Button Pad Hookup Guide

Introduction

This tutorial introduces matrix-scanning tecnniques, using the SparkFun 4x4 Button Pad to build an illuminated keypad.

4x4 Button Pad with Arduino Mega 2560

More importantly, we'll introduce the concepts underlying the design and implementation of matrix scanning, so the reader can adapt and extend the techniques for their own projects.

Covered in this Tutorial

This tutorial is structured as a series of progressive exercises. First, we'll explore the underlying concepts and underpinnings of the design. Then we'll apply what we've learned, assembling the PCB, and working through several applications, starting simply, then adding features and complexity.

1. We'll start by assembling the the keypad.
2. Next, we'll introduce matrix scanning by getting a single color of LED to light.
3. From there, we'll add the buttons as an input device.
4. Finally, we'll enable to other colors of the LEDs in the matrix.

Along the way, we'll explore some of the design decisions that contribute to the keypad, and look at some of the coding techniques that are used to control the hardware.

Suggested Reading

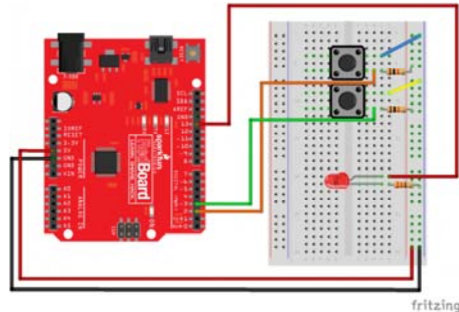
- We're going to be doing some low-level programming in this tutorial. This assumes that you're familiar with Binary numbers and converting them to and from hexadecimal and decimal representations.
- In particular, we're going to be using bitwise operators to perform the matrix scan.

- You'll need to be comfortable with Through-hole soldering. Assembling the button pad involves some more sophisticated soldering techniques than most of our other soldering projects.
- The matrix scanning will take advantage of internal Pull-up Resistors
- The matrix scanning also uses diodes to isolate the switches from each other.

Background

Even if you're just getting into the world of microcontrollers and embedded programming, you've probably hooked some buttons and LEDs up to your system, and written programs that let you turn the LEDs on and off using the buttons. It's fun to make an LED light up, and even more fun to have a switch that turns it on and off.

Exercises #3, #4, and #5 in the SIK Experiment Guide are versions of exactly that. In those experiments, a variety of LEDs and buttons are attached to a RedBoard, usually with one pin attached to a single button or LED.



If a couple switches and LEDs are fun, then heaps of LEDs and buttons are even better, right?

It naturally leads to questions about the limits of button and LED attachment. If a RedBoard allows 20 pins for digital I/O (digital pins 0-13, and repurposing A0 through A5 as digital), that allows a mix of buttons and LEDs totaling 20 units to be connected. This device-directly-to-pin connection strategy also assumes that we don't need to use pins for other purposes, such as serial communication.

That's a pretty serious constraint for many applications.

A Classic Example

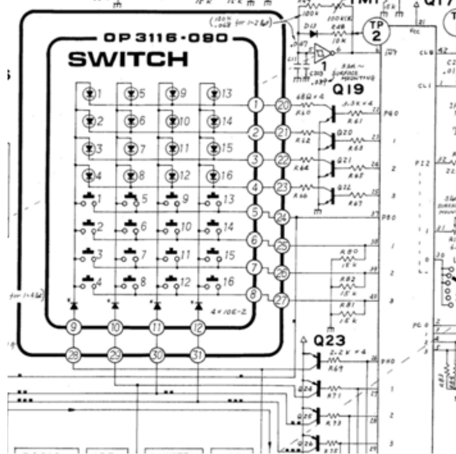
Let's look at a classic microcontroller system with a stylish "buttons & LEDs" user interface, the Roland TR-808 drum machine.

Across the lower edge of the control panel are 16 pushbutton switches, each with a captive red LED. The machine uses the LEDs to indicate its current status, and pressing a button causes the associated LED to toggle on and off.

The 808 was introduced in 1980. The NEC µPD650C microprocessor inside was state of the art for its day, running at 500 KHz (a screaming *half megahertz!*). It had nine 4-bit wide ports for I/O, but a number of those ports were consumed by the external memory bus (accessing the four KB of external RAM), leaving 20 pins for I/O.

But, the designers of the TR-808 connected all 32 of those components using only *twelve* I/O pins.

How did they do it? Let's look at that section of the schematic.



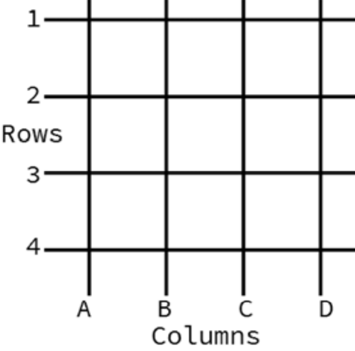
TR-808 keys and LEDs (Courtesy Archive.org)

Here you see the LEDs near the top, the buttons below that, and the microcontroller along the left. The LEDs and buttons are in a **scan matrix** arrangement.

An Introduction To Matrix Scanning.

Matrix scanning is a common technique used to expand the number of inputs or outputs beyond the number of available pins. Matrix scanning requires some cleverness on both the hardware and software sides of the system – there are some subtle factors at play.

To create a scan matrix, instead of using *n* pins directly as input or outputs, we allocate them as the axes of a Cartesian coordinate system – think of them like lines of latitude and longitude. Matrices are usually described using the terms **row** for an X-axis group, and **column** for a Y-axis group. Any intersection in the matrix can be described using its *X,Y* or *row, column* coordinates.



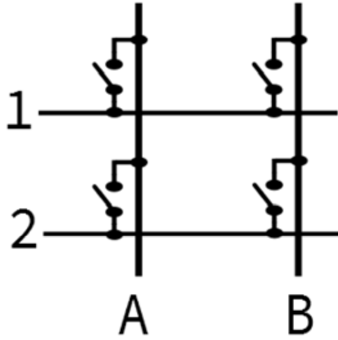
If you're having trouble keeping them straight, remember that Doric and Ionic columns are vertical structures in classical architecture.

Also keep in mind that the physical layout of a matrix circuit may not correspond to the conceptual axes used in the design. We could lay LEDs out in a circle, but still use X,Y based scanning to address each of them.

The scan matrix circuit places electronic components at the row, column intersections. By selecting a single row and column at a time, each component can be addressed uniquely.

A Simple Example

Let's explore a simple example. Using four pins, we'll use two of them as row selectors, and the other two as column selectors. At each intersection in the matrix, we'll install a pushbutton switch that shorts the row to the column.



- The row pins are configured as inputs, with pull-up resistors. We'll refer to them as 1 and 2.
- The column pins are configured as outputs. We'll call them A and B.
- At the intersection of each pair of pins, we'll place a momentary-contact switch that bridges the row to the column when it is pressed.

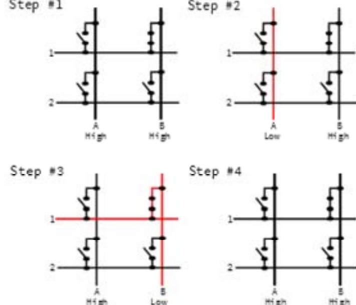
To use this matrix, we drive a single output at a time to select a column. While it's driven, we read the inputs. If we see the drive signal coming back on the input pins, we know that the switch at that X,Y position is closed.

Practical scan matrices often invert the voltages on the pins for a couple of clever reasons. Using inverted logic is called an **active low** system.

Many microcontrollers have input pins that can be configured with an internal pull-up resistor. If nothing is driving a pulled-up pin, it will read as a logic high. It's uncommon for port pins to have an internal pull-down.

To take advantage of those internal pull-up resistors, scan matrices often use inverted logic. The selected row is set to a low level, and the others are set high. Reading the columns, we look for a logic low to indicate that a button is closed, allowing the selection signal through. This is also why we'll use the relative terms "selected" or "driven," rather than the absolute terms "high" and "low."

The scan consists of walking a logic low around the output pins, and looking for that low on the input pins.

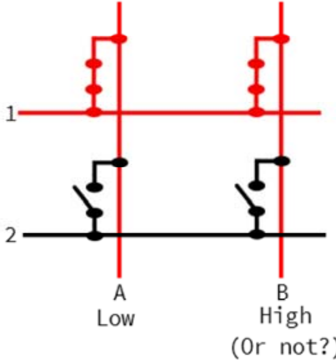


Above, the button at B1 is held. As the scan progresses, it does the following:

1. Nothing is selected. Outputs A and B are both high, and we don't worry about inputs 1 and 2.
2. Column A is selected with a logical low.
   - The system reads inputs 1 and 2. Both are open, so the pull-up resistors cause the inputs to be pulled high. Since this is an active low matrix, the high inputs indicate nothing is pressed.
3. The system deselects column A by driving it high and selects column B is by driving it low.
   - The system reads inputs 1 and 2. Since switch B1 is held, the low level from the column selection shows up at input 1.
   - By pairing the column output (B) with the detected switch (1), the system knows that switch B1 is pressed.
4. Finally, everything is deselected by driving both outputs high.

The Problem With Simple

The process described above works well when a single button is held, but it can be problematic when more than one button is held at a time. Let's look at what happens when we press buttons A1 and B1 together.



When the scanning sets column output A low, button A1 puts that low voltage onto row 1. Because A2 is also held, the low selection voltage from A is put onto column B, even though B is not selected, and putting a high level on the output pin.

This is a problem!

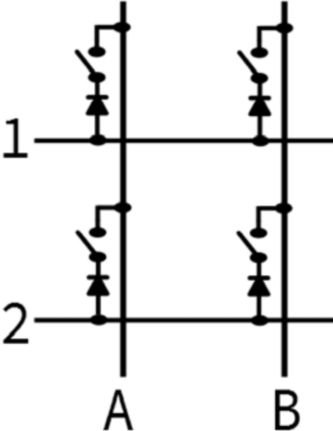
We have inadvertently connected two outputs together, and driven them to different logic levels. It's hard to predict what the voltage read by input 1 will be. Outputs A and B are contending, and the results depend on the specific architecture of the input and output pins.

- In a perfect universe, the two outputs would balance each other, and the row would sit halfway between the voltages (IE: at 2.5V on a 5V system).
- The universe isn't usually so perfect. It's more likely that one pin can drive harder than the other, and row 1 will be either high or low.
- Sometimes the universe is downright malevolent. By shorting two outputs together, it's possible that the circuitry inside one or both pins could be damaged, and possibly burned out.

Regardless of the actual result, connecting port pins together like this is considered poor practice. Ultimately, the circuit is too simple to be very useful.

A Step Above Simple

To fix the above situation, diodes are put in series with each button, as shown below. The diodes isolate the scan columns from each other, even when multiple buttons are held simultaneously.



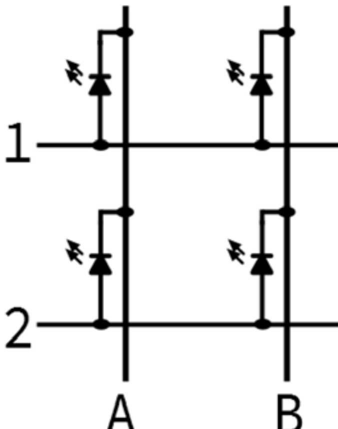
With the diodes in place, and we hold A1 and B1 at the same time, the diode on B1 won't conduct the row voltage back to column B. The short circuit no longer exists, and we can individually detect that A1 and B1 are pressed at the same time.

This circuit is commonly found in MIDI keyboards, where the microcontroller needs to be able to correctly detect that multiple keys are held simultaneously, in order to play chords.

This is also the sort of circuit used in high-quality PC keyboards. Being able to correctly detect arbitrary key combinations is known as N-Key Rollover. While it's not especially useful for everyday touch-typing, it can be very important for gaming, and alternate typing systems, such as braille, and chord typing.

Turning Things Around

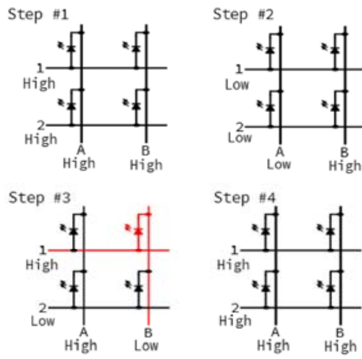
We've seen how to scan a 2x2 key matrix for registering input. Now let's turn things around, and use a 2x2 LED matrix for output. The matrix places LEDs at the junctions of the matrix.



The LED matrix takes advantage of what we discussed in the switch matrix analysis, above – diodes (or, more specifically, *Light Emitting Diodes*) only conduct in one direction, when the anode is at a higher voltage than the cathode. If both ends are at the same voltage (both high, or both low), or the diode is *reverse biased* (anode lower than cathode), then current doesn't flow, and the LED doesn't light up.

Differing from the button examples above, the driving pins are all configured as outputs. By carefully steering voltage onto the columns and rows, we control the voltage across the LEDs, allowing us to address each LED individually.

Let's look at how an active low scan would proceed if we want to light up B1, while leaving the other diodes dark.



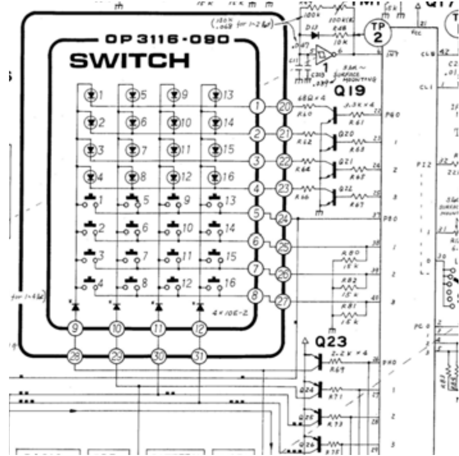
1. At the start of the scan, all pins are high.
2. The first column is selected by driving pin A low. We don't want the LEDs in column A to light, so we also drive pins 1 and 2 low. There is no voltage across the LEDs, which are therefore dark.
3. Column A is deselected be driving it high, and Column B is selected by driving it low. To get LED B1 to light, we also drive row 1 high. B1 is forward biased, and illuminates.
4. At the end of the scan, column B is deselected, and rows 1 and 2 are also driven high.

The LED is only illuminated while column B is being scanned. This takes advantage of how our eyes perceive the world around us – when an LED flashes on and off quickly enough, we see it as being solidly illuminated (although it may appear to be less bright than a continuously lit LED.). This is one example of the phenomenon of **persistence of vision**.

Growing larger

To address more buttons or LEDs, we simply add columns and rows. For instance, if we wanted 16 buttons , we could add two rows and two columns, making a 4x4 matrix.

A 4x4 matrix almost brings us full circle, back to the Roland TR-808, with its 16 buttons and 16 LEDs. By including both buttons and LEDs, the 808 hits one last design optimization: the scan outputs for the switches are shared with the LEDs.



If we look a little more closely at the schematic, we see (from top to bottom)

- Port G pins 0 through 3 are the LED row select inputs.
- Port B pins 0 through 3 are the switch row inputs.
- Port H pins 0 through 3 are the column outputs, common to both the switches and LEDs.

The output pins in ports G and H are buffered using discrete transistors, since the port pins of the µPD650C aren't capable of sourcing or sinking the current required to get the LEDs to light up.

Port B has discrete pull-down resistors, an indication that this is an active-high scan matrix. The µPD650C doesn't have internal pull-ups, so there's no reason to invert the scan just to take advantage of them.

You'll also notice that instead of a diode per button, the column select lines each have a single diode – this isolates the column selection outputs, but doesn't prevent misleading behavior if multiple switches are held simultaneously (although such presses can be detected and ignored in software).

A Few More Details

We mentioned above that the scan matrix requires cleverness in both the hardware and software. Our discussion above has covered some of the clever hardware design issues, such as installing diodes to prevent contention, using active-low logic to take advantage of the internal pull-ups on port pins, and sharing select lines between the button and LED matrices.

There are a handful of other issues that a scanned matrix need to take into account, often handled by the associated software.

- The rate at which the scan progresses is a key parameter. When the scan is too slow, the LEDs will visibly flicker, and the buttons will seem unresponsive.
- A faster scan smooths out the LEDs, but becomes susceptible to detecting small glitches in the button actuation. We'll explore software techniques for removing button glitches when we hook up the buttons.

With the basic concepts of matrix scanning covered, lets look at how they're implemented in the 4x4 RGB Button Pad!

The 4x4 RGB Button Pad

With the basics of matrix scanning established, let's look at how they work out in the SparkFun 4x4 RGB Button Pad.

Button Pad Schematic

From what we explored in the previous section, we recognize the columns and rows. Each junction in the matrix consists of a pushbutton switch and RGB LED. The LEDs are a single envelope containing separate red, green and blue LEDs, which share a common cathode. The LEDs are set up as three overlapping 4x4 matrices, one for each color.

Errata

The Button pad PCB design dates from around 2008, relatively early in SFE history. As such, it's not completely up to our modern standards for labeling marking and labeling. Let's clarify a few points.

- First, the RGB LED that the PCB was designed for is no longer available. We stock a nearly-equivalent one, but there's one catch – the Green and Blue pins have traded places.
- The PCB silkscreen has component designators for the LEDs printed on top, but they are marked as C1 to C4. The *C* (usually used for capacitors) should actually be a *D* (for diode).
- The row signals are combined into busses (the horizontal dark blue lines). This is a schematic shorthand that results in a simpler schematic, but can make it harder to follow individual wires around the page. In this case, four signals enter the bus on the left and are pulled out at each junction in the matrix.
- The connections at the bottom are marked "ground". These would more accurately be labeled "column".
  - For the buttons, they could be marked column select.
  - For the LEDs, they are the LED cathodes.
- Rows and columns are both labeled numerically (1 through 4), while it might be easier to understand if one axis were alphabetical, A through D.

Let's assemble our keypad, and start working with it.

Materials

The following exercises use the items on the following wishlist.

4x4 Button Pad Exercises SparkFun Wish List

Button Pad 4x4 - LED Compatible
COM-07835
This is a translucent silicon rubber button pad with 16 buttons originall…

Button Pad 4x4 - Breakout PCB
COM-08033
This a simple breakout board for the button pads. Each LED and butt…

Arduino Mega 2560 R3
DEV-11061
Arduino is an open-source physical computing platform based on a si…

Screw - Phillips Head (1/2", 4-40, 10 pack)
PRT-10452
Standard Philips-head 4-40 screws. They are 1/2" long and come in p…

Standoff - Nylon (4-40; 3/8"; 10 pack)
PRT-10927
These nylon standoffs are 3/8" long and tapped for a 4-40 screw. The…

Hook-Up Wire - Assortment (Solid Core, 22 AWG)
PRT-11367
An assortment of colored wires: you know it's a beautiful thing. Six diff…

(16) LED - RGB Clear Common Cathode
COM-00105
Ever hear of a thing called RGB? Red, Green, Blue? How about an R…

(2) Button Pad 2x2 Bottom Bezel
COM-08747
This is a black plastic bezel, custom made to mate with the silicon rub…

(2) Button Pad 2x2 Top Bezel
COM-08746
This is a black plastic bezel, custom made to mate with the silicon rub…

(16) Diode Small Signal - 1N4148
COM-08588
This is a very common signal diode - 1N4148. Use this for signals up…

Wishlist for 4x4 Button Pad Components

Tools

You'll also need to following tools.

- A soldering iron
- Leaded or lead free solder
- Flush cutters
- Solid core wire
- Wire strippers

Before we assemble the button pad, let's take a closer look at some of the parts.

The PCB

The PCB is designed with geometric features that interface with the button pad. The most obvious are the circular traces printed on top, with interlocking fingers that are bridged by the rings in the keypad, completing the button circuit.

Button Contacts

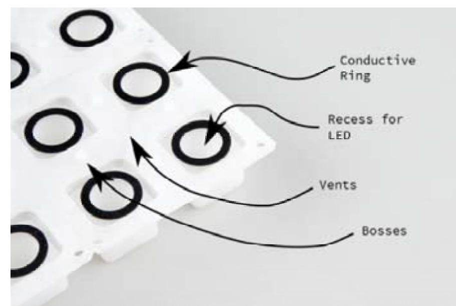
Inside each set of circles is the footprint for a common-cathode RGB LED.

The Button Pad

4x4 Rubber Keypad

The 4x4 Rubber Keypad is molded from Dow Corning Silastic[tm] Silicone rubber. It's similar to the keypads used by television remote controls.

If we flip it over, we'll see a some particular details molded into the rubber.



Back of Keypad

There are several features on the back of the button pad worth noting.

1. **Contact rings** – The most important feature is probably the conductive ring molded into the back of each button. It makes contact with fingers on the printed circuit board.
2. **Recesses** – Each button also has a hollow cavity in it, leaving room for a 3mm or 5mm PTH LED.
3. **Vents** – There are little troughs in the rubber that allow air to escape when the button is pressed. This make it easier to press the buttons, and prevents them from suctioning onto the PCB.
4. **Bosses** – These are little pegs that interface with holes in the PCB, to keep the keypad from sliding around.

The Bezels

When the keypad is placed on the PCB, it is held in place by a set of brackets known as the "bezel."

Sandwich

There are actually four bezels in the wish list, two marked “bottom,” and two marked “top.” These designators indicate the positions the the bezels get installed in. The bottom bezels have tabs that stick out, and the top bezels have recesses those tabs fit into.

Bottom and Top Bezels

Assembly

As we saw in the last section, the rubber button pad needs to sit on the PCB to work properly. If anything sticks up too far, the pad won’t sit properly, and the buttons will be hard to press.

To keep the components appropriately short, we’ll take a little extra care as we install them.

Diodes

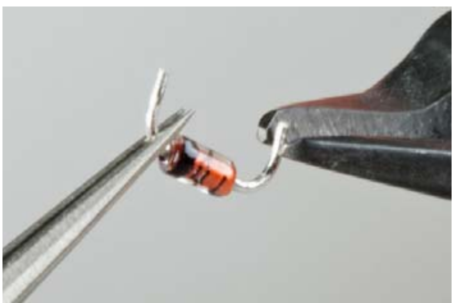
The 1n4148 diodes are installed on the back of the PCB, and would usually be soldered on the back of the board. However, since the resulting solder fillets would stick up too far, we’ll actually trim them short, and then solder them from the back side of the PCB. It’s a bit more work, but the end result is worth the effort.

There are 16 diodes. The diode locations on the PCB are simply marked with a rectangular outline, with a extra stripe at the cathode end.

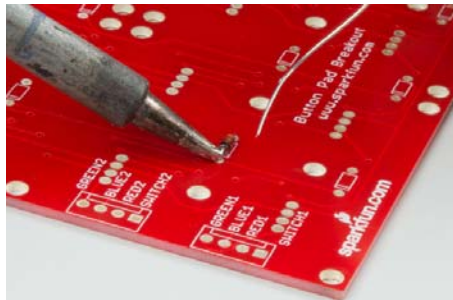
To install the diodes, first, bend the leads of the diode to fit the holes in the PCB.

Push the diode down until the body touches the PCB, but don't solder it in. Instead, flip the board over and trim the leads flush.

Then remove the trimmed diode, and snip a tiny bit more off the leads. We want to have enough lead left to fit into the hole on the PCB, but not so much that it protrudes.



Finally, put the diode back in the footprint, taking care to keep the cathode stripe pointed the correct direction, matching the silkscreen. Solder it in by tacking the remaining lead, adjacent to the diode body.



Before moving on, double-check that solder hasn't flowed through the hole, making a bump on the top of the PCB. If it has, you can cut it away with your flush cutters, or remove the bump with a bit of solder wick.

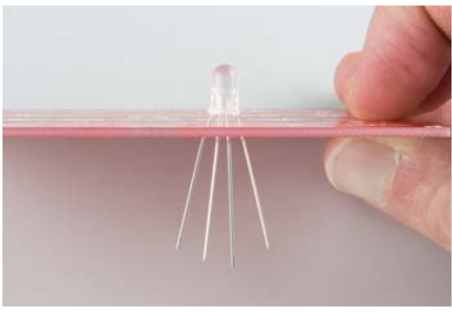
LEDs

Each RGB LED has four leads. The anode of each color has its own pin, and they share a common cathode. The cathode is the longest pin of the bunch.

The LEDs are polarized. The body has a small flat edge at the base, that corresponds to the flat on the PCB silkscreen

To put the LED on the PCB, insert the diode into the footprint. The longest lead is the second hole in from the flat, the other leads are each a bit shorter. With practice, you can insert the LED by “walking” it from side to side, and each lead will fall into place.

We need to cinch the LED up close to the PCB, to ensure that it will fit inside the recess in the button. If you’re unsure about how tall it can be, you can test the fit with the keypad over the LED.



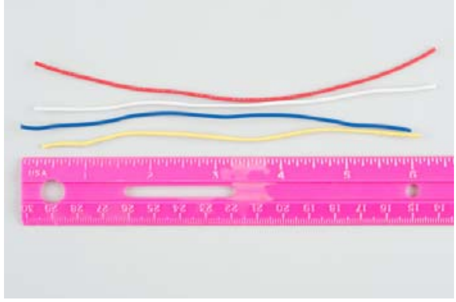
Before soldering, doublecheck the alignment of the flat side of the LED. When you’re sure it’s the right way around, solder it down, and snip the leads.

Wires

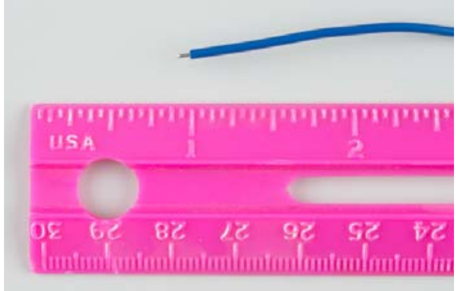
For the following exercises, we’ll simply be using solid-core wire to connect the keypad to the Arduino Mega. If you’ve got a different application in mind, removable connectors or snappable headers might be more suitable. However you chose to connect the button pad, the goal should be to keep the connectors from protruding through the PCB, interfering with the rubber buttons

A fully populated 4x4 RGB button pad requires 24 wires to connect it to the microcontroller. We’ll install the wires incrementally in the following sections, but each one will be installed using the same basic method. Like the 1N4148 diodes, we’ll be soldering them from the back of the PCB, so they don’t protrude and create bumps under the keypad.

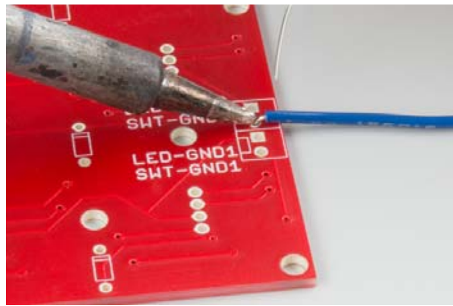
The exercises in this guide were done on a workbench, with the button pad sitting adjacent to the Mega. For that application, each wire was about 6" (15 cm) long.



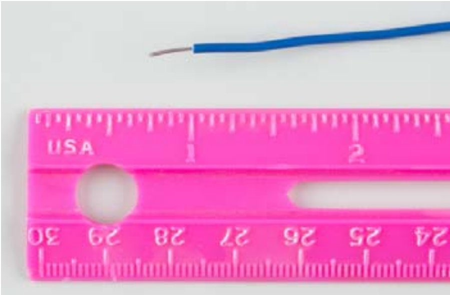
The PCB end of the wire was stripped to reveal about 0.1" of copper.



That end was bent into a gentle curve, and soldered to the PCB. Like the diodes, it was soldered from the back, and anything protruding on the button-side on the board was trimmed flush.



The other end was stripped about 0.25", so it could be inserted into the headers on the Mega.



Final Assembly

The last assembly step is to place the buttonpad over the PCB…

…and secure it with the plastic bezels. Start with the “bottom” bezels (the ones with the protruding tabs), on opposite corners.

Then add the “top” bezels on the remaining corners.

Finally, insert the Philips screws, and thread them into the standoffs.

Take care to not overtighten the screws. We want to hold the button pad in place, but not tighten it so much that it gets damaged.

Exercise #1: Monochrome LEDs

Our first exercise is to get the red LED scan working.

Wiring

To start out, we need to wire up the parts of the matrix that hit the red LEDs – the LED columns and the red LED row pins.

To make it easier to think about, set the button pad so that the button side is facing you, and connectors are on the left and bottom edges.

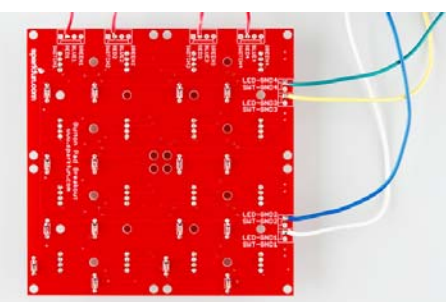
This keeps the rows and columns in a fairly sensible order and orientation. The columns are the connections across the bottom, and the rows are the connections up the left edge.

For 16 LEDs in a 4x4 matrix, that makes 4 rows + 4 columns = 8 connections total. These 8 connections are detailed in the following table.

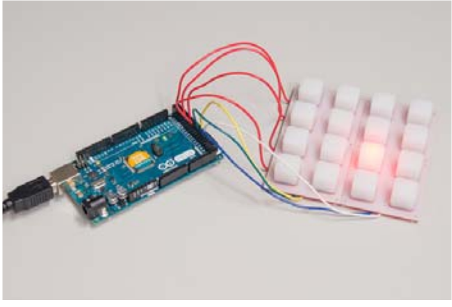
Function	Color	Button Pad Connection	Mega Connection
Red LED Row 1	Red	Red1	22
Red LED Row 2	Red	Red2	30
Red LED Row 3	Red	Red3	33
Red LED Row 4	Red	Red4	36
LED Column A	Green	LED-GND-4	42
LED Column B	Yellow	LED-GND-3	43
LED Column C	Blue	LED-GND-2	44
LED Column D	White	LED-GND-1	45

You'll notice that the column selects are on adjacent pins, but the red LED rows are spaced apart – this leaves the interceding pins for the corresponding green and blue connections.

With the wires attached to the PCB, it will look like this:



Each wire was prepared and soldered as described in the assembly section, and the end was stuck into the header on the Mega.



4x4 Button Pad with Arduino Mega

Code

The following sketch illuminates a single red LED at a time. The illuminated

LED walks around the matrix.

```
/*************************************************************
****************
red-only.ino
Byron Jacquot @ SparkFun Electronics
1/6/2015

Example to drive the red LEDs in the RGB button pad.

Exercise 1 in a series of 3.
https://learn.sparkfun.com/tutorials/button-pad-hookup-guide/e
xercise-1-monochrome-leds

Development environment specifics:
Developed in Arduino 1.6.5
For an Arduino Mega 2560

This code is released under the [MIT License](http://opensourc
e.org/licenses/MIT).

Distributed as-is; no warranty is given.
*************************************************************
****************/
//config variables
#define NUM_LED_COLUMNS (4)
#define NUM_LED_ROWS (4)
#define NUM_COLORS (1)

// Global variables
static bool LED_buffer[NUM_LED_COLUMNS][NUM_LED_ROWS];
static int32_t next_advance;
static uint8_t led_index;

static const uint8_t ledcolumnpins[NUM_LED_COLUMNS]   = {42,4
3,44,45};
static const uint8_t colorpins[NUM_LED_ROWS] = {22,30,33,36};

static void setuppins()
{
  uint8_t i;

  // initialize all of the output pins

  // LED column lines
  for(i = 0; i < NUM_LED_COLUMNS; i++)
  {
    pinMode(ledcolumnpins[i], OUTPUT);

    // with nothing selected by default
    digitalWrite(ledcolumnpins[i], HIGH);
  }

  // LED row lines
  for(i = 0; i < NUM_LED_ROWS; i++)
  {
    pinMode(colorpins[i], OUTPUT);

    // with nothing driven by default
    digitalWrite(colorpins[i], LOW);
  }

}

static void scan()
```

```
{
  static uint8_t current = 0;
  uint8_t val;
  uint8_t i, j;

  // Select a column
  digitalWrite(ledcolumnpins[current], LOW);

  // write the row pins
  for(i = 0; i < NUM_LED_ROWS; i++)
  {
    if(LED_buffer[current][i])
    {
      digitalWrite(colorpins[i], HIGH);
    }
  }

  delay(1);

  digitalWrite(ledcolumnpins[current], HIGH);

  for(i = 0; i < NUM_LED_ROWS; i++)
  {
    digitalWrite(colorpins[i], LOW);
  }

  // Move on to the next column
  current++;
  if (current >= NUM_LED_COLUMNS)
  {
    current = 0;
  }

}

void setup()
{
  // put your setup code here, to run once:
  Serial.begin(115200);
  Serial.print("Starting Setup...");

  // setup hardware
  setuppins();

  // init global variables
  next_advance = millis() + 1000;
  led_index = 0;

  // Initialize the LED display array
  for(uint8_t i = 0; i < NUM_LED_COLUMNS; i++)
  {
    for(uint8_t j = 0; j < NUM_LED_ROWS; j++)
    {
      LED_buffer[i][j] = false;
    }
  }
  // Set the first LED in the buffer on
  LED_buffer[0][0] = true;

  Serial.println("Setup Complete.");

}

void loop()
{
```

```
  // put your main code here, to run repeatedly:

  scan();

  if(millis() >= next_advance)
  {
    next_advance = millis()+1000;

    LED_buffer[led_index/NUM_LED_COLUMNS][led_index%NUM_LED_CO
LUMNS] = false;
    led_index++;
    led_index %= (NUM_LED_COLUMNS * NUM_LED_ROWS);
    LED_buffer[led_index/NUM_LED_COLUMNS][led_index%NUM_LED_CO
LUMNS] = true;
  }
}
```

The code is an implementation if what we described in the background section. A column is selected, and the corresponding row pins are driven to get the LEDs to light.

Let's look at a few of the finer points in the code.

- In an effort to make the code more portable and configurable, the basic parameters are defined in a set of definitions at the top of the sketch.
- The image displayed by the LEDs is declared as a two-dimensional array of `bool` . The array dimensions match the rows and columns.
- The pins themselves are defined as constant one-dimensional arrays. This makes it easy to:
  - Reassign the pins, by simply editing the array initialization values.
  - Walk from pin to pin, by incrementally indexing the array.
- The matrix scan is performed by the `scan()` function.
  - On each invocation, the scan selects the next column, then writes the corresponding row pins with the values from the LED array.
  - It pauses for a millisecond, which allows the LED to glow for a moment.
    - *If you're curious, remove this `delay(1)` , and you'll find that the LEDs get significantly dimmer*.
  - Then it deselects the column and stops driving the rows.
- `loop()` calls the scan function every time it is invoked. Because it contains the `delay(1)` mentioned above, the scan updates at most once every millisecond.
- Every second (or 1000 `millis()`), the loop walks the illuminated LED to the next position.

The next exercise is to add button inputs to this, allowing you to turn each LED on or off by pressing the corresponding button.

Exercise #2: Monochrome plus Buttons

Our Second exercise is to add the buttons to the scan matrix. We'll use them as inputs, to toggle the corresponding LEDs on and off.

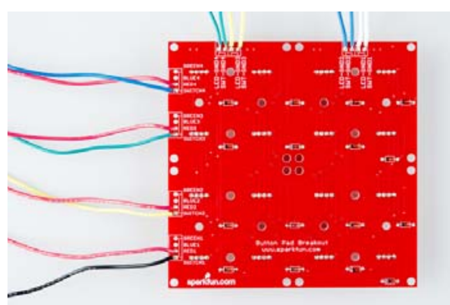
We're going to assume that you're following this guide in order, and have just completed exercise #1. This exercise will describe the additions needed to add the button matrix, building incrementally on top of exercise #1.

Wiring

We're going to add another 8 wires to interface the button matrix. This breaks down as four button columns and four button rows, and described in the following table.

Function	Wire Color	Button Pad Connection	Mega Connection
Button Row 1	Black	Switch1	46
Button Row 2	Yellow	Switch2	47
Button Row 3	Green	Switch3	48
Button Row 4	Blue	Switch4	49
Button Column A	Green	SWT-GND-4	50
Button Column B	Yellow	SWT-GND-3	51
Button Column C	Blue	SWT-GND-2	52
Button Column D	White	SWT-GND-1	53

Again, the wires are prepared and soldered as described in the assembly section, and the other end is stuck into the header on the Mega. With these wires, we're adding column and row connections for the buttons. When they're in place, the connections to the PCB look like this.



Red LED and Button Connections

Code

The following sketch adds button support to the previous sketch.

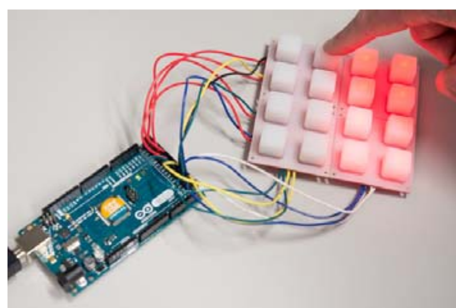
```
/*************************************************************
****************
red-plus-buttons.ino
Byron Jacquot @ SparkFun Electronics
1/6/2015

Example to drive the red LEDs and scan the buttons of the RGB
button pad.

Exercise 2 in a series of 3.
https://learn.sparkfun.com/tutorials/button-pad-hookup-guide/e
xercise-2-monochrome-plus-buttons

Development environment specifics:
Developed in Arduino 1.6.5
For an Arduino Mega 2560

This code is released under the [MIT License](http://opensourc
e.org/licenses/MIT).

Distributed as-is; no warranty is given.
**************************************************************
****************/
//config variables
#define NUM_LED_COLUMNS (4)
#define NUM_LED_ROWS (4)
#define NUM_BTN_COLUMNS (4)
#define NUM_BTN_ROWS (4)
#define NUM_COLORS (1)

#define MAX_DEBOUNCE (3)

// Global variables
static bool LED_buffer[NUM_LED_COLUMNS][NUM_LED_ROWS];

static const uint8_t btncolumnpins[NUM_BTN_COLUMNS] = {50, 5
1, 52, 53};
static const uint8_t btnrowpins[NUM_BTN_ROWS]       = {46, 4
7, 48, 49};
static const uint8_t ledcolumnpins[NUM_LED_COLUMNS] = {42, 4
3, 44, 45};
static const uint8_t colorpins[NUM_LED_ROWS]        = {22, 3
0, 33, 36};

static int8_t debounce_count[NUM_BTN_COLUMNS][NUM_BTN_ROWS];

static void setuppins()
{
  uint8_t i;

  // initialize
  // select lines
  // LED columns
  for (i = 0; i < NUM_LED_COLUMNS; i++)
  {
    pinMode(ledcolumnpins[i], OUTPUT);

    // with nothing selected by default
    digitalWrite(ledcolumnpins[i], HIGH);
  }

  // button columns
  for (i = 0; i < NUM_BTN_COLUMNS; i++)
  {
```

```
    pinMode(btncolumnpins[i], OUTPUT);

    // with nothing selected by default
    digitalWrite(btncolumnpins[i], HIGH);
  }

  // button row input lines
  for (i = 0; i < NUM_BTN_ROWS; i++)
  {
    pinMode(btnrowpins[i], INPUT_PULLUP);
  }

  // LED drive lines
  for (i = 0; i < NUM_LED_ROWS; i++)
  {
    pinMode(colorpins[i], OUTPUT);
    digitalWrite(colorpins[i], LOW);
  }

  // Initialize the debounce counter array
  for (uint8_t i = 0; i < NUM_BTN_COLUMNS; i++)
  {
    for (uint8_t j = 0; j < NUM_BTN_ROWS; j++)
    {
      debounce_count[i][j] = 0;
    }
  }
}

static void scan()
{
  static uint8_t current = 0;
  uint8_t val;
  uint8_t i, j;

  // Select current columns
  digitalWrite(btncolumnpins[current], LOW);
  digitalWrite(ledcolumnpins[current], LOW);

  // output LED row values
  for (i = 0; i < NUM_LED_ROWS; i++)
  {
    if (LED_buffer[current][i])
    {
      digitalWrite(colorpins[i], HIGH);
    }
  }

  // pause a moment
  delay(1);

  // Read the button inputs
  for ( j = 0; j < NUM_BTN_ROWS; j++)
  {
    val = digitalRead(btnrowpins[j]);

    if (val == LOW)
    {
      // active low: val is low when btn is pressed
      if ( debounce_count[current][j] < MAX_DEBOUNCE)
      {
        debounce_count[current][j]++;
        if ( debounce_count[current][j] == MAX_DEBOUNCE )
        {
          Serial.print("Key Down ");
```

```
          Serial.println((current * NUM_BTN_ROWS) + j);

          // Do whatever you want to with the button press her
e:
          // toggle the current LED state
          LED_buffer[current][j] = !LED_buffer[current][j];
        }
      }
    }
    else
    {
      // otherwise, button is released
      if ( debounce_count[current][j] > 0)
      {
        debounce_count[current][j]--;
        if ( debounce_count[current][j] == 0 )
        {
          Serial.print("Key Up ");
          Serial.println((current * NUM_BTN_ROWS) + j);

          // If you want to do something when a key is release
d, do it here:

        }
      }
    }
  }// for j = 0 to 3;

  delay(1);

  digitalWrite(btncolumnpins[current], HIGH);
  digitalWrite(ledcolumnpins[current], HIGH);

  for (i = 0; i < NUM_LED_ROWS; i++)
  {
    digitalWrite(colorpins[i], LOW);
  }

  current++;
  if (current >= NUM_LED_COLUMNS)
  {
    current = 0;
  }

}

void setup()
{
  // put your setup code here, to run once:
  Serial.begin(115200);

  Serial.print("Starting Setup...");

  // setup hardware
  setuppins();

  // init global variables
  for (uint8_t i = 0; i < NUM_LED_COLUMNS; i++)
  {
    for (uint8_t j = 0; j < NUM_LED_ROWS; j++)
    {
      LED_buffer[i][j] = 0;
    }
  }
```

```
  Serial.println("Setup Complete.");

}

void loop() {
  // put your main code here, to run repeatedly:

  scan();

}
```

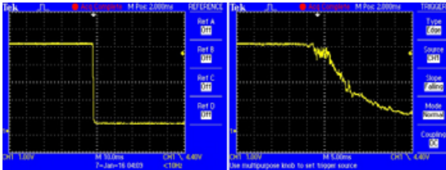
Once the code is loadeed, the keypad doesn't do anything on its own, but when you press a button, the corresponding LED toggles on and off.



Button matrix in command of LED matrix

The button scan is added in parallel to the LED scan. A column is selected, and a row is read. Some new definitions have been added to dimension the button matrix, plus some new data tables to define the pins for those functions.

The most complex part of this code is the button press detection, which involves a little extra work. When the buttons open or close, sometimes the change isn't especially clean – the contacts might chatter or bounce when they close. The waveforms below show a couple of different switch actuations, captures from this button pad.



Captured switch closures

The nice closure on the left was the result of pressing the button quickly and decisively, straight down. The closure on the right is the result of pushing the button slowly, at an oblique angle. The ring inside twisted down onto the fingers of the PCB contacts. In this instance it takes roughly 20 milliseconds for the button to close solidly.

The glitchiness can cause the scanning system to misinterpret the chatter as multiple keypresses, when in fact the button was only actuated once.

To prevent this, the software uses some logic to perform **debouncing**. The scan system has to find a switch closed for several scans in a row before it decides that the switch is actually closed. It uses the two-dimensional array `debounce_count` to keep track of the number of successive scans. When a switch is found closed, the counter is incremented. When the counter reaches the constant `MAX_DEBOUNCE`, it decides that the switch has been solidly and cleanly closed, and acts on the keypress.

When a key is released, the counter counts back down to zero. This example prints a message when it detects a release. Some applications don't take any action on key releases (like the keypad on an ATM), but it's necessary in others (such as MIDI keyboards).

For the rubber button pad, a `MAX_DEBOUNCE` of 2 or 3 seems to work fairly well. Other types of buttons might require different debounce values.

With the red LEDs and buttons working, the final step is to get the green and blue LEDs working.

Exercise #3: RGB LEDs and Buttons

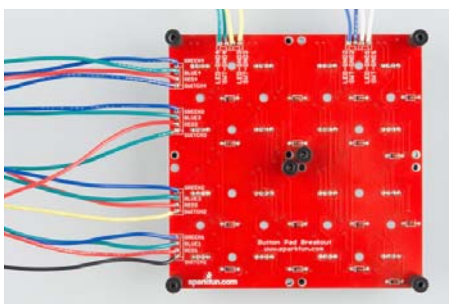
Wiring

Hang in there, only 8 more wires to go!

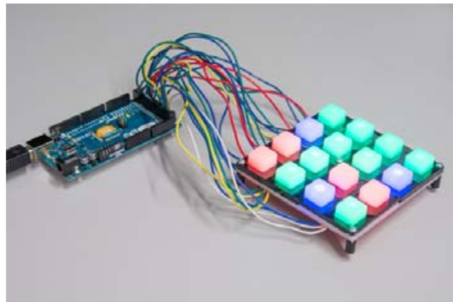
With the red LEDs and buttons working, let's finish up by adding the blue and green LEDs to the mix.

Function	Wire Color	Button Pad Connection	Mega Connection
Green Row 1	Green	Blue1*	24
Blue Row 1	Blue	Green1*	26
Green Row 2	Green	Blue2*	31
Blue Row 2	Blue	Green2*	32
Green Row 3	Green	Blue3*	34
Blue Row 3	Blue	Green3*	35
Green Row 4	Green	Blue4*	37
Blue Row 4	Blue	Green4*	38

** Keep in mind that the LEDs we've used have their green and blue legs transposed when referencing the PCB markings. Yes, we are intentionally swapping green and blue in the wiring!*



With all of these wires, it's starting to look like a bird's nest!



Code

The final sketch builds on the previous two.

```
/************************************************************
*****************
rgb-plus-buttons.ino
Byron Jacquot @ SparkFun Electronics
1/6/2015

Example to drive the RGB LEDs and scan the buttons of the RGB
button pad.

Exercise 3 in a series of 3.
https://learn.sparkfun.com/tutorials/button-pad-hookup-guide/e
xercise-3-rgb-leds-and-buttons

Development environment specifics:
Developed in Arduino 1.6.5
For an Arduino Mega 2560

This code is released under the [MIT License](http://opensourc
e.org/licenses/MIT).

Distributed as-is; no warranty is given.
*************************************************************
****************/
//config variables
#define NUM_LED_COLUMNS (4)
#define NUM_LED_ROWS (4)
#define NUM_BTN_COLUMNS (4)
#define NUM_BTN_ROWS (4)
#define NUM_COLORS (3)

#define MAX_DEBOUNCE (3)

// Global variables
static uint8_t LED_outputs[NUM_LED_COLUMNS][NUM_LED_ROWS];
static int32_t next_scan;

static const uint8_t btnselpins[4]   = {50,51,52,53};
static const uint8_t btnreadpins[4] = {46,47,48,49};
static const uint8_t ledselpins[4]   = {42,43,44,45};

// RGB pins for each of 4 rows
static const uint8_t colorpins[4][3] = {{22,24,26}, {30,31,3
2},{33,34,35},{36,37,38}};


static int8_t debounce_count[NUM_BTN_COLUMNS][NUM_BTN_ROWS];

static void setuppins()
{
    uint8_t i;

    // initialize
    // select lines
    for(i = 0; i < NUM_LED_COLUMNS; i++)
    {
        pinMode(ledselpins[i], OUTPUT);

        // with nothing selected by default
        digitalWrite(ledselpins[i], HIGH);
    }

    for(i = 0; i < NUM_BTN_COLUMNS; i++)
    {
```

```
        pinMode(btnselpins[i], OUTPUT);

        // with nothing selected by default
        digitalWrite(btnselpins[i], HIGH);
    }

    // key return lines
    for(i = 0; i < 4; i++)
    {
        pinMode(btnreadpins[i], INPUT_PULLUP);
    }

    // LED drive lines
    for(i = 0; i < NUM_LED_ROWS; i++)
    {
        for(uint8_t j = 0; j < NUM_COLORS; j++)
        {
            pinMode(colorpins[i][j], OUTPUT);
            digitalWrite(colorpins[i][j], LOW);
        }
    }

    for(uint8_t i = 0; i < NUM_BTN_COLUMNS; i++)
    {
        for(uint8_t j = 0; j < NUM_BTN_ROWS; j++)
        {
            debounce_count[i][j] = 0;
        }
    }
}

static void scan()
{
  static uint8_t current = 0;
  uint8_t val;
  uint8_t i, j;

    //run
    digitalWrite(btnselpins[current], LOW);
    digitalWrite(ledselpins[current], LOW);

    for(i = 0; i < NUM_LED_ROWS; i++)
    {
        uint8_t val = (LED_outputs[current][i] & 0x03);

        if(val)
        {
            digitalWrite(colorpins[i][val-1], HIGH);
        }
  }


  delay(1);

  for( j = 0; j < NUM_BTN_ROWS; j++)
  {
    val = digitalRead(btnreadpins[j]);

    if(val == LOW)
    {
      // active low: val is low when btn is pressed
      if( debounce_count[current][j] < MAX_DEBOUNCE)
      {
        debounce_count[current][j]++;
        if( debounce_count[current][j] == MAX_DEBOUNCE )
```

```
        {
          Serial.print("Key Down ");
          Serial.println((current * NUM_BTN_ROWS) + j);

          LED_outputs[current][j]++;
        }
      }
    }
    else
    {
      // otherwise, button is released
      if( debounce_count[current][j] > 0)
      {
        debounce_count[current][j]--;
        if( debounce_count[current][j] == 0 )
        {
          Serial.print("Key Up ");
          Serial.println((current * NUM_BTN_ROWS) + j);
        }
      }
    }
  }// for j = 0 to 3;

  delay(1);

  digitalWrite(btnselpins[current], HIGH);
  digitalWrite(ledselpins[current], HIGH);

  for(i = 0; i < NUM_LED_ROWS; i++)
  {
    for(j = 0; j < NUM_COLORS; j++)
    {
      digitalWrite(colorpins[i][j], LOW);
    }
  }

  current++;
  if (current >= NUM_BTN_COLUMNS)
  {
    current = 0;
  }
}

void setup()
{
  // put your setup code here, to run once:
  Serial.begin(115200);

  Serial.print("Starting Setup...");

  // setup hardware
  setuppins();

  // init global variables
  next_scan = millis() + 1;

  for(uint8_t i = 0; i < NUM_LED_ROWS; i++)
  {
    for(uint8_t j = 0; j < NUM_LED_COLUMNS; j++)
    {
      LED_outputs[i][j] = 0;
    }
  }

  Serial.println("Setup Complete.");
```

```
}

void loop() {
  // put your main code here, to run repeatedly:

  if(millis() >= next_scan)
  {
    next_scan = millis()+1;
    scan();
  }
}
```

Now, instead of toggling the red LEDs between on and off, pressing a button cycles between the different colors. The cycle is off, then red, then green, then blue, then off again.

If you've been following the code changes between each exercise, you can probably anticipate what has been added.

- The array that defines the color output pins has been expanded to be two-dimensional. The second dimension adds pins for the green and blue LEDs.
- When the button scan detects a button press, it increments the corresponding value in the `LED_outputs` array.
- When the LED scan sets the LED outputs, it uses some bitwise operations to convert the button counter into a color bit, and apply it to the LED row outputs.
- This version doesn't allow for color mixing – we'll explore why below.

The Limits of I/O Pins

If you've worked with modern microcontrollers, you're probably used to hanging LEDs off digital output pins, as we've been doing in these examples.

We're actually being a little sloppy by doing this - we're taking advantage of the fact that the Atmel ATMega2560 microcontroller has very robust digital inputs and outputs. They are specified to an absolute maximum of 40 mA, and they behave reasonably well when this limit is exceeded, limiting the current.

Similarly, the overall maximum rated current for the entire processor is 200 mA.

The reason that this example constrains each LED to only red, green or blue is to stay safely within these limits. If we were to light an entire column of LEDs (that's twelve LEDs total, each at roughly 20 mA), they'd add up to 240 mA!

Older microcontrollers allowed much less current per port pin, often in the range of 1 mA. If that range was exceeded, the pin could be fatally damaged. Designing with such controllers meant considering the current consumption per pin more carefully - refer back to the TR-808 schematic, and you'll notice that the pins that drive the LEDs use discrete transistors to increase the current drive capabilities.

For added safety, we should put a resistor inline with each LED scan line. The appropriate values can be calculated using the formula in our resistor applications tutorial.

Extra Credit

Through this guide, we've intentionally kept things simple, to demonstrate the matrix scanning concepts. We've only scratched the surface of the permutations of matrix scanning.

There are a lot of other variations on these ideas to consider, but we're leaving them as exercises for the reader.

More Colors

As we discussed in the previous section, we're only lighting the red, green or blue section of each LED individually, to keep current consumption low. When the author was developing exercise #4, the initial implementation allowed for the RGB LEDs to be on in any combination. The end result was that if too many LEDs were lit simultaneously, they got dim, and the processor heated up. Ultimately, the exercise was revised to only use single colors.

If we want to light more than one section at a time, we can mix them to create other colors, but to do that, we'll need to source and sink additional current. One solution is to add current limiting resistors in series with each LED Anode, and buffer the signal with high-current line drivers.

If you're really clever, you can use pulse-width modulation to vary the LED brightness, for an even wider range of colors.

Code Optimizations

The sketches in the exercises were written to be portable. They use the Arduino pin definitions, and `digitalRead` , and `digitalWrite` functions, which makes them easy to move to other systems. For instance, you could move this code to a Teensy 3.2 by adjusting the pin numbers in the row and column arrays. But this means handling those individual bits one at a time.

Most microcontrollers define the port pins as adjacent bits in byte or word wide registers. If you're working with those registers, you can arrange the matrix to gang the digital I/O using byte and word reads and writes. It's just not as easy to move to a different microcontroller!

As an even more optimized keypad scan can take advantage of external pin interrputs. The scanning is idle until a key is pressed. The key triggers an interrupt, which performs a single scan to determine which key is pressed. When all keys are released, the system idles again.

Bigger and Better

We can combine multiple keypads into a larger matrix by joining the adjacent rows and columns.

Offload The Keypad Scanning

There are peripheral chips that can do the button and LED scanning. The SX1509 I/O expander has a mode that allows it to scan an 8x8 key matrix, and interrupt the host when a key press is detected.

The 2x2 button Pad

The 4x4 button pad has a little brother, the 2X2 button pad, that pairs with a matching PCB. If you're building a 2x2 button pad, you'll need a single top bezel to hold it together.

The circuit of the 2x2 button pad is actually set up like a single row of the 4x4 pad, and could be added to the same scan matrix.

Resources and Going Further

Resources

- The complete Roland TR-808 service notes can be found at archive.org.

Going Further

- Our Serial LED Matrix uses matrix scanning techniques to address an 8x8 matrix of RGB LEDs.
- Charlieplexing is a more advanced scanning technique for driving many LEDs with very few pins.
- Matrix scanning is only one way to interface a bunch of switches. This article compares and contrasts a number of different methods.