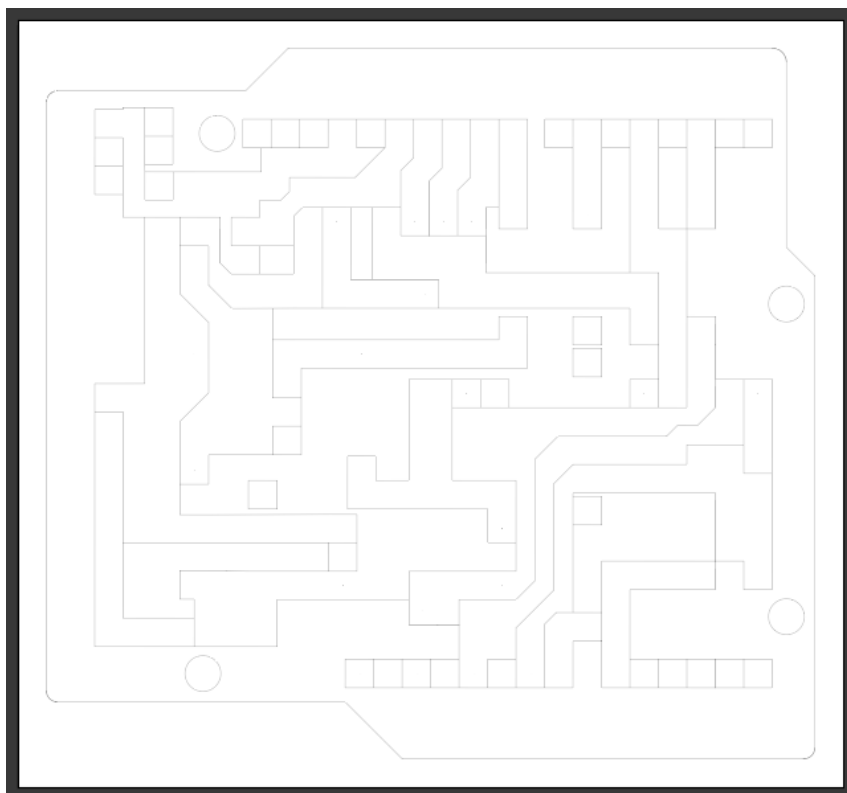


Figure 9: Processed top layer prior to milling, I went through and maximised the width of the traces by ensuring there weren't two edges in the same position

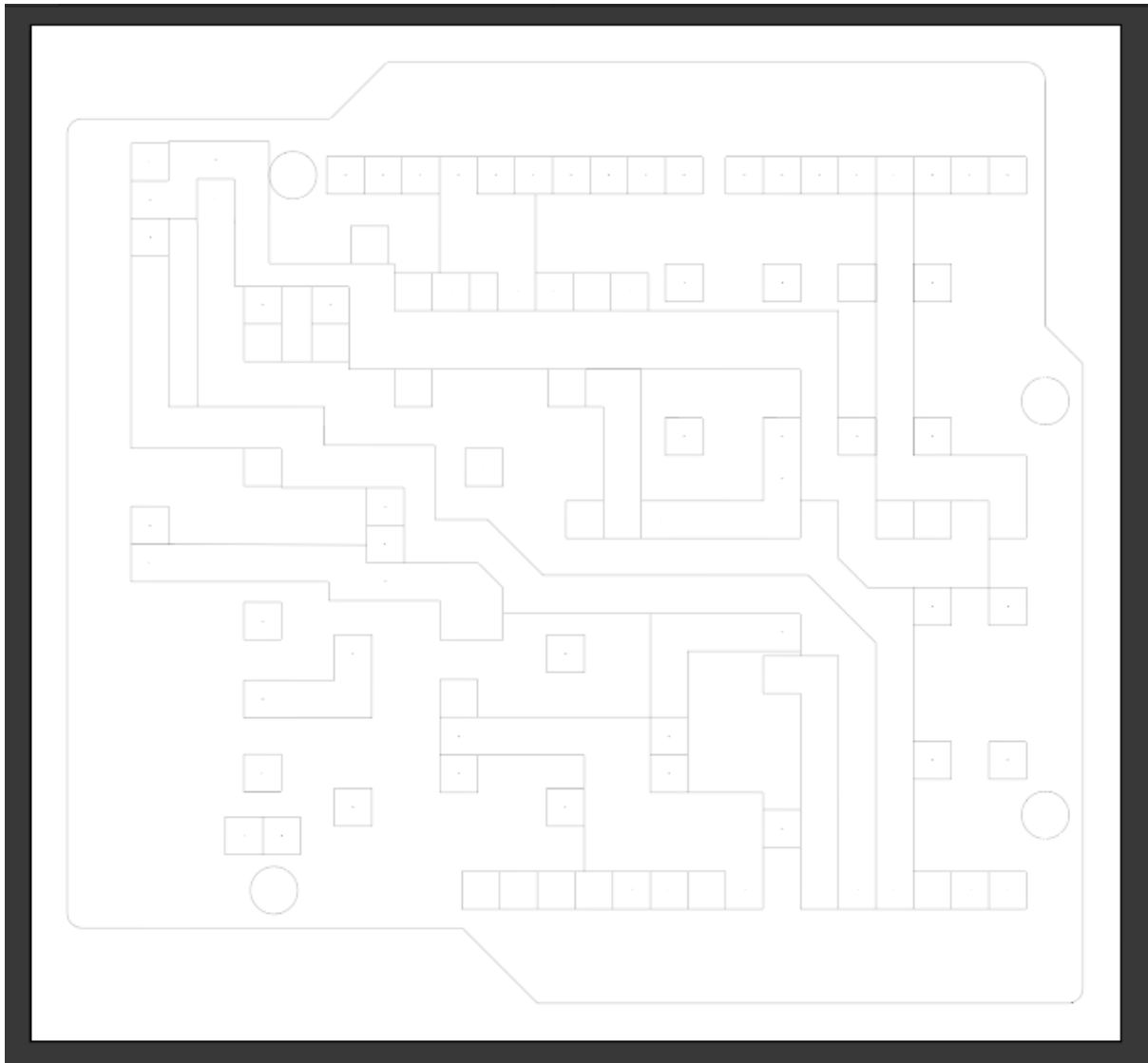


Figure 10: Processed bottom layer



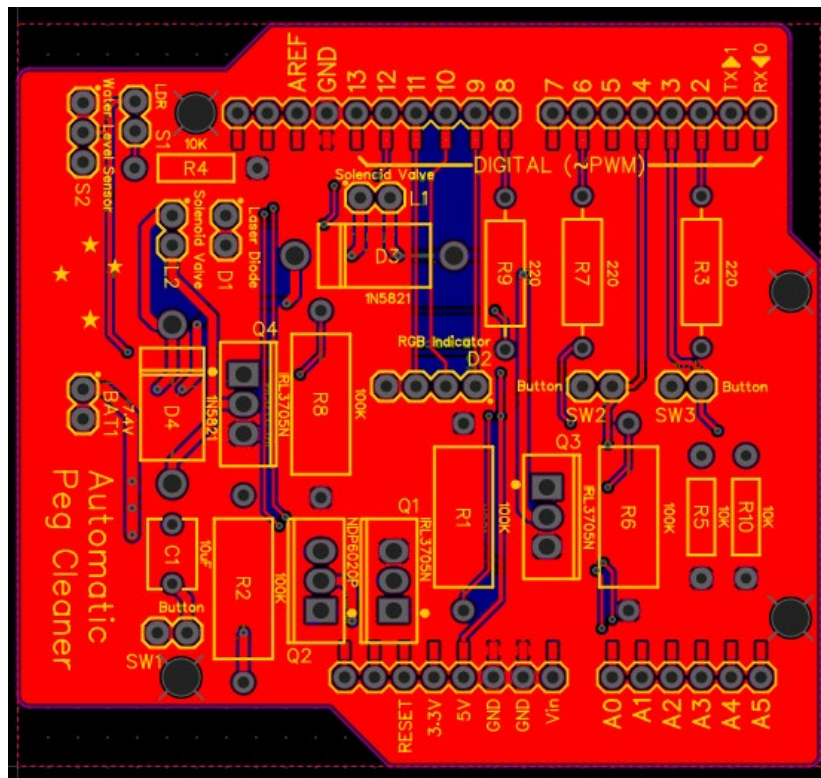


Figure 11: Fabrication PCB top layer, before re-design with more appropriate components

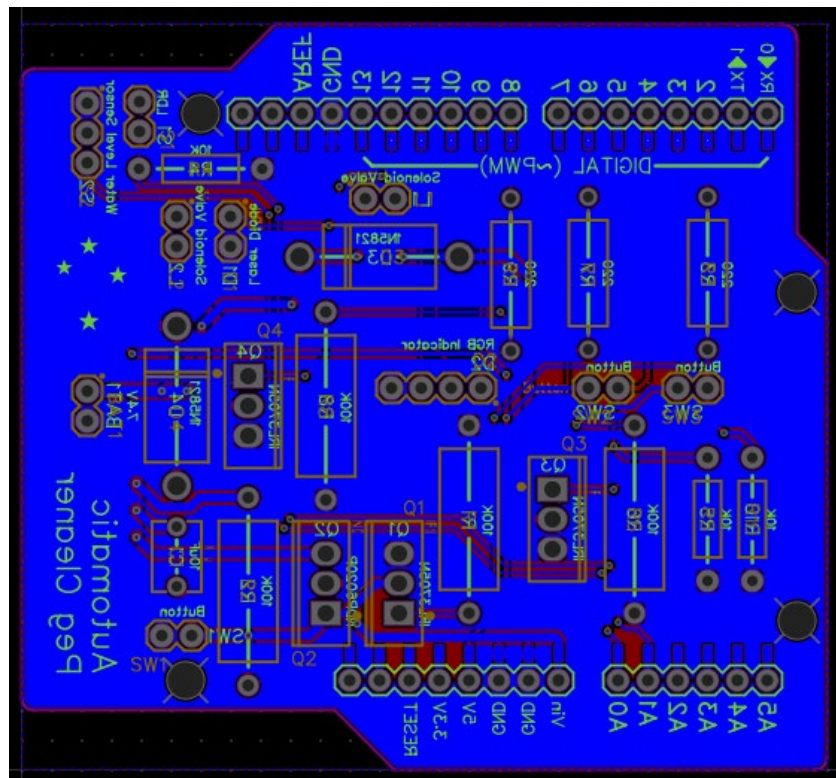


Figure 12: Bottom layer of Figure 11

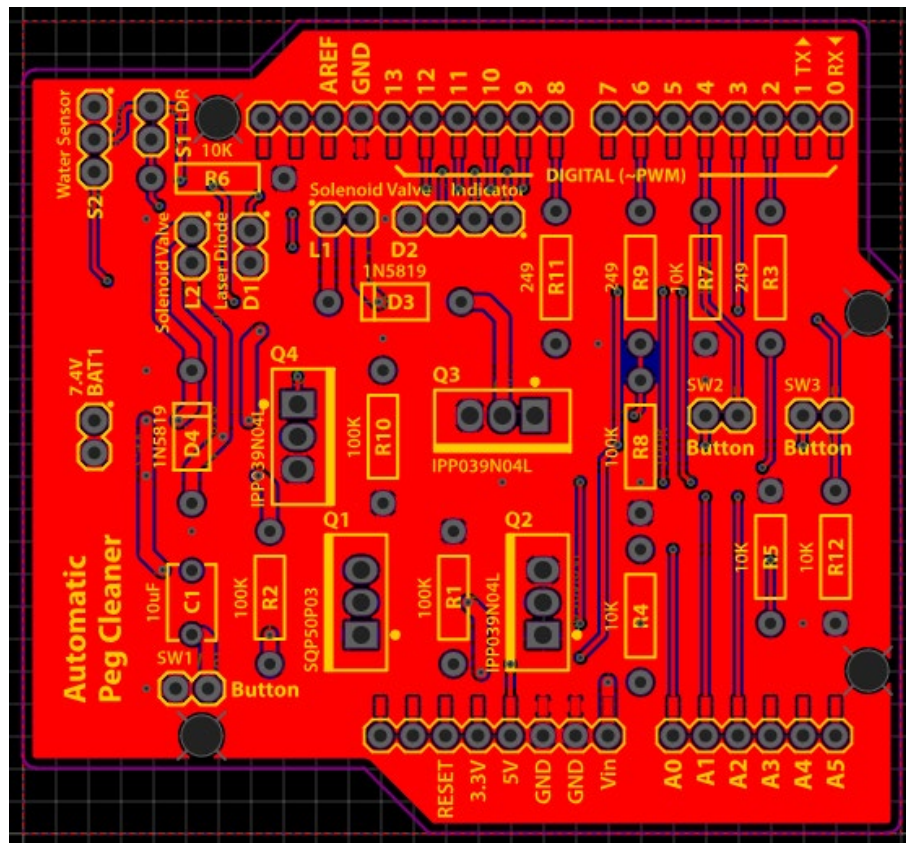
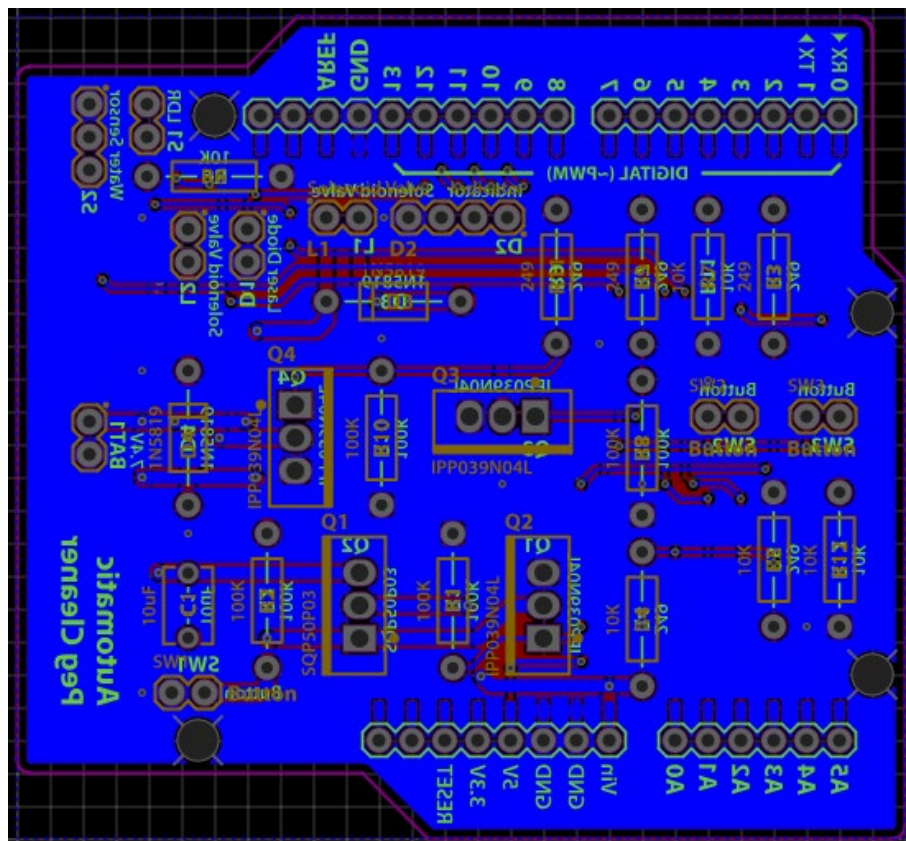
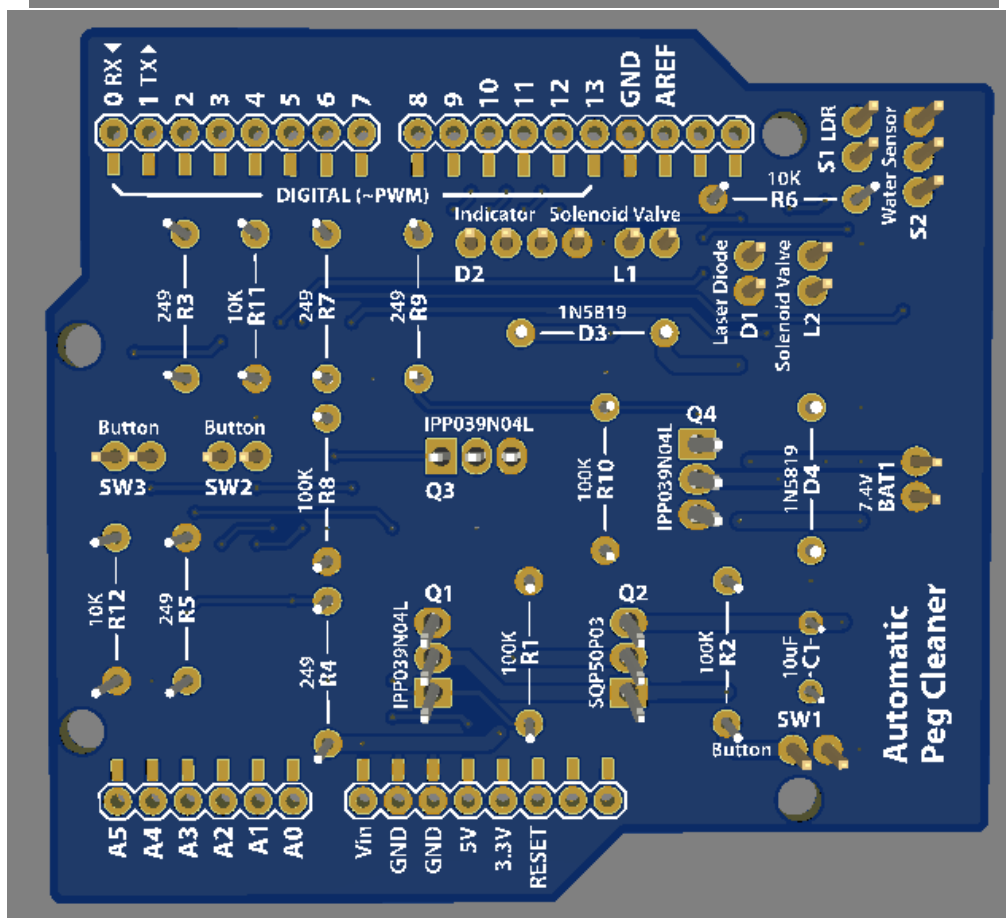
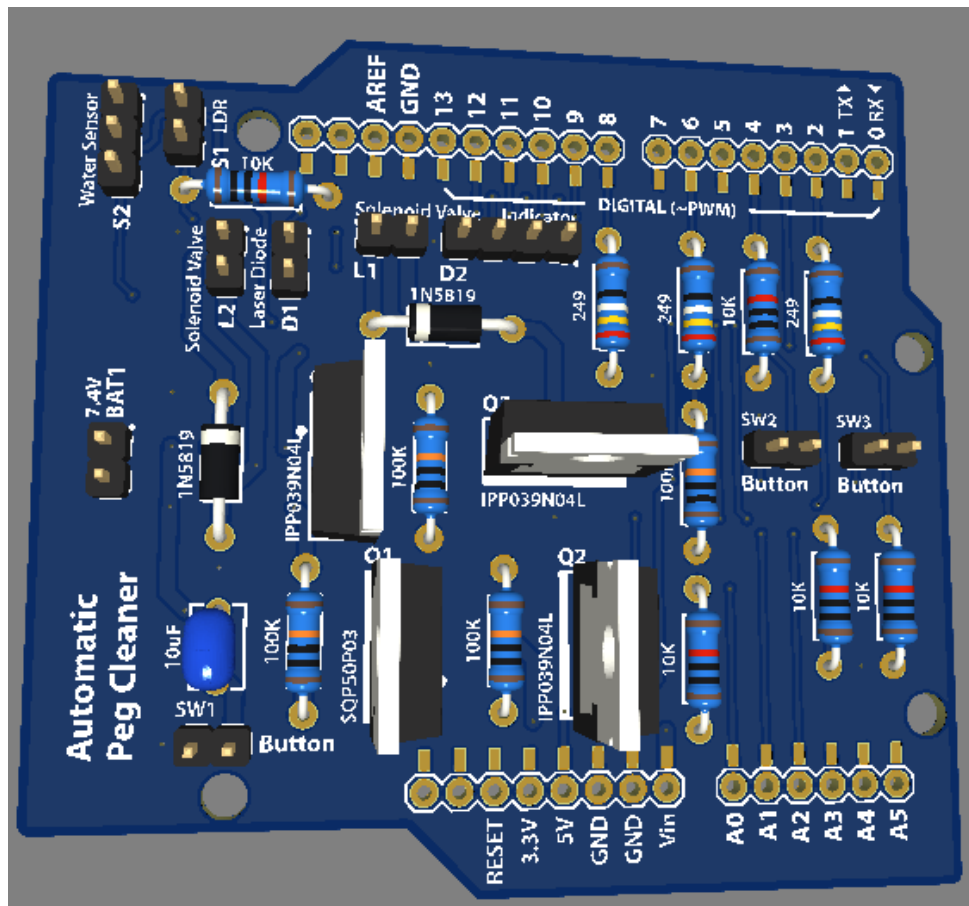


Figure 13: Final PCB designed for fabrication







For my early versions such as the one I designed for the Voltera, my only goal was to ensure good quality electrical connections between A and B, in order to verify the performance and functionality of the circuit. Once I achieved that, I then went back and continued to iterate and improve, using design best practises such as using a GND plane/copper fill on my boards, enforcing a convention of North-South and East-West on my top/bottom layers to avoid routing conflicts, using vias to tidy up board layouts, and using GND stitching vias. By aiming to have my designs fabricated professionally, I was able to incorporate these elements into my designs as I could be confident that they would be able to be produced within the capabilities of the machines, unlike if I were to make them myself.

As we eventually moved away from using the Voltera at school, it meant the capabilities and restrictions I had to prototype my board at school changed drastically, as the CNC miller offered some significant advantages but also some disadvantages over the Voltera. The miller is far more robust and easier to operate than the Voltera, resulting in a board that is easier to debug and test, as well as to solder to. However, the design process to actually fabricate the board became far more complex, where while the Voltera simply needed the exported Gerber files directly from the PCB design software, the CamCut CNC required a proprietary and non-standardised .FAB fabrication file. This meant I had to export my designs from EasyEDA as a PDF rather than as a set of Gerber files, such that I could end up with a file readable by 2D Design. The only way we knew to get a workable .FAB file for the CNC miller was to export from 2D Design using a pre-loaded export engine and then run an executable on one of the school's computers, and so I kept jumping through applications and file formats until I found one that resulted in a workable work-around. Eventually, I found I needed to export as a PDF and use Adobe Illustrator to convert it to a .DWG file readable by AutoCAD, which could then export as a .PLOT file that was recognisable by 2D design. We could then export using the CamCut PLOT settings, and putting that through the convert.exe, we ended up with a file that could be milled.

This meant it was somewhat more difficult to quickly prototype iterations of my PCB and especially if I wanted to use Illustrator to process my exported EasyEDA pdf so that the lines were as efficient as possible, however in the end resulted in a functional and working PCB.

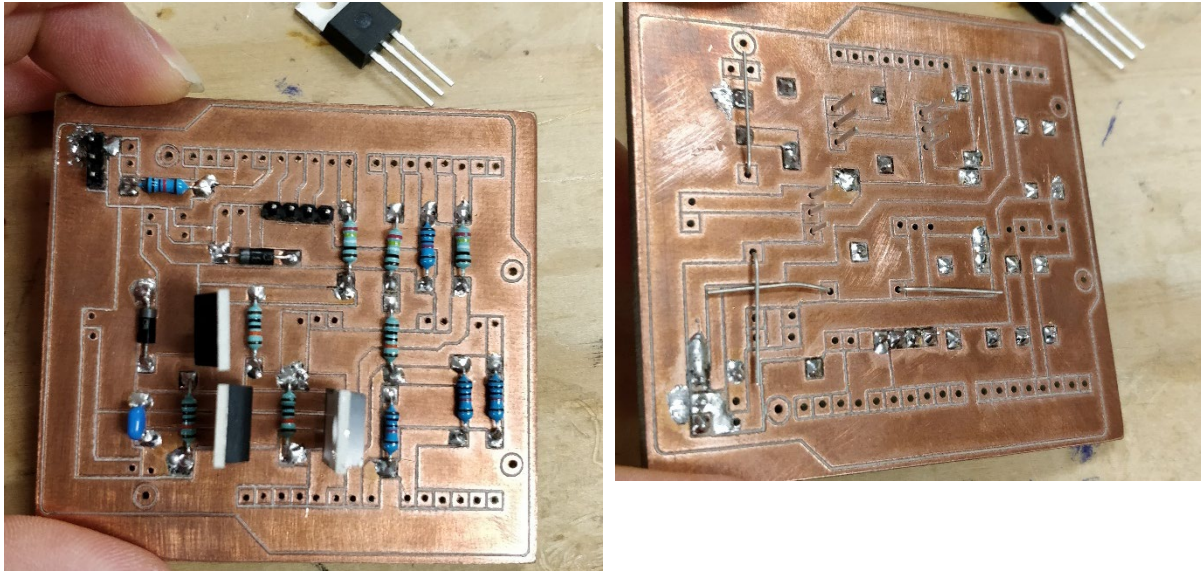


Figure 14: Soldering components

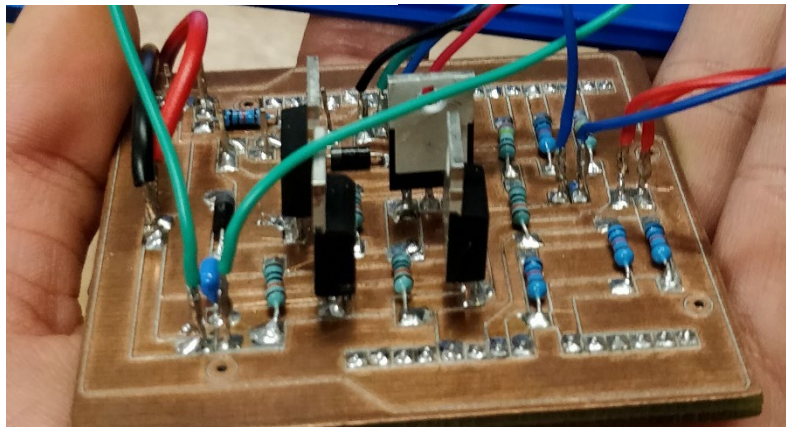
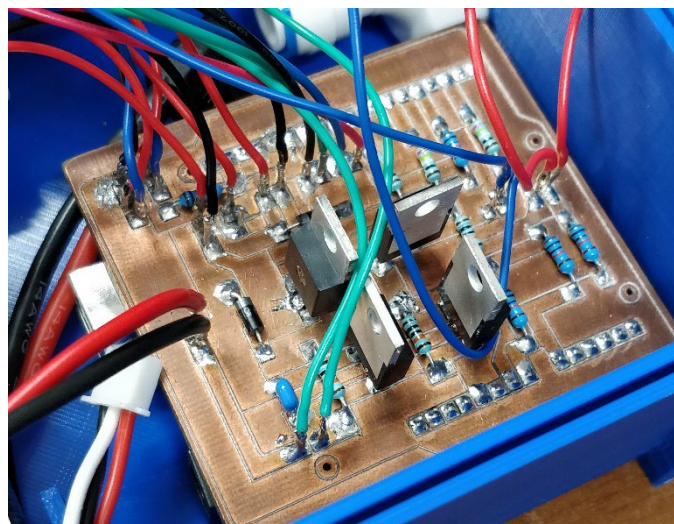


Figure 15: Completed PCB



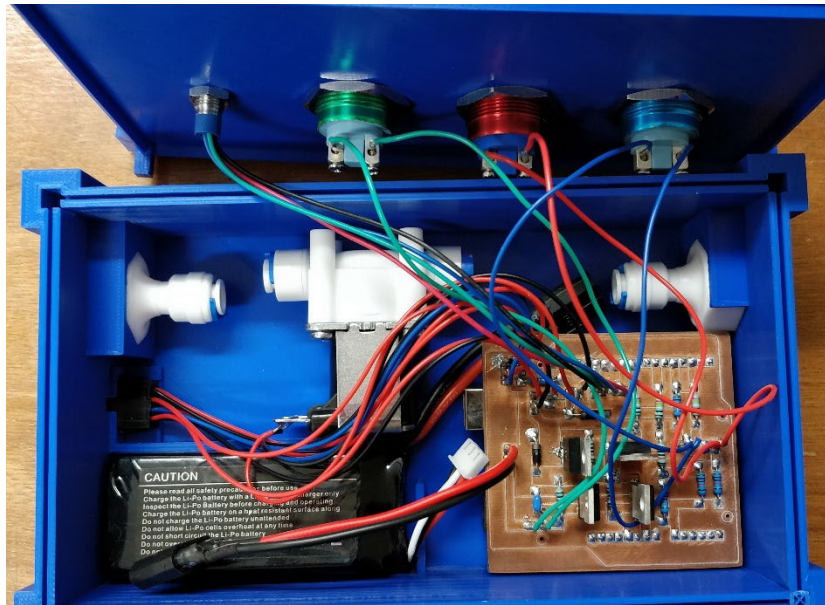


Figure 16: Completed PCB in enclosure

## Arduino as ISP

To program my Arduino, I used a second Arduino as a Programmer and used the ArduinoAsISP functionality to program the one I was using in my project. By using this method, I am ensuring

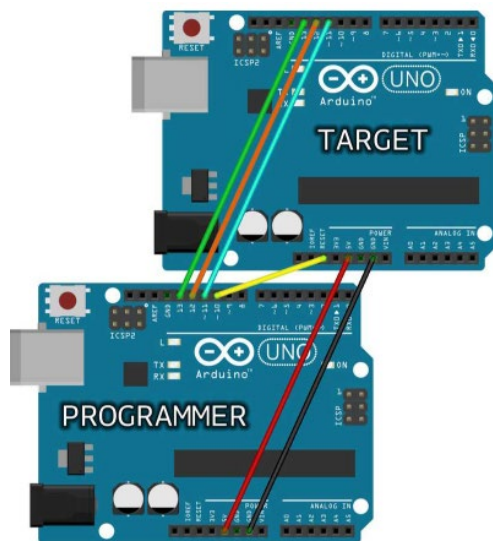


Figure 17: From

<https://www.arduino.cc/en/Tutorial/BuiltInExamples/ArduinoISP>

the Arduino bootloader is not loaded on to my target Arduino, as therefore does not execute the bootloader processes when it first starts up. This consequently means that on start-up it immediately begins running my project code, and therefore will switch the pin connected to the MOSFET on extremely quickly. I decided to use this method of programming my target Arduino instead of the usual direct upload as by using the conventional upload, the IDE will automatically also upload Arduino's bootloader to the target alongside the project code, which runs on start-up. What the bootloader does is check if the Arduino is trying to be programmed, and if so establishes the connection in order to accept the incoming code. Although this only takes roughly a second to fully complete and to move on to the actual project code, I needed the target to turn on as

quickly as possible in order to ensure the capacitor did not discharge below the threshold needed to have the Arduino turn on, in the event that the user only gave the button a short tap.

Arduino as ISP uploads the program code from the programmer to the target using the SPI interface/protocol, or Serial Peripheral Interface. SPI is a synchronous interface, meaning that there is a clock signal controlled by the master device, i.e. the programmer, and used by the



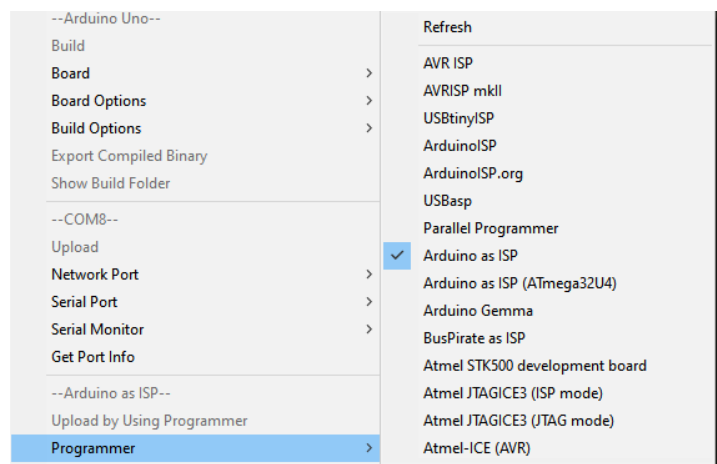
slave device(s), i.e. the target. SPI uses four pins as shown in the diagram above (not including the +5V and GND connections), by convention called SCLK, MOSI, MISO and SS.

SCLK	Serial clock	Controlled/output by the master device for synchronising the serial communication
MOSI	Master Out Slave In	Communication line from master to slave
MISO	Master In Slave Out	Communication line from slave to master
SS	Slave select	Controlled by master to select the target slave device

On the Arduino Uno, these are both stand-alone on the edge of board as in the Figure below, however they are also accessible through the digital pins 10-13, for SS, MOSI, MISO and SCLK



respectively. In order for me to use this protocol to interface with the Arduino and upload my program code to my target device, all I needed to do was upload the Arduino as ISP sketch that is built-in to the Arduino IDE to my master/programmer, make the connections between programmer and target, and upload the code to the programmer while 'Arduino as ISP' was selected as the Programmer interface.



For the rest of my project, I did not need to use any other communication protocols as my subsystem was wired directly to my PCB using a connector and did not have any active components that needed to be communicated with, as all the MOSFET switching circuits were controlled on my PCB. If I were to re-make this project, I would look more in depth into creating a wireless control module for the sub-system, and communicating over Bluetooth or another wireless interface, so that I do not need to have wires connecting the PCB to the actual peg cleaner.

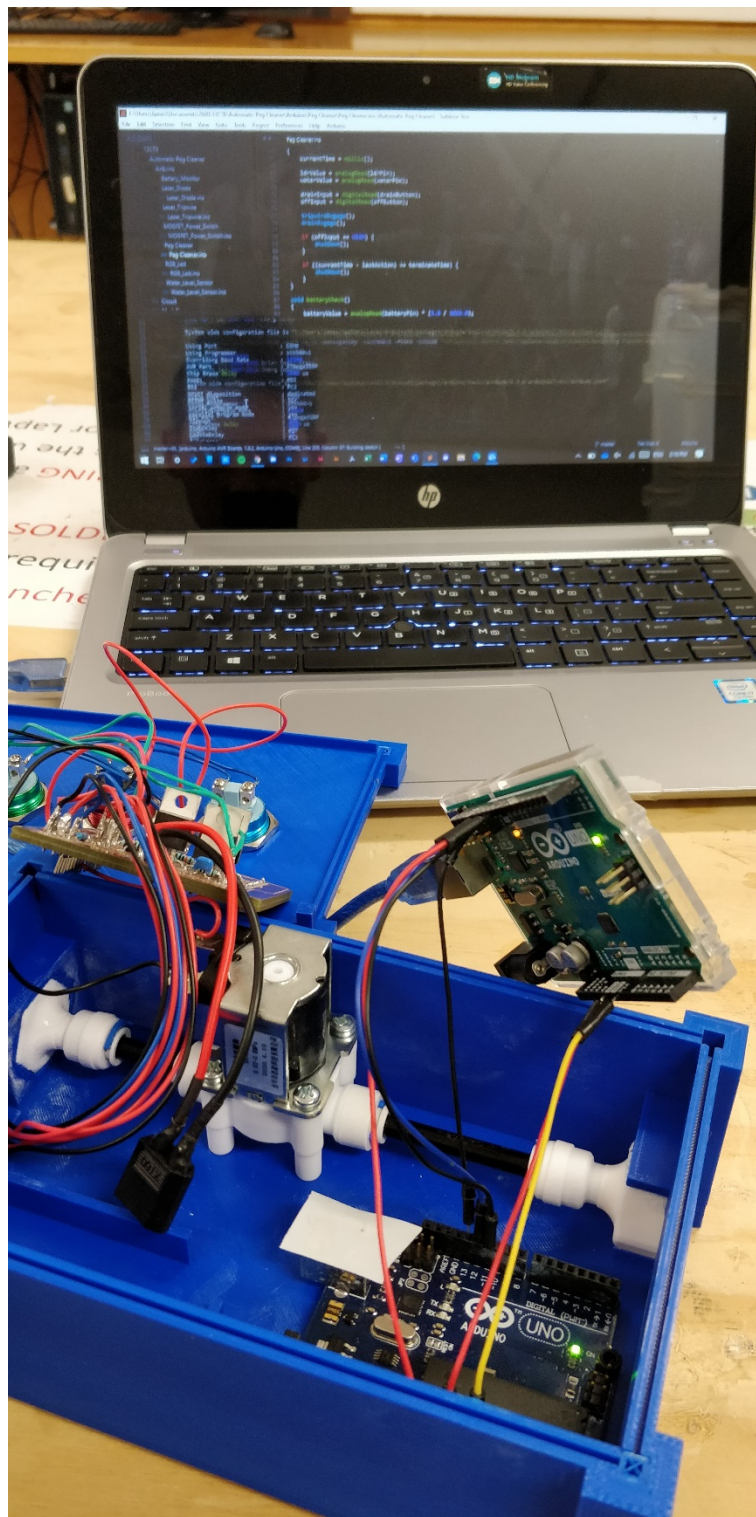


Figure 18: Uploading using Arduino as ISP



## Appendix of Evidence

<https://youtu.be/9pyqVOIHjWM> - Testing using a single NMOS for the switching circuit, does not work as Arduino turns on even before button is pressed

<https://youtu.be/B2iy4qF7UdA> - Switching circuit with relay, functions perfectly however relay was not suitable for final product due to power consumption and physical size

<https://youtu.be/7AMME0NAls> - Testing analogRead values with water level sensor

<https://youtu.be/zFWam2sqd8s> - Testing accuracy of the defined water danger value

<https://youtu.be/cPWKj5jhYVU> - Changed minimum value as a result of testing

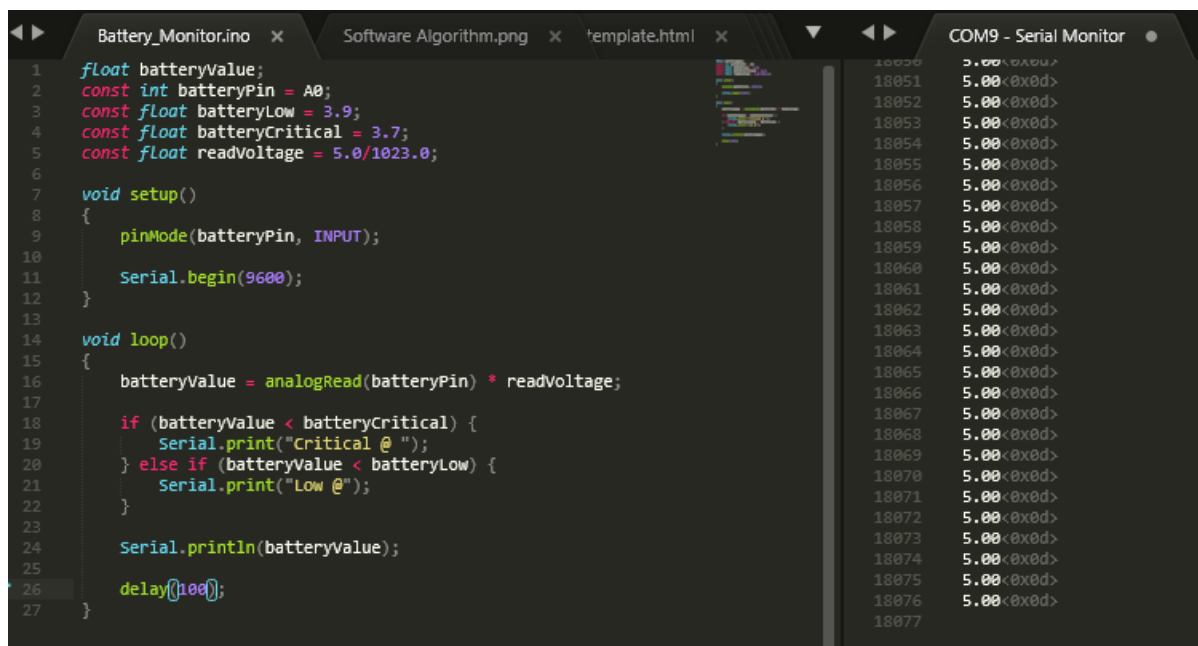
[https://youtu.be/0\\_3hpAmIlrM](https://youtu.be/0_3hpAmIlrM) - Testing the MOSFET switching circuit, Arduino programmed conventionally

<https://youtu.be/LsTzG-rKk8Q> - Testing MOSFET switching circuit with target Arduino programmed with Arduino as ISP

<https://youtu.be/iif9ogpAt94> - Testing turning off indicator LED after capacitor discharges, i.e. when Arduino fully shuts down (feels laggy)

<https://youtu.be/blNLxr6ts74> - Testing turning off indicator LED before capacitor discharges, i.e. before Arduino fully shuts down (feels responsive, however not fully accurate to Arduino being off)

<https://youtu.be/aM0B3x9BfSc> - Testing indicator LED colour conditions to ensure it is fully functional



The screenshot shows the Arduino IDE with the file 'Battery\_Monitor.ino' open. The code is as follows:

```

1 float batteryValue;
2 const int batteryPin = A0;
3 const float batteryLow = 3.9;
4 const float batteryCritical = 3.7;
5 const float readVoltage = 5.0/1023.0;
6
7 void setup()
8 {
9   pinMode(batteryPin, INPUT);
10  Serial.begin(9600);
11 }
12
13 void loop()
14 {
15   batteryValue = analogRead(batteryPin) * readVoltage;
16
17   if (batteryValue < batteryCritical) {
18     Serial.print("Critical @ ");
19   } else if (batteryValue < batteryLow) {
20     Serial.print("Low @");
21   }
22
23   Serial.println(batteryValue);
24
25   delay(1000);
26 }
27

```

The Serial Monitor (COM9) shows the following output:

```

18050 5.00<0x0d>
18051 5.00<0x0d>
18052 5.00<0x0d>
18053 5.00<0x0d>
18054 5.00<0x0d>
18055 5.00<0x0d>
18056 5.00<0x0d>
18057 5.00<0x0d>
18058 5.00<0x0d>
18059 5.00<0x0d>
18060 5.00<0x0d>
18061 5.00<0x0d>
18062 5.00<0x0d>
18063 5.00<0x0d>
18064 5.00<0x0d>
18065 5.00<0x0d>
18066 5.00<0x0d>
18067 5.00<0x0d>
18068 5.00<0x0d>
18069 5.00<0x0d>
18070 5.00<0x0d>
18071 5.00<0x0d>
18072 5.00<0x0d>
18073 5.00<0x0d>
18074 5.00<0x0d>
18075 5.00<0x0d>
18076 5.00<0x0d>
18077

```

Figure 19: Testing battery monitoring circuit

```

Battery_Monitor.ino x Software Algorithm.png x template.html x
float batteryValue;
const int batteryPin = A0;
const float batteryLow = 3.9;
const float batteryCritical = 3.7;
const float readVoltage = 5.0/1023.0;

void setup()
{
  pinMode(batteryPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
  batteryValue = analogRead(batteryPin) * readVoltage;

  if (batteryValue < batteryCritical) {
    Serial.print("Critical @ ");
  } else if (batteryValue < batteryLow) {
    Serial.print("Low @");
  }

  Serial.println(batteryValue);

  delay(100);
}

```

COM9 - Serial Monitor

```

18395 Critical @ 3.28<0x0d>
18396 Critical @ 3.28<0x0d>
18397 Critical @ 3.28<0x0d>
18398 Critical @ 3.28<0x0d>
18399 Critical @ 3.28<0x0d>
18400 Critical @ 3.28<0x0d>
18401 Critical @ 3.28<0x0d>
18402 Critical @ 3.28<0x0d>
18403 Critical @ 3.28<0x0d>
18404 Critical @ 3.28<0x0d>
18405 Critical @ 3.28<0x0d>
18406 Critical @ 3.28<0x0d>
18407 Critical @ 3.28<0x0d>
18408 Critical @ 3.28<0x0d>
18409 Critical @ 3.28<0x0d>
18410 Critical @ 3.28<0x0d>
18411 Critical @ 3.28<0x0d>
18412 Critical @ 3.28<0x0d>
18413 Critical @ 3.28<0x0d>
18414 Critical @ 3.28<0x0d>
18415 Critical @ 3.28<0x0d>
18416 Critical @ 3.28<0x0d>
18417 Critical @ 3.28<0x0d>
18418 Critical @ 3.28<0x0d>
18419 Critical @ 3.28<0x0d>
18420 Critical @ 3.28<0x0d>
18421 Critical @ 3.28<0x0d>
18422 Critical @ 3.28<0x0d>
18423 Critical @ 3.28<0x0d>
18424 Critical @ 3.28<0x0d>
18425 Critical @ 3.28<0x0d>
18426 Critical @ 3.28<0x0d>
18427 Critical @ 3.28<0x0d>
18428 Critical @ 3.28<0x0d>
18429 Critical @ 3.28<0x0d>
18430

```

```

Battery_Monitor.ino x Software Algorithm.png x template.html x
float batteryValue;
const int batteryPin = A0;
const float batteryLow = 3.9;
const float batteryCritical = 3.7;
const float readVoltage = 5.0/1023.0;

void setup()
{
  pinMode(batteryPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
  batteryValue = analogRead(batteryPin) * readVoltage;

  if (batteryValue < batteryCritical) {
    Serial.print("Critical @ ");
  } else if (batteryValue < batteryLow) {
    Serial.print("Low @");
  }

  Serial.println(batteryValue);

  delay(100);
}

```

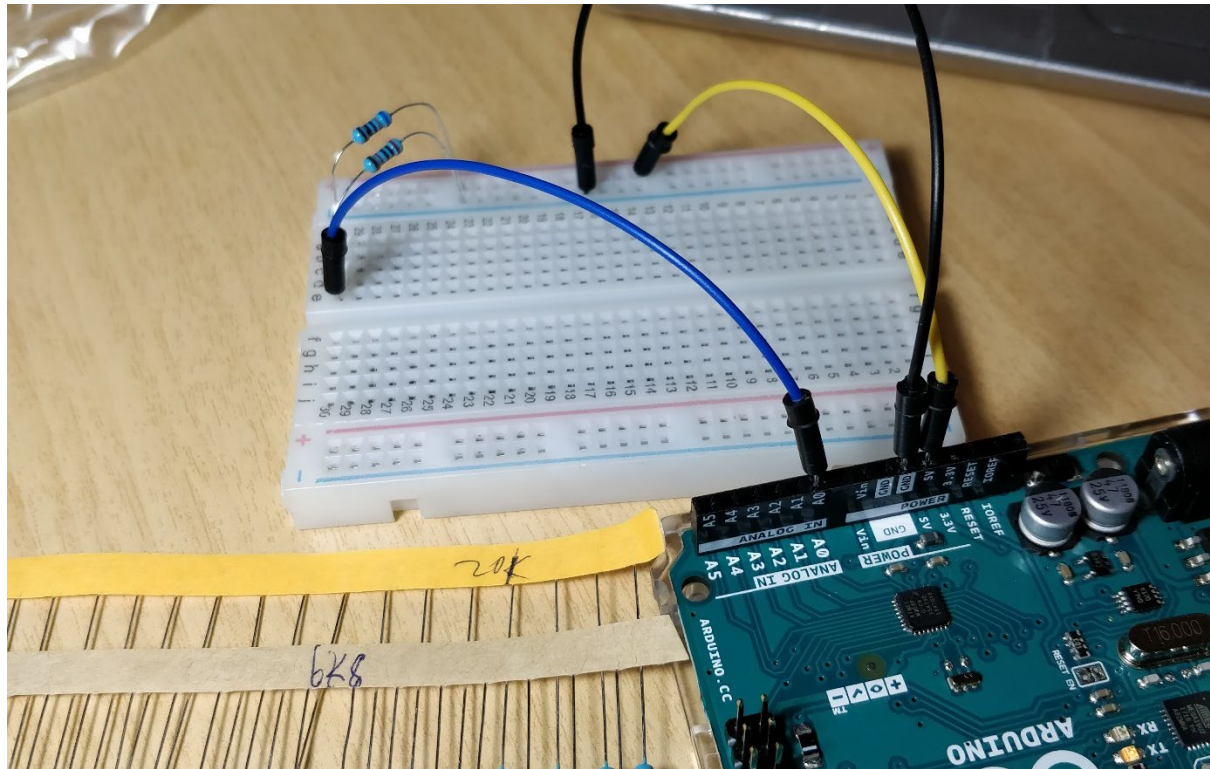
COM9 - Serial Monitor

```

16900 Low @ 3.74<0x0d>
16901 Low @ 3.74<0x0d>
16902 Low @ 3.74<0x0d>
16903 Low @ 3.74<0x0d>
16904 Low @ 3.74<0x0d>
16905 Low @ 3.74<0x0d>
16906 Low @ 3.74<0x0d>
16907 Low @ 3.74<0x0d>
16908 Low @ 3.74<0x0d>
16909 Low @ 3.74<0x0d>
16910 Low @ 3.74<0x0d>
16911 Low @ 3.74<0x0d>
16912 Low @ 3.74<0x0d>
16913 Low @ 3.74<0x0d>
16914 Low @ 3.73<0x0d>
16915 Low @ 3.74<0x0d>
16916 Low @ 3.74<0x0d>
16917 Low @ 3.74<0x0d>
16918 Low @ 3.74<0x0d>
16919 Low @ 3.74<0x0d>
16920 Low @ 3.74<0x0d>
16921 Low @ 3.74<0x0d>
16922 Low @ 3.74<0x0d>
16923 Low @ 3.74<0x0d>
16924 Low @ 3.74<0x0d>
16925 Low @ 3.74<0x0d>
16926 Low @ 3.74<0x0d>
16927 Low @ 3.74<0x0d>

```

I tested the battery monitoring voltage divider by using the Arduino's 5V and 3.3V pins, and 20k/6.8k ohm voltage divider to test supplying pin A0 with differing voltage levels, and testing the processing/output value.



## Final Code Development

Final version -

<https://github.com/JamesNZL/13CTE/blob/master/Arduino/Peg%20Cleaner/Peg%20Cleaner.ino>

I started off duplicating my most recent copy of Laser\_Tripwire.ino, which was a finished copy of the laser tripwire module with indicator LED integration, and now I needed to add the control for the solenoids, Arduino, and water level sensor, and make sure they all work alongside each other.

First I adapted my variable names and pin numbers into my final configuration as per my schematic, and then I added code to have the water level sensor turn the drain on and turn the LED red if it was above the danger level.

81	-	Serial.println(ldrValue);
	81	+ digitalWrite(jetGate, HIGH);
	82	+ lastAction = currentTime;
	83	
82	84	}
83	85	
84	-	Serial.print("Triggered @ ");
85	-	Serial.println(ldrValue);
86	-	}
	86	+ digitalWrite(jetGate, LOW);
	87	+ waterValue = analogRead(waterPin);
	88	+ if (waterValue > waterMinimum) {
	89	+ digitalWrite(drainGate, HIGH);
	90	+ digitalWrite(redPin, HIGH);
	91	+ digitalWrite(greenPin, LOW);
	92	+ digitalWrite(bluePin, LOW);
	93	+ }
	94	+ digitalWrite(drainGate, LOW);
	95	+ }
	96	
	97	
	98	
	99	+)

The next significant change I made was to move from the 'blocking' while loops towards if statements, which would allow other code to run alongside.

79	-	while (ldrValue >= triggerAt) {
80	-	ldrValue = analogRead(ldrPin);
81	-	}
	79	+ if (ldrValue >= triggerAt) {
82	80	digitalWrite(jetGate, HIGH);
83	81	lastAction = currentTime;
	82	+ } else {
	83	+ digitalWrite(jetGate, LOW);
84	84	}
85	85	
86	-	digitalWrite(jetGate, LOW);
87	-	

Some more changes and fixes I made include adding the time out functionality:

```

-unsigned long currentTime;
-unsigned long lastAction = 0;

24  bool jetsEngaged = false;
25  bool waterDanger = false;
26  bool buttonPressed = false;
27  +unsigned long currentTime;
28  +unsigned long lastAction = 0;
29  +unsigned long terminateTime = 180000;
30
31  void setup()
32  {
@@ -158,4 +159,13 @@ void loop()
159      digitalWrite(bluePin, LOW);
160      digitalWrite(powerGate, LOW);
161  }
162  +
163  +  if lastAction >= terminateTime {
164  +      digitalWrite(jetGate, LOW);
165  +      digitalWrite(drainGate, LOW);
166  +      digitalWrite(redPin, LOW);
167  +      digitalWrite(greenPin, LOW);
168  +      digitalWrite(bluePin, LOW);
169  +      digitalWrite(powerGate, LOW);
170  +  }

```

Which I had to fix as lastAction would go over terminateTime almost immediately, and so I needed to find the difference between the current time and the last action:

```

-  if lastAction >= terminateTime {
+  if ((currentTime - lastAction) >= terminateTime) {

```

I then began to refactor code into functions so that it was not all in one massive loop() function, instead the loop

() simply calls the sub-modules:

```

155 - digitalWrite(jetGate, LOW);
156 - digitalWrite(drainGate, LOW);
157 - digitalWrite(redPin, LOW);
158 - digitalWrite(greenPin, LOW);
159 - digitalWrite(bluePin, LOW);
160 - digitalWrite(powerGate, LOW);
155 + shutdown();
161 156 }
162 157
163 158 if ((currentTime - lastAction) >= terminateTime) {
164 - digitalWrite(jetGate, LOW);
165 - digitalWrite(drainGate, LOW);
166 - digitalWrite(redPin, LOW);
167 - digitalWrite(greenPin, LOW);
168 - digitalWrite(bluePin, LOW);
169 - digitalWrite(powerGate, LOW);
159 + shutdown();
170 160 }
161 +}
162 +
163 +void shutdown()
164 +{
165 + digitalWrite(jetGate, LOW);
166 + digitalWrite(drainGate, LOW);
167 + digitalWrite(redPin, LOW);
168 + digitalWrite(greenPin, LOW);
169 + digitalWrite(bluePin, LOW);
170 + digitalWrite(powerGate, LOW);

```

I needed to fix the battery voltage reading by actually converting it to a voltage:

```

- batteryValue = analogRead(batteryPin);
+ batteryValue = analogRead(batteryPin) * (5.0 / 1023.0);

```

My laser tripwire statement was backwards, meaning it would turn the jets on etc. if the laser was NOT blocked:

```

- if (ldrValue >= triggerAt) {
+ if (ldrValue <= triggerAt) {

```

I also added a 'fluctuation trigger', which if the LDR value changed outside of a tolerance while collecting testTotal, i.e. if it was reading an ambient light value of 100 then suddenly became 300, the fluctuation check would ensure the test is cancelled and restarted.

```

for (int i = 0; i < testIterations; i++) {
  int fluctuation = abs(analogRead(ldrPin) - (testTotal/i));
  Serial.print("Fluctuation: ");
  Serial.println(fluctuation);

  if (fluctuation > 10 && i != 0) {
    Serial.println("Fluctuation triggered");
  }

  if (fluctuation > 10) {
    Serial.println("Fluctuation triggered");
    break;
  }
}

```

Figure 20: Adding break statement so calculation reset