ENGR 40M Project 3b: Programming the LED cube

Prelab due 24 hours before your section, July 31–August 3
Lab due before your section, August 8–11

1 Introduction

Our goal in this week's lab is to put in place the mechanics to drive your LED array. Next week, you'll figure out something cool to do with your array.

When writing software to display patterns, we'd like to think about it in its *physical* layout, because it's a lot easier to conceptualize. But *electrically*, your cube is really a 2-D array and time-multiplexing runs in 2-D. Therefore, you'll want to write software to *abstract* such hardware details away from the rest of your software. This requires you to write a *mapping* between the two representations.

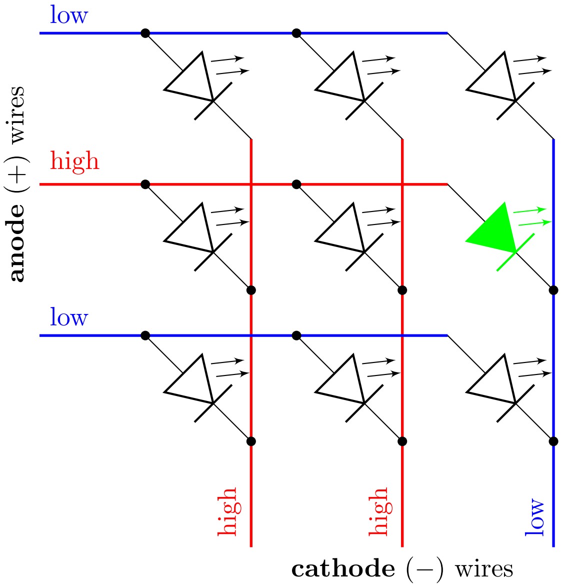
By completing this lab you will:

- Implement a time-division multiplexed driver, which is used in many electronic displays.
- Build a reasonably complex circuit on your breadboard, and learn how to debug it.
- Organize software to abstract hardware details from the rest of the code.
- Gain skills debugging a hardware/software system (because you will make mistakes along the way!)

2 Prelab

2.1 Our LED multiplexing strategy

An LED is on when, and only when, its anode (+ side) is high and its cathode (− side) is low. Any other combination (high/high, low/low, low/high) will keep the LED off. Therefore, we can turn on any *single* LED by settings its anode (+) wire high and its cathode (−) wire low, and setting *all other* anodes low and all other cathodes high.

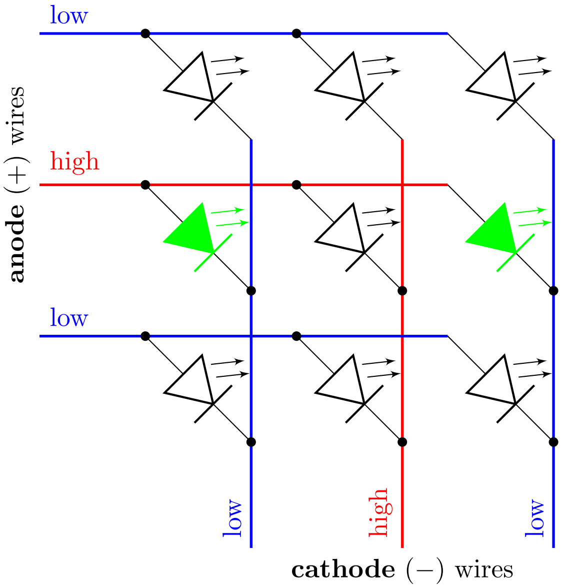


While the figures show a 3 × 3 array, but the following questions refer to the **8 × 8 array** that you will actually build.

A one simple way to time-multiplex the LEDs would be just to turn them on one at a time. If cycle through all 64 LEDs fast enough, your eyes will "fuse" them together and it will look like the LEDs that are on are on constantly—just dimmer, since they're not always on. To understand this, if an LED is turned on and off very rapidly such that on average it is on $\frac{1}{n}$th of the time, we say that its *relative brightness* is $\frac{1}{n}$.

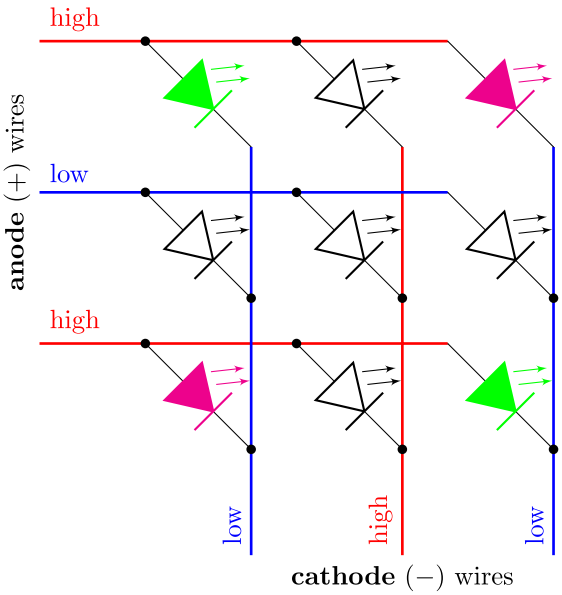
P1: Consider a single LED that is turned on. Under the above regime, where we cycle through LEDs one by one, what is the relative brightness of this LED?

We can improve this by turning on more than one LED at a time. More precisely: We can cycle through *anode (+) wires*, setting them to high one at a time, but now set *all* of the *cathode (−) wires* corresponding to LEDs we want to be on *in that row* to be low. Because every other row is forced to be off (by a low anode wire), and each LED in a single row is on a different cathode wire, this still allows us to maintain independent control of the LEDs.



P2: What is the relative brightness of a single LED under this scheme?

We might be a little more ambitious still, and ask: if we can have multiple *cathode (−) wires* low at the same time, can we also have multiple *anode (+) wires* high? Sadly, things fall apart. Suppose we wanted to light up the top-left and bottom-right LEDs in the diagram below. We can't do this without turning the two other LEDs on the same wires:

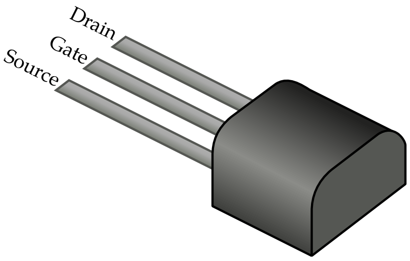


To keep this from happening, we need to make sure we're only driving one anode (+) wire at a time.

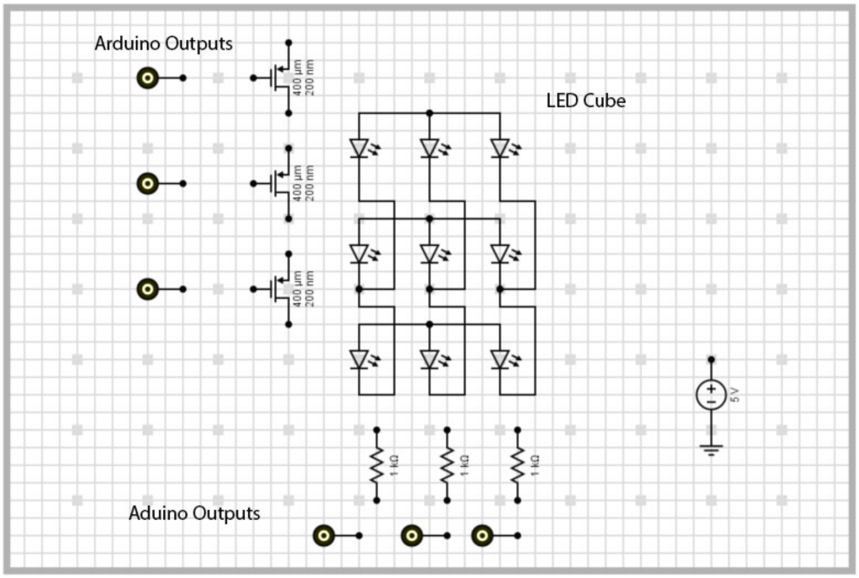
2.2 LED driver circuitry

Driving a whole row at once requires that we supply much more current to the + (anode) wires: depending on the pattern being displayed, we may need to supply up to eight LEDs. During lab 2b (smart useless box), we found that the Arduino can only supply a limited amount of current: about 40 mA per output pin. In order to supply more, we use the same type of solution we used in the useless box: a transistor.

For this project, we've given you a handful of BS250 PMOS transistors in a TO-92 package (cylindrical with a flat side). Please note that **the pins on this PMOS are different from those from lab 2b**.



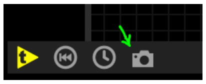
To help you figure out how to drive your LED cube we have created an EveryCircuit template for you to use. The template is located at http://everycircuit.com/circuit/6685272118919168, and you should be able to find it by searching for E40M Cube Driver. The template is shown below.



The circles labeled "Arduino outputs" model digital outputs. If you click or tap them they change state (LOW to HIGH or vice versa) so you can use them to simulate your Arduino outputs. When testing this array, make sure that only one pMOS transistor is turned on at a time.

P3: Complete the LED driver in EveryCircuit, and adjust the resistor values to get the right currents (20 mA) in the LED array. Please submit a screenshot of the completed circuit.

This button might be useful:



2.3 Arranging it on the breadboard

When you get to lab you will need to build eight transistor drivers, and add a resistor for each of the cathode (−) wires. Since this is a lot of stuff to fit, you should do a little planning before you get into the lab about how you will layout these components on your large breadboard. Remember to use the power and ground rails of your breadboard. You will need to have some long wires in this design (since you are using so many Arduino pins) but you should try to minimize the number of long wires. You should try to create one layout for the transistor driver that you can replicate, which will make it easier to build.

Note: Do not use I/O pins 0 or 1, because these pins are used by the Arduino for serial communication.

P4: Sketch how you plan to arrange your driver circuit on your breadboard, using either the breadboard template below or an online program such as Fritzing (http://fritzing.org/download/). This sketch doesn't need to include every component (you can indicate you replicate a circuit 7 more times), but should figure out where each transistor and resistor will go. Provide enough detail so your TA can catch any strategies that might lead to problems in the lab.

Arduino
MICRO
TX RX

3 Building the driver circuitry

1. Double-check your circuit with your TA before you start building.
2. Solder each of your eight anode (+) and eight cathode (−) wires to your cube (to lead to your breadboard). It's convenient to use CAT5 (Ethernet) cable for this, because it contains eight uniquely-colored wires inside.
3. Build the circuit you designed on your breadboard. Use the 82 Ω resistors from your kit. Use your large breadboard for this project. Talk with your TA to get any feedback on your approach to creating this circuit, and then start building. We suggest that you build a couple of transistor drivers and check them out first (either with the cube or a single LED) to make sure you are connecting the transistors correctly before doing all right of them.

That's all the hardware for this project! Now it's time to tackle the programming.

4 Software

4.1 First steps: Checking each LED individually

The first thing to do is to turn on each LED individually, to make sure all your circuitry is working.

1. Set up two arrays with the "names" of the anode (+, "row") and cathode (−, "column") pins. Having these in an array means that the rest of the code need not care which pins get used; it can just index into the array. This makes your code much easier to read, and to fix if you need to reorder your wires.

   You can use any I/O pins **except** (digital) pins 0/1, which are used by the Arduino for serial communication. A0–A5 of the analog pins can also act as digital I/O pins, so there are 18 pins at your disposal. But, **note**: for next week's project it would be good to leave a couple of analog (A0–A5) pins free.

```
// Your pins will probably be different.
// Remember that analog pins (A0, A1, ...) can also act as digital
const byte ANODE_PINS[8] = {2, 3, 4, TODO};
const byte CATHODE_PINS[8] = {13, A0, TODO};
```

2. Complete the code below to create a nested loop that flashes all the LEDs:[1]

```
// TODO: Turn off everything

for (byte aNum = 0; aNum < 8; aNum++) {
  // TODO: Turn "on" the anode (+) wire (high or low?)
  // You can get the pin name with ANODE_PINS[pNum], and pass
  // that to digitalWrite.

  for (byte cNum = 0; cNum < 8; cNum++) {
    // TODO: Turn "on" the cathode (-) wire (high or low?)
    // Again, you can get the pin with CATHODE_PINS[nNum]
    // TODO: Wait a little while
    // TODO: Turn " off" the cathode (-) wire
  }
  // TODO: Turn "off" the anode (+) wire
}
```

3. Run the code and make sure that all of the LEDs do in fact turn on. It will be helpful for the next part if you set up `ANODE_PINS[]` and `CATHODE_PINS[]` so that the lights go in some logical order. If you can't wait for all 64 LEDs, make the delay very short so that all the lights appear to be on at once, and make sure they are all lit up. If only one or two LEDs don't light up, check to make sure you did not put them in backward.

L1: Attach your completed code to cycle through each LED one by one.

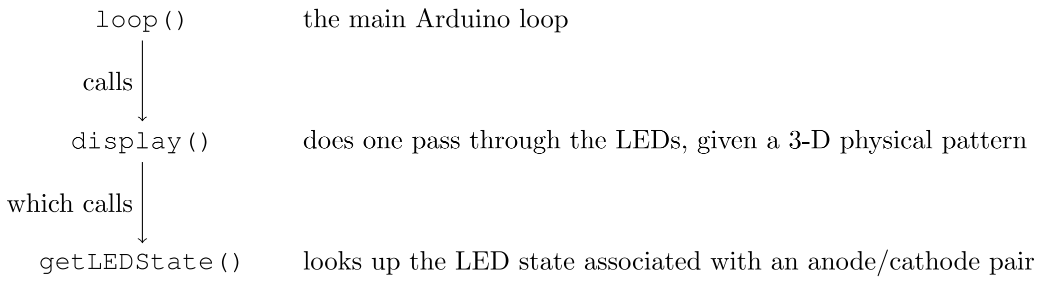
4.2 Decomposing a complex task into simpler ones

We can now turn on each LED individually, but we'd really like to display whatever pattern we like on them. Moreover, we'd like to specify these patterns using a natural representation of what we want to *see*—that is, their physical layout—rather than the obscure anode/cathode pairs.

What might this "natural representation" look like? The most obvious idea is a $4 \times 4 \times 4$ multidimensional array, representing the cube. Each element can be either `1` (if the LED at that position is on) or `0` (off).

```
byte ledPattern[4][4][4];
```

Getting from `ledPattern[4][4][4]` to time-multiplexing an anode/cathode grid is a little complex. To make this easier, we *decompose* the task into several functions, each with a simpler job.



`loop()` the main Arduino loop

calls

`display()` does one pass through the LEDs, given a 3-D physical pattern

which calls

`getLEDState()` looks up the LED state associated with an anode/cathode pair

Our `display()` routine will do one time-division multiplexing cycle through the LEDs. Time-division multiplexing runs most naturally in terms of the anode/cathode grid, so `display()` will "think" in those terms. It *delegates* the task of interpreting the *physical* representation of the LEDs to `getLEDState()`, which runs the *mapping* from anode–cathode coordinates (a, c) (`aNum`, `cNum` in code) to physical coordinates (x, y, z).

[1] You can download the file `everylight_cube.ino` or `everylight_plane.ino` from the class webpage.

4.3 Time-division multiplexing in `display()`

Recall that our time-division strategy goes like this:

- Go through the anodes (+, "rows") one by one. For each anode:
    1. Set each cathode (−) wire ("column") to the appropriate level (`HIGH` or `LOW`)
    2. Activate the anode (+) wire to turn the row on
    3. Wait a (very) short time
    4. Deactivate the anode (+) wire to turn the row off

We discussed this in lecture, and it's very important that you understand this routine fully, so please ask in office hours and discuss with your classmates if you're not confident about how this works.

Below is the outline of the code; `TODO` lines are for you to fill in. You can download a starter file `display_cube.ino` or `display_plane.ino` from the class website. The starter file also has hints in it. ☺

```
void display(byte pattern[4][4][4])
{
  for (byte aNum = 0; aNum < 8; aNum++) { // iterate through anode wires

    // Set up all the cathode (-) wires first
    for (byte cNum = 0; cNum < 8; cNum++) { // iterate through cathode wires
      byte value = getLEDState(pattern, aNum, cNum); // look up the value
      // TODO: Activate the cathode wire if value is > 0, otherwise deactivate it
    }

    // TODO: Activate the anode wire (without condition)
    // TODO: Wait a short time
    // TODO: Now done with this row, so deactivate the anode wire
  }
}
```

L2: Download the starter file `display_cube.ino` or `display_plane.ino` from the class website, and fill in the `display()` function.

4.4 Coordinate conversion in `getLEDState()`

If you did an LED plane, this subsection doesn't apply to you.

The `display()` function you just wrote delegated interpretation of `pattern` to another function, `getLEDState()`. The role of this function is, *given* a 3-D array `pattern[][][]` and anode/cathode wire numbers (a, c) (`aNum`, `cNum` in code), to find the corresponding physical location (x, y, z) and to return `pattern[`x`][`y`][`z`]`.

Your (nontrivial, but fun) job, then, is to find the conversion mapping (a, c) to (x, y, z). As a reminder:

- a, c each take values between 0 and 7 (inclusive), corresponding to the pins you listed in `ANODE_PINS[]` and `CATHODE_PINS[]`, respectively.
- x, y, z each take values between 0 and 3 (inclusive), corresponding to locations in the `pattern[][][]` array, which in turn corresponds to your physical cube.
- You can (and should) reorder the entries in `ANODE_PINS[]` and `CATHODE_PINS[]` to make life easier for yourself—that is, to put them in some sort of logical *physical* ordering.
- Your function can use logical (`&&`, `||`, `!`) and bitwise (`&`, `|`) operators as well as mathematical ones (`*`, `/`, `%`, `+`). You can also use `if` statements or the ternary operator (`?:`), but you shouldn't need more than one or two. (Certainly, please don't write 64 `if-else if` statements!)

Below is an empty function definition:

```
inline byte getLEDState(byte pattern[4][4][4], byte aNum, byte cNum)
{
  // TODO: fill this in to return the value in the pattern array corresponding
  // to the anode (+) wire and cathode (-) wire number (aNum and cNum) provided.
  return 0;
}
```

L3: Fill in the `getLEDState()` function.

4.5 Putting it together

In the starter code, we've written a `loop()` function that reads values (x, y, z) from the Serial Monitor (or (x, y) for the plane), and toggles (flips) the state of the light at that position. So, *e.g.*, if you type in `0 0 0` then hit Enter, an LED in your $(0, 0, 0)$ corner should turn on if it was previously off, and off if it was previously on. You'll need to set the Serial Monitor to "Newline" and baud rate 115200. You shouldn't need to edit this `loop()` code, at least not this week.

Once you've written `display()` and `getLEDState()`, run the code and see if it works. If it doesn't, don't panic—for most people this won't work the first time. Start debugging:

- Try different combinations (x, y, z) and try to see patterns in what's wrong.
- If exactly one LED is toggling but it's the wrong one, check what your mapping function does for those coordinates.
- You could slow down the `display()` function by increasing the `delay()` length in it, to help gain visibility into what the cube is doing "in slow motion".

L4: Please submit your completed display code, including the `display()` and `getLEDState()` functions.

5 Analysis

The display function that you created in the lab has only two states for each LED: on or off. It would be nice if we could store brightness values in the LED pattern, and have the code create lights of different intensity. Sixteen brightness levels would probably be good enough for most displays.

We could change the brightness by varying the current, but it's difficult to change the current flowing through the diodes. There's only a small range of current for which the LEDs will turn on without blowing out. We'll have to vary something else to set the brightness.

A1: Without changing any hardware, what property could you change to set the brightness of each LED? (Think about what determines the relative LED brightness.)

A2: Write pseudocode for the `display()` function, showing how you would modify it for variable brightness. You should start with the code you've already written. Assume that `getLEDState()` returns a number between 0 (off) and 15 (fully on) representing the brightness for the LED specified. *There are a number of easy of mistakes to make on this problem. If you want to be sure your method works, you should try it on your cube!*

6 Reflection

Individually, answer the questions below. Two to four sentences for each question is sufficient. Answers that demonstrate little thought or effort will not receive credit!

R1: What was the most valuable thing you learned, and why?

> **R2:** What skills or concepts are you still struggling with? What will you do to learn or practice these concepts?

> **R3:** If this lab took longer than the regular 3-hour allotment: what part of the lab took you the most time? What could you do in the future to improve this?

7 Build Quality rubric

Your build quality grade in this project is based on both the quality of your breadboarding and the clarity of your code.

Plus

- Connections to the cube have neat, compact solder joints, with sensibly-chosen connection points
- Breadboard layout is clean and organized, taking largely no effort to follow
- Wires are color coded and easy to trace
- Wire lengths are about right
- Software is well-commented and easy to follow, with good use of constants and data types
- Mapping function is straightforward

Check

- Breadboard layout is organized, but could be improved by rearranging some components
- Wires can be traced without too much difficulty, but lacked some planning
- Code is properly indented and variable names make sense, but would benefit from more comments

Minus

- Clear lack of breadboard and/or software planning
- Some LEDs do not work
- Breadboard layout doesn't follow a consistent pattern
- Wires can be difficult to trace
- Breadboard circuitry prone to short-circuiting
- Software isn't commented, or is difficult to follow
- Mapping function is unnecessarily complex or incorrectly maps some LEDs