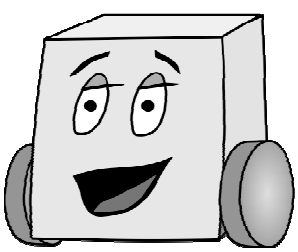

E11: Autonomous Vehicles

Fall 2010

Harris & Lape with Keeter & Ong

PS 4: Gold Code Detection

For this assignment you will read a code that is being transmitted, and compare it to a known Gold Code (which we will give you). The biggest challenge associated with this task is compensating for signal interference, caused by a combination of distance from the signal and atmospheric "noise". This assignment will have two sketches, named `ps41.pde` and `ps42.pde` (naturally).

Part 0: Setting an LED Signal

With this part of the code, start with the provided program (below and on the hints page), get to know how it works, and try it at different "speeds." Save this sketch as `ps41.pde`. Then, you'll alter this provided program to create your `ps41.pde` sketch, which will blink an LED according to the data in an array, e.g., a Gold Code.

For now, however, the largest new piece of the program below is the *timing* calls. Its overall goal is to perform some task every `LOOP_TIME` microseconds. The first task you work with will be turning on and off an LED. Since `LOOP_TIME` starts at one million, this means that – to begin – that LED task will occur once per second. ***Read over this code carefully – and run it – in order to understand what it is doing and how.*** The next paragraph explains a bit of the *why*.

As is typical in Arduino programs, this program uses the `loop()` function as its basic loop. At the top of that loop, it checks the current time. If not enough time has elapsed since the last time the task was performed, nothing happens. On the other hand, if enough time ***has*** elapsed since the last time, the conditional test is true, and the following actions occur: (1) the `next_time` variable is updated – this resets it for subsequent LED flashes, (2) the `LED_level` variable is toggled (from 0 to 1 or from 1 to 0), (3) one of the LEDs is flashed on or off, depending on the value of the `LED_level` variable, and (4) some printing and a small delay are run – mostly to show how much *actual* time passed and to demonstrate how to add small delays to your code.

Here is the code; you shouldn't need to add anything else to compile and run it:

```
int PHOTO_PIN = 1;    // the phototransistor
int LED_PIN = 7;      // the GREEN LED is 7
long LOOP_TIME = 1000000;  // number of microseconds per loop

void setup()
{
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(9600);
  Serial.println("Hi there!");
}

long curr_time = 0;  // the current time
// the next time we want to take some action...
long next_time = curr_time + LOOP_TIME;
int LED_level = 0;
int sensed_light_level = 0;

void loop()
{
  // An example of how to loop every LOOP_TIME microseconds
  curr_time = micros();      // get current time...
  if (curr_time > next_time) // is it time yet?
  {
    // add our timer interval to next_time
    next_time = next_time + LOOP_TIME;
    // do our task - change the light level
    if (LED_level == 0) {
      LED_level = 1;
    } else {
      LED_level = 0;
    }
    // set the light level
    digitalWrite( LED_PIN, LED_level );
    // print the time - note that this takes a lot of time!
    Serial.print("Time was ");
    Serial.println( curr_time );
    // you will need to get rid of the printing, above,
    // in order to run at 4000 Hz! (250 microseconds per loop)

    // We can wait now, but remove this delay later...
    delayMicroseconds( 100 );
```

```
    // this is code for reading from the phototransistor
    // it is commented out for the moment
    /*
    sensed_light_level = analogRead( PHOTO_PIN );
    Serial.print("light level: ");
    Serial.println(sensed_light_level);
    */
  }
}
```

Be sure to try out this code and read it over so that you understand how it works.

As a useful task to help get to know this code, try ***reducing*** the value of the `LOOP_TIME` variable to see how fast you can flash your LED. Of course, you can reduce `LOOP_TIME` all the way to zero, but there will still be time taken by the statements within the if. See if you can get the loop to run at 4000 hertz (once every 250 microseconds): this is the rate that the beacons transmit codes in the test arena, so it will be a useful capability to be able to replicate. To do this, you will have to comment out the printing and the delays in the loop.

For debugging however, bring back those printing lines before moving on to the next part, which will ask you to change the LED flashes to the transmission of a code stored in an array. Also, change `LOOP_TIME` back to one million (a one-hertz update rate) in order to make it possible to verify by eye that things are working correctly!!

Part 1: Generating a Code

For this part, continue working with the `ps41.pde` sketch that you used in the above problem. Next, improve the program in these ways:

(1) Change the program so that it has an array that holds a binary code of your choice. Eventually this will be a 31-element Gold code, but I found a 5-element array easier to debug initially. For example, above the loop() function you could declare the following:

```
const int LEN = 5;
int code[LEN] = {1,0,1,1,0};
```

(2) Change the `loop()` function so that, rather than flash its LED on and off repeatedly, it instead repeatedly ***cycles through*** the values in the array named `code`, above.

Thus, your program should first set its LED high (on), then low (off), then high (on), then high (on), then low (off), and then back to the beginning of the sequence again, repeatedly.

To do this, you *won't* need a loop beyond the existing `loop()` function! However, you *will* need a counter variable that keeps track of *which* element in your array is currently determining the state of the LED. Each time the LED updates, that counter variable will update, too.

(3) Test again that this works at faster speeds than one LED update per second! Also, iot would be a good idea to try different length arrays (codes), to ensure that your program can handle those, too. For example, here is one of the Gold Codes you worked with last week, although *any* 31-bit sequence will do for the moment:

```
const int GCLEN = 31;
int GC[GCLEN] = {0,1,0,0,0,1,1,0,0,1,1,0,0,1,1,1,1,0,0,1,0,1,0,0,1,0,1,1,1,1,0};
```

Once you have this working, you are done with `ps41.pde` – go ahead and submit that file… . Next you'll create a new ***copy*** of this sketch (under the name `ps42.pde`) and write a similar program that ***reads*** light levels using the phototransistor.

Part 2: Single Signal Reading

To start, be sure to save a copy of that previous program under the new name `ps42`.

Also, you will want to join **with a partner** for this portion of the problem set. After all, the goal here is to ***detect*** the code that a flashing LED is transmitting. Thus, one partner can use the LED flashing program, and the other (likely the team) can be developing the light-*reading* program here.

Warning: Remember that the phototransistor has a very small viewing angle, so you will need to line up the LED and the phototransistor very precisely.

In fact, I used the very bright blue LED (with the same setup as in the phototransistor problem of week 2) instead of the red, yellow, and green LEDs, which are a bit dimmer. It is true, though, that by placing the phototransistor very near the flashing LED, it's possible to use any of them.

What to do: Part A ~ single light readings

First, remove from this *light-reading* program the code that flashes the Mudduino board's LED.

Next, set up your phototransistor in a manner identical to week 2's problem that used that sensor. If you use a different analog pin to read inputs from that sensor, be sure to make the appropriate change in the code itself.

Finally, take a reading once for each update loop and print its value to the Serial console. By "update loop" we mean whenever the value of the `curr_time` variable exceeds the value of the `next_time` variable, just as in the LED-flashing part of this problemset.

Make sure that your phototransistor is working, i.e., that the values it is reporting are changing with changing light that hits it (cover and uncover it, perhaps). Lower values indicate *more* light; higher values indicate *less* light.

What to do next: Part B ~ an array of light readings

For this part, you will adapt your program (the same one, ps42.pde), so that it reads in values from the light sensor ***at the same rate*** as the LED-output program (ps41.pde) is emitting them and ***into an array of the same size*** as the length of the sequence that LED-output program is using.

Not quite as many details are provided here as in the array-based LED-flashing example, but the ideas are quite similar. You will want to (1) create an array of the appropriate length into which you will store those values, then (2) make sure that you use a counter to

keep track of ***which*** array location the next value will go. Finally, (3) after filling the array each time, **print to the Serial console** the values from the whole array.

In order to accomplish this last part, we would suggest grabbing and using your `printArray` function that you used in prior weeks!

To test this new capability, be sure to set up a flashing LED that is running the ps41.pde program very near (and central) to the phototransistor. Be sure the update rates and array lengths match. Then, make sure that the code read is the same as the code emitted – but remember that the code may be *shifted* and that the code is binary, but the light levels are not… . We will address these two differences in the final two parts of this problem set.

What to do next: Part C ~ an array of ***binary*** light readings

Of course, the light-levels read will ***not*** be the same as the code emitted, because the light levels have values between 0 and 1023, but the code is binary. You will address that in this part of the problem set.

So, once you have Part B, above, working, alter your program so that

(1) ***After getting one array-full of light-level values***, your program finds the *average of the values in that array.*

(2) Your program then ***binarizes*** the array of light values. That is, values **above** the average for that row are set to 0 (remember that higher values are lower light!!) and values **below** the average for the row are set to 1.

(3) Keep the other behavior of the program the same, so that you can ***check to be sure*** *that the binary code received is the binary code emitted*! Again, it may be shifted, however. We turn to that next.

What to do next: Part D ~ correlating to determine a matching Gold Code

Gold Codes are used because they have very nice properties! In particular, a Gold Code of length 31 has a maximum correlation of 31 among the shifted versions of itself, but a correlation of no more than 9 (in absolute value) with shifted versions of other Gold Codes.

Thus, for this final part of this problem set, make sure you are using one of the Gold Codes provided in problem set 3 (and on the website). Then, extend your program so that

- After taking a complete 31-element array of light values, …
- It finds the average of all 31 values and creates the binary code as above, …
- It then correlates the resulting binary sequence ***with each possible shift*** of the Gold Code that was transmitted or another Gold Code (remember it will have a high maximum correlation with the correct Gold code and a much lower maximum correlation with any other Gold Code in the same family).
- And as it does so, it should print out the correlation results for each of those shifts.

Finally, your program should report whether the received signal ***is*** or ***is not*** the Gold Code that was being checked.

Try it out both ways! Use the code you wrote last week to help you!!

Of course, all of this processing and printing is liable to take a lot of time – this is OK.

After all, it's only the individual light readings that must be taken on a strict schedule (whatever rate they're being transmitted).

Once the light levels are read, then lots of processing can go on to determine ***which*** Gold Code best matches the values seen, i.e., which beacon your board is close to.

Speed! Try speeding up the light-emitting and light-reading programs – if you can get them to work while updating at 4000 hertz (a 250 microsecond LOOP_TIME), then your system can handle the beacons on the vehicles' playing field. It's ***not necessary*** to have this working this week!! It does require do very little work other than reading and storing values in the loop – until it's time to print out the results, that is!

Congratulations! you're finished with ps41.pde and ps42.pde – go ahead and submit those files. (You might want to be sure that each partner tries each one… .)

Some optional ideas for improving this Gold-Code detection appear below – you can try them, or you might wait until later when additional robustness starts to become more important… .

Part 3: Robust Gold Code detection

However, taking only ***one*** set of light-level readings might not be ideal – there are lots of sources of light noise (other sources in the room, other codes being transmitted, noise in the sensor, etc.) This section provides some ideas for making your Gold-Code detection more robust.

You ***do not*** need to work on this for this fourth problem set… these suggestions are really meant as possibilities that we suspect will be important as you start to deploy your vehicle in less-controlled settings.

One way to make Gold Code detection more robust is to take ***several*** sets of readings and use that redundancy to eliminate some of the effects of noise.

For example, you could take 8 sets of 31 light readings in a row, for a total of 248 read values. Keep in mind that reading and storing a value from the phototransistor must not take more than 250 microseconds. To keep things simple and fast, we recommend storing the values in an array of length 248, instead of trying to store the values in a 2-d array.

Thus, you could write a new copy of your program, call it ps43.pde, to do this.

Remember that, for debugging purposes, you should also be printing these values to the Serial Monitor – but only ***after all eight sets have been read!!*** It is most useful to print them in rows of 31, that is, have each set of 31 values on a separate line, where the 8

different readings for each bit should be visible in their own column. For example, here are some printouts from our program. The light values are small, so that each row of 31 fits on one line; the binary values are larger:

```
Light values:
69 16 70 164 270 375 477 562 633 690 16 15 16 69 165 269 16 69 16 16 16 71 16 15 16 72 17 71 167 275 16
15 15 70 169 275 380 478 565 636 696 16 16 16 70 163 271 15 71 16 16 15 71 16 16 15 71 16 71 158 258 15
16 16 69 160 266 370 467 553 627 689 16 16 16 71 165 268 16 70 16 16 16 70 16 16 17 71 15 70 171 286 16
16 16 72 170 273 377 473 560 633 695 15 16 16 72 166 272 16 71 15 16 16 72 15 16 16 71 15 71 164 271 15
16 15 69 160 259 357 455 539 610 671 16 16 15 69 166 273 16 69 16 16 16 68 16 16 16 69 16 70 164 270 16
15 16 72 178 291 401 496 583 651 705 15 16 16 71 166 275 15 70 16 16 16 71 16 16 16 70 15 74 164 267 16
16 16 70 161 267 364 455 534 607 659 16 16 16 69 162 266 16 71 16 16 16 72 16 16 16 70 16 68 163 267 16
16 16 71 166 272 384 483 576 657 722 16 16 15 71 164 269 15 72 16 16 16 73 16 16 15 71 16 71 164 272 16
```

```
Binary values:
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
```

Although this data worked well, as we moved the emitting LEDs away, the rows became noisier and noisier.

With this redundant data, you could take the majority bit in each column in order to allow for possible noise in the signal. If a column had an equal number of 1s and 0s, it would be an indication of a poor data set or other unfavorable conditions, for example.

Even further, you could add robustness to your system by correlating the beacons' Gold codes with white noise (the binarized values obtained when ***no*** signal is present). You will want your threshold for "matching" Gold Codes to be far enough above the correlation with white noise so that false positives do not happen very often!

Have fun with Gold Codes!