Stamp Applications no. 22 (December '96):

Scan a Keypad with the BS2 For Pushbutton User Input

16-key matrix keypad software
plus beginner's race-timer project
by Scott Edwards

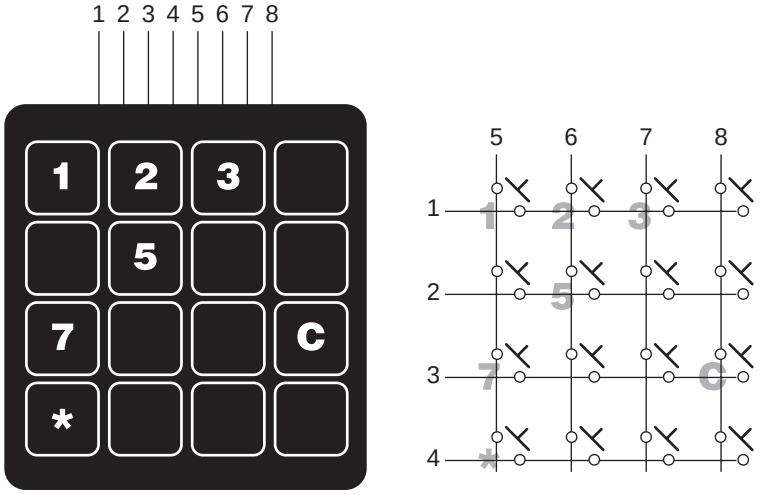
THE BUTTON instruction offered by both of the BASIC Stamps makes it easy to interface one or two buttons to your project. But if your application requires more sophisticated input, a keypad is almost a necessity.

Button doesn't do keypads.

This month, we'll look at a typical keypad and write a general-purpose BS2 subroutine for reading it.

In BASIC for Beginners, we'll use Boolean logic to craft an elegant three-lane race timer demo with the BS1.

Matrix Keypad. Figure 1 is a 16-button keypad made by Grayhill and sold through Digi-Key (Sources). As the circuit shows, a keypad is more than just a bunch of buttons bolted to a panel. It's a *matrix* arrangement designed to minimize the number of I/O lines required to read the buttons. Ordinarily, reading 16 button states would require 16 inputs. By arranging the buttons into rows and columns as shown in the figure, we cut the number of pins to eight. This savings in hardware requires some additional software—a keypad scanning routine.



Grayhill Keypad
(DK: GH5003-ND)

Equivalent Circuit
1–4 rows; 5–8 columns

Figure 1. Typical matrix keypad.

The duties of keypad-scanning code are:

- To determine whether or not a key is pressed.
- If a key is pressed, to determine which one.
- To notify the rest of the program that keypad input is available.
- To avoid registering more than one input for a given keypress.

Although that sounds like a lot, a keypad scanning routine can be quite compact, as listing 1 shows.

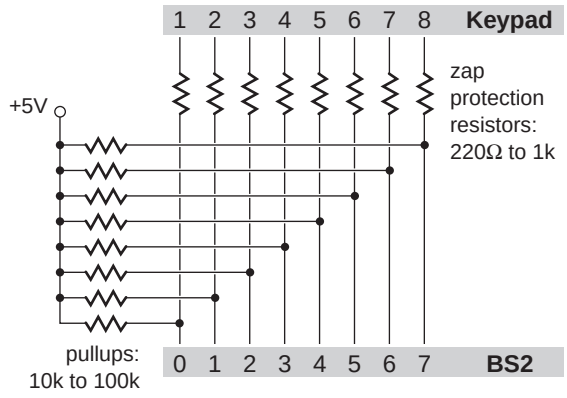


Figure 2. Hookup for listing 1.

The keypad scanner is a systematic search. It starts with all connections to the keypad set to input. The pins are set high (1) by pullup resistors. The object of the search is to determine whether one of the rows of the keypad is shorted to one of the columns. The scanner outputs a 0 to one of the rows, then looks at the column inputs to see whether that 0 shows up. If not, it turns off that row output and tries the next row.

When the scanner does find a 0 on a column input, it can easily determine which key is pressed using this logic: If we number the keys from top-left to bottom-right, the first row of keys consists of 0, 1, 2, 3; second—4, 5, 6, 7; third—8, 9, 10, 11; and fourth—12, 13, 14, 15.

So the key number is (4 x row) + column. Of course, that orderly numbering of the keys does not match the labels on the keypad itself, which is scrambled to conform to the arrangement of a phone keypad, but a lookup table takes care of that.

That's the core of the listing-1 keypad scanner. There are some other subtleties:

• The scanner uses the NCD operator to convert the column bits into a bit position. For example, suppose the column bits are arranged with the 0 in bit 2, %1011 (% is the PBASIC symbol for binary numbers). The program inverts %1011 to 0100 using the logical-not operator (~). Then it applies the "priority encoder" operator NCD, which returns the bit position plus 1, which is 3.

• The NCD writeup in the current BS2 manual is wrong. It indicates that NCD returns the bit number, 0—15, of the highest bit position of a value that contains a 1. If the value doesn't contain a 1 in any bit position (value = 0), NCD is supposed to return 255. However, NCD actually returns 0 for a input value of 0, and 1—16 as the highest bit position containing a 1. That's why the program subtracts 1 from the NCD result before computing the key number.

• Efficiency-minded programmers might be wondering why I used separate program lines to invert the column bits and compute their NCD. Why not do 'em both with one expression: NCD ~ col? Try it. The program won't work. The reason is that PBASIC2 does all of its math and logic in a 16-bit workspace. Only when the result is written to a variable are the extra bits trimmed off. Consider what happens when you invert a 4-bit number like %1011 in a 16-bit workspace. First the number is converted to its 16-bit equivalent, %0000000000001011. It is inverted to %1111111111110100. Then the NCD of this value is computed, yielding 16. But the answer we expect is 3, based on the 4-bit value %0100. To get the correct result we have to trim off the additional bits, either by ANDing the intermediate result with %1111, or by simply writing it into the 4-bit variable. I chose the latter, because it seems clearer to me.

• The scanning routine uses a bit variable, db, to *debounce* the keypad input. Debouncing prevents multiple responses to a single keypress by setting some condition for registering a new press. In this case, the condition is that the previously pressed key must be released before a new press will register. You can test this by holding a key down—only one response will

appear on the Debug screen.

• Another bit variable, press, is used to tell the main program that the scan routine has detected a keypress. It's the program's responsibility to clear this bit once it has processed the input.

• Finally, a hardware note: In figure 1 there are resistors in series with the lines to the keypad. These are not strictly necessary; they are just a precaution against static damage. If you are just bench-testing the circuit, feel free to omit them. But if you are building a device that will be used in the real world, spring for the resistors. They are cheap, partial protection against static zaps from users' fingertips.

BASIC for Beginners. Last month we started work on a three-lane race timer using the BS1. We wrote some code to detect the start-of-race pulse, but ran into difficulty with maintaining independent timing on the three lanes while reliably detecting an end-of-race condition. After exploring some blind alleys, we concluded that the solution might be to keep a record of each car's status in terms of "in-the-race" or "finished." If a particular car is in the race, then we update its elapsed time, otherwise we stop the timer on that car. If all cars are finished, the race is over.

To save you the trouble of dragging out the previous issue of *N&V*, figure 3 once again shows the race-timer switches.

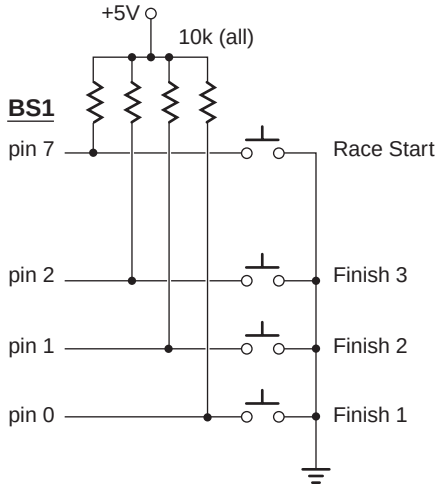


Figure 3. Arrangement of race-timer switches.

Now that we have a logical way to keep times and detect the end of the race, let's see how our ideas translate into a BASIC program. Let's start with variables.

First, we need to decide how to store the cars' times. We can choose from three storage sizes: bits (0,1), bytes (0—255) or words (0—65535). Clearly, bits are not suitable for storing race times, so the choice is between bytes and words.

It would be nice to base the decision on some hard facts, like the timing resolution in units/second and the maximum length in seconds of a race. For example, if races last 10 seconds or less and timing resolution is 1000 units/second, then we need variables that can hold values up to 10,000.

At this point, though, we don't know what our timing resolution might be. A stock BS1 executes about 2000 instructions per second; a quad-speed Counterfeit, about 8000. It will certainly take several instructions to keep time and detect the end-of-race, so we'll estimate a maximum of 2000 units/second at quad speed. At that rate, a byte variable would give us a maximum racing time of 255 x 1/2000 = 0.12 seconds. A word variable would provide 65535 x 1/2000 = 32.7 seconds. So the race timers will be word variables.

How about race status? We really need only two conditions—racing and finished. That's an ideal job for bit variables. We can use a 1 to represent racing and 0 to represent finished.

Next we need to determine how these variables will work together. Without writing any real code, let's sketch the main loop of the program. (I'm using a different typeface to indicate that this is not PBASIC, but *pseudocode*, a sort of pidgin-English/BASIC):

```
Timing:
  Get state of Finish switches
  Update racing/finished status bits
  If status1 = racing, then increment time1
  If status2 = racing, then increment time2
  If status3 = racing, then increment time3
  If status1=done AND status2=done AND...
    ...status3=done then Finished, else goto Timing
Finished:
  Report race times
```

Seems like a lot of If...Thens. Thinking about the status bits; they are going to contain 1 for racing and 0 for done. What if we change that to:

```
Timing:
  Get state of Finish switches
  Update racing/finished status bits
  time1= time1 + status1
  time2= time2 + status2
  time3= time3 + status3
  If status1=done AND status2=done AND...
    ...status3=done then Finished, else goto Timing
Finished:
  Report race times
```

Adding the status bit to the race time accomplishes the same thing as using the status bit to decide whether or not to add 1 to the race time. And it's a heckuva a lot simpler.

Now, what about those vague phrases regarding getting the state of the finish switches and updating the status bits? We need translate those into something closer to BASIC.

The finish switches pulse low (0) briefly as a car crosses the finish line. The status bits must change from high to low when the corresponding finish switch pulses. The tricky part is that while the finish switch may return high, the status bit must remain low after the car has finished. We need the logic equivalent of a trap door that can go from 1 to 0, but not from 0 to 1.

Sounds like a job for AND (&).

In column 14 (available from the *N&V* Internet library) we discussed the Boolean logic operators, including AND. Here's how AND works:

AND (symbol: &)

first bit	second bit	result
0	0	0
0	1	0
1	0	0
1	1	1

AND is just the trap door we need. During the race, a car's status bit is 1 and its finish switch is 1. AND 'em together and you get 1. Write that back into the status bit. When the car crosses the finish line, the status bit is 1 and the finish switch is 0. 1 AND 0 = 0, which goes back into the status bit. Now the car is officially out of the race. After the car is completely across the finish line, the status bit is 0 and the finish switch returns to 1. 0 AND 1 is still 0, so the status bit remains correct.

That leaves just the final detail of determining when the race is over. Our pseudocode shows this as a compound If...Then instruction combining the states of the three status bits. Maybe this can be simplified, too.

Although PBASIC lets us use individual bit variables, those bits are also accessible as portions of a byte variable. For example bit0, bit1, and bit2 can be addressed individually, or as the lowest three bits of the byte variable b0. There are eight possible states for those three bits, and all we need to do is identify one of them—the case in which all three bits are 0. If we ignore the other five bits of b0, when those bits are 0 the byte b0 is 0. So that complicated If...Then can be reduced to IF b0 = 0 THEN... provided we can clear the upper five bits of b0.

Once again we can call upon the bit-clearing power of AND. If we AND b0 with a number that has 0s in the upper five bits and 1s in the lower three bits, we can zero out the bits that we want to ignore.

And that's all we need to form the core of the race-timing application. See listing 2 for a demo.

Next time we'll clear up the loose ends of converting our race times to units and displaying race results.

Sources

The Grayhill keypad is available from Digi-Key, Digi-Key, 701 Brooks Avenue South, PO Box 677, Thief River Falls, MN 56701-0677; phone 1-800-344-4539, fax 218-681-3380, net http://www.digikey.com.

NOTE: This article was originally published in 1996. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.

Listing 1. BS2 Program to scan a 16-key matrix keypad

```
' Program: KEYP.BS2 (Scan a 16-key matrix keypad)
' This program shows how to scan  a 4x4 matrix
' keypad using a BASIC Stamp II (BS2). A subroutine
' scans the keypad when it is called. On return, a
' flag bit (press) will contain a 1 if a key was
' pressed, and a nibble variable (key) will contain
' the key number (0-15). Once the program has
' responded to a key press, it must clear the
' press bit to prevent multiple actions triggered
' by the same key press. In a similar way, the
' keyScan subroutine uses another bit, db, to
' avoid responding to a key press until the key
' previously pressed has been released.

db      var    bit     ' Debounce bit for use by keyScan.
press   var    bit     ' Flag to indicate keypress.
key     var    nib     ' Key number 0-15.
row     var    nib     ' Counter used in scanning keys.
cols    var    INB     ' Input states of pins P4-P7.

' Demo loop. Waits for press to indicate a keypress, then
' displays the key on the debug screen. Note that that
' this code clears the press bit when done in order to
' prepeare for the next press.

again:
  gosub keyScan
  if press = 0 then again
  debug "key pressed = ", hex key,cr
  press = 0
goto again

' ==================== KEYPAD SUBROUTINE ====================
' This code scans a 0 across the row connections of the keypad,
' then looks at the column nibble to see if that 0 has shown up
' on any of those bits. If the column bits are all 1s, then
' no key is pressed. If a column bit is 0, then a key is pressed
' at the intersection of the current row and that column.
keyScan:
for row = 0 to 3        ' Scan rows one at a time.
  low row               ' Output a 0 on current row.
  key = ~cols           ' Get the inverted state of column bits.
  key = NCD key         ' Convert to bit # + 1 with NCD.
  if key <> 0 then push ' No high on cols? No key pressed.
  input row             ' Disconnect output on row.
next                    ' Try the next row.
  db = 0                ' Reset the debounce bit.
return                  ' Return to program.
```

```
push:
  if db = 1 then done   ' Already responded to this press, so done.
  db = 1: press = 1     ' Set debounce and keypress flags.
  key = (key-1)+(row*4) ' Add column (0-3) to row x 4 (0,4,8,12).

' Key now contains 0-15, mapped to this arrangement:
'           0    1   2   3
'           4    5   6   7
'           8    9  10  11
'           12  13  14  15
' A lookup table is translates this to match the actual
' markings on the key caps.

  lookup key,[1,2,3,10,4,5,6,11,7,8,9,12,14,0,15,13],key
done:
  input row             ' Disconnect output on row.
return                  ' Return to program.
```

Listing 2. BS1 race timer (BASIC for Beginners)

```
' Program RACE1.BAS (Prototype three-lane race timer)
' This program shows how the BS1 (or Counterfeit) can
' be used to time a three-lane Pinewood Derby race
' without complicated IF..THEN programming. This
' program is a prototype; when the race is over it
' displays raw data on the PC screen via Debug. Later
' versions will convert the data to units (fractions of
' a second) and display them on a freestanding display.

SYMBOL time1 = w2      ' Word variable for lane-1 time.
SYMBOL time2 = w3      ' Word variable for lane-2 time.
SYMBOL time3 = w4      ' Word variable for lane-3 time.

SYMBOL start = pin7    ' Start-switch on pin 7; 0=start.

SYMBOL status1 = bit0  ' Status of lane 1; 1=racing, 0=done.
SYMBOL status2 = bit1  ' Status of lane 2; 1=racing, 0=done.
SYMBOL status3 = bit2  ' Status of lane 3; 1=racing, 0=done.
SYMBOL stats = b0      ' Byte variable containing status bits.

stats = %111           ' All cars in the race to begin.

hold:
if start =1 then hold  ' Wait for start signal.

timing:                ' Time the race.
 stats = stats & pins & %111    ' Put lowest 3 pin states into stats.
 if stats = 0 then finish       ' If all cars done, then race over.
 time1 = time1 + status1        ' If a car is in race (status=1) then
 time2 = time2 + status2        '  increment its timer. If it's done
 time3 = time3 + status3        '  (status=0) don't increment.
goto timing                     ' Loop until race over.

finish:
 debug cr,"Checkered flag!",cr  ' Race over.
 debug time1                    ' Display results.
 debug time2
 debug time3
```