Extending Encoder Button Functionality on your DDS Development Kit

By Bruce Hall, W8BH

The DDS kit by W8DIZ gives us a really useful input device: a rotary encoder with a pushbutton on its shaft. In my VFO memory project, described at http://w8bh.net/avr/AddMemories.pdf, I used the pushbutton to switch between tuning and memory preset modes. Since the pushbutton is already used to change the cursor position, another pushbutton function is needed. I wrote a few routines to add a 'hold' function, triggered when the button has been depressed for an extended amount of time.

How many different functions can be signaled by a pushbutton? You can do a quick press, which I call 'tap', and a longer press, which I call 'hold'. Suppose you want more than two functions for the pushbutton. Can we do better? Yes, we can. Both tap and hold contain two distinct physical and electrical events: the initial button-down event, and the subsequent button-up event. This gives us four separate events that can be associated with a single button press: tap down, tap up, hold down, and hold up. If we needed more, we could even have events for double taps, triple taps, double holds, etc. Any why stop there? We could have events for Morse 'a', Morse 'b', etc. To keep things simple, and considering that our encoder pushbutton is not an optimal key, I chose to stick with the 4 single-press events.

To extend our pushbutton functions from 2 (tap and hold) to 4 (tap up/down, hold up/down), we need a way to detect the rising and falling edges of the encoder-button input. Our button input is active low, so a button-down event is the 1->0 or falling-edge transition. When the button is released, the event is a level change from 0->1 or rising-edge transition.

One way to detect edge transitions is to continuously look at the button pin input, and wait for a change in the logic level. That would waste a lot of microcontroller time! Fortunately we don't have to. There are two hardware interrupts on the ATmega88 chip, which can detect logic transitions on their associated input pins. Our pushbutton is already physically connected to an external interrupt pin, so we are all set.

The Pushbutton Interrupt routine

In the original source code, the interrupt routine for a pushbutton event is very simple:

```
EXT_INT1:
       push   temp1
       in     temp1,SREG
       inc    press
       out    SREG,temp1
       pop    temp1
       reti
```

There is only a single instruction which does anything at all: inc press. This instruction increments the 'press' register, signaling a button press that needs to be dealt with. The remainder of the code saves the contents of the status register while the interrupt is running. So where do we set up the interrupt, and program it to detect rising or falling edges? This is done in the initialization part of the program, near the beginning. The following lines set up our pushbutton interrupt:

```
       ldi    temp1,$03
       out    EIMSK,temp1          ; enable int0 and int1 interrupts
       ldi    temp1,0b00001011     ; int1 on falling edge & int0 on rising edge
       sbic   PIND,STATE           ; test state of encoder
       ldi    temp1,0b00001010     ; int1 on falling edge & int0 on falling edge
       sts    EICRA,temp1
```

OK, this is little meatier! The first two lines tell the microcontroller to enable the interrupts. In other words, they turn pin #4, which is normally Port D, bit 2 into External Interrupt 0. Chip pin #5, which is normally Port D, bit 3, turns into External Interrupt 1. The next four lines establish the interrupt mode, and cause an interrupt to occur whenever a falling edge is detected. Notice that the modes of the pushbutton and encoder interrupts are set at the same time, a complicating issue that will need to be dealt with later.

The existing interrupt routine detects only the falling-edge (button down) event. When you press the button, your cursor moves with the push-down action, but nothing happens when the button is released. Try modifying the initialization code above, substituting the values $0F for 0b00001011 and $0C for 0b00001010. This will set the interrupt to look for a rising-edge (button up) event instead.

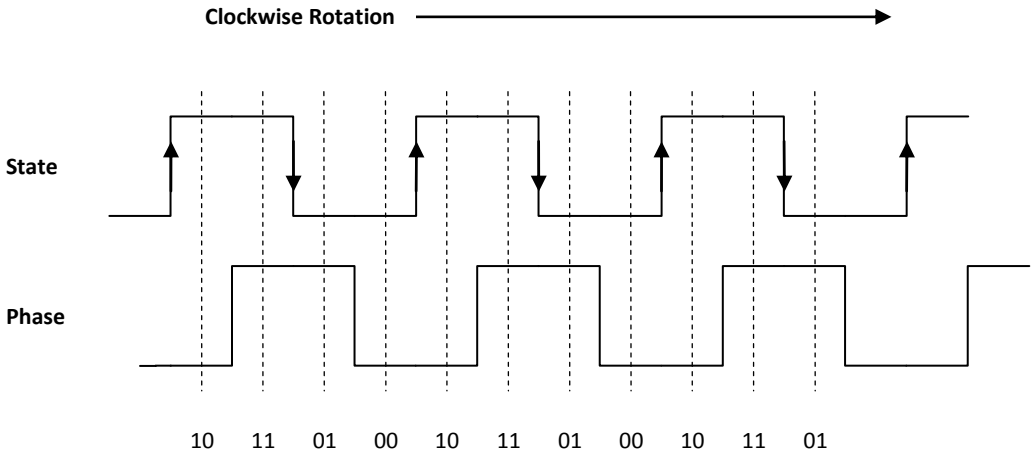
Recompile with the modification above, and you'll notice that the cursor does not move when you press the button down; it moves when the button is released. Now turn the encoder knob and press again. The cursor moves on button-down, not button-up! We just lost our interrupt-on-rising-edge programming. The reason is found in the encoder interrupt routine, show below.

The Encoder Interrupt routine

```
EXT_INT0:
       push   temp1               ;save temp1 register
       in     temp1,SREG          ;save the status register
       push   temp1
       lds    temp1,EICRA
       cpi    temp1,0b00001010    ;test falling edge
       breq   int05
       ldi    temp1,0b00001010    ;set int0 for falling edge and int1 on falling edge
       sts    EICRA,temp1
       sbis   PIND,PHASE          ;test PHASE
       rjmp   int01
       dec    encoder
       rjmp   int09
int01:
       inc    encoder
       rjmp   int09
int05:
       ldi    temp1,0b00001011    ;set int0 for rising edge and int1 on falling edge
       sts    EICRA,temp1
       sbis   PIND,PHASE          ;test PHASE
       rjmp   int06
       inc    encoder
       rjmp   int09
int06:
       dec    encoder
int09:
       pop    temp1
       out    SREG,temp1          ;restore the status register
       pop    temp1               ;restore temp1 register
```

It is worth a few minutes to study this code. The first three and last three lines just save and restore registers. The remaining code handles encoder rotation, which is implemented using the Gray code. If you are not familiar with Gray code, there is a detailed article in Wikipedia. Our encoder uses a 2 bit code, which means the two encoder outputs change states during rotation as follows: 00 -> 01 -> 11 -> 10. The code is similar to binary code 00 -> 01 -> 10 ->11, isn't it? But it is not the same. In our modern digital world, why don't manufacturers use binary code? Because binary requires both bits to change their state between 01 and 10. It is difficult to make a control that will guarantee that both bits change at exactly the same time. The gray code encoder only changes one bit at a time, and is simpler to manufacture.

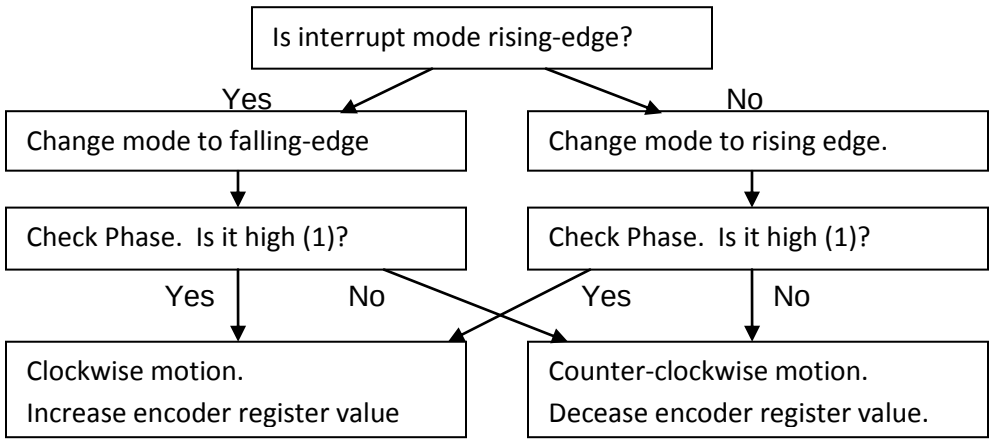
Diz calls the encoder output attached to the interrupt pin as STATE and the non-interrupting encoder output as PHASE. Here is a diagram of how the two outputs, state and phase, change as we turn the encoder shaft clockwise:



A key fact is that if we set the State interrupt to fire on a given edge, then reading the second output (Phase) will tell us the direction of spin. Look at the graph above to confirm. Change the direction of spin, reversing the arrows, and you'll come up with the table below:

Encoder Outputs:	Phase = 0	Phase = 1
State = Rising-Edge	Forward spin (CW)	Reverse spin (CCW)
State = Falling-Edge	Reverse spin	Forward spin

We can only detect one edge at a time, so we must look for the first edge, and then switch detection to the other edge. The algorithm can now be written as follows:



Now look back at the code, and you'll see that it follows this algorithm exactly. Remember that this routine somehow affected the pushbutton edge-detection. The problem is with these two instructions:

```
       ldi    temp1,0b00001010    ;set int0 for falling edge and int1 on falling edge
       ldi    temp1,0b00001011    ;set int0 for rising edge and int1 on falling edge
```

They are used to change the edge-detection for the encoder, but a side effect is they also change the pushbutton (int1) edge-detection. Whenever the encoder turns, this interrupt routine is called, and our pushbutton interrupt is affected. When we were only looking for button down event (falling edge), this wasn't a problem. But now it is. We'll need to modify the encoder routine, removing the pushbutton side effect, but keeping the encoder functionality intact. Here is one way to do it. Additional comments added, following the algorithm above.

```
EINT0:
       push   temp1                    ;save temp1 register
       in     temp1,SREG               ;save the status register
       push   temp1
       lds    temp1,EICRA              ;get current interrupt mode
       sbrs   temp1,0                  ;is mode rising-edge?
       rjmp   i02                      ;no, so go to falling edge (bit0=0)
       cbr    temp1,$01                ;yes, clear bit 0
       sts    EICRA,temp1              ;change mode to falling-edge
       sbis   PIND,PHASE               ;is PHASE=1?
       rjmp   i01                      ;no, increase encoder (CW rotation)
       dec    encoder                  ;yes, decrease encoder (CCW rotation)
       rjmp   i04
i01:   inc    encoder
       rjmp   i04
i02:                                   ;current mode = falling-edge
       sbr    temp1,$01                ;set bit 0
       sts    EICRA,temp1              ;change mode to rising-edge
       sbis   PIND,PHASE               ;is PHASE=1?
       rjmp   i03                      ;no, decrease encoder (CCW rotation)
       inc    encoder                  ;yes, increase encoder (CW rotation)
       rjmp   i04
i03:   dec    encoder
i04:   pop    temp1
       out    SREG,temp1               ;restore the status register
       pop    temp1                    ;restore temp1 register
       reti
```

The two offending LDI instructions are replaced with CBR 'clear bits in register' and SBR 'set bits in register'. These two instructions operate on individual bits instead of the entire byte. They allow us to change the bits in interrupt control register dealing with INT0 (encoder), and leave the INT1 (pushbutton) control bits intact. If you try this new code, you'll see that turning the encoder still works, but does not affect our pushbutton interrupt settings anymore.

The encoder routine shows us an easy way to expand the pushbutton routine. Whenever we detect a rising edge, flag it and change edge detection to falling edge. Similarly, whenever a falling edge is detected, flag it and switch the edge detection to rising edge. In this way we can capture both button-up and button-down events.

Now, what about hold-up and hold-down? The hold routine needs to be modified a bit, in order to correctly capture the hold-up. Our previous hold routine generated hold events when the

button was held down, but the hold counter is reset every time the hold condition is recognized. While you are holding down the button, the hold counter is incrementing, but there is no record that the hold condition was previously met and currently in progress. For this we'll use a register bit, or flag. I've created a byte in SRAM called Flags, which gives us 8 separate bits that we can set/reset as needed. Bit 0 is used to flag the 'hold-in-progress' condition. You can use the other bits for whatever you like. We can use the SBR and CBR instructions to change these bits individually, and use SBRS and SBRC instructions to branch depending on the bit state. Here is the modified CheckHold routine:

```
CHECKHOLD:
       tst    hold                    ;any new hold event?
       brpl   ck1                     ;no, so quit
       lds    temp1,flags
       sbr    temp1,$01               ;flag the hold
       sts    flags,temp1             ;save it
       rcall  ButtonHoldDown          ;do the hold event
       clr    hold                    ;reset = allow future holds
ck1:   ret
```

The SBR instruction sets our new flag, so that whenever we see a button-up event, we can check to see if the up-event if from a preceeding hold (flag=1) or just a tap (flag=0). The LDS and STS instructions load and store the Flags, using the temp1 register as a location to test and or modify the value.

The CheckButton routine will need to modified, too, since we have both button down and button up events to look for. I kept it as a single routine, but you may split this into two separate routines if it looks too cluttered for you. Here it is:

```
CHECKBUTTON:
       tst    press                   ;any button down events?
       breq   cb1                     ;no, check for button up events?
       rcall  ButtonTapDown           ;do the button down
       dec    press                   ;one less button tap to do
cb1:   tst    release                 ;any button up events?
       breq   cb4                     ;no, so quit
       lds    temp1,flags
       sbrs   temp1,0                 ;is there a hold in progress?
       rjmp   cb2                     ;no
       cbr    temp1,$01               ;yes, remove hold flag
       sts    flags,temp1             ;save un-held state
       rcall  ButtonHoldUp            ;do hold release
       rjmp   cb3
cb2:   rcall  ButtonTapUp             ;to the Tap Release
cb3:   dec    release                 ;one less release to do
cb4:   ret
```

It first checks for button-down events, which are counted by the button register, and then button-up events, which are counted by the release register. A release, starting at label cb1, has two possible causes: the release from a tap and the release from a hold. The hold-flag is checked to see if there is a hold in progress. If not, the event must have been a tap release. If there is a hold in progress, then it is a hold release and the hold flag is cleared.

There are now four named routines for the four button events: ButtonTapDown, ButtonTapUp, ButtonHoldDown, and ButtonHoldUp. Time to declare them and do something! What you do with them depends on your application, of course. For demonstration here, I'll just flash the LED and display a message and the LCD:

```
ButtonTapDown:
       ldi    temp1,5                  ; Display 'TAP DOWN'
       rjmp   dd1
ButtonTapUp:
       ldi    temp1,6                  ; Display 'TAP RELEASE'
       rjmp   dd1
ButtonHoldDown:
       ldi    temp1,7                  ; Display 'HOLD DOWN'
       rjmp   dd1
ButtonHoldUp:
       ldi    temp1,8                  ; Display 'HOLD RELEASE'
dd1:   rcall  QuickBlink
       rcall  DisplayLine2
       ret
```

We are nearly done. I wrote two routines, DisplayLine1 and DisplayLine2, to show messages on the first and second line of the LCD. The display routine in the source code takes a pointer to a null-terminated string. You can use the original DISPLAY_LINE routine instead, by labeling each message and loading Z with the address of the label. It works fine, and this is what I did at first. But you'll either need to clear the display line before you start, or be sure to write spaces for each of the 16 characters that you don't send. In the end, I decided that creating 16 character messages was easier.

```
DISPLAYLINE1:
;      displays a 16-character msg on line 1
;      call with msg# in temp1

       mov    temp2,temp1
       ldi    temp1,$80                ;use line 1
       rcall  LCDCMD
       rcall  DISPLAY16                ;send 16 characters
       ret

DISPLAYLINE2:
;      displays a 16-character msg on line 2
;      call with msg# in temp1

       mov    temp2,temp1
       ldi    temp1,$C0                ;use line 2
       rcall  LCDCMD
       rcall  DISPLAY16                ;send 16 characters
```

```
        ret

DISPLAY16:
;       displays a 16-character msg
;       call with msg# in temp2

        ldi     ZH,high(messages*2-16)
        ldi     ZL,low(messages*2-16)
di1:    adiw    Z,16                    ;add 16 for each message
        dec     temp2                   ;add enough?
        brne    di1                     ;no, add some more
        ldi     temp3,16                ;16 characters
di2:    lpm     temp1,Z+                ;get the next character
        rcall   LCDCMD                  ;put character on LCD
        dec     temp3                   ;all 16 chars sent?
        brne    di2                     ;no, so repeat
        ret
```

Both routines position the cursor to the start of an LCD line, and then call Display16 to send the characters. Display16 points to the start of the message block, which is at the end of the program. It finds the correct offset by multiplying the message number by 16. I used a small, 3 instruction addition loop to do the multiplication. The ATmega88 has a hardware-multiply instruction, which you can try instead.

The button demonstration

The final code shown here is for demonstration only, and shows how we can detect 4 different button states. It doesn't do anything useful. To access the demonstration, hold the button down until the LCD displays 'Button test mode'. Then press the button and see what happens. Press the reset button to return to normal mode. Some routines are not defined in this demonstration code and are 'commented out' with a semicolon at the start of the line.

Choices, choices

There are lots of ways to check button states and trap button events. In fact, when I started writing this article I used a different method. My original idea was to use have a single register that contained flags for button-up, button-down, and hold. The interrupt routines set the flags, and the program loop tested and reset the flags. It sounds efficient, and it worked. Unfortunately, it did not blend well with the existing source code, and the 'glue' code that kept both parts working was confusing. So I dumped it in favor of the routines presented here. Instead, we use 2 registers (press and release) and an additional hold flag. It isn't too efficient, register-wise, but works well with the existing code. Experiment and find a better way!

Source Code

```
;*********************************************************
;*     W8BH - INTERRUPT VECTOR TABLE
;*********************************************************
;  use RJMP instructions with ATmega88 chips
;  use JMP instructions with ATmega328 chips

.cseg
.org $000
       jmp    RESET
.org INT0addr
       jmp    EINT0                    ; New External Interrupt Request 0
.org INT1addr
       jmp    EINT1                    ; New External Interrupt Request 1
.org OVF0addr
       jmp    OVF0                     ; Timer/Counter0 Overflow
.org OVF2addr
       jmp    OVF2                     ; Timer/Counter2 overflow
.org INT_VECTORS_SIZE


;insert the following instruction below the 'menu' label
menu:       ;main program
       rjmp  W8BH                      ;!! go to new main program


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

; Before compiling, manually make the following changes to the source code:
; 1.  Add/Change the following register definitions:
;      .def release = R21
;      .def hold    = R15
; 2.  In .dseg, add the following line
;      mode:  .byte 1    ; 0=tuning mode; 1=button test


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
       clr    temp1
       sts    mode,temp1               ;start mode0 = normal operation
       sts    flags,temp1              ;nothing to flag yet
```

```
       ldi    temp1, $07                ;set timer2 prescale divider to 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 events
       rcall  CheckHold                 ;check for button holds
;      rcall  Keypad                    ;check for keypad action
       rjmp   Main                      ;loop forever


CHECKENCODER:
       tst    encoder                   ;any encoder requests?
       breq   ce9                       ;no, so quit
       lds    temp1,mode
       cpi    temp1,0                   ;are we in normal mode (0)?
       brne   ce1                       ;no, skip
       rcall  EncoderMode0              ;yes, handle it
       rjmp   ce9
ce1:   cpi    temp1,1                   ;are we in mode 1?
       brne   ce2                       ;no, skip
;      rcall  EncoderMode1              ;yes, handle it
       rjmp   ce9
ce2:   cpi    temp1,2                   ;are we in mode 2?
       brne   ce3                       ;no, skip
;      rcall  EncoderMode2              ;yes, handle it
       rjmp   ce9
ce3:
ce9:   ret


CHECKHOLD:
       tst    hold                      ;any new hold event?
       brpl   ck1                       ;no, so quit
       lds    temp1,flags
       sbr    temp1,$01                 ;flag the hold
       sts    flags,temp1               ;save it
       rcall  ButtonHoldDown            ;do the hold event
       clr    hold                      ;reset = allow future holds
ck1:   ret


CHECKBUTTON:
       tst    press                     ;any button down events?
       breq   cb1                       ;no, check for button up events?
       rcall  ButtonTapDown             ;do the button down
       dec    press                     ;one less button tap to do
```

```
cb1:    tst     release                 ;any button up events?
        breq    cb4                     ;no, so quit
        lds     temp1,flags
        sbrs    temp1,0                 ;is there a hold in progress?
        rjmp    cb2                     ;no
        cbr     temp1,$01               ;yes, remove hold flag
        sts     flags,temp1             ;save un-held state
        rcall   ButtonHoldUp            ;do hold release
        rjmp    cb3
cb2:    rcall   ButtonTapUp             ;do the Tap Release
cb3:    dec     release                 ;one less release to do
cb4:    ret


BUTTONTAPUP:
        lds     temp1,mode              ;get mode
        cpi     temp1,0                 ;are we in mode0?
        brne    tu1                     ;no, skip
;       rcall   TapUp0                  ;yes, handle it
        rjmp    tu9
tu1:    cpi     temp1,1                 ;are we in mode1?
        brne    tu2                     ;no, skip
        rcall   TapUp1                  ;yes, handle it
        rjmp    tu9
tu2:    cpi     temp1,2                 ;are we in mode2?
        brne    tu3                     ;no, skip
;       rcall   TapUp2                  ;yes, handle it
        rjmp    tu9
tu3:                                    ;placeholder for higher modes
tu9:    ret

BUTTONTAPDOWN:
        lds     temp1,mode              ;get mode
        cpi     temp1,0                 ;are we in mode0?
        brne    td1                     ;no, skip
        rcall   TapDown0                ;yes, handle it
        rjmp    td9
td1:    cpi     temp1,1                 ;are we in mode1?
        brne    td2                     ;no, skip
        rcall   TapDown1                ;yes, handle it
        rjmp    td9
td2:    cpi     temp1,2                 ;are we in mode2?
        brne    td3                     ;no, skip
;       rcall   TapDown2                ;yes, handle it
        rjmp    td9
td3:                                    ;placeholder for higher modes
td9:    ret

BUTTONHOLDUP:
        lds     temp1,mode              ;get mode
        cpi     temp1,0                 ;are we in mode0?
        brne    hu1                     ;no, skip
;       rcall   HoldUp0                 ;yes, handle it
        rjmp    hu9
hu1:    cpi     temp1,1                 ;are we in mode1?
```

```
        brne    hu2                     ;no, skip
        rcall   HoldUp1                 ;yes, handle it
        rjmp    hu9
hu2:    cpi     temp1,2                 ;are we in mode2?
        brne    hu3                     ;no, skip
;       rcall   HoldUp2                 ;yes, handle it
        rjmp    hu9
hu3:                                    ;placeholder for higher modes
hu9:    ret

BUTTONHOLDDOWN:
        lds     temp1,mode              ;get mode
        cpi     temp1,0                 ;are we in mode0?
        brne    hd1                     ;no, skip
        rcall   HoldDown0               ;yes, handle it
        rjmp    td9
hd1:    cpi     temp1,1                 ;are we in mode1?
        brne    hd2                     ;no, skip
        rcall   HoldDown1               ;yes, handle it
        rjmp    hd9
hd2:    cpi     temp1,2                 ;are we in mode2?
        brne    hd3                     ;no, skip
;       rcall   HoldDown2               ;yes, handle it
        rjmp    hd9
hd3:
hd9:    ret


CHANGEMODE:
;       call this routine with new mode in temp1
;       only action is to change the message on Line 1
        sts     mode,temp1              ;save the new mode
        cpi     temp1,0                 ;mode 0?
        brne    cm1                     ;no, skip
        inc     temp1
        rcall   DisplayLine1            ;yes, show normal title
        rjmp    cm9
cm1:    cpi     temp1,1                 ;mode 1?
        brne    cm2                     ;no, skip
        inc     temp1
        rcall   DisplayLine1            ;yes, show mode 1 title
        rjmp    cm9
cm2:    cpi     temp1,2                 ;mode 2?
        brne    cm3                     ;no, skip
        inc     temp1
        rcall   DisplayLine1            ;yes, show mode 2 title
        rjmp    cm9
cm3:                                    ;placeholder for higher modes
cm9:    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 - 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

TAPDOWN0:
;      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    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
```

```
HOLDDOWN0:
;       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     temp1,1
        rcall   ChangeMode              ;go to next mode
        rcall   ClearLine2
        ret


;*********************************************************
;*  W8BH - MODE 1 ROUTINES
;*********************************************************

TapDown1:
        ldi     temp1,5                 ;Display 'TAP DOWN'
        rjmp    dd1
TapUp1:
        ldi     temp1,6                 ;Display 'TAP RELEASE'
        rjmp    dd1
HoldDown1:
        ldi     temp1,7                 ;Display 'HOLD DOWN'
        rjmp    dd1
HoldUp1:
        ldi     temp1,8                 ;Display 'HOLD RELEASE'
dd1:    rcall   QuickBlink
        rcall   DisplayLine2
        ret


;*********************************************************
;*  W8BH - Timer 2 Overflow Interrupt Handler
;*********************************************************
;       This handler is called every 8 ms @ 20.48MHz clock
;       Increments HOLD counter (max 128) when button held
;       Resets HOLD counter if button released before hold met
;       Sets hold & down flags in button state register.

OVF2:
        push    temp1
        in      temp1,SREG              ;save status register
        push    temp1
        ldi     temp1,90                ;256-90=160; 160*50us = 8ms
        sts     TCNT2,temp1             ;reduce cycle time to 8 ms
        tst     hold                    ;counter at max yet?
        brmi    ov1                     ;not yet
        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 - External Interrupt 1 Handler
;*********************************************************
;      This handler is replaces the original EXT_INT1 code
;      It is called when a logic-level change on the
;      external interrupt 1 (pushbutton) pin occurs.
;      Press is incremented on button-down events.
;      Release is incremented on button-up events.

EINT1:
       push   temp1                    ;save temp1 register
       in     temp1,SREG
       push   temp1                    ;save status register
       lds    temp1,EICRa              ;get interrupt control register
       sbrs   temp1,2                  ;bit2: rising edge =0, falling edge =1
       rjmp   ei1
                                       ;here is the falling-edge code
       cbr    temp1,$04                ;falling edge '11' -> rising edge '10'
       inc    release                  ;count the button-up
       rjmp   ei2
                                       ;here is the rising-edge code
ei1:   sbr    temp1,$04                ;rising edge '10' -> falling edge '11'
       inc    press                    ;count the button-down

ei2:   sts    EICRa,temp1              ;save interrupt control register
       pop    temp1
       out    SREG,temp1               ;restore status register
       pop    temp1                    ;restore temp1 register
       reti

;*********************************************************
;*  W8BH - External Interrupt 0 Handler
;*********************************************************
;      This handler is replaces the original EXT_INT0 code
;      It is called when a logic-level change on the
;      external interrupt 0 (encoder state) pin occurs.
;      Press is incremented on button-down events.
;      Release is incremented on button-up events.

EINT0:
       push   temp1                    ;save temp1 register
       in     temp1,SREG               ;save the status register
       push   temp1
       lds    temp1,EICRA              ;get current interrupt mode
       sbrs   temp1,0                  ;is mode rising-edge?
       rjmp   i02                      ;no, so go to falling edge (bit0=0)
       cbr    temp1,$01                ;yes, clear bit 0
       sts    EICRA,temp1              ;change mode to falling-edge
       sbis   PIND,PHASE               ;is PHASE=1?
       rjmp   i01                      ;no, increase encoder (CW rotation)
       dec    encoder                  ;yes, decrease encoder (CCW rotation)
       rjmp   i04
i01:   inc    encoder
       rjmp   i04
i02:                                   ;current mode = falling-edge
```

```
       sbr    temp1,$01                ;set bit 0
       sts    EICRA,temp1              ;change mode to rising-edge
       sbis   PIND,PHASE               ;is PHASE=1?
       rjmp   i03                      ;no, decrease encoder (CCW rotation)
       inc    encoder                  ;yes, increase encoder (CW rotation)
       rjmp   i04
i03:   dec    encoder
i04:   pop    temp1
       out    SREG,temp1               ;restore the status register
       pop    temp1                    ;restore temp1 register
       reti


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


;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

DISPLAYLINE1:
;      displays a 16-character msg on line 1
;      call with msg# in temp1

       mov    temp2,temp1
       ldi    temp1,$80                ;use line 1
       rcall  LCDCMD
       rcall  DISPLAY16                ;send 16 characters
       ret

DISPLAYLINE2:
;      displays a 16-character msg on line 2
;      call with msg# in temp1

       mov    temp2,temp1
       ldi    temp1,$C0                ;use line 2
       rcall  LCDCMD
       rcall  DISPLAY16                ;send 16 characters
       ret

DISPLAY16:
;      displays a 16-character msg
;      call with msg# in temp2

       ldi    ZH,high(messages*2-16)
       ldi    ZL,low(messages*2-16)
```

```
di1:   adiw   Z,16                    ;add 16 for each message
       dec    temp2                   ;add enough?
       brne   di1                     ;no, add some more
       ldi    temp3,16                ;16 characters
di2:   lpm    temp1,Z+                ;get the next character
       rcall  LCDCHR                  ;put character on LCD
       dec    temp3                   ;all 16 chars sent?
       brne   di2                     ;no, so repeat
       ret

CLEARLINE2:
       ldi    temp1,$C0
       rcall  LCDCMD
       ldi    temp3,16
cl1:   ldi    temp1,' '
       rcall  LCDCHR
       dec    temp3
       brne   cl1
       ret


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


;    12345678901234567890123456789012345678901234567890
msg1:
.db "Kits and Parts",0,0

messages:
.db "W8BH - Mode 0   "           ;1
.db "Button Test Mode"           ;2
.db "Mode 3          "           ;3
.db "Mode 4          "           ;4
.db " TAP DOWN       "           ;5
.db " TAP RELEASE    "           ;6
.db " HOLD DOWN      "           ;7
.db " HOLD RELEASE   "           ;8
.db " -5-----        "           ;9
```