Add VFO Memories

to your DDS Development Kit

By Bruce Hall, W8BH

The DDS kit by W8DIZ is a tinkerer's delight. In my first project, I added a keypad. The whole process is described at http://w8bh.net/avr/AddKeypadFull.pdf. The keypad is great for entering frequencies directly, but it would be even better if we could have some memories. I wanted a way to quickly change bands, and go directly to the qrp calling frequencies. The following pages will describe how I added my VFO frequency memories. You can access the memories with or without a keypad.

I thought it would be neat if I could turn the encoder shaft and scroll through a list of memories, and then select the frequency I want by pressing the button. The encoder currently has only one function: changing the frequency. Can we add more functions? There are plenty of examples in consumer gear where a single input device has multiple uses.

To give the encoder multiple modes of operation, we need to create a variable 'mode'. Then, wherever we code for encoder action, check the mode variable and act according to the current mode. For example, in normal mode we change the frequency, but in a second mode we scroll through preset frequencies instead. We can create as many modes of operation as we want, coding the encoder behavior to whatever the mode requires.

In the source code, handling encoder rotation is an integral part of the main program loop. Changing modes and encoder behavior would mean a lot of changes to the existing code, and would be hard to support if and when the source code is updated. For me, the easiest approach was to keep the existing code intact, and create a new main program loop. Reverting to the original code would only require commenting out a single program line:

```
menu:  ;main program
       rjmp   W8BH                       ;!! go to new main program
```

In the beginning of the main program loop I add a single line that jumps to the new code, including some additional initialization. With this line in place, the original program loop is bypassed and never executed. Put a semicolon in front of the rjmp instruction above, and all of my inserted code will be bypassed instead. Neat and simple!

In the keypad article, I started from some simple routines, like blinking the LED, and built larger and more complex routines in a step-wise fashion. In computer class this is called ‘bottom-up’ programming, and is often frowned upon. Personally, I prefer it: I learn better by starting small. But when rewriting a main program loop, it makes more sense to think in a bigger, top-down approach, and then fill in the details later. Doing so keeps our loop simpler to code and simpler to read. What does the DDS program do? It checks for encoder action, then checks for button action, and repeats forever. So my new encoder routine is really simple:

```
MAIN:
       rcall  CheckEncoder                ;check for encoder action
       rcall  CheckButton                 ;check for button taps
       rcall  CheckHold                   ;check for button holds
       rcall  Keypad                      ;check for keypad action
       rjmp   Main                        ;loop forever
```

The only things I added are checks for keypad and button-hold activity. This new routine works for the original code, and for any additional functions we may assign to the encoder and/or button. I added a button-hold as a way to change modes. Hold the button down, and we change from normal mode to our new ‘scrolling presets’ mode. But how do we code for a button-hold? In top-down thinking, we’ll make a place for it now and worry about the details later.

Dealing with multiple modes

As I mentioned before, each mode will have an associated behavior. In mode 0, the original mode, the encoder increases/decreases the frequency and the button press advances the cursor. So our CheckEncoder routine will need to check the mode, and branch to the appropriate routines. The pseudo-code for this routine looks something like this:

a. Bypass this routine if no encoder requests pending
b. If we’re in mode 0, do the original encoder routine
c. If we’re in mode 1, scroll through the presets
d. If we’re in modes 2+, create a space for that behavior

The actual code follows this form almost exactly

```
CHECKENCODER:
       tst    encoder                     ;any encoder requests?
       breq   ce2                         ;no, so quit
       cpi    mode,0                      ;are we in normal mode (0)?
       brne   ce1                         ;no, skip
       rcall  EncoderMode0                ;yes, handle it
ce1:   cpi    mode,1                      ;are we in mode 1 = presets?
       brne   ce2                         ;no, skip
       rcall  EncoderMode1                ;yes, handle it
ce2:   ret
```

Each line in the pseudo-code equates to 3 lines of code - the cpi, brne, and rcall instructions. Future modes can be added at the end of the routine. The CheckButton and CheckHold routines follow the exact same sequence.

Mode 0 Routines

In mode 0, the encoder behaves normally, and the code is an almost exact copy of the original source code. In EncodeMode0 routine we handle encoder behavior, and in ButtonMode0 we handle the button behavior. I won't go into all the details, but I have added comments in the code. A new routine, HoldMode0, is called when the button is held down. I wanted a button-hold to change the modes, so this is the place to handle it:

```
HOLDMODE0:
;      Called when button has been held down for about 1.6 seconds.
;      In mode 0, action should be to invoke mode1 = scrolling freq. presets

       ldi    mode,1
       rcall  ChangeMode                  ;go to scrolling preset mode
       rcall  CurrentPreset               ;return to most-recent preset
       ret
```

There isn't much to it. In top-down approach, we list what we want to happen and code it later. The call to CurrentPreset ensures that any time we change modes that we return to the last-used preset. For example, I might choose the 7.030 MHz preset, then move up & down the band a bit. When I return to presets, this call brings me back to 7.030.

Mode 1 Routines

It's time to make our encoder do something different. In mode1, the encoder will scroll through our frequency presets. Here is the code:

```
ENCODERMODE1:
       tst    encoder                     ;which way did encoder turn?
       brmi   e11
       rcall  CyclePresetUp               ;CW = increase freq
       rjmp   e12
e11:   rcall  CyclePresetDown             ;CCW = decrease freq
e12:   clr    encoder                     ;ignore any more requests
       ret
```

We check the encoder variable to see which way the encoder turned. Two 'clicks' of the encoder to the left (CCW) results in an encoder value of -2, and two 'clicks' to the right (clockwise) result in a +2. The branch instruction *brmi* ‘branch on minus’ distinguishes the positive from negative encoder values. Then we call the main actions, CyclePresetUp or CyclePresetDown. After taking the action we are done, so the variable is cleared.

When we are scrolling, what should a button press do? I decided that it should cancel the scrolling and return us to normal mode. You may decide to do something else. The code for this is ButtonMode1.

Button Hold routines

I’ve run out of top-down programming. At some point we have to code what our routines say they are going to do! It is time to start the trickier stuff.

We need a way to determine if the button is being held down longer than a simple press-and-release. This means that we need a way of measuring the amount of time that the button is in the pushed-down state. There are several ways to do this, but using a hardware timer seemed like a good choice. The ATmega88 has three built-in timers. The source code already uses one of these timers. Although it’s possible to use the same timer for more than one function, I decided to use another, unused timer instead.

The three timers are timer0, timer1, and timer2. Both timer0 and timer2 are 8-bit timers, which means that they can measure time in 256 increments. Timer1 is 16-bit, and can count up to 65535 time-increments. Timer1 is also more complex and versatile. Since 8-bit resolution was good enough for a button-hold, and timer0 was already in use, I chose timer2.

At a clock rate of 20.48 MHz, how can a count of 256 possibly be enough time to measure a second or more? The answer is to prescale the clock to a slower and more useful frequency. I divided the clock by 1024, which gives a period of 1024/20.48 = 50 microseconds. Notice that Diz chose the crystal frequency to give us a nice, even number. Now, every cycle of 256 is completed in 245*50 = 12.8 milliseconds, a more reasonable unit of time. Timer2 is programmed almost exactly like Timer0, except for a longer cycle.

I configured timer2 to interrupt the program every 13 millisecond cycle, so that we can check the status of the button. If the button is down, increment a counter. If the button is released, restart the counter. Here is the code of our new interrupt routine:

```
;*********************************************************
;*  W8BH - Timer 2 Overflow Interrupt Handler
;*********************************************************
;      This handler is called every 12.8 ms @ 20.48MHz clock
;      Increments HOLD counter (max 128) when button held
;      Resets HOLD counter if button released

OVF2:
       push   temp1
       in     temp1,SREG                 ;save status register
       push   temp1
```

```
        tst     hold                    ;counter at max yet?
        brmi    ov1                     ;dont count above maxcount (128)
        sbic    pinD,PD3
        clr     hold                    ;if button is up, then clear
        sbis    pinD,PD3
        inc     hold                    ;if button is down, then count
ov1:    pop     temp1
        out     SREG,temp1              ;restore status register
        pop     temp1
        reti
```

To enable the interrupt routine, we need to add its address to the interrupt table:

```
.org OVF2addr
        jmp     OVF2                    ; Timer/Counter2 overflow
```

And we need to configure the interrupt's control registers:

```
        ldi           temp1, $07        ;set timer2 prescaler to clk/1024
        sts           TCCR2B,temp1
        ldi           temp1, $01        ;enable TIMER2 overflow interrupt
        sts           TIMSK2,temp1
```

Now we are able to measure button-hold times in terms of seconds. When the hold counter reaches 128, we've been holding the button down for 128*12.8 = 1.6 seconds. I thought this was a reasonable hold time. You can easily decrease it by increasing the timer2 variable from 0 to a higher value at the start of each new cycle.

The Push/In/Push sequence at the start of the routine is needed to preserve the status register. Why? Because our interrupt routine can be called at any time during program execution, and we don't want our interrupt routine to unexpectedly change the register in the middle of some routine that uses it.

I've assigned the hold counter to variable R0. Variables below R16 cannot use certain common instructions, and therefore are somewhat less useful. R0 seemed like a good choice, since we do not need HOLD to do much more than count. You could have used an upper register instead, but it is better to save them for more complex operations. A third option would be to use an SRAM memory location. It is the programmer's choice.

We check the status of our button pin using the sbis/sbic instructions, and update our hold counter accordingly. Notice that once we reach a count of 128, the counter is 'stuck' there, waiting for our software to recognize the hold condition. Viola! We have button-hold. We can now code the button hold routine that we've put off.

```
CHECKHOLD:
        tst     hold                    ;is hold a positive/zero value?
        brpl    ch2                     ;yes, not a hold yet
        clr     hold                    ;its a hold.  Reset counter.
```

```
        cpi    mode,0                    ;normal mode (0)?
        brne   ch1                       ;no, skip
        rcall  HoldMode0                 ;yes, handle button
ch1:    cpi    mode,1                    ;presets mode (1)?
        brne   ch2                       ;no, skip
        rcall  HoldMode1                 ;yes, handle button
ch2:    ret
```

Notice the use of the branch-if-plus *brpl* instruction to check for the hold. I set the hold counter to max out at 128. Any count up to 127 is positive, but in signed-binary the next-incremented value is -128. I used this technique because the hold register (R0) cannot use the compare-immediate instruction. Another method would be to move the value into a temporary register, like temp1, and then do the compare. For example: mov temp1, hold; cpi temp1,150; brlo ch2.

Memory routines

I entered my memory presets into a table at the end of the source code like this:

```
.EQU NumPresets = 9                      ;Enter # of presets here

presets:                                 ;One line for each preset freq
.db 0,3,5,6,0,0,0,0                      ;80M qrp calling =  3.560 MHz
.db 0,7,0,3,0,0,0,0                      ;40M qrp calling =  7.030 MHz
.db 1,0,0,0,0,0,0,0                      ;--- --- --- WWV = 10.000 MHz
.db 1,0,1,0,6,0,0,0                      ;30M qrp calling = 10.106 MHz
.db 1,4,0,6,0,0,0,0                      ;20M qrp calling = 14.060 MHz
.db 1,8,0,9,6,0,0,0                      ;17M qrp calling = 18.096 MHz
.db 2,1,0,6,0,0,0,0                      ;15M qrp calling = 21.060 MHz
.db 2,4,9,0,6,0,0,0                      ;12M qrp calling = 24.906 MHz
.db 2,8,0,6,0,0,0,0                      ;10M qrp calling = 28.060 MHz
```

Each memory entry is 8 bytes, and each byte corresponds to a digit of the desired frequency. Now we need a way of converting these digits into the actual frequency. To do so requires knowing something about the DDS chip. To get a desired frequency output, you cannot just tell it the frequency you want; you must send it a 28-bit value which specifies the frequency in units of 100 MHz/(2^^28) = 0.373 Hz. For example, to get 10 MHz out, you need to supply a value of 26,843,545! In addition, there need to be a few added control bits. I found it complicated and confusing, honestly. Hocus pocus. And so, for a lack of a better term, I call this required value our 'magic number'. Sorry if this seems immature, but it made it easy for me to remember. Send the magic number to the DDS chip, and you get your frequency output.

The tricky part, of course, is being able to generate the magic number. The original source code handles it in a really slick way: the magic numbers for 1 Hz, 10Hz, 100Hz, …, 10 MHz are all stored in a table. If the encoder moves the displayed frequency up 100 Hz, then it also grabs the 100 Hz value and adds it to our magic number. No need for complicated formulas.

After studying the code, I realized that it would not be hard to generate the magic number for any frequency. All we need to do is look at each digit in the desired frequency, and add up the corresponding magic components. For example, the magic number for 20 MHz would be twice the value for 10 MHz. The table value for 10 MHz is $33333333, so the 20 MHz magic number is $66666666. To get 11 MHz, add the 10 MHz ($33333333) and 1 MHz ($051EB852) values to get the magic number of $3851EB85.

```
BuildMagic:
       push   StepRate                    ;save StepRate
       ldi    XH,high(LCDrcve0)           ;point to LCD digits
       ldi    XL,low(LCDrcve0)            ;16bit pointer
       ldi    StepRate,7                  ;Start with 10MHz position
bm1:   ld     temp3,X+                    ;get next LCD digit
       tst    temp3                       ;is it zero?
       breq   bm3                         ;yes, so go to next digit
bm2:   rcall  AddMagic                    ;no, so add magic component
       dec    temp3                       ;all done with this component
       brne   bm2                         ;no, add some more
bm3:   dec    StepRate                    ;all done with freq. positions?
       brne   bm1                         ;no, go to next (lowest) position
       pop    StepRate                    ;restore StepRate
       ret
```

The routine above looks at the current frequency digits, which are pointed to by LCDrcve0. At each digit, starting at the 10 MHz position, add the corresponding magic number. The digit is loaded into temp3, which is used to count the number of magic units added. For example, if the frequency is 25 MHz and we are on the first digit, then ‘2’ gets loaded into temp3 and we will add the 10 MHz component ($33333333) twice. When this digit is done (at bm3), we go to the next digit ‘5’ and do the same thing. And so on, until all of the digits in the displayed frequency are done.

I added a small routine, ShowMagic, to verify that BuildMagic worked. It displays the magic number on the top line of the LCD. I kept the code in case I needed it later.

Once we have the magic number, a call to FREQ_OUT will update the DDS with the corresponding frequency. All we need is a way to move a memory frequency to the display buffer. It is a simple copy operation. But since the source and destination are in memory rather than registers, we need to point to them with 16-bit pointers. Here is the code:

```
LoadPreset:
       ldi    ZH,high(freqLCD*2)          ;point to the preset table (-8 bytes)
       ldi    ZL,low(freqLCD*2)           ;16bit pointer
lp1:   adiw   ZL,8                        ;point to next frequency preset
       dec    temp1                       ;get to the right preset yet?
       brne   lp1                         ;no, keep looking
       ldi    YH,high(LCDrcve0)           ;yes, point to LCD digits
       ldi    YL,low(LCDrcve0)            ;16bit pointer
       ldi    temp2,8                     ;there are 8 frequency digits
lp2:   lpm    temp1,Z+                    ;get an LCD digit from FLASH mem
       st     Y+,temp1                    ;and put into LCD display buffer
```

```
        dec     temp2                      ;all digits done?
        brne    lp2                        ;not yet
        ret
```

The memory values are stored at the end of the program. And because a quirk in the AVR assembler, which counts program lines differently than data, we must multiply the source address by 2. We also need to use the lpm 'load from program memory' instruction instead of the regular load instruction.

Now, with all of these pieces in place, using a VFO memory is easy: just put the memory number into temp1 and call the following routine:

```
GetPreset:
        rcall   LoadPreset                 ;get the preset in LCD buffer
        ldi     StepRate,3                 ;put cursor on KHz value
        rcall   ShowFreq                   ;show preset on LCD
        rcall   ZeroMagic                  ;clear out old magic number
        rcall   BuildMagic                 ;build new one based on current freq
        rcall   Freq_Out                   ;send new magic to DDS
        ;rcall  ShowMagic                  ;show magic# on line 1 (debugging)
        ret
```

Bits and Pieces

There isn't much more to it. The new encoder behavior is moving up or down the list of memories, so we can just cycle through them with each encoder update:

```
CyclePresetUp:
        ldi     ZH,high(prset)             ;point to current preset
        ldi     ZL,low(prset)              ;16bit pointer
        ld      temp1,Z                    ;get current preset
        cpi     temp1,NumPresets           ;end of list?
        brne    cp1                        ;no, so can save
        ldi     temp1,0                    ;yes, cycle back to start
cp1:    inc     temp1
        st      Z,temp1                    ;save preset
        rcall   GetPreset                  ;load & display preset
        ret
```

The key instruction is the increment instruction at cp1. The routine also checks to see if we've reached the top of the list, and to cycle back to zero if we're at the top. Notice that I've put the preset variable in SRAM. I could have used a lower register instead. Either way is acceptable. I used SRAM since this variable is only used for storing a value. The upper registers should probably be saved for variables that need more complex operations.

Source Code

```
.cseg
.org $000
       jmp    RESET
.org INT0addr
       jmp    EXT_INT0                   ; External Interrupt Request 0
.org INT1addr
       jmp    EXT_INT1                   ; External Interrupt Request 1
.org OVF0addr
       jmp    OVF0                       ; Timer/Counter0 Overflow
.org OVF2addr
       jmp    OVF2                       ; Timer/Counter2 overflow
.org INT_VECTORS_SIZE


menu:  ;main program
       rjmp   W8BH                       ;!! go to new main program


;*********************************************************
;* W8BH - INITIALIZATION CODE
;*********************************************************
W8BH:
       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

       rcall  InitPreset                 ;frequency presets
       ldi    mode,0                     ;start mode0 = normal operation

       ldi    temp1, $07                 ;set timer2 prescaler to clk/1024
       sts    TCCR2B,temp1
       ldi    temp1, $01                 ;enable TIMER2 overflow interrupt
       sts    TIMSK2,temp1


;*********************************************************
;*  W8BH - REVISED MAIN PROGRAM LOOP
;*********************************************************

MAIN:
       rcall  CheckEncoder               ;check for encoder action
       rcall  CheckButton                ;check for button taps
       rcall  CheckHold                  ;check for button holds
       rcall  Keypad                     ;check for keypad action
       rjmp   Main                       ;loop forever
```

```
CHECKENCODER:
       tst    encoder                    ;any encoder requests?
       breq   ce2                        ;no, so quit
       cpi    mode,0                     ;are we in normal mode (0)?
       brne   ce1                        ;no, skip
       rcall  EncoderMode0               ;yes, handle it
ce1:   cpi    mode,1                     ;are we in mode 1 = presets?
       brne   ce2                        ;no, skip
       rcall  EncoderMode1               ;yes, handle it
ce2:   ret


CHECKBUTTON:
       tst    press                      ;any button requests?
       breq   cb2                        ;no, so quit
       cpi    mode,0                     ;normal mode (0)?
       brne   cb1                        ;no, skip
       rcall  ButtonMode0                ;yes, handle button
cb1:   cpi    mode,1                     ;presets mode (1)?
       brne   cb2                        ;no, skip
       rcall  ButtonMode1                ;yes, handle button
cb2:   ret


CHECKHOLD:
       tst    hold                       ;is hold a positive/zero value?
       brpl   ch2                        ;yes, not a hold yet
       clr    hold                       ;its a hold.  Reset counter.
       cpi    mode,0                     ;normal mode (0)?
       brne   ch1                        ;no, skip
       rcall  HoldMode0                  ;yes, handle button
ch1:   cpi    mode,1                     ;presets mode (1)?
       brne   ch2                        ;no, skip
       rcall  HoldMode1                  ;yes, handle button
ch2:   ret


;*****************************************************
;*  W8BH - MODE 0 (NORMAL MODE) ROUTINES
;*****************************************************


ENCODERMODE0:
;      This code taken from original program loop.
;      Called when there is a non-zero value for encoder variable.
;      Negative encoder values = encoder has turned CCW
;      Positive encoder values = encoder has turned CW
;      In mode 0, encoder should increase/decrease the DDS freq

       tst    encoder
       brpl   e02                        ;which way did encoder rotate?
       inc    encoder                    ;remove 1 negative rotation
       rcall  DecFreq0                   ;reduce displayed frequency
       cpi    temp1,55                   ;55 = all OK
       brne   e01
```

```
       rcall  IncFreq0                    ;correct freq. underflow
       rjmp   e05
e01:   rcall  DecFreq9                    ;reduce magic number
       rjmp   e04

e02:   dec    encoder                     ;remove 1 positive rotation
       rcall  IncFreq0                    ;increase displayed frequency
       cpi    temp1,55                    ;55 = all OK
       brne   e03
       rcall  DecFreq0                    ;correct freq. overflow
       rjmp   e05
e03:   rcall  IncFreq9                    ;increase magic number

e04:   rcall  FREQ_OUT                    ;update the DDS
       rcall  ShowFreq                    ;display new frequency
e05:   rcall  QuickBlink
       ret


BUTTONMODE0:
;      This code taken from original program loop.
;      Called when there is a non-zero value for press variable.
;      Non-zero value = number of times button has been pressed
;      In mode 0, button should advance cursor to the right

       tst    encoder                     ;check for pending encoder requests
       brne   b01                         ;dont advance cursor until encoder done
       dec    press                       ;reduce button press count
       dec    StepRate                    ;advance cursor position variable
       brpl   b01                         ;position >= 0 (Hz position)
       ldi    StepRate,7                  ;no, so go back to 10MHz position
b01:   rcall  ShowCursor
       rcall  QuickBlink                  ;flash the LED
       ret

HOLDMODE0:
;      Called when button has been held down for about 1.6 seconds.
;      In mode 0, action should be to invoke mode1 = scrolling freq. presets

       ldi    mode,1
       rcall  ChangeMode                  ;go to scrolling preset mode
       rcall  CurrentPreset               ;return to most-recent preset
       ret


;**********************************************************
;*  W8BH - MODE 1 (SCROLL FREQUENCY PRESET) ROUTINES
;**********************************************************


ENCODERMODE1:
       tst    encoder                     ;which way did encoder turn?
       brmi   e11
       rcall  CyclePresetUp               ;CW = increase freq
       rjmp   e12
```

```
e11:   rcall  CyclePresetDown            ;CCW = decrease freq
e12:   clr    encoder                    ;ignore any more requests
       ret


BUTTONMODE1:
       clr    press                      ;ignore any more requests
       ldi    mode,0
       rcall  ChangeMode                 ;go to mode 0 = normal op.
       ret

HOLDMODE1:
       ret                               ;dont do anything special


CHANGEMODE:
;      call this routine when mode changes
;      only action is to change the message on Line 1

       cpi    mode,0                     ;mode 0?
       brne   cm1                        ;no, skip
       rcall  DisplayMsg1                ;yes, show normal message
cm1:   cpi    mode,1                     ;mode 1?
       brne   cm2                        ;no, skip
       rcall  DisplayMsg2                ;yes, show 'Scroll Presets'
cm2:   ret


QUICKBLINK:
       cbi    PORTC,LED                  ;turn  LED on
       ldi    delay,15                   ;keep on 20 ms
       rcall  wait
       sbi    PORTC,LED                  ;turn LED off
       ret


;**********************************************************
;*   W8BH - KEYPAD ROUTINES
;*
;*   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
;*   * = frequency presets.
;**********************************************************

KEYPAD:
       tst    encoder                    ;is encoder busy?
       brne   kp0                        ;wait for encoder to finish
       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
kp0:    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
        ldi     temp1,1                 ;1 blink for switch debouncing
        rjmp    kp6                     ;done with '#'

kp3:    cpi     temp1,10                ;special case:"*" command
        brne    kp4                     ;no, try next command
        rcall   CyclePresetUp           ;yes, get next preset
        rjmp     kp6                    ;done with '*'

kp4:    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
kp5:    dec     ZL                      ;advance through frequency digits
        dec     temp2                   ;and advance through cursor positions
        brpl    kp5                     ;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

kp6:    ldi     delay,150                   ;simple key debouncer
        rcall   wait                        ;give the LED a rest!
        ret


;*******************************************************
;*  W8BH - FREQUENCY PRESET ROUTINES
;*******************************************************

ZeroMagic:
        ldi     ZH,high(rcve0)              ;point to magic#
        ldi     ZL,low(rcve0)
        ldi     temp1,0
        st      Z+,temp1                    ;zero first byte (MSB)
        st      Z+,temp1                    ;zero second byte
        st      Z+,temp1                    ;zero third byte
        st      Z+,temp1                    ;zero fourth byte (LSB)
        ret

ShowMagic:
        ldi     ZH,high(rcve0)              ;point to magic number
        ldi     ZL,low(rcve0)               ;2 byte pointer
        ldi     temp3,4                     ;counter for 4 byte display
        ldi     temp1,$80                   ;display on line1
        rcall   LCDCMD
sh1:    ld      temp1,Z+                    ;point to byte to display
        rcall   SHOWHEX                     ;display first nibble
        ldi     temp1,' '                   ;add a space
        rcall   LCDCHR                      ;display the space
        dec     temp3                       ;all 4 bytes displayed yet?
        brne    sh1                         ;no, so do the rest
        ret

AddMagic:
;       adds one component to magic according to StepRate
;       0 = Hz rate, 3=Khz rate, 6=MHz rate, 7=10MHz rate
        Rcall   IncFreq9                    ;already coded!
        ret

BuildMagic:
        push    StepRate                    ;save StepRate
        ldi     XH,high(LCDrcve0)           ;point to LCD digits
        ldi     XL,low(LCDrcve0)            ;16bit pointer
        ldi     StepRate,7                  ;Start with 10MHz position
bm1:    ld      temp3,X+                    ;get next LCD digit
        tst     temp3                       ;is it zero?
        breq    bm3                         ;yes, so go to next digit
bm2:    rcall   AddMagic                    ;no, so add magic component
        dec     temp3                       ;all done with this component
        brne    bm2                         ;no, add some more
bm3:    dec     StepRate                    ;all done with freq. positions?
        brne    bm1                         ;no, go to next (lowest) position
        pop     StepRate                    ;restore StepRate
```

```
        ret

LoadPreset:
        ldi     ZH,high(freqLCD*2)          ;point to the preset table (-8 bytes)
        ldi     ZL,low(freqLCD*2)           ;16bit pointer
lp1:    adiw    ZL,8                        ;point to next frequency preset
        dec     temp1                       ;get to the right preset yet?
        brne    lp1                         ;no, keep looking
        ldi     YH,high(LCDrcve0)           ;yes, point to LCD digits
        ldi     YL,low(LCDrcve0)            ;16bit pointer
        ldi     temp2,8                     ;there are 8 frequency digits
lp2:    lpm     temp1,Z+                    ;get an LCD digit from FLASH mem
        st      Y+,temp1                    ;and put into LCD display buffer
        dec     temp2                       ;all digits done?
        brne    lp2                         ;not yet
        ret

GetPreset:
        rcall   LoadPreset                  ;get the preset in LCD buffer
        ldi     StepRate,3                  ;put cursor on KHz value
        rcall   ShowFreq                    ;show preset on LCD
        rcall   ZeroMagic                   ;clear out old magic number
        rcall   BuildMagic                  ;build new one based on current freq
        rcall   Freq_Out                    ;send new magic to DDS
        ;rcall ShowMagic                    ;show magic# on line 1 (debugging)
        ret

InitPreset:
        ldi     zh,high(prset)              ;point to freq. preset counter
        ldi     ZL,low(prset)
        ldi     temp1,1                     ;start with first preset
        st      Z,temp1                     ;initialize counter
        ret

CurrentPreset:
        ldi     ZH,high(prset)              ;point to current preset
        ldi     ZL,low(prset)               ;16bit pointer
        ld      temp1,Z                     ;get current preset
        rcall   GetPreset                   ;load & display preset
        ret

CyclePresetUp:
        ldi     ZH,high(prset)              ;point to current preset
        ldi     ZL,low(prset)               ;16bit pointer
        ld      temp1,Z                     ;get current preset
        cpi     temp1,NumPresets            ;end of list?
        brne    cp1                         ;no, so can save
        ldi     temp1,0                     ;yes, cycle back to start
cp1:    inc     temp1
        st      Z,temp1                     ;save preset
        rcall   GetPreset                   ;load & display preset
        ret

CyclePresetDown:
        ldi     ZH,high(prset)              ;point to current preset
```

```
       ldi    ZL,low(prset)              ;16bit pointer
       ld     temp1,Z                    ;get current preset
       dec    temp1                      ;point to prior preset
       brne   cd1                        ;not zero = OK
;comment out one of the next two lines, depending on action you want
       ldi    temp1,1                    ;stop at bottom of list
;      ldi    temp1,NumPresets           ;cycle back to top of list
cd1:   st     Z,temp1                    ;save preset
       rcall  GetPreset                  ;load & display preset
       ret



;**********************************************************
;*  W8BH - Timer 2 Overflow Interrupt Handler
;**********************************************************
;      This handler is called every 12.8 ms @ 20.48MHz clock
;      Increments HOLD counter (max 128) when button held
;      Resets HOLD counter if button released

OVF2:
       push   temp1
       in     temp1,SREG                 ;save status register
       push   temp1
       tst    hold                       ;counter at max yet?
       brmi   ov1                        ;dont count above maxcount (128)
       sbic   pinD,PD3
       clr    hold                       ;if button is up, then clear
       sbis   pinD,PD3
       inc    hold                       ;if button is down, then count
ov1:   pop    temp1
       out    SREG,temp1                 ;restore status register
       pop    temp1
       reti

;**********************************************************
;*  W8BH - Message Display routines
;**********************************************************

DISPLAYMSG1:
       ldi    ZH,high(2*msg1)
       ldi    ZL,low(2*msg1)
       rcall  DisplayMsg
       ret

DISPLAYMSG2:
       ldi    ZH,high(2*msg2)
       ldi    ZL,low(2*msg2)
       rcall  DisplayMsg
       ret

DISPLAYMSG:
;      displays a null-terminated message on line 1
;      call with pointer to message in Z
```

```
        ldi    temp1,$80                  ;use line 1
        rcall  LCDCMD
        rcall  DISPLAY_LINE               ;display the message
        ldi    StepRate,3                 ;put cursor at KHz posn
        rcall  ShowCursor
        ret

;**********************************************************
;*  W8BH - END OF INSERTED CODE
;**********************************************************


FreqLCD: .db 1,0,0,0,0,0,0,0,0 ;LCD for 10,000,000 Hz

;**********************************************
;*
;*    USER-ADDED FREQUENCY PRESETS
;*
;**********************************************

.EQU NumPresets = 9                       ;Enter # of presets here

presets:                                  ;One line for each preset freq
.db 0,3,5,6,0,0,0,0                       ;80M qrp calling =  3.560 MHz
.db 0,7,0,3,0,0,0,0                       ;40M qrp calling =  7.030 MHz
.db 1,0,0,0,0,0,0,0                       ;--- --- --- WWV = 10.000 MHz
.db 1,0,1,0,6,0,0,0                       ;30M qrp calling = 10.106 MHz
.db 1,4,0,6,0,0,0,0                       ;20M qrp calling = 14.060 MHz
.db 1,8,0,9,6,0,0,0                       ;17M qrp calling = 18.096 MHz
.db 2,1,0,6,0,0,0,0                       ;15M qrp calling = 21.060 MHz
.db 2,4,9,0,6,0,0,0                       ;12M qrp calling = 24.906 MHz
.db 2,8,0,6,0,0,0,0                       ;10M qrp calling = 28.060 MHz

;    1234567890123456
msg1:
.db "W8BH - Hold 'em ",0,0
msg2:
.db "* Scroll Presets",0,0
```