Add a Keypad

to your DDS Development Kit

By Bruce Hall, W8BH

The DDS kit by W8DIZ is a real beauty. Built-in encoder, dual outputs, 0-30MHz range, LCD display, ready to pop in your latest radio project. Go ahead. But if you are like me, you like to get under the hood and tinker a bit. See what makes it tick. The following experiments in this eight-part series will show how to interface a keypad in a simple, from-the-ground-up approach. If you just want the end result, skip to the final code. But if you want to see how I learned to do it, step by step, keep reading.

It's hard to know exactly where to start. I downloaded and read a very nice tutorial at www.avr-asm-tutorial.net. For the uninitiated it can be overwhelming. But if you have any programming experience you will understand the concepts pretty well. Look at some sample code for making an LED blink. Turn the LED pin on, wait, turn the pin off, wait, and repeat. Simple, right? Now download and print the DDS kit source code at w8diz.com/qq-fc-project/ATmega88PA/. Yes, I said print. Find the routine called Blink_LED. Spend some time reading the code and annotating the printout, figuring out what each section does. Some parts will be nearly impossible at first glance, others easy. Find the main program loop, which goes from 'main program' to 'rjmp menu'. After staring at it for a while, you'll realize that all it does is check for encoder data and button presses. Sure, lots of jumping here and there, but nothing conceptually difficult.

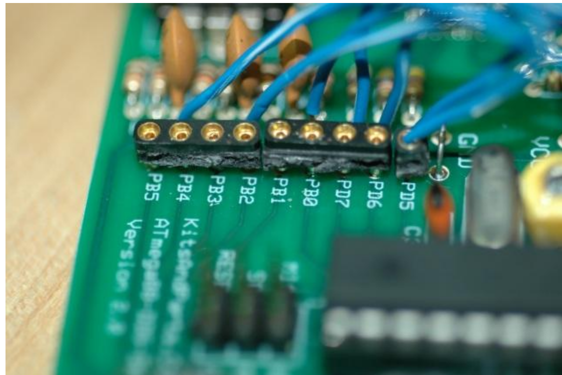
Let me speed through the really time-consuming parts. First, you need to get an AVRISP-II programming cable, available at DigiKey [ATAVRISP2-ND] for about $35. When I was ordering the cable I also ordered a few spare ATmega88 chips. I didn't want to ruin the pre-programmed chip that came with my kit. Next, download AVR Studio 4 from the Atmel website at http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725. While waiting for the hardware, I spent quite a few hours poring over the source code, trying to figure out the nitty-gritty details. The more time you spend on it, the better. Learn how to use AVR studio and your programmer. Pat yourself on the back when you can put a blank chip in your kit, set the fuses, program it with the source code, and watch it come to life. Let the fun begin!

The first step for me was figuring out how to physically connect the keypad. The easiest way is connecting each of the 4 row and 3 column lines to a microcontroller pin. Are there enough I/O pins available? Here is a table of the ATmega88's I/O pins and how they are used in your DDS

kit. The available pins I will use for row & column connections are marked in red.

Of the 22 total I/O lines (not counting RESET), your DDS board uses 12, leaving your with 10 I/O lines to play with. We only need 7, so we are in good shape. These lines are brought out and conveniently labeled for you in the center of the board. I picked the bottom seven lines, PD7 and PB0 through PB5, since they were all available.

Bit Number	Port B	Port C	Port D
0	(AVAIL: c2)	LCD	DDS
1	(AVAIL: c3)	LCD	DDS
2	(AVAIL: r1)	LCD	Encoder
3	(AVAIL: r2)	Led	Encoder
4	(AVAIL: r3)	(AVAIL)	Encoder
5	(AVAIL: r4)	(AVAIL)	DDS
6	xtal	---	(AVAIL)
7	xtal	---	(AVAIL: c1)



Now that we know where to physically attach the keyboard, let's start to program. Before we use any I/O pins we must define them as inputs or outputs. This is done by writing to the Port's data direction register. In this keypad implementation we are going to (arbitrarily) assign the columns as outputs and rows as inputs. You could do it the other way around, if you wanted.

Here is the first bit of code:

```
;*****************************************
;* PORT B&D INITIALIZATION for KEYPAD
;*****************************************
ldi         temp1,$03          ;binary 0000.0011
out         DDRB,temp1         ;set PB0,1 as output

ldi         temp1,$3C          ;binary 0011.1100
out         PORTB,temp1        ;set pullups on PB2-5

ldi         temp1,$A3          ;b1010.0011 (add bit PD7)
out         DDRD,temp1         ;set PD0,1,5,7 outputs
```

This code replaces the two existing code lines (ldi temp1,$a3; out DDRD,temp1) in the 'init port' section of your source code. These 6 lines of code contain 3 sets of 2. In each set, we load a register with a value, and then in output that value to a port. The first 2 lines we send a value to the Data Direction Port B (DDRB), which specifies for each port B bit whether it is an input or an output. To make a bit an output, we send a '1' to the corresponding bit in the DDRB. The value 00000011 sets the lowest 2 bits in the port B register (PB0 and PB1) as outputs, and the

remainder as inputs. The output pins correspond to what we've assigned to column 2 and column 3.

The next two lines of code enable internal pull-up resistors on Port B, bits 2 through 5. Internal pull-ups are activated by writing to an input pin. Here we are outputting the value 00111100, which activates 4 pull-ups resistors (the four '1' values) on bits 2 through 5. The internal pull-ups allow us to keep the inactive row inputs at a logic 1 state, unless brought low by an actively-low column.

The last two lines set the PD7 pin as an output. But the DDS kit also uses three other outputs on port D, so we must include them, too.

Our first 6 lines don't actively do anything – they just configure the pins that we want use with our keypad. Now that we've finished that chore, let's try something more interesting.

A beginner's first program conventionally displays 'Hello, world'. And a microcontroller's version of hello world is to blink an LED. So that's the first thing I tried to do on my kit. Here is the code:

```
      ;**************************************
      ;* Experiment #1
      ;* Make the LED flash.
      ;**************************************
KEYPAD:
      ldi         temp1,1             ;set up for 1 blink
      rcall       blink_led           ;already programmed for us!
      ret
```

You can put this code just about anywhere. I inserted it just after the main program loop, and before the label 'DISPLAY_LINE'. This code won't work by itself. You must call this routine periodically from somewhere else. And a great place to call it from is the main program loop. After 'menu9:' and before 'rjmp menu', insert the following line:

```
      rcall       keypad              ;check for keypad entry
```

Now assemble your new source code, attach your programming cable, and program the ATmega88. You should see the LED blink now. Of course, this is not very useful, and in fact it slows down your system so that the encoder will not work too well. But, you have verified that your programming method is working, and you now have a great starting point to build your code.

The next few experiments will control the LED, based on keypad input.

Experiment #2

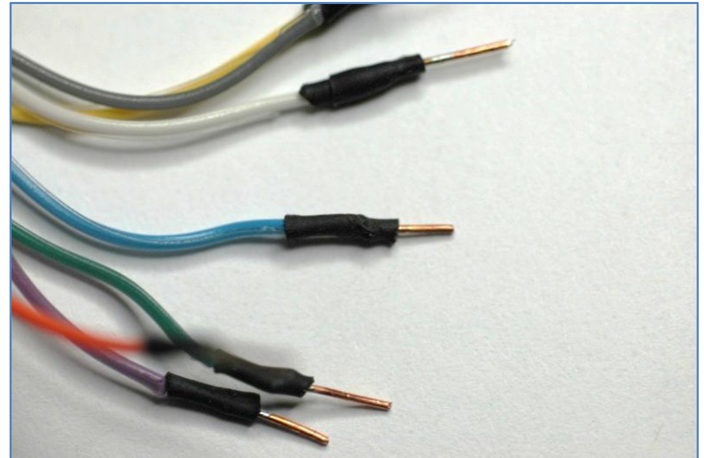
My first 'real' experiment on this kit was to make the LED blink whenever a row input (PB5) when to ground. We only need to add one extra instruction. Easy! Here is the code:

```
      ;**************************************
      ;* Experiment #2
      ;* tap PB5 to ground and watch LED flash.
      ;**************************************
KEYPAD:
      ldi         temp1,1           ;set up for 1 blink
      sbis        PinB,PB5          ;don't blink if PB5 is 1
      rcall       Blink_led         ;get here only if PB5 is 0
      ret
```

There are several different ways to check an input pin, and do something based on the input. I am using the 'sbis' instruction here, which skips the following instruction if the bit is set (1). Our next instruction is to blink, so we blink only if the bit is clear (0).

Now assemble the code and program your chip, as before. Each time you bring PB5 to ground, the LED should blink.

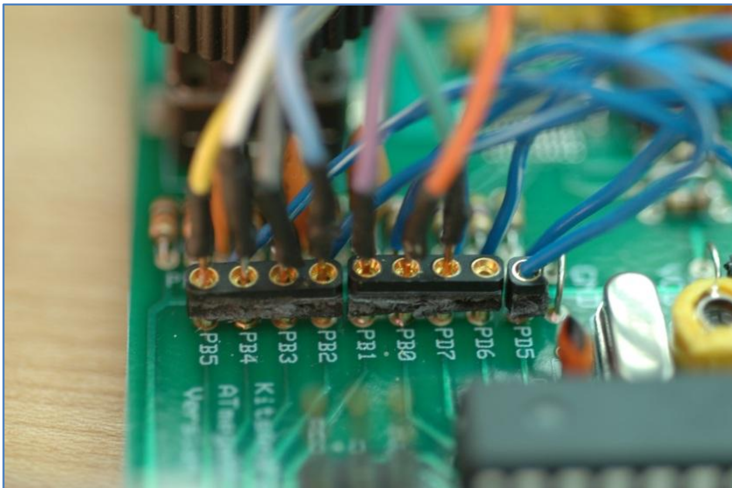
Writing code is interesting, but building is better! Now that the row inputs are finally getting action, I decided to start wiring the keypad. I already had machine pin sockets soldered on my board, so it was easy to plug in 24 gauge solid hookup wires (Cat5 cable works great). But I also wanted to use rainbow ribbon cable to connect the keypad to the DDS. Add some solder and heat-shrink tubing, and you have a flexible way of connecting your wires to any available pin. Here is a photo of all seven keypad wires attached to the DDS board:



If you have gotten to this point, the rest is easy – and fun! You have a place to do your own experimenting. Try blinking twice instead of once. Try using one of the other row inputs.

In the next part, we will connect multiple row inputs, and use the software to determine which row was selected.

By the way, at some point you will want to see all of the instructions available to you. Visit the ATMEL website and look at the datasheet for the ATmega88 chip. Here are the links:



- 32 page summary: http://www.atmel.com/dyn/resources/prod_documents/8271S.pdf
- 562 page complete: http://www.atmel.com/dyn/resources/prod_documents/doc8271.pdf

- Instruction set manual:
  http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf

Experiment #3

OK, now you are comfortable with programming your chip, and have poked around the source code a bit. After I got to this point, I wanted to see if I could look at the row inputs and figure out which one was being grounded. But how would I display the answer? Write it out to the LCD? I think in Morse code more than RTTY, so I decided to blink the LED with the pin number instead. Here is the code I tried:

```
      ;***************************************
      ;* Experiment #3
      ;* tap PB2-5 to ground and
      ;* watch LED flash the pin#.
      ;***************************************
KEYPAD:
      ldi         temp1,0           ;assume no row input
      sbis        pinB,PB2          ;is PB2 low?
      ldi         temp1,2           ;yes, 2 blinks
      sbis        pinB,PB3          ;is PB3 low?
      ldi         temp1,3           ;yes, 3 blinks
      sbis        pinB,PB4          ;is PB4 low?
      ldi         temp1,4           ;yes, 4 blinks
      sbis        pinB,PB5          ;is PB5 low?
      ldi         temp1,5           ;yes, 5 blinks
      tst         temp1             ;how many blinks?
      breq        kp1               ;if none, we're done
      rcall       Blink_LED         ;do the blinking
kp1:
      ret
```

OK, OK, I am being less than honest here. This is *not exactly* what I tried. My first version of experiment #3 did not check for 'number of blinks = 0'. I ran it and my little LED just blinked away. The problem is that calling blink_led with a value of 0 will result in 256 blinks. Take out the two lines 'tst temp1; breq kp1' for fun and you'll see what I mean.

So how does it work? Notice that most of this code is exactly like experiment #2: Load a value in temp1, then skip the following instruction if the bit is set. I've just extended it for four different pins, and adjusted the number of blinks according to the pin that was brought low.

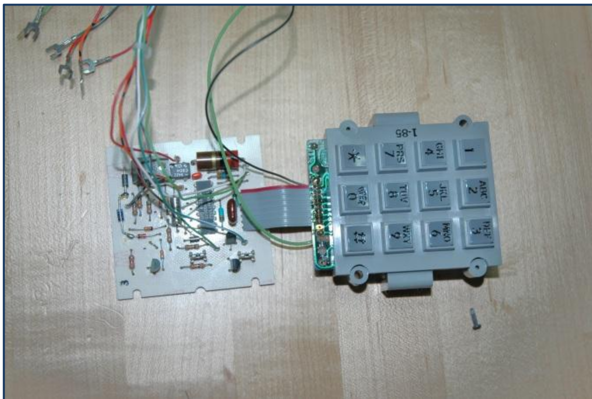
Your code at this point is the foundation of a decent keypad routine. Now you have a good way to decode the rows inputs. All you need now is to take the column pins low, one at a time, and see if any of the rows go low.

That sounds easy, but it was still too much for my brain when I started to type into the code editor. So I went for something simpler. What about taking a single column low, and then looking for an active row? Take a break to build something, and then go on to experiment 4!

Get your keypad ready

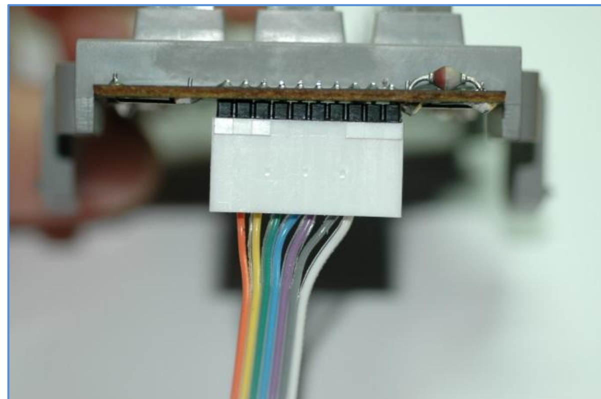
My junkbox keypad was given to me by none other than W8DIZ. He put it in the bag of parts for the FDIM buildathon project he hosted. Since then I've wondered what I'd use it for.

The keypad was attached to a pulse dialer circuit board. I disconnected the board and ribbon cable, solder-wicked the pads, then attached a 10-pin male header to the keypad. The header



allows me to connect the keypad directly to a solderless breadboard for testing, or to a socketed rainbow ribbon cable I used for this project. Did you know that ribbon cable colors are in resistor color-code order?

By the way, this keypad is +12V backlit with translucent keys. Very nice.



Let's see if we can add a column outputs and a pushbutton or two from our keypad. If you haven't found yourself a keypad yet, now is the time.

Experiment #4

```
;***********************************
;* Experiment #4
;*
;* Put a pushbutton
;* switch between PD7 and one of PB2-5.
;*
;* Press the switch and the LED should
;* flash the PortB pin#
;***********************************


KEYPAD:
      cbi         PORTD,PD7         ;take column output low
      rcall       ScanRow           ;see if a row goes low
      sbi         PORTD,PD7         ;restore column output high
      ret


SCANROW:
      ldi         temp1,0           ;assume no row input
      sbis        pinB,PB2          ;is PB2 low?
      ldi         temp1,2           ;yes, 2 blinks
      sbis        pinB,PB3          ;is PB3 low?
      ldi         temp1,3           ;yes, 3 blinks
      sbis        pinB,PB4          ;is PB4 low?
      ldi         temp1,4           ;yes, 4 blinks
      sbis        pinB,PB5          ;is PB5 low?
      ldi         temp1,5           ;yes, 5 blinks
      tst         temp1             ;how many blinks?
      breq        kp1               ;if none, we're done
      rcall       blink_led         ;do the blinking
kp1:
      ret
```

The first thing to notice is that I didn't make many changes from the last experiment. I renamed Keypad to ScanRow, and added a four instruction section to briefly take a column output low. The only difference is that instead of touching a row input to ground, we can now touch to a column output. Add 2 more column outputs to the mix, and we've got ourselves enough connections for a full 3x4 keypad.

My next step was mostly just thinking. How can you go from row and columns to an output value, like 1 through 12? I searched the internet and there are many sorts of keypad algorithms. Even Atmel, creator of the AVR chips, publishes several examples. See http://www.atmel.com/dyn/resources/prod_documents/doc1232.pdf for my favorite.

I could have tried using a published example, but I find that starting small and building my code step-by-step gives me the confidence I need to keep coding. Do you like to build a whole radio, start-to-finish, and then flip the switch? Or do you prefer to build the oscillator, then the mixer,

etc, and test as you go? So I decided to forget the published methods and roll my own. You may decide differently. You have choices here.

1	2	3
4	5	6
7	8	9
*	0	#

Looking at a keypad, you'll notice that columns increment the value by one, and that rows increment the value by 3. For example, in column 1, going from row1 to row2 changes the value from 1 to 4. So a reasonable approach would be to assign a preliminary value of 1-3, based on the column you are currently scanning, and then add 0 for row1, add 3 for row2, add 6 for row3, and add 9 for row4. This will give you an output of 1-12, and we can fix up the zero later. Try programming it this way. There are a few examples online for you to look at.

Oops. There is funny quirk to the AVR instruction set: there is *no* instruction for adding an immediate value. So you can't 'add 3', like I suggested above. What are the workarounds? You could put your value of 3 in another register, and add the registers together. Or, you could subtract -3, the two's complement of 3. Both ways bothered me a bit. Could I find another way?

	2	1	0
3	-1	-2	-3
6	-4	-5	-6
9	-7	-8	-9
12	-10	-11	-12

Here is my oddball solution: instead of looking for a positive 1 to 12 result, let's see if we can get a negative 1 to 12 result. Can we come up with row and column values, when subtracted (not added), give a matrix of negatives values instead? In fact, yes we can. Here is a matrix of negative values, shown in blue, derived by subtracting the row from the column values shown in brown. Now we no longer need to subtract negative numbers, or use an extra register. All we need to do is negate the final answer. There are simpler ways for sure, but this way works and I like it. The code for it was my next experiment.

Experiment #5

```
      ;************************************
      ;* Experiment #5: Add the Keypad!
      ;*
      ;* 7-wire interface:
      ;* Row1 to PB5, Row2 to PB4,
      ;* Row3 to PB3, Row1 to PB2,
      ;* Col0 to PD7, Col1 to PB1,
      ;* Col2 to PB0
      ;*
      ;************************************

KEYPAD:
      cbi         PORTD,PD7         ;take column1 low
      ldi         temp1,2           ;col1 value is 2
```

```
        rcall       ScanRow             ;see if a row went low
        sbi         PORTD,PD7           ;restore column1 high

        cbi         PORTB,PB0           ;take column2 low
        ldi         temp1,1             ;col2 value is 1
        rcall       ScanRow             ;see if a row went low
        sbi         PORTB,PB0           ;restore col2 high

        cbi         PORTB,PB1           ;take column3 low
        ldi         temp1,0             ;col3 value is 0
        rcall       ScanRow             ;see if a row went low
        sbi         PORTB,PB1           ;restore column3 high
        ret


SCANROW:
        clc                             ;clear carry
        sbis        pinB,PB5            ;is row1 low?
        subi        temp1,3             ;yes, subtract 3
        sbis        pinB,PB4            ;is row2 low?
        subi        temp1,6             ;yes, subtract 6
        sbis        pinB,PB3            ;is row3 low?
        subi        temp1,9             ;yes, subtract 9
        sbis        pinB,PB2            ;is row4 low?
        subi        temp1,12            ;yes, subtract 12
        brcc        kp1                 ;no carry = no keypress
        neg         temp1               ;negate answer
        rcall       Blink_LED           ;let's see it!
kp1:    ret
```

Compare this code to Experiment #4. We haven't actually added much at all. The whole keypad decoder is only 26 lines. Like I said before, most of it was just a thinking exercise. In Keypad, we've added lines to set each of the columns low, not just a single line. And we've assigned values of 2,1,0 to the columns and 3,6,9,12 to the rows. Review the last part of this series for why I used those values.

But sheesh, what are all those subi/sbis instructions? The sbis instruction is what we've used all along to check the rows. The new subi instructions "subtract immediate" just subtracts our row-column numbers to get the keypress value, albeit in a negative form. We correct that with the 'neg temp1' instruction near the end.

It is finally time to hook up that keypad. Figure out which lines are which on your keypad, and connect the lines to the row/column pins as indicated in the code comments above. My keypad pinout is (BkLight, C2, R1, C1, R4, C3, R3, R2, nc, GND). Yours is probably different. Hook it up and try it out.

Finally we have something useful attached to our DDS kit. Too bad it just flashes at us! And it doesn't even get zero right. Integrating our keyboard with the DDS is the topic of Experiment #6.

Experiment #5 extras

Did all of that subtraction and negativity bother you? There are other ways to tackle the problem. I tried a few ideas of my own, and found one that I like better than the original. If you are a code junkie, read on. Otherwise you can skip this section!

Idea #1: Just use all positive values, the way they are supposed to be. Set the column values to 1, 2, and 3. We'll get around the lack of an 'add immediate' instruction by loading the value into another register (temp3), and then adding it later. You might start out with something like this:

```
SCANROW:
      sbis        pinB,PB5           ;is row1 low?
      ldi         temp3,0            ;yes, add 0
      sbis        pinB,PB4           ;is row2 low?
      ldi         temp3,3            ;yes, add 3
      sbis        pinB,PB3           ;is row3 low?
      ldi         temp3,6            ;yes, add 6
      sbis        pinB,PB2           ;is row4 low?
      ldi         temp3,9            ;yes, add 9
      add         temp1,temp3        ;add row & columns
      tst         temp1              ;find a row match?
      breq        kp1                ;zero = no match
      rcall       Blink_LED          ;show result
kp1:  ret
```

Too bad it doesn't work! The problem is that the value of temp1 will always be greater than zero (it was set by the columns), so the LED will always be blinking.

Idea #2: Add 3 to the row values, then check to see if the final result was greater than three. If it was, then you know that you've found a row, and you can subtract the excess. This one works, and you can use it if you like:

```
SCANROW:
      ldi         temp3,3
      sbis        pinB,PB5           ;is row1 low?
      add         temp1,temp3        ;yes, add 3 (not 0)

      ldi         temp3,6
      sbis        pinB,PB4           ;is row2 low?
      add         temp1,temp3        ;yes, add 6 (not 3)

      ldi         temp3,9
      sbis        pinB,PB3           ;is row3 low?
      add         temp1,temp3        ;yes, add 9 (not 6)

      ldi         temp3,12
      sbis        pinB,PB2           ;is row4 low?
      add         temp1,temp3        ;yes, add 12 (not 9)

                                     ;valid key values are 4-15
```

```
      cpi         temp1,4           ;is value 4 or more?
      brlo        kp1               ;less than 4 = no active row
      subi        temp1,3           ;remove the excess
      rcall       Blink_LED         ;let's see it!
kp1:  ret
```

Idea #3: Don't use any addition or subtraction tricks, just jump to the 'success' part of the code on any row match. We only do the addition once. Very jumpy, but it works, too:

```
SCANROW:
      ldi         temp3,0           ;set up row1 value (0)
      sbis        pinB,PB5          ;is row1 active?
      rjmp        kp1               ;yes, so do it

      ldi         temp3,3           ;set up row2 value (3)
      sbis        pinB,PB4          ;is row2 active?
      rjmp        kp1               ;yes, so do it

      ldi         temp3,6           ;set up row3 value (6)
      sbis        pinB,PB3          ;is row3 low?
      rjmp        kp1               ;yes, so do it

      ldi         temp3,9           ;set up row4 value (9)
      sbis        pinB,PB2          ;is row4 low?
      rjmp        kp1               ;yes, so do it
      rjmp        kp2               ;got here = no row match

kp1:  add         temp1,temp3       ;success: add column value
      rcall       Blink_LED         ;and show result
kp2:  ret
```

Idea #4: What about making the column values negative, and then adding them to the row if there is a match? If the final result was greater than zero, you found a row match. [Note: to use this routine, you'll need to modify the column values in the keypad routine to col1 = -2, col2 = -1, col3 = 0]

```
SCANROW:
      ldi         temp3,-1          ;assume no row match
      sbis        pinB,PB5          ;is row1 low?
      ldi         temp3,3           ;yes, add 3
      sbis        pinB,PB4          ;is row2 low?
      ldi         temp3,6           ;yes, add 6
      sbis        pinB,PB3          ;is row3 low?
      ldi         temp3,9           ;yes, add 9
      sbis        pinB,PB2          ;is row4 low?
      ldi         temp3,12          ;yes, add 12
      add         temp1,temp3       ;add row & column
      brmi        kp1               ;negative = no match
      rcall       Blink_LED         ;show result
kp1:  ret
```

That's pretty compact. But wait, why subtract three from the columns, then add it right back when we get to the rows?

Idea #5: Based on idea #4, but let's put the column values back to 1, 2, and 3. Adjust the row values accordingly:

```
SCANROW:
      ldi         temp3,-1          ;assume no row match
      sbis        pinB,PB5          ;is row1 low?
      ldi         temp3,0           ;yes, add 0
      sbis        pinB,PB4          ;is row2 low?
      ldi         temp3,3           ;yes, add 3
      sbis        pinB,PB3          ;is row3 low?
      ldi         temp3,6           ;yes, add 6
      sbis        pinB,PB2          ;is row4 low?
      ldi         temp3,9           ;yes, add 9
      tst         temp3             ;find a row match?
      brmi        kp1               ;negative = no match
      add         temp1,temp3       ;add row & column
      rcall       Blink_LED         ;show result
kp1:  ret
```

Now we have a nice routine that uses positive values. Compared to the original it takes an extra line of code, but is easier to read and more intuitive. I think I'll use it! Notice how close it is to idea #1, which didn't work. Sometimes you just have to try out a bunch of ideas and see what happens.

Our keypad decoder is finished, but it doesn't interact with the DDS at all. It's time to start integrating.

Experiment #6

After finishing the keypad decoder, it is time to think again. What small thing can we make our keypad do? I decided that trying to enter frequencies was too hard at this point. But moving the cursor sounded easy. No numbers to worry about. The source code already does it for you, somehow, every time you press down on the encoder knob. So I looked at the main program loop again for the part where it checks for encoder knob presses. Can you find it? It is labeled 'menu 5'. I'll reproduce it here, with added comments:

```
Menu5:
      tst         press       ;are there any knob presses?
      breq        menu9       ;no, so we are done
      dec         press       ;yes, so decrease count of presses
      …                       ;and do what we need to do
```

All we need to do is *emulate* a knob press. And the only thing a knob press does is increment a register labeled 'press'. The main program loop will handle the rest.

```
;******************************************
;* Experiment #6: Emulate encoder knob press.
;
;* KEYPAD CONNECTIONS:
;* Row1 to PB5, Row2 to BP4,
;* Row3 to PB3, Row4 to PB2,
;* Col0 to PD7, Col1 to PB1, Col2 to PB0
;*
;* Moves cursor right when '#' pressed,
;* Otherwise LED flashes the keypad value
;*
;*******************************************


KEYPAD:
      cbi         PORTD,PD7         ;take column1 low
      ldi         temp1,2           ;col1 value is 2
      rcall       ScanRows          ;see if a row went low
      sbi         PORTD,PD7         ;restore column1 high

      cbi         PORTB,PB0         ;take column2 low
      ldi         temp1,1           ;col2 value is 1
      rcall       ScanRows          ;see if a row went low
      sbi         PORTB,PB0         ;restore col2 high

      cbi         PORTB,PB1         ;take column3 low
      ldi         temp1,0           ;col3 value is 0
      rcall       ScanRows          ;see if a row went low
      sbi         PORTB,PB1         ;restore column3 high
      ret


SCANROWS:
      clc                           ;clear carry
      sbis        pinB,PB5          ;is row1 low?
      subi        temp1,3           ;yes, subtract 3
      sbis        pinB,PB4          ;is row2 low?
      subi        temp1,6           ;yes, subtract 6
      sbis        pinB,PB3          ;is row3 low?
      subi        temp1,9           ;yes, subtract 9
      sbis        pinB,PB2          ;is row4 low?
      subi        temp1,12          ;yes, subtract 12
      brcc        kp1               ;no carry = no keypress
      neg         temp1             ;negate answer
      rcall       PadCommand        ;do something
kp1:
      ret


PadCommand:
      cpi         temp1,12          ;special case: "#" = cursor right
      brne        kp2               ;not the # command
      ldi         temp1,1           ;allow 1 blink for debouncing
      inc         press             ;increment the keypress
kp2:
      rcall       Blink_LED         ;blink for everything else.
      ret
```

The only new code is PadCommand at the end. It looks to see if '#' has been pressed. Instead of flashing 12 times, it increments the press register instead. The main program loop will think the knob button has been pressed, and will do what it needs to do to advance the cursor.

Emulating the encoder button saved us a lot of programming time. We didn't have to know how to move the cursor – we let the existing code do that for us. Can we emulate the encoder rotation, too? See for yourself in the next experiment.

Experiment #7

```
;*****************************************
;* Experiment #7: cursor & frequency
;
;* Keypad code '#' = move cursor right
;* Keypad code '*' = flash LED with value
;*                    of current LCD digit
;******************************************

;Keypad & ScanRow routines omitted for brevity.  See Experiment #6 for code.

PadCommand:
      cpi         temp1,11           ;special case: is it "0"?
      brne        kp2                ;no, continue
      ldi         temp1,0            ;yes, replace with real zero

kp2:  cpi         temp1,12           ;special case: "#" command?
      brne        kp3                ;no, try next command
      inc         press              ;emulate button press = cursor right
      ldi         temp1,1            ;1 blink for switch debouncing
      rjmp        kp5                ;done with "#"

kp3:  cpi         temp1,10           ;special case:"*" command
      brne        kp5                ;no, must be a keypad digit
      mov         temp2,StepRate     ;yes, get current cursor position
      ldi         ZH,high(rcve0)     ;point to frequency value in memory
      ldi         ZL,low(rcve0)      ;16 bits, so need two instructions
kp4:  dec         ZL                 ;advance through frequency digits
      dec         temp2              ;and advance through cursor positions
      brpl        kp4                ;until we get to current digit
      ld          temp1,Z            ;load value at cursor

kp5:  tst         temp1              ;get a zero result?
      breq        kp6                ;no flashes for zero
      rcall       Blink_LED          ;still doin' lots of flashin'
kp6:  ret
```

I chickened out. I like to do things in small steps, and I wasn't ready to tackle the whole frequency/numbers thing yet. Maybe by now you are braver than me. I am just showing you my steps, without any substantial editing of the process.

If you are an encoder, you just count up (or down) by one. But a keypad digit can be a jump of 0 to 9 from the value of the current digit. First we must know the value at the current cursor position. The code above will LED flash the value of the LCD digit at the cursor position. Turn the encoder knob to something like 15,000,000 and then press '*'. You should get 5 LED blinks.

The code that determines the digit at the current cursor position is between labels kp3 and kp5. It is a bit complicated, and requires you to know the variable that stores the current cursor position (StepRate) and the variable that points to the displayed frequency (rcve0). You can find similar logic in the source code where the frequencies are incremented and decremented.

Now that we have the current digit, can we emulate encoder rotation that will get us to the keypad's digit? For example, if the current display is 700 Hz and I press a '9' on the keypad, how do I get to 900 Hz? It would take an encoder movement of +2. The answer is just the *difference* of the requested digit (keypad) and the current digit. First, remove the '*' command by changing 'brne kp5' to 'breq kp6'. Substitute the following code, and your LED will flash the difference between the current digit on the LCD and what you pressed.

```
;above this level, change brne kp5 to breq kp6
kp4:  dec         ZL                  ;advance through frequency digits
      dec         temp2               ;and advance through cursor positions
      brpl        kp4                 ;until we get to current digit
      ld          temp3,Z             ;load value at cursor
      sub         temp1,temp3         ;subtract from keypad digit
      brpl        kp5                 ;convert to absolute value
      neg         temp1               ;was negative result, so negate it.
kp5:
```

Wow, now we are flashing the value we want to encoder to move. But what is the name of the register in the source code that handles encoder requests? Not surprisingly, is called 'encoder'. Now, try this amazingly simple change:

```
kp4:  dec         ZL                  ;advance through frequency digits
      dec         temp2               ;and advance through cursor positions
      brpl        kp4                 ;until we get to current digit
      ld          temp3,Z             ;load value at cursor
      sub         temp1,temp3         ;subtract from keypad digit
      mov         encoder,temp1       ;set up difference for encoder routines.
      ret                             ;done! Keep kp5 & kp6 below for now.
```

Press a digit, and the DDS frequency changes! We are nearly done. In the next and final part, I will add a few things to make our keypad easier to use.

Full Code:

```
;*****************************************
;* W8BH - KEYPAD INSERTION SEQUENCE
;*
;* Adds a 3x4 keypad interface to the
;* DDS Development kit by W8DIZ
;*
;* KEYPAD CONNECTIONS (7 wires)--
;* Row1 to PB5, Row2 to BP4,
;* Row3 to PB3, Row4 to PB2,
;* Col0 to PD7, Col1 to PB1, Col2 to PB0
;*
;* Functions
;* # = cursor right
;* * not implemented, consider cursor left.
;*
;******************************************

KEYPAD:
      cbi         PORTD,PD7         ;take column1 low
      ldi         temp1,2           ;col1 value is 2
      rcall       ScanRows          ;see if a row went low
      sbi         PORTD,PD7         ;restore column1 high

      cbi         PORTB,PB0         ;take column2 low
      ldi         temp1,1           ;col2 value is 1
      rcall       ScanRows          ;see if a row went low
      sbi         PORTB,PB0         ;restore col2 high

      cbi         PORTB,PB1         ;take column3 low
      ldi         temp1,0           ;col3 value is 0
      rcall       ScanRows          ;see if a row went low
      sbi         PORTB,PB1         ;restore column3 high
      ret


SCANROWS:
      clc                           ;clear carry
      sbis        pinB,PB5          ;is row1 low?
      subi        temp1,3           ;yes, subtract 3
      sbis        pinB,PB4          ;is row2 low?
      subi        temp1,6           ;yes, subtract 6
      sbis        pinB,PB3          ;is row3 low?
      subi        temp1,9           ;yes, subtract 9
      sbis        pinB,PB2          ;is row4 low?
      subi        temp1,12          ;yes, subtract 12
      brcc        kp1               ;no carry = no keypress
      neg         temp1             ;negate answer
      rcall       PadCommand        ;do something
kp1:  ret
```

```
PADCOMMAND:
      cpi         temp1,11          ;special case: is it “0”?
      brne        kp2               ;no, continue
      ldi         temp1,0           ;yes, replace with real zero

kp2:  cpi         temp1,12          ;special case: "#" command?
      brne        kp3               ;no, try next command
      inc         press             ;emulate button press = cursor right
      rjmp        kp5               ;done with “#”

kp3:  cpi         temp1,10          ;special case:"*" command
      breq        kp5               ;yes, ignore it for now.
      mov         temp2,StepRate    ;no, get current cursor position
      ldi         ZH,high(rcve0)    ;point to frequency value in memory
      ldi         ZL,low(rcve0)     ;16 bits, so need two instructions
kp4:  dec         ZL                ;advance through frequency digits
      dec         temp2             ;and advance through cursor positions
      brpl        kp4               ;until we get to current digit
      ld          temp3,Z           ;load value at cursor
      sub         temp1,temp3       ;subtract from keypad digit
      mov         encoder,temp1     ;set up difference for encoder routines.
      inc         press             ;advance cursor position

kp5:  ldi         delay,150         ;simple key debouncer
      rcall       wait              ;give the LED a rest!
      ret
```

Now the cursor advances after each entered digit. Unfortunately, it advances too quickly! A few more changes are needed to improve reliability. Since updating each digit takes time, we must suspend keypad scanning until all pending encoder requests are met. A few inserted lines in the main menu will do the trick:

```
menu9:
 tst encoder      ;!!any pending encoder requests?
 brne menu        ;!!if so, ignore keypad
 rcall keypad     ;!!check keypad
 rjmp menu
```

- and -

```
menu5:
 tst press
 breq menu9
 tst encoder      ;!!check for pending encoder requests
 brne menu9       ;!!dont advance cursor until encoder done
 dec press
```

Final Thoughts

I hope you had some fun learning how to add a keypad to your DDS. You probably have thought of some improvements or new things to try. For example, I might turn the '*' into a cursor-left key. If you have a 4x4 keypad, you can add another column line to your code and have additional function keys to play with.

It takes some time for the DDS to 'catch up' when you enter a digit on the keypad. I'd like to be able to enter the numbers faster. For starters, you can try shortening some of the key-debounce wait times. Or you can abandon the idea of emulating the encoder, and write your own direct keypad-to-DDS interface. Have fun!