PIC n' Mix

Mike O'Keeffe

Our periodic column for PIC programming enlightenment

Four-digit, seven-segment LED display – Part 3

WE'VE BUILT a 24-hour clock with an LED display, but we want more! This month, we're going to add to the design and build a simple calculator – but, all the available GPIO pins have been used up; what to do?

In *Part 1* and *Part 2* of the four-digit seven-segment display, we built a simple clock, and you will need to re-read *Part 1* in order to build and understand this month's design. However, as just noted, we find we have a few problems. The biggest head scratcher is the number of available pins. We have used up all the GPIO pins on the PIC16LF1829 to control the LED display. So, to add extra functionality, we will need to figure out a way to connect any additional hardware.

Another problem to work around is the limited number of digits. There are only four digits on the display, limiting the range of numbers from 0 to 9,999. We also have decimal points on the display, which should allow us to display numbers as low as 0.001. For the moment, however, we will treat this as a software issue, which will be covered next month.

The keypad

We want to convert our design to behave like a calculator, which means we will need a keypad of some sort. Ideally, we want a keypad displaying the numbers from zero to nine, and keys for the basic arithmetic functions of addition, subtraction, multiplication and division. We also need a key for the equals sign to indicate we have completed entering our calculation. In total, this means fifteen keys. A 4×4 keypad would be ideal for this situation, giving us one extra key, which can be used as a clear display button, or maybe we could use it for the decimal point. Fig.1 shows the layout of these keys on a keypad, with the suggested numbering.

Now let's consider the behaviour we want for the calculator. We need to enter a first number followed by some mathematical function (add, subtract, divide and multiply) and then another number. Once the second number has been entered, we need some means of indicating we want the answer. The equals key tells the 'calculator' that we have entered the two values with some math function in between, and it's time to calculate and display the answer.

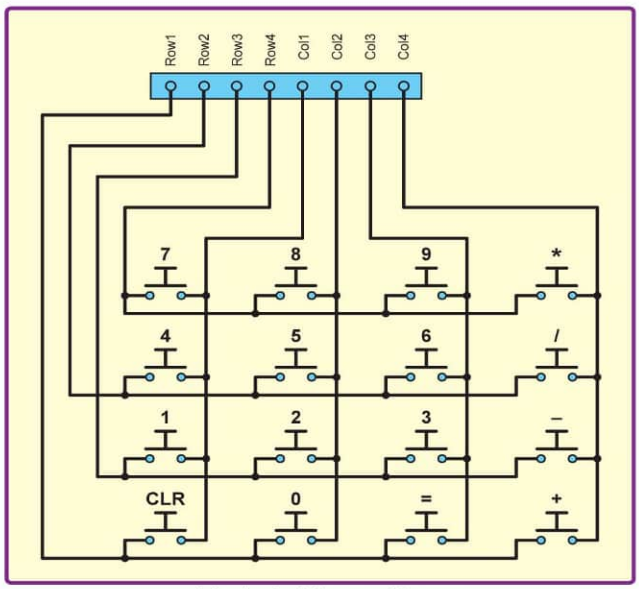


Fig.1. 4×4-keypad layout

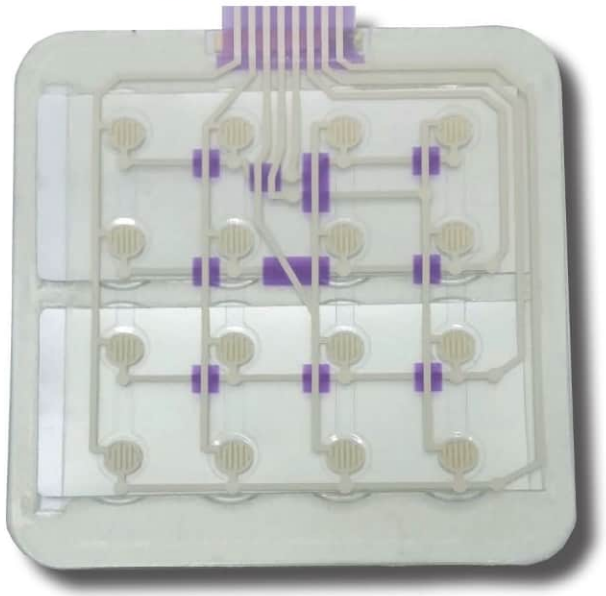


Fig.2. The underside internal working of a 4×4 keypad

There are going to be a few error scenarios that will need to be covered, which we will look at in the software next month. Examples include getting negative numbers, whole numbers rounding up and results that exceed the displays capacity. There are creative ways to work around these obstacles, but it's best to start simple and build on that.

The keypad operates in a 4×4 matrix. Fig.1 shows the schematic for the keypad, which is arranged into rows and columns. Fig.2 shows the underside internal workings of the keypad used in this project (the 4×4 AC3561 by APEM). The pads themselves are interleaved pieces of exposed copper. When a key is pressed, two pieces of copper are connected. The PIC microcontroller detects which row and column have been connected, allowing it to determine, which key was pressed.

The keypad's eight output pins are connected to each of these four rows and four columns. Typically, these keypads are configured by connecting all of the row and column pins to individual pins on a microcontroller. These pins are also connected to ground via a resistor, and are known as weak pull downs. The microcontroller cycles through each *column*, pulsing it high, and checking the state of each *row*. If a row pin goes high, then it knows a button has been pressed relating to that row and column.

There's just one problem here; we don't have eight pins available on the PIC. We have three options: 1) we could redesign our original design, cutting tracks and wires and adding in awkward connections, or 2) we could add a port extender, adding more GPIOs to our design, adding further complexity, or 3) something else!

In the original design, we switch between the common cathodes for each of the four digits. We could potentially hijack this behaviour and attach it to our row pins. This could cause some timing problems with our display. Not to mention, we're still four pins short.

All on one pin

In fact, we still have one pin available, which is currently used to program the PIC. Port RA0 is connected to the

ICSPDAT on the programmer. This is a digital data pin used for programming the PIC. It is possible to use this one pin to figure out what key has been pressed on the keypad. By using seven resistors, we present a unique voltage to RA0 for each key. RA0 will use an analogue-to-digital conversion (ADC) to translate these unique voltages into button presses.

By using a series of resistors that will behave as multiplexed voltage dividers, we can establish unique voltages for each key, and that unique voltage will be seen by the ADC module. We can then map these values to each key being pressed. The resistor values need to be calculated to ensure sufficient spacing between each voltage value. A 10-bit ADC typically yields an accuracy of about 2-3 bits. At 3.3V, this gives us around 10mV margin of error. Ideally, we want to ensure the difference between each value is much more than this.

Using an excel spreadsheet, I quickly worked out one set of values for these resistors, giving a decent spacing between each value. This could be optimised further. The spreadsheet will be included with the software download on *EPE's* website next month. Check it out and see if you can improve the values to reduce possible errors.

Fig.3 shows the schematic of our modified design. We use a 1kΩ pull-up resistor on RA0 to V_{CC}. This ensures when no button is pressed, our ADC value should be 1023 bits or 3.3V. When a button is pressed, what we are really seeing is a multiplexed voltage divider circuit. By pressing the number 2 on the keypad (Fig.1), we connect row 3 and column 2, which adds R13, R14 and R17 into the circuit.

A typical voltage divider circuit in shown in Fig.4, and the voltage divider equation is:

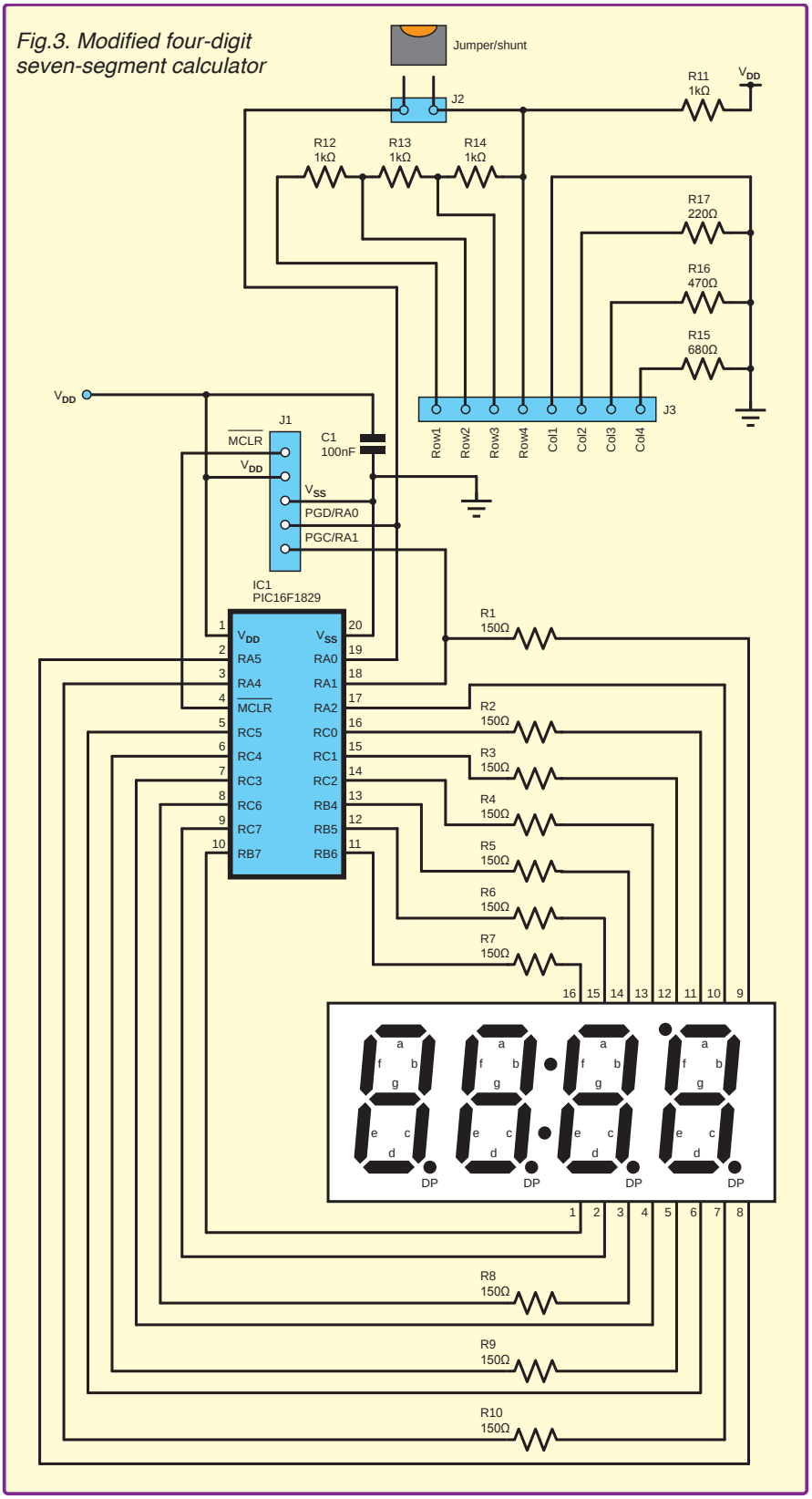
$$V_{out} = V_{CC} \times R2 / (R1 + R2)$$

Here, V_{out} is our ADC voltage. In the above '2' example, V_{CC} = 3.3V, R1 refers to the R11 1kΩ pull-up resistor, and R2 is the sum of R13, R14 and R17 (1kΩ + 1kΩ + 220Ω = 2.2kΩ). This should give us a value of 2.275V or 716 bits. When we capture this value, we now know the number 2 has been pressed. And we can add a debounce or delay afterwards to ensure we only recognise one press of the button (a particularly nice example of why debounce is important).

One small problem with programming

I mentioned earlier that RA0 is also a programming pin. The addition of the resistor network in Fig.3 poses problems when trying to program or debug the microcontroller. The 1kΩ pull up on R11 will hinder programming the board by making it harder for the programmer (eg, PICKit3 or ICD3) to output a logic low. In theory, we could scale the resistor values up so that they do not affect the programming. This could be done by multiplying each value by ten or a hundred. We would still get the same unique voltage for each key. The only problem here is that the resistors will start affecting the capture time of the ADC. The capture and conversion for the ADC needs to be quick to avoid problems on the display. Increasing resistor values will slow the voltage rise on the ADC pin as a key is pressed.

Fig.3. Modified four-digit seven-segment calculator



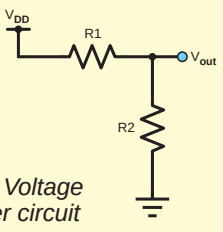


Fig.4. Voltage divider circuit

The only way to make this work is to disconnect the resistor circuit during programming. Soldering and

resoldering would be a nuisance, so I recommend using a two-pin header and a jumper or a shunt bar across the header to reconnect it – see J2 in Fig.3.

Constructing the circuit

To build the circuit, you need the following components:

Four-digit seven-segment display circuit from last month (December 2017)

Resistors

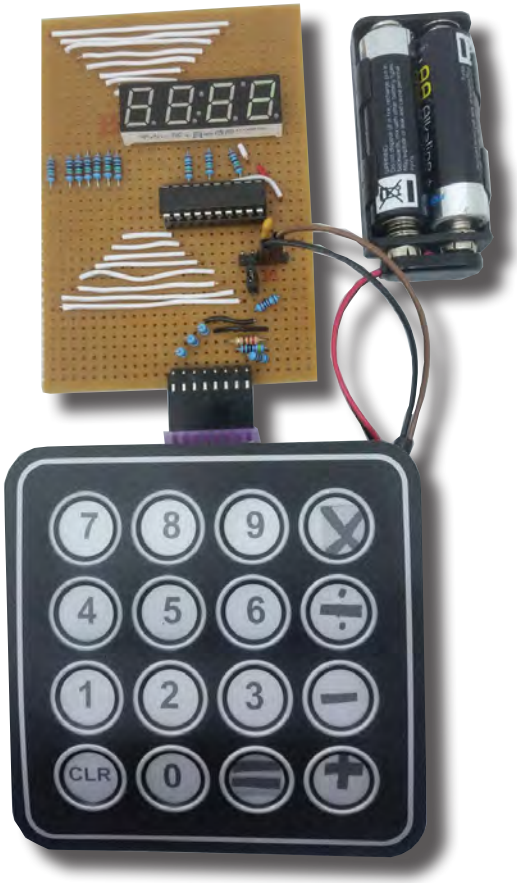
4 1kΩ (R11, R12, R13 and R14)
1 220Ω (R17)
1 470Ω (R16)
1 680Ω (R15)

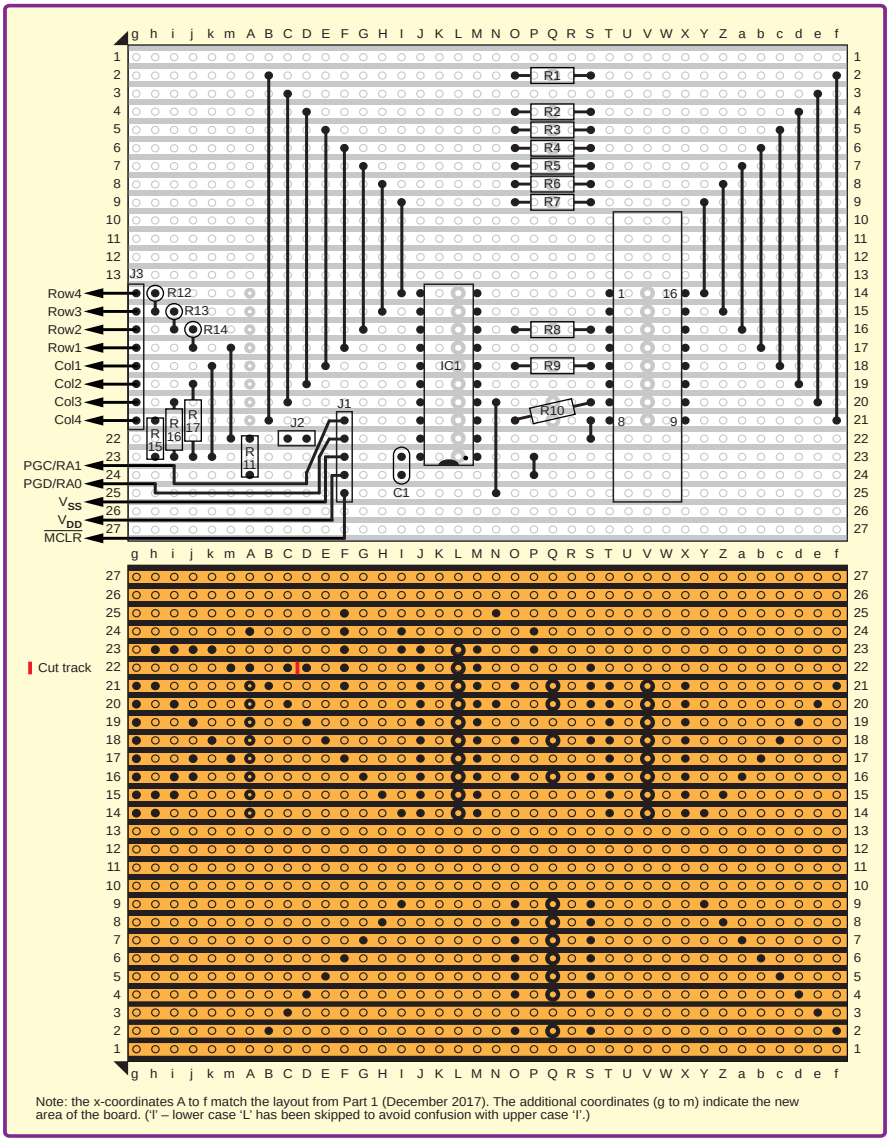
Miscellaneous

1 2-pin header (J2)
1 Jumper or shunt bar (J2)
1 8-pin right-angled header (J3)
1 4×4 keypad (eg, AC3561 by APEM, as used here)

Fig.5 shows the veroboard layout for the components on the top side and the underside of the board. Don't forget to carefully make the jumper track cut between the holes on C22 and D22. Everything from Column B upwards is from the original design (excluding the connector J2). There are only nine components and two wires to add. The spacing for some of the resistors is a bit tight, so it might be easier to stand these up instead (see Fig.6).

Fig.6 shows the complete working calculator. The keypad from APEM comes with two silver inserts with the digits printed on them. These inserts originally came with other signs on them, which have been covered over for this project. Notice in Fig.6 that I have made my own mathematical symbols using a bit of black electrical tape. Feel free to move these around, but don't forget to make the adjustments in the software next month.





Note: the x-coordinates A to f match the layout from Part 1 (December 2017). The additional coordinates (g to m) indicate the new area of the board. ('I' – lower case 'L' has been skipped to avoid confusion with upper case 'I'.)

Fig.5. Veroboard project layout

It is interesting to consider this solution compared to the alternative methods of adding functionality to an existing design. It happens a lot more than I like to admit, where a project suffers from feature creep. Often this means a complete redesign, but in this case, a little bit of creative ingenuity saved us the hassle of creating a completely new design or some awkward wiring and track cuts.

All in all, I believe this method is a lot less complicated than the original method of controlling the keypad. There is a little more in the build, albeit only seven resistors. In fact, the true beauty of this design will reveal itself in the software next month. If we did have the necessary pins and we had to look at several inputs to discover what key has been pressed, this would add significant delays to the code, which would affect the behaviour of the LED display.

The software involves using an ADC to capture the input voltage and a function that maps that value to the corresponding button pressed. After that, it's a matter of calculating everything quickly enough without affecting the display.

Next month

We've now built the guts of the simple calculator; next month we'll look at how to program the microcontroller to capture the keys pressed on the keypad, display them on screen and perform simple calculations. As we mentioned last month, any delays in our code will cause the segment LEDs to flicker. This means our code will need to be quick enough that we don't notice what is happening in the background.

Not all of Mike's technology tinkering and discussions make it to print.

You can follow the rest of it on Twitter at *@MikePOKeeffe*, on the *EPE Chat Zone* or *EEWeb's* forums as '*mikepokeeffe*'

and from his blog at *mikepokeeffe.blogspot.com*