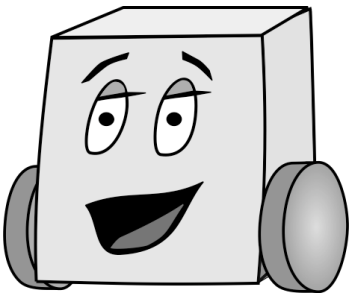


E11: Autonomous Vehicles

Fall 2010

Harris & Lape with Keeter & Ong

PS2: Arrays and Feedback

Part 0: Feedback Loop

For the first part of this assignment you will be using an LED and a light sensor to create a feedback loop. Name this sketch `ps20`.

Feedback loops are systems in which the sensing of an event leads to the modification of that event, which in turn affects the sensing of the event, and so on. There are two predominant types of feedback loops:

- Positive Feedback Loops -- These loops *increase* the events which trigger them. A common example of this is audio feedback, which results from a microphone "hearing" its own speaker, which results in an awful screeching sound. Although set up slightly differently, the video at http://www.youtube.com/watch?v=t-7mQhSZRgM demonstrates this concept well.

- Negative Feedback Loops -- These loops change their output in an effort to reach *equilibrium*. A common example of this is an air-conditioning system: if too high of a temperature is sensed, the system lowers the temperature; if a low enough temperature is sensed, the air conditioning stops running until the temperature rises again.

For this problem you will be creating a negative feedback loop in order to very carefully control an LED's brightness. You will use the phototransistor in order to sense whether the LED is too bright or not bright enough, and then your program will adjust accordingly.

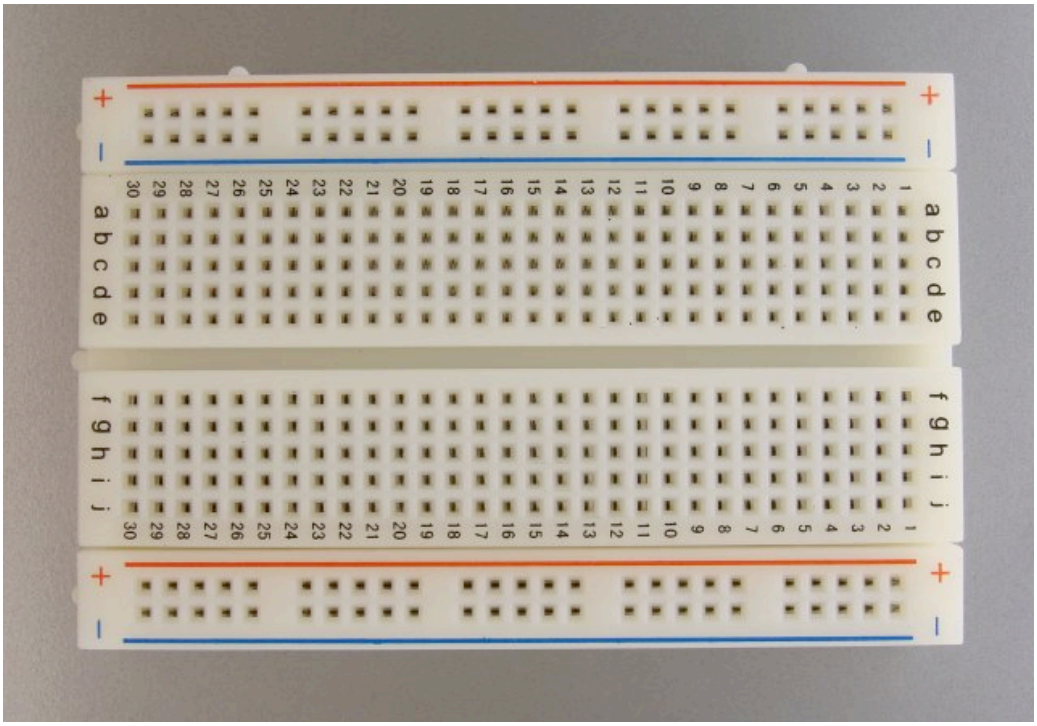
Thus, using Arduino, your program will take readings from the photosensor, and if the reading is *higher than desired*, your program will *lower* the output of the LED, whereas if the reading is *lower than desired*, your program will augment the output of the LED.

First, however, we need to set up the phototransistor and the LED!

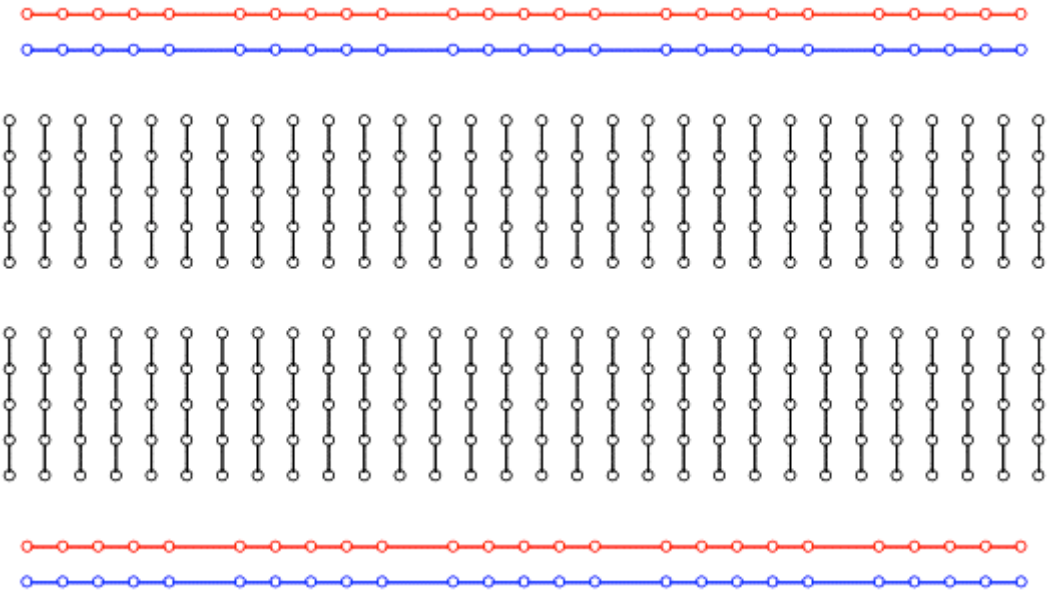
Setting up your Phototransistor

At the moment, you may not have a phototransistor on your Mudduino board. Therefore, the first thing to do is to connect the phototransistor (in your kit) to your Mudduino using a prototyping board, sometimes called a *breadboard*.

Here's a picture of your kit's breadboard:

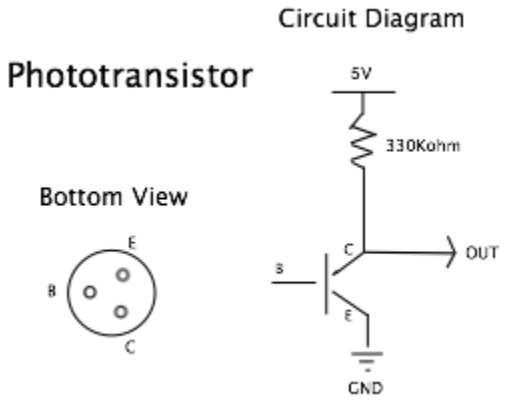


As you can see, the breadboard is just a piece of plastic with a bunch of holes in it. The usefulness of breadboards comes from the fact that many of these holes are electrically connected. This diagram shows those connections:



Holes connected to each other by a line in the diagram above are electrically connected in the board. Your goal is to use this fact, and a bunch of wires, to connect the parts of your phototransistor to the necessary parts of your board. The red and blue columns are conventionally used for power and ground, respectively, but this is not strictly necessary.

Next, let's look at the circuit diagram we will follow:



Where B is the Base, C is the Collector and E is the Emitter. Your phototransistor has a small metal tab jutting from its case near the collector.

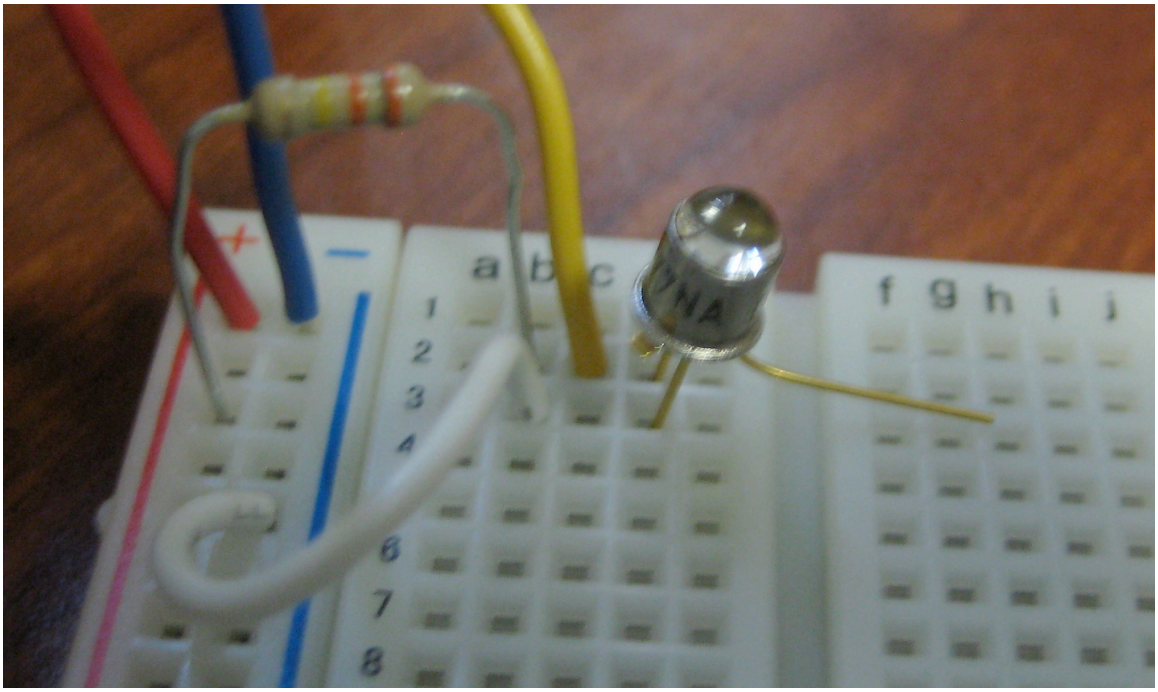
All the above diagram means is that:

- The emitter must be connected to the ground.
- The collector must be connected to the output
- The collector must be connected to the power through a 330 kiloohm resistor.
    - Note: 330 kiloohms is orange-orange-yellow
- The base need not be connected to anything.

We have bent back the base of your phototransistor for you already, so that you can tell the parts of your phototransistor apart more easily.

When we connected these different pieces of the above circuit, our result looked like the following picture.

Here, the red wire is coming from the Mudduino's 5V power header, the blue is running to the GND header, and the yellow wire is running to analog pin A1.



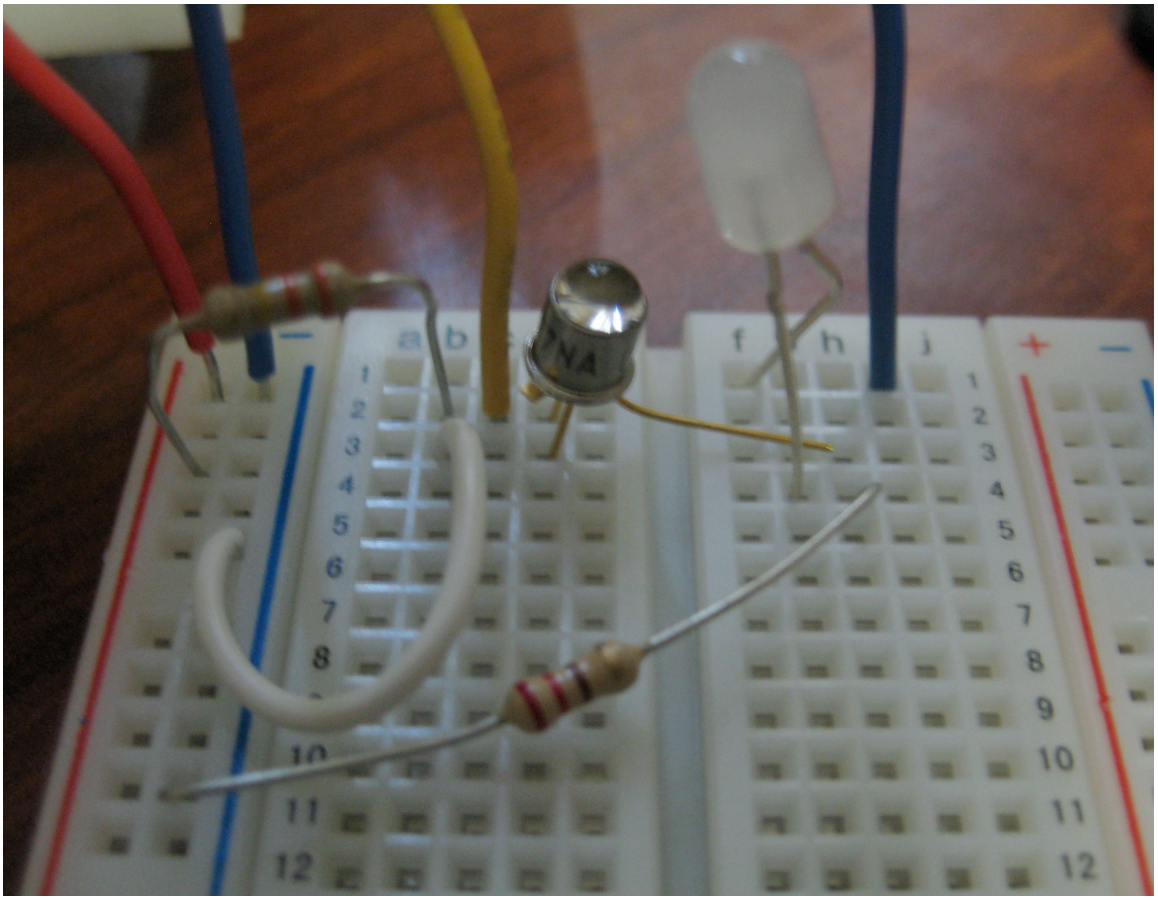
Making sure the light sensor works…

First, create a program to make sure your light sensor is working: you might use the example program from problemset #1 that tested the range sensor – the only difference will be the pin on which analog values are read.

The phototransistors are not completely identical, but you should be able to change the readings coming from yours by shining a light onto the sensor or by covering it with your hand. The readings run from 0 (brightest) to 1023 (darkest). Remember that to take analog input from your sensor, use `analogRead(pin)`.

Adding an LED to your breadboard…

Next, you'll put an LED on your breadboard so that the light it emits impinges on the phototransistor. The LED's shorter leg should be connected through a 220 ohm resistor (red-red-brown) to ground. The LED's longer leg should be connected to one of the digital output pins. We used pin D3 in this example:



In this picture, you see the blue LED has been taken out of the Mudduino and placed into the breadboard near the phototransistor. The leftmost blue wire leads from the D3 pin's header on the Mudduino board: it is placed in the same breadboard row as the *longer* LED lead. The 220 ohm resistor connects from the same row as the *shorter* LED lead to the same ground shared by the phototransistor.

With this set up in place, you can use the command `analogWrite(3,led_value)` in order to change the brightness of the LED. However, in order to make sure that `led_value` is in bounds, you should use these two lines before calling `analogWrite`:

```
if (led_value < 0) {led_value = 0;)

if (led_value > 255) {led_value = 255;}
```

These lines make sure that the value of `led_value` does not go above 255 or below 0.

Putting it all together, programmatically…

Finally, alter your program (ps20) so that the following steps happen in order that the system adjusts the LED output to achieve a desired level of light on the phototransistor:

- First, create an integer variable named `SET_POINT` at the top of your program. We started this at a value of 200. This variable holds the *desired light level* for the phototransistor.

- Next, create an integer variable named `error` that will represent the difference (or error) between the `SET_POINT` value and the value read by the phototransistor.

- Create a double-precision variable named `gain` that will act as a multiplier that converts error in phototransistor levels to the desired difference in led brightness levels.

- Finally, write a small program (or adapt the starter program below) that

    - finds the `error`, as defined above (based on the `SET_POINT` and the actual phototransistor level)

    - computes the change that it will make in the `led_value` (this is the `gain` times the `error`)

    - makes that change to the LED

    - prints out both the actual phototransistor level and the actual `led_value` level to the Serial port so that the user can watch the adjustment…

- Optionally, tune your `gain` and your delay so that the system hits its setpoint as quickly as possible without too much overshoot!

Here is a starting program that has a little bit of code to accept new `SET_POINT` values from the Serial interface.

```
// ps21 (light sensor)
// Light sensing with feedback!

int DIST_PIN = 0;     // the range sensor is on A0
int LIGHT_SENSOR = 1; // the phototransistor
int LED_PIN = 3;      // the pin that will control the LED

int sensed_value = 0; // variable for raw light readings
int SET_POINT = 200;  // the desired raw light readings

double led_value = 0.0;  // the strength of the LED


void setup()
{
  pinMode(LED_PIN, OUTPUT);
```

```
  Serial.begin(9600);
  Serial.println("Hello, World!");
}

void loop()
{
  // get current reading
  // see the warning below to improve this!!
  sensed_value = analogRead( LIGHT_SENSOR );

  // here is where your feedback goes!

  analogWrite(LED_PIN, led_value);

  Serial.print(sensed_value);
  Serial.print(" ");
  Serial.println(led_value);

  delay(250); // you may want to change this...

  // if you enter a number key N, this will
  // make the SET_POINT 100*N
  if (Serial.available())
  {
    int value = Serial.read() - 48;
    SET_POINT = value * 100;
    Serial.print("SET_POINT is ");
    Serial.println(SET_POINT);
  }
}
```

The field of feedback control is a rich one that is at the cornerstone of many branches of engineering. This example is only one of many times you'll have the chance to explore it in this project – and beyond.

Warnings, hints and things to consider:

- You will want to use analogWrite(LED_PIN, led_value), though perhaps with your own variable names, in order to set an analog brightness of the LED. analogWrite allows its second argument to range from 0 to 255.
- **WARNING!** If you take a ***single*** reading of the phototransistor and use it for feedback control, you may get a chaotically variable light level from the LED! This is because the analogWrite command (see the point above) uses *pulse width modulation* in order to achieve analog control over the LED's light level. That is, the LED is being flashed on an off very quickly to achieve a desired brightness. However, the phototransistor can sense those rapid flashes. To it, the LED is either all the way on or all the way off, so the outputs are variable!

- ***But how to fix this?*** Fixing this gives you a chance to practice using a for loop! Rather than taking a single analogRead reading from the phototransistor pin, you need to average several readings. We found that it worked well when we took 100 readings with a 50 microsecond delay between them. Here is an example for loop to get you started:

```
// take N readings and average them
total = 0;
for (int i=0 ; i<N ; ++i) {
  total += analogRead( LIGHT_PIN );
  delayMicroseconds(50);
}
```

- Remember that the phototransistor's readings are opposite of what you might expect: very high readings mean low light; low readings mean it has sensed a lot of light
- Ambient light: as the other lights in the room change, your feedback system should adjust accordingly. We started out with a setpoint of 200, which worked well.
- Low-light setpoints that are high work less well, because the LED's contribution is relatively low – that is, the fraction of the system that is under the board's control is less than for setpoints demanding more light.
- Your phototransistor has a relatively narrow viewing angle, so that you will need to ensure that the LED shines *directly* at the phototransistor

Part 1: Continuous Readings and Graphing

This part of the assignment will allow you to develop your skills with arrays and, optionally, work with the graphical language *Processing*. Processing is a sibling language to Wiring, the language you have been using on the Arduino.

You will begin by continuously taking readings from your phototransistor every tenth of a second. You will then store the latest ten of these readings in an array, and send this array over Serial every half second (so that half of the values are updated every time Serial is updated). For this problem, all that is required is that you print these values to the Serial monitor. However, see the extra credit for a more elaborate interface! For this problem, name your Arduino sketch `ps21`.

First, make sure you have `Serial.begin(9600);` in your `setup()` function. The rest of your code can then go in the `loop()` function.

The first thing we will consider is an alternative to the `delay()` function. This function is certainly useful, but the problem is that when `delay` is called *no other code can*

execute in the meantime. Thus, if you want to perform one action every second but also be able to do other things in between, then `delay` will not suit your needs.

To work around this, consider the following example program that toggles an on-board LED every 10th of a second while sending a friendly message twice per second.

```
//
// ps21 using delays
// Fun!

int DIST_PIN = 0;     // the range sensor is on A0
int LIGHT_PIN = 1;    // the phototransistor
int SPEAKER_PIN = 4;  // for playing notes
int LED_RED = 5;      // red led pin

unsigned long prev_LED_time = 0; // previous LED time
unsigned long prev_MSG_time = 0; // previous MSG time
unsigned long curr_time = 0; // current time

int LED_interval = 100;  // 100 milliseconds
int MSG_interval = 500;  // 500 milliseconds

int led_status = LOW;

void setup()
{
  pinMode(LED_RED, OUTPUT);
  Serial.begin(9600);
  Serial.println("Salam, Dunya!");  // Azeri
}

void loop()
{
  curr_time = millis();  // get the current time

  // check if it's time to toggle the LED
  if (curr_time - prev_LED_time > LED_interval)
  {
    if (led_status == LOW) { led_status = HIGH; }
    else { led_status = LOW; }  // toggle led status
    digitalWrite( LED_RED, led_status ); // go!
    prev_LED_time = curr_time;  // reset previous!
  }

  // check if its time to send a message
  if (curr_time - prev_MSG_time > MSG_interval)
  {
    Serial.println("Still going... :-)");
```

```
    prev_MSG_time = curr_time;  // reset previous!
  }

  // no delay needed!
}
```

You can use this technique for any number of tasks you need to accomplish: you will simply need to have different variables to keep track of time elapsed for each task.

For practice, try changing this program so that it blinks the red LED 10 times per second, the yellow LED 3 times per second, and the green LED 7 times per second… some interesting syncopation will result!

Using an array to store light readings…

We will use an array of integers to store the readings from the phototransistor.

Recall that, to declare an array, you will need a line like the following at the top of your program:

```
int light[10];
```

As you can see above, you first declare the type of the *contents* of your array, and the variable name must have `[]` brackets at the end, with the length of the array in the square brackets. Inside the brackets you should put the desired *length* of your array.

Recall that, using the array declared above, you can store up to 10 values of the type `int`. Accessing and changing the contents of an array location is a simple matter: you can use the expressions

```
light[0]  light[1]  light[2]  . . .  light[9]
```

anywhere that you would use another variable or value of type `int`. These array location numbers are called the index, and keep in mind that the index runs from 0 up to one less than the length of the array.

For printing your array to Serial you will want to use a `for` loop. Remember that `for` loops follow this format:

```
for (i=0; i<10; i=i+1)
{
   // this code will loop for each index i, from 0 to 9
}
```

Connecting all of these pieces…

So, to finish this program, alter the framework that you started, above, so that

- Every tenth of a second, a new value is read in from the phototransistor
- The value read is placed in the array at the location of the OLDEST data value.
- Every half second, a line of ten integers is sent over Serial, drawn from the array from oldest to newest.

Your output to the serial console should look something like this, though not these exact values!

```
380 390 381 391 355 358 348 370 361 0
370 366 348 352 366 395 374 370 381 354
381 388 381 354 345 348 367 347 349 376
363 374 368 381 378 391 383 385 364 369
351 374 352 341 365 364 371 452 468 347
601 785 857 892 909 903 917 914 916 535
921 920 917 919 927 918 918 927 917 917
950 922 918 924 917 920 938 803 917 916
375 357 353 366 370 356 360 364 415 392
382 383 264 92 41 34 39 24 381 374
9 8 19 7 6 15 12 17 23 31
9 11 544 402 368 373 356 365 14 8
370 377 376 382 389 412 389 363 372 365
```

If you'd like, you may want to use the Processing language to make a serial connection to your Mudduino, grab the values, and plot them graphically on the screen. The starter code to do this is online at the E11 programming hints page. This is entirely optional, but it's fun and not a bad introduction to the graphical language Processing!

Part 2: Memory Game

In the final part of this assignment, you will implement a Memory Game similar in spirit to Simon (see http://en.wikipedia.org/wiki/Simon_(game)) – call this Arduino sketch ps22. In creating this memory game, you will gain practice with

- creating and using arrays
- writing loops that run through the values within arrays
- writing functions in order to encapsulate behaviors

First, we will describe the game to implement. Then, we will provide an example function to use (named `getKey`) and details on two more functions that you will write for this program (named `printToSerial` and `correlate`).

The memory game…

The game should be played as follows:

- First, your Mudduino should create 8 random LED flashes, some of them green and some of them red.

- Then, the user should enter eight keypresses (hitting enter each time) into the Mudduino – this should be an attempt to replicate the LED flashes. For convention, let's let 1 represent green LED flashes and let 0 represent red LED flashes.

- Then, the Mudduino should compute the user's *score*, by comparing the record of the correct LED flashes (1s and 0s) with the keypresses the user entered (1s and 0s). The score will be the *correlation* between the two sequences, as defined below.

- Finally, the Mudduino should ask the user to hit a key before playing again.

How to represent this data…

In contrast to some of the previous problems, we won't specify exactly how to create this program. However, we will provide some hints here. As you go, be sure to build small pieces of functionality, test them, and then continue – and seek out help (from your partner, from a tutor, or from one of us) if you run into trouble!

This game requires at least two arrays, each of length 8. The first array will hold the actual sequence of LED flashes. The second array will hold the sequence of keypresses that the user typed in at the Serial console. (If you do want to emulate our solution, we called these arrays `correct_sequence` and `user_sequence`.)

At the start of the program (in `loop`), you will need to create 8 random values (each either 0 or 1) and store them into your "correct" array. Either at the same time – or just afterwards – you will want to flash the LEDs according to that same pattern. We used 0 to represent red and 1 to represent green.

Then, you will need to input 8 keypresses from the user. ***Use the following function in your*** code in order to get those eight keypresses:

```
int getKey( int msg )
{
  Serial.print("Input # "); // prompt the user
  Serial.println(msg);      // finish the prompt

  // if no keypress has come in, wait for 100 ms
  while (!Serial.available()) { delay(100); }

  int value = Serial.read();  // read the keypress that came in
  return value - 48;  // return its numeric, not ASCII, value
}
```

Paste that function above `loop` in your code. Then, you will be able to use it whenever you want to *wait* for the user to type a key. The above function returns the numeric value

of the key pressed (for 0-9, at least). The input `msg` allows you to prompt the user with where they are in the list (or, it could be used to "cheat," by telling the user what to input!)

Computing the score…

The score will depend on the similarity between the "correct" array of values and the "user" array of values. You should implement a function that computes the *correlation* between those two arrays. That correlation – the output of the function you write -- should be the score for the game.

For the purposes of this problem, to calculate the correlation of two n-bit binary sequences, you should do the following:

Start with a correlation of 0 and march over each of the n bits:

- If a pair of bits at the same location are the same, add 1 to the correlation
- If a pair of bits at the same location are not the same, subtract 1 from the correlation

For example, for the following 8-bit binary sequences:

```
 0  1  1  0  0  1  1  0
 0  1  0  0  0  1  1  1
----------------------
+1 +1 -1 +1 +1 +1 +1 -1  = 4
```

So the above sequences have a correlation of positive 4: that is, the number of correct guesses reduced by the number of incorrect guesses.

Thus, the only possible correlations (scores) you *can* get for an 8-bit sequence are:

8 – for 8 correct; 0 incorrect
6 – for 7 correct; 1 incorrect
4 – for 6 correct; 2 incorrect
2 – for 5 correct; 3 incorrect
0 – for 4 correct; 4 incorrect
-2 – for 3 correct; 5 incorrect
-4 – for 2 correct; 6 incorrect
-6 – for 1 correct; 7 incorrect
-8 – for 0 correct; 8 incorrect

Another way to calculate correlation, which you are welcome but not required to use here, follows these rules:

- Change every 0 to a -1

- *Multiply* the two bits in every place
- Sum the results

Thus for the 8-bit sequences that we used above:

```
-1  1  1 -1 -1  1  1 -1
-1  1 -1 -1 -1  1  1  1
----------------------
 1  1 -1  1  1  1  1 -1  = 4
```

with the same result as before.

So, create a function named **`correlate`** with the following signature:

```
int correlate( int A[], int B[] )
```

that computes the above-defined correlation between array A (the first input) and array B (the second input). How will you know the lengths of the arrays A and B? The simplest way is to use a globally available variable to store the lengths. For example, you could declare your arrays as follows:

```
const int LEN = 10;
int A[LEN];
int B[LEN];
```

Then, you should be able to use **`LEN`** throughout your program. If you'd like your **`correlate`** function to work without reference to the global **`LEN`** you could pass in the length as a third input: **`int correlate(int A[], int B[], int length)`**.

Be sure to test out your function! A good way to do this is to simply place values of your own – perhaps the example values from higher on this page – into each array and the print out the result of the correlate function.

Another function to write…

A second function that you need to write for this program has the signature

```
void printToSerial( int A[] )
```

This function's first input is an array of integers. Like **`correlate`** it can obtain the length of the array from the global variable **`LEN.`** Alternatively, you could include a second input representing the length of that array.

This **`printToSerial`** function does not return any value! Instead, it should print the contents of the input array to the Serial console, with a single space between each one, and with a newline at the end.

This **printToSerial** function will help because it will make checking your overall memory game's performance as easy as the following lines:

```
printToSerial( correct_sequence, LEN );
printToSerial( user_sequence, LEN );
correlation_value = correlate( correct_sequence,
                                    user_sequence, LEN );
Serial.print("Correlation: ");
Serial.println(correlation_value);
```

Other Notes:

- Keep in mind that you will want a `delay()` of some length in your program, so that the LEDs are not flashing too quickly for the user to see. It is up to you how long this delay is, and therefore how difficult your game is! You will also want a short delay *between* the flashes, when both LEDs are off.
- On making your random binary sequence: You will of course want to use the `random()` function that you used in the last assignment, but you will need to account for something that we did not mention before. If you play the game several times, you'll notice that `random()` is not so random after all – in fact, it starts at the same value each time the board is reset! This isn't a problem for debugging (in fact, it's great!), but it makes the game too easy after a while. If you'd like to make it more challenging with truly random values, read how **`randomSeed()`** is used in the Arduino reference.

Deliverables

You are responsible for turning in the following Arduino files:

- **ps20.pde**
- **ps21.pde**
- **ps22.pde**