Encode dozens of buttons with only four lines

Jim Brannan - January 31, 2018

In this **Design Idea**, an old technique of diode-based 16-to-4 encoding is combined with a new design, to scan a record number of buttons – 74 – with only four I/O pins!

The diode encoder: Background

First start with the ancient technique of using an array of diodes, **Figure 1**, to construct a 16-to-4 binary encoder. The cathodes of each diode connect to an output which has a pull-down resistor (or anodes & pull-ups). A single positive input, with the rest left open, passes through the diodes and raises the outputs in a binary pattern. In this example, zero is indistinguishable from no button pressed and is used to indicate that condition.

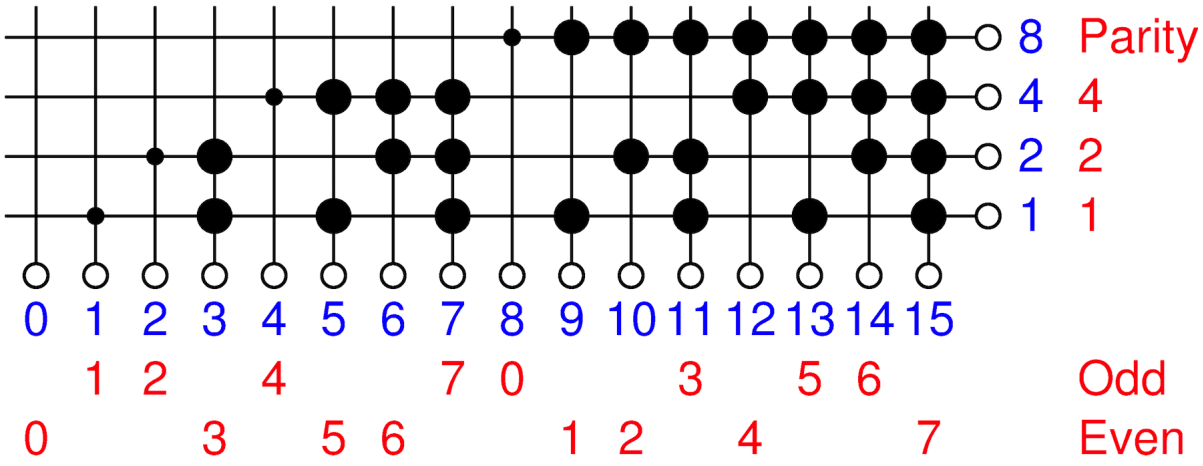


Figure 1 Classic diode 16-to-4 encoder. The small dots are direct connections, the large ones, diodes. *Parity* is explained later.

Because our inputs are either positive or open we can just make a direct connection on inputs 1, 2, 4, & 8, where only one diode would be indicated.

We can consider the output as a four-bit number, or use just half of the inputs, and define the output to be a three-bit number with parity, either even or odd. It's the same diode arrangement, just a different way of looking at the output.

More buttons through multiplexing

If we switch one input pin to output mode to drive a row, we can use the three remaining pins as inputs to detect up to seven buttons. We can choose any encoding that is conveniently detectable.

Now comes the clever bit. Since the data paths that output the row choice are the same as those that input the column result, we end up corrupting the column value. But in **Figure 2**, that's ok, because we imagine the value read as protected by a parity bit. Not only do we know that at most one bit has been damaged, we know which one. This is how we can share the column encoding

diodes with all the rows. Note that one location in each row represents "no button pressed". For even parity, they all fall in one column; for odd parity, in different columns.

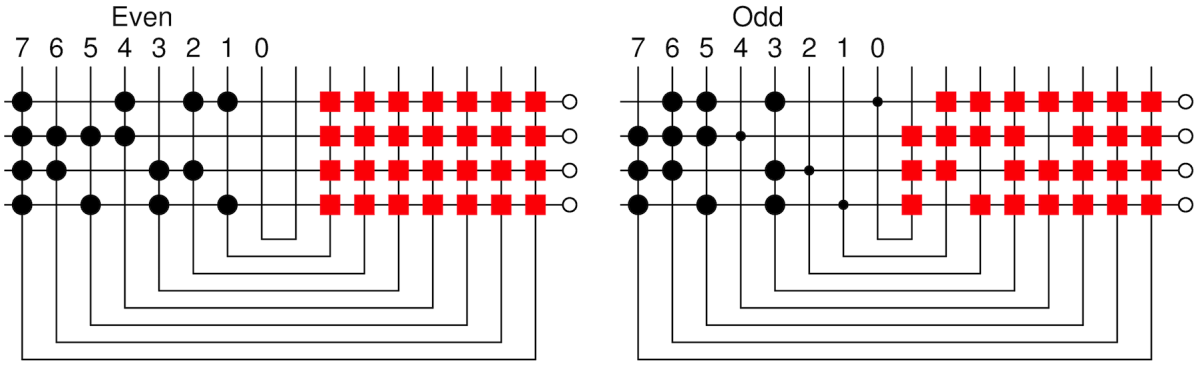


Figure 2 Multiplexing and parity encode more buttons. Squares are switches.

Figures 3 and **4** are complete arrays with both true and complementary diode encoders, for a total of 74 usable button locations in four banks.

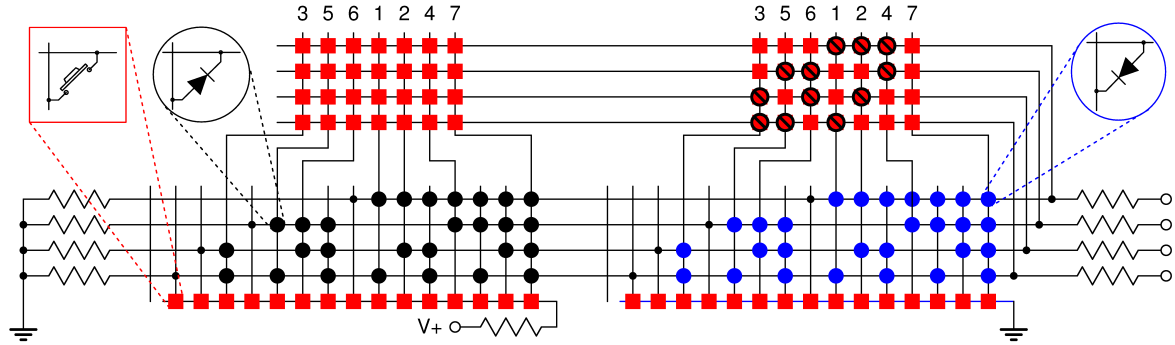


Figure 3 Pulling it all together: 74 button locations in four banks. The animation is explained later in the article.

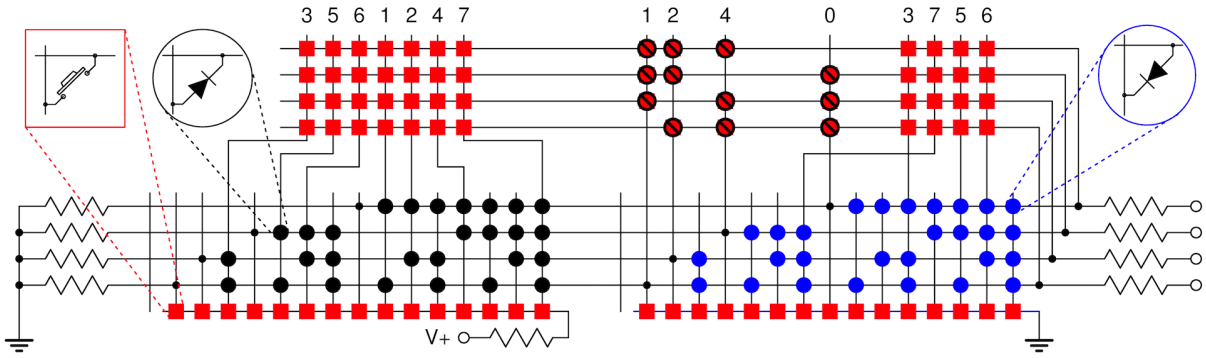


Figure 4 Using odd parity to rearrange the upper-right bank.

Reading the arrays

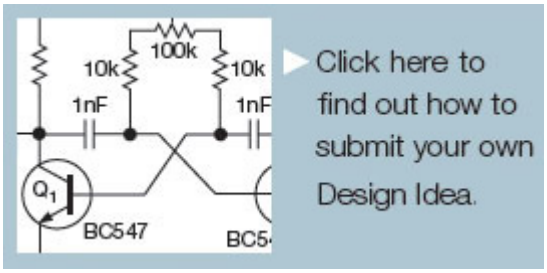
1. Lower Left: Set the port to input with pull-ups OFF. The pull-downs make a zero, and any button pressed creates the column number.
2. Lower Right: Set the port to input with pull-ups ON. Read the port and invert the resulting column number.
3. Upper Left: A ONE output on one pin selects a row while the remaining bits, with pull-ups OFF, read the column. After reading the port, check for a button press and use the parity to correct the

row bit in the column value.

4. Upper Right: A ZERO output on one pin selects a row while the remaining bits, with pull-ups ON, read the column. After inverting the value and checking for a button press, use the parity to correct the row bit.

The pull-downs must be large enough to allow the pull-ups to create a voltage that the micro accepts as a one: typically four to five times larger than the built-in pull-up resistors.

The port pins and the lower-left bank's V+ have current limiting resistors to prevent damage when multiple buttons are pressed, or one is pressed in a bank not currently being scanned. The value is not critical, but should be small relative to the pull-ups and pull-downs – typically between 1kΩ and 10kΩ.



There's a lot of capacitance out there: buttons, wires, diodes, and IC pins. So wait a number of microseconds after changing the port configuration before reading its value, allowing pull-ups/downs to get voltages where they should be.

Multiple button presses may be read as some other location and are not supported. When scanning the upper banks, ignore anything seen in a bank that detects button presses in multiple rows. A button press detected in the lower banks overrides anything seen in the upper ones.

In all cases buttons bounce when pressed and released, so repeat the above until the answer doesn't change for at least 50 ms. [ed: Or, scan only every 50 ms or so. *[MD]*]

Conflicts arise

Though a parity based array can be used with either a true or complementary encoder, a problem results when both are used simultaneously, as in Figure 3. Imagine we are trying to scan the upper-left bank, and output a ONE on a row. We intend (green highlight) for it to "travel" to a button, cross over to a column, activate some diode combination, and return to the inputs. Ah, but instead (blue highlight) it reaches a complementary diode, travels up a column, finds a button in the upper-right array, and returns back through the row to the inputs. This can be read as a button in more than one row of the upper-left bank and be rejected. But if there are two or fewer diodes in the column, and we are pressing one of the buttons in the same row as a diode, it will be misread as a button in just one row of the upper-left bank. Thus, there are twelve locations in the upper right bank that are unusable.

Remember that the three bits read can use any encoding, so by switching the upper-right bank to odd parity (Figure 4), the good locations shift to complete columns.

Time to rein in the imagination a bit

Though this is an interesting exercise, there are, after all, better ways to interface large keyboards to micro-controllers. An I²C I/O expander (PCF8575, TCA9555 , MCP23017, etc.) with 16 bidirectional pins and built-in pull-ups can scan 240 keys – more than 3 times the buttons – with 50% fewer pins, and likely less PCB area.

The point is not so much the large number of buttons that can be connected, but rather the large number of choices you have when connecting any number of buttons. One doesn't have to implement them all.

For example, if no buttons on the left side are used, an entire encoder and the pull-down resistors are unneeded. If fewer buttons are needed, chose the columns with fewer diodes first.

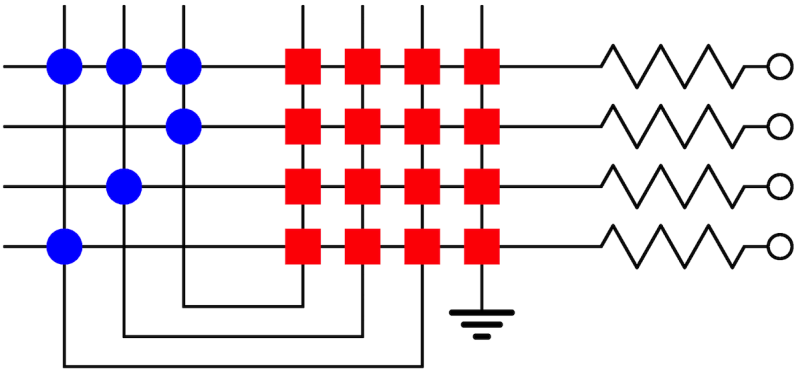


Figure 5 Minimalist example of a 4×4 keypad

Multiple contact closures are not generally supported, but if the aliased positions are not used, no confusion results. Consider a digital joystick, **Figure 6**: four buttons in a diamond pattern, with a handle that allows movement in each direction, plus diagonal, plus all-down. The diagonals and all-down press multiple buttons at once, but that just appears as a different column. And there are still six button combinations available in that bank for further expansion.

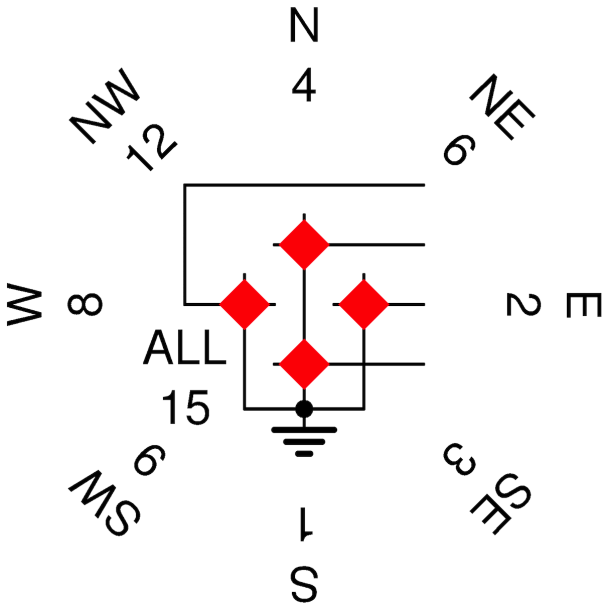


Figure 6 Digital joystick connected to lower-right bank

Testing

I did not construct the complete array in Figure 4. Rather, I made most of the two diode encoders. I connected two 4×4 keypads to the upper-left bank and a third 4×4 keypad to the upper-right bank. I connected a joystick to the lower-right bank, and added a few more buttons to different locations in the bottom banks.

Firmware scans for a button press, waits to debounce the result, and then draws a diagram on the terminal to indicate which buttons *seem* to be pressed, and the final button code composed of the row and column numbers (with special attention to the joystick).

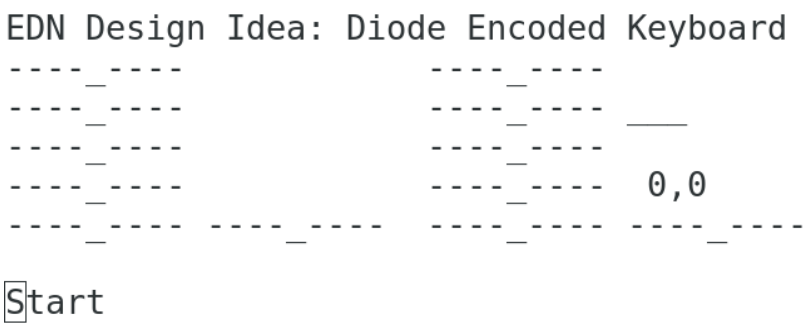


Figure 7 Screen shots with various buttons pressed

Now for one last extravagance

A typical microcontroller interface to a character LCD module (**Figure 8**) often uses only four data bits and three control lines. When *E* is high, the module responds, but when low, the remaining six bits can be used for other things. One can write to or read from the module – the most useful read action is to check the busy flag. If we isolate that bit (D7) from the rest, four of the remaining five can be used to scan a button array. We can use the fifth to implement four *shift* keys (e.g., Control, Shift, Alt, Other), which can be pressed simultaneously with themselves and any other button. We scan them by setting the fifth pin as input with pull-ups ON, while the other four are outputs. We select a shift button one at a time with a zero, and read the fifth pin. We end up with a nearly complete keyboard with no additional pins beyond those necessary for the LCD.

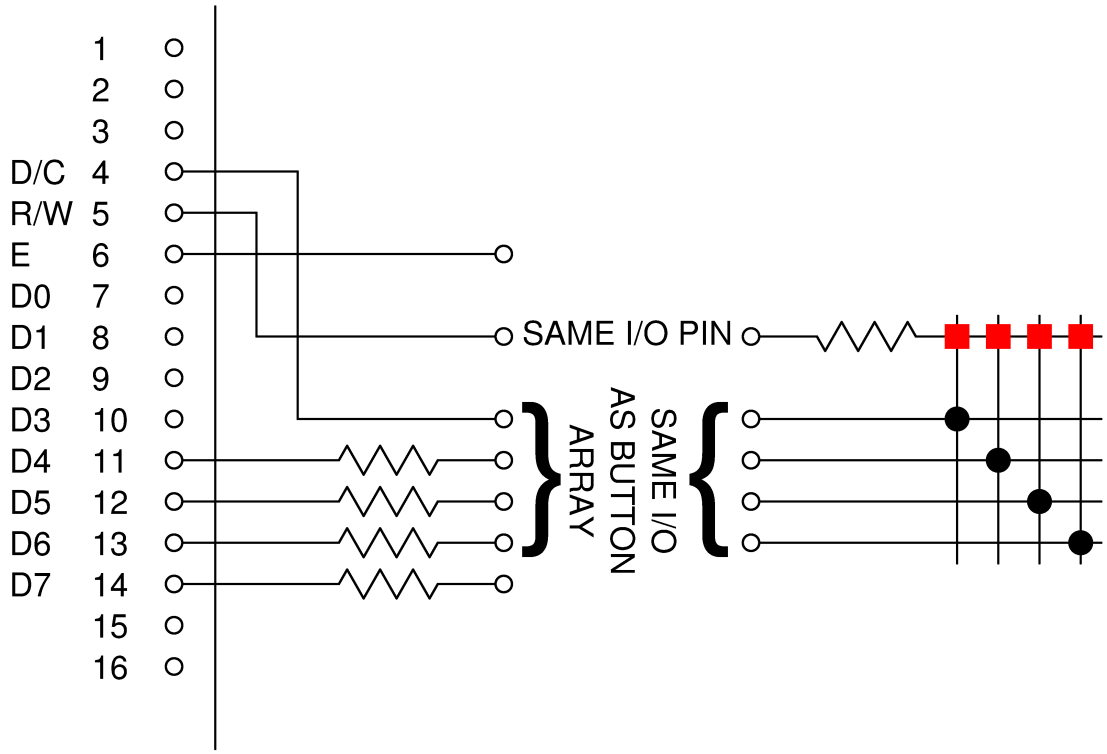


Figure 8 Sharing I/O pins with an LCD module

The current-limiting resistors of the button array prevent it from interfering with any data sent to the LCD, though a button press will corrupt anything read from the LCD on the shared pins. The current limiting resistors in the data lines of the LCD aren't strictly necessary, but if you accidentally output when you should input, it's far better to warm up a resistor than fry a chip.

The key thing to know is that the six I/O pins are reconfigured each time they are used for a particular purpose. The LCD E pin is always an output.

Know thy microcontroller

As this design is implemented using only diodes and resistors, it is critical to consider the I/O specs of your microcontroller. **Figure 9** shows thresholds and loaded output levels of various microcontrollers I've encountered. The red lines represent diode voltage drops down from 5 V and 3.3 V and up from 0 V.

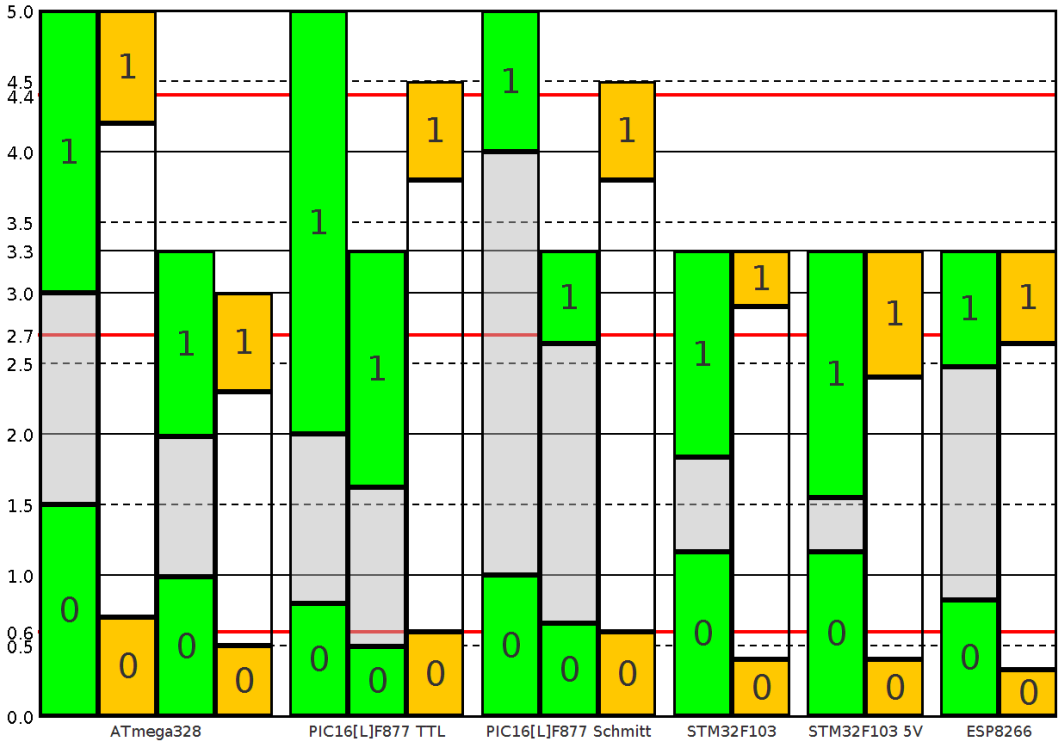


Figure 9 Input thresholds (green) & output levels (orange) for several microcontrollers

There are often complicating details. For example, PICs mostly have Schmitt trigger inputs, but some are TTL, and others are unique due to the requirements of multiplexed functions on the same pin. STMs have CMOS or TTL (5V-tolerant) pins, each with different specs.

Data sheets rarely give output details, often listing one or two cases with a specific supply voltage and (usually midrange) current. In this application, there is usually little voltage drop, but outputs won't reach rail-to-rail, so be sure to leave as much margin as possible. Noise is all around, so be careful how close you crowd the limits. Schottky diodes may have to be used in some cases to meet the particular requirements of a given microcontroller and supply voltage.

Download assembly code implementation.

Related articles:

- **Read 10 or more switches using only two I/O pins of a microcontroller**
- **Two PIC pins drive six LEDs**
- **Active multiplexing saves inputs**
- **3 pins, 3 LEDs, 3 buttons**
- **Charlieplexing at high duty cycle**
- **"Chipiplexing" efficiently drives multiple LEDs using few microcontroller ports**
- **Increase efficiency in embedded digital-I/O lines**

- **Keyboard and display multiplexing -- the traditional approach**
- **Keyboard and display multiplexing -- Charlieplexing**

—Jim Brannan is a systems programmer who also loves designing hardware and messing with microcontrollers.