A Memory Keyer for your DDS Development Kit

By Bruce Hall, W8BH

This article will describe how to add a memory keyer to your DDS Development kit. In the last project (http://w8bh.net/avr/IambicKeyer.pdf), I created a simple iambic keyer. This project will build on that keyer, adding the ability to send two user-defined messages.

The first thing I did was search for similar projects. Many different keyers, beacons, repeater controllers and such have been developed, and surely hams have published ways of automating Morse code output. I found three different methods, each one using the idea of encoding Morse dits and dahs as binary 1's and 0's.

For example, some keyers use a binary 1 to indicate key down, and a 0 to indicate key up. A 'C' would be encoded as 11101011101: three ones for a dah, then a one for a dit, etc. This method is described by David Robinson WW2R/G4FRE (http://g4fre.com/keyers2.htm). A second method, used by Hans Summers in his QRSS beacons (http://www.hanssummers.com/qrsskeyer), is to code a dit as a '0' and dah as a '1'. The byte is scanned from right to left, looking for a '0' start bit. Once found, the remaining bits are converted to Morse output. This method works well in the C programming language. Using this method, 'C' is encoded as 11101010 or $EA. The third and oldest method was described by Jeff Otterson N1KDO in his Feb 1997 QST article. Using '0' for dit and '1' for dah, the byte was shifted to the right until a single '1' remained. For example, a 'C' would be coded as 00010101 or $15.

For my method I use a combination of the second and third methods above. I like the N1KDO idea of shifting the byte to get the next bit: it's an assembly-friendly method, but the codes must be entered and read from right to left. I prefer the more natural left-to-right encoding that Hans uses. I also visually think of dits as skinny 1's and dahs as rounder 0's, but that's just me. My algorithm for stuffing Morse characters into bytes goes like this:

- Shift the byte to the left, into the carry bit, so that we're reading left to right
- If the carry bit is set (1), then send a dit. Otherwise send a dah.
- If we are left with a stop bit of '1' ($80), we're done. Otherwise repeat.

My 'C' is 01011000 or $58. The code for this algorithm is just as simple:

```
MorseOut:
;       call this routine with the encoded morse byte in temp1
        cpi     temp1,$80               ;found stop bit yet:
        breq    mo2                     ;yes, so quit
        lsl     temp1                   ;no, get next bit into carry
        brcs    mo1                     ;is the bit a dit? (bit=1)
        rcall   dah                     ;no, so send a dah
        rjmp    morseout
mo1:    rcall   dit                     ;yes, so send a dit
        rjmp    morseout
mo2:    rcall   ditwait                 ;end of char spacing
        rcall   ditwait
        ret

CQtest:
;       sends a CQ
        ldi     temp1,$58               ;binary 0101.1000 = 'C'
        rcall   MorseOut
        ldi     temp1,$28               ;binary 0010.1000 = 'Q'
        rcall   MorseOut
        ret
```

I like to test things as I go along, so I added the CQtest routine. The C (dah-di-dah-dit) is coded as 0101, with a 1 stop bit added to the end. Similarly Q is 0010 with an extra 1 on the end.

Add the two routines above to the Iambic Keyer project, and put a call to CQtest in the initialization part of the program. When your DDS kits resets, you'll see the LED flash 'CQ'. Very cool.

We could code all of our messages this way, but it would be a little tedious. Every time we want to create a message, we would have to mentally convert each letter into 'coded-morse', then send those coded bytes to the MorseOut routine. It would be much nicer if we could just send a text message using plain ASCII characters, like 'CQ de W8BH'. To do this we'll need a second routine to convert ASCII characters to coded-morse.

To convert an ASCII character to coded-morse, I use a simple lookup table. Each text character is mapped to its corresponding coded-morse byte. For example, assume our table begins with 'A'. The first entry will be the coded-morse for 'A', which is 1010000. (Remember, the extra one at the end is the stop bit.) The second entry will be for 'B', and so on. If the input character is an 'E', all we need to do is grab the fifth table entry and we're done.

The ASCII-to-Morse conversion table never needs to be changed, so program memory is a good place to put it. I put my table at the end of the program, where the strings and default VFO frequencies are kept.

So, can we begin this table with the letter 'A', as I suggested above? I guess so, but that is a poor choice. The ASCII code (see http://ascii-code.com) is a 7-bit code with 128 entries. The first 32 entries are nonprintable control codes, and the next 95 are printable characters. The 95 printables break down as follows: symbols (16), digits (10), symbols (7), upper case alphabet (26), symbols (6), lower case alphabet (26) and symbols (4). We don't need our table to include all of these. All we need are digits, alpha characters, and a few symbols. I started my table at

position #42, which is hex $2A, the asterisk. You could easily start anywhere from $20 to $2F, depending on how many symbols and punctuation you wanted to include. I found it helpful to include some symbols, so that I could assign them to prosigns like BT and SK. Here is the code. Half of the code is just making sure that we exclude ASCII codes that we don't need.

```
AsciiToMorse:
;       Call with an ASCII character in temp1
;       This routine will convert it into a coded-morse character
;       If input is an invalid character, output = $80 (stop bit)
        ldi     ZH,high(2*mtable)           ;point to morse table
        ldi     ZL,low(2*mtable)
        cpi     temp1,$20                   ;is it a space character?
        brne    am1                         ;no
        rcall   WordWait                    ;yes, so wait appropriate time
        rjmp    am4
am1:    cpi     temp1,$2A                   ;ignore control chars
        brmi    am3
        cpi     temp1,$7A                   ;ignore graphic chars
        brpl    am3
        cpi     temp1,$60                   ;is it an lower-case char?
        brmi    am2                         ;no
        andi    temp1,$DF                   ;yes, convert to upper-case
am2:    subi    temp1,$2A                   ;start table at $2A='*'
        add     ZL,temp1                    ;add char offset to table pointer
        clr     temp1                       ;keep only the carry bit
        adc     ZH,temp1                    ;add carry, if any, to ZH
        lpm     temp1,Z                     ;get character from table
        rjmp    am4                         ;done
am3:    ldi     temp1,$80                   ;output stop-bit for invalid chars
am4:    ret
```

Notice that I subtract $2A from the input, so that ASCII code $2A (asterisk) corresponds to the first entry in my conversion table.

Here is another neat trick: I save a lot of table space by mapping all of the lower case ASCII characters to the corresponding upper case character. For example, the letter 'q' (ASCII 81) is mapped to 'Q' (ASCII 113) by the ANDI temp1,$DF instruction. This single instruction works because the difference between all of the lower case and upper case codes is a single bit: bit 5. The ANDI instruction turns this bit from 1 into 0. Bit 5 has a value of 32 (00100000), so you could also subtract 32 for the same result: 113-32 = 81.

Let's try another test. Add the following code and put a call to CQtest2 in the initialization section of your program. Again, the LED will flash CQ. This time, however, it is doing it from text input.

```
CQtest2:
        ldi     temp1,'c'                   ;send a 'c'
        rcall   AsciiToMorse
        rcall   MorseOut
        ldi     temp1,'q'                   ;send a 'q'
        rcall   AsciiToMorse
        rcall   MorseOut
        ret
```

Now that we can send single characters, it is time to send some longer messages. All we need is a single loop that gets each character from the message, convert it to coded-morse, and then output it. We'll use a 0 to mark the end of the message.

```
MorseMsg:
;       call with Z pointing to message
;       will output Morse
mm1:    lpm     temp1,Z+                ;get next ASCII character
        tst     temp1                   ;look for 0=stop byte
        breq    mm2                     ;done
        rcall   AsciiToMorse            ;convert char to morse
        rcall   MorseOut                ;and send it
        rjmp    mm1                     ;no, keep going
mm2:    ret

SendMorse1:
        ldi     ZH,high(2*cwmem1)       ;point to first CW msg
        ldi     ZL,low(2*cwmem1)
        rcall   MorseMsg
        ret
```

We can send a very, very long message with these routines. Just put your message at the end of your program with the label 'cwmem1:'. Terminate your message with a zero byte. I sent some really long ones, just to make sure the table was working correctly.

One of my goals was to display messages on the LCD screen as they were being sent. The MorseMsg routine will need to be modified, calling the LCDCHR routine each time a letter is sent. I tried this:

```
MorseMsg:
;       call with Z pointing to message
;       will output Morse
mm1:    lpm     temp1,Z+                ;get next ASCII character
        tst     temp1                   ;look for 0=stop byte
        breq    mm2                     ;done
        rcall   LCDCHR                  ;show char on LCD?  DOESN'T WORK!
        rcall   AsciiToMorse            ;convert char to morse
        rcall   MorseOut                ;and send it
        rjmp    mm1                     ;no, keep going
mm2:    ret
```

Guess what? It doesn't work. Going back to the original source code, LCDCHR does not preserve its input register, temp1. We'll need to save this value, then recall it after the routine completes. I used PUSH and POP for this:

```
MorseMsg:
;       call with Z pointing to message
;       will output Morse
mm1:    lpm     temp1,Z+                ;get next ASCII character
        tst     temp1                   ;look for 0=stop byte
        breq    mm2                     ;done
        push    temp1                   ;save char
        rcall   LCDCHR                  ;show char on LCD
        pop     temp1                   ;restore char
        rcall   AsciiToMorse            ;convert char to morse
        rcall   MorseOut                ;and send it
```

```
        rjmp    mm1                     ;no, keep going
mm2:    ret
```

Now we can see the text on the LCD, but it only works if the LCD cursor is at a visible location. In other words, letters can be sent 'off the screen' and be invisible unless we are carefully controlling where the text is sent. Calling LCDCHR puts the character at the current cursor location, but doesn't guarantee that it will be visible! Time for some more modifications:

```
ResetLine2:
;       used by MorseMsg to prep LCD line 2
        rcall   ClearLine2              ;erase line2
        ldi     temp1,$C0               ;set cursor to start of line
        rcall   LCDCMD
        clr     temp3                   ;clear char counter
        ret

MorseMsg:
;       call with Z pointing to message
;       will output Morse and show it on LCD line 2
        rcall   ResetLine2              ;prep line2 for display
mm1:    lpm     temp1,Z+                ;get next ASCII character
        tst     temp1                   ;look for 0=stop byte
        breq    mm2                     ;done
        push    temp1                   ;save char
        rcall   LCDCHR                  ;show char on LCD
        pop     temp1                   ;restore char
        rcall   AsciiToMorse            ;convert char to morse
        rcall   MorseOut                ;and send it
        inc     temp3                   ;incr character counter
        cpi     temp3,16                ;is line2 full=16 chars?
        breq    MorseMsg                ;yes, clear it & continue
        rjmp    mm1                     ;no, keep going
mm2:    ret

SendMorse:
        ldi     temp1,10                ;Show 'Sending Message'
        rcall   DisplayLine1            ;on LCD line 1
        rcall   MorseMsg                ;send the message
        ldi     temp1,0                 ;restore LCD
        rcall   ChangeMode              ;to tuning mode
        ret

SendMorse1:
        ldi     ZH,high(2*cwmem1)       ;point to first CW msg
        ldi     ZL,low(2*cwmem1)
        rcall   SendMorse
        ret
```

Now we confine our text to the second line of the LCD display, and count characters using the temp3 register. Once the line is full, the next character will erase the line and start back at the beginning. You can do some fancy scrolling if you like, but this works and looks OK. The modified code also restores our display after the message has finished. Try putting a call to SendMorse1 in your initialization code, and it will play back to you on startup.

These messages are helpful only if we have a quick way to send them. I thought about putting them in another 'mode', but it isn't very convenient: holding down the encoder button for several seconds to select a message is cumbersome and takes too long. We need a quicker method, like the individual buttons found on the side of many keyers. There aren't any extra button inputs handy, but we do have our paddles and the encoder button. Each of them already has a function. But what about combining them?

My idea for memory keying is to combine the button and paddles: push the button down, and then hit a paddle key while the button is depressed. The left paddle is keyer memory #1 and the right is memory #2. To do this we'll need to check the button state in the paddle input routines. If the button is down, do the memory keying; otherwise, send the dit/dah. Here is the revised code for the paddledown routines:

```
LPADDLEDOWN:
;       Come here is the left (dit) paddle is pressed
        sbis    PinD,PD3                    ;is encoder button down?
        rjmp    SendMorse1                  ;yes, so do message1
        sbis    PinC,RPaddle                ;are both paddles pressed?
        rjmp    Iambic                      ;yes, so iambic mode
        rcall   Dit                         ;no, so just send a dit
        ret

RPADDLEDOWN:
; Come here is the left (dit) paddle is pressed
        sbis    PinD,PD3                    ;is encoder button down?
        rjmp    SendMorse2                  ;yes, so do message2
        sbis    PinC,LPaddle                ;are both paddles pressed?
        rjmp    Iambic                      ;yes, so iambic mode
        rcall   Dah                         ;no, so just send a dah
        ret
```

That's it. The added SBIS instructions check the button state, and cause a jump to the memory keyer routines if the button is pressed.

73,

Bruce.

Final Code

Instructions on adding these routines to the source code are found in the iambic keyer article.

```
;*****************************************************
;*  W8BH - Iambic Keyer routines
;*****************************************************
;
; Left paddle (dit) = Port C, bit 5
; Right paddle (dah) = Port C, bit 4
```

```
; Keyer output line = Port D, bit 6

.equ          LPaddle       = PC5
.equ          RPaddle       = PC4
.equ          DahFlag       = 1           ;0=dit, 1=dah
.equ          KeyOut        = PD6



CHECKKEY:
;       Checks to see if either of the paddles have been pressed.
;       Paddle inputs are active low
        lds     temp2,flags               ;get flags in register
        sbis    PinC,LPaddle              ;dit (left) paddle pressed?
        rcall   LPaddleDown               ;yes, so do it
        sbis    PinC,RPaddle              ;dah (right) paddle pressed?
        rcall   RPaddleDown               ;yes, so do it
        sts     flags,temp2               ;save flags
        ret

LPADDLEDOWN:
;       Come here is the left (dit) paddle is pressed
        sbis    PinD,PD3                  ;is encoder button down?
        rjmp    SendMorse1                ;yes, so do message1
        sbis    PinC,RPaddle              ;are both paddles pressed?
        rjmp    Iambic                    ;yes, so iambic mode
        rcall   Dit                       ;no, so just send a dit
        ret

RPADDLEDOWN:
; Come here is the left (dit) paddle is pressed
        sbis    PinD,PD3                  ;is encoder button down?
        rjmp    SendMorse2                ;yes, so do message2
        sbis    PinC,LPaddle              ;are both paddles pressed?
        rjmp    Iambic                    ;yes, so iambic mode
        rcall   Dah                       ;no, so just send a dah
        ret

IAMBIC:
;       Come here if both paddles are pressed
        sbrc    temp2,DahFlag             ;was the last element a Dah?
        rjmp    Dit                       ;yes, so do a dit now
        rjmp    Dah                       ;no, so do a dah now

DIT:
        rcall   KeyDown
        rcall   DitWait                   ;key down for 1 dit
        rcall   KeyUp
        rcall   DitWait                   ;key up for 1 dit
        cbr     temp2,1<<DahFlag          ;remember dit sent
        ret

DAH:
        rcall   KeyDown
        rcall   DahWait                   ;key down for 1 dah
        rcall   KeyUp
        rcall   DitWait                   ;key up for 1 dit
        sbr     temp2,1<<DahFlag          ;remember dah sent
        ret

KEYDOWN:
        cbi     PortD,KeyOut              ;turn on output line
        cbi     PortC,LED                 ;turn on LED
```

```
        ret

KEYUP:
        sbi     PortD,KeyOut            ;turn off output line
        sbi     PortC,LED               ;turn off LED
        ret

DITWAIT:
        ldi     delay, 120              ;set speed at 10 WPM
        rcall   wait                    ;and wait that long
        ret

DAHWAIT:                                ;wait for 3 dits
        rcall   DitWait
        rcall   DitWait
        rcall   DitWait
        ret

MORSEOUT:
;       call this routine with the encoded morse byte in temp1
        cpi     temp1,$80               ;found stop bit yet:
        breq    mo2                     ;yes, so quit
        lsl     temp1                   ;no, get next bit into carry
        brcs    mo1                     ;is the bit a dit? (bit=1)
        rcall   dah                     ;no, so send a dah
        rjmp    morseout
mo1:    rcall   dit                     ;yes, so send a dit
        rjmp    morseout
mo2:    rcall   ditwait                 ;end of char spacing
        rcall   ditwait
        ret

CQTEST:
;       sends a CQ
        ldi     temp1,$58               ;binary 0101.1000 = 'C'
        rcall   MorseOut
        ldi     temp1,$28               ;binary 0010.1000 = 'Q'
        rcall   MorseOut
        ret


ASCIITOMORSE:
;       Call with an ASCII character in temp1
;       This routine will convert it into a coded-morse character
;       If input is an invalid character, output = $80 (stop bit)
        ldi     ZH,high(2*mtable)       ;point to morse table
        ldi     ZL,low(2*mtable)
        cpi     temp1,$20               ;is it a space character?
        brne    am1                     ;no
        rcall   WordWait                ;yes, so wait appropriate time
        rjmp    am4
am1:    cpi     temp1,$2A               ;ignore control chars
        brmi    am3
        cpi     temp1,$7A               ;ignore graphic chars
        brpl    am3
        cpi     temp1,$60               ;is it an lower-case char?
        brmi    am2                     ;no
        andi    temp1,$DF               ;yes, convert to upper-case
am2:    subi    temp1,$2A               ;start table at $2A='*'
        add     ZL,temp1                ;add char offset to table pointer
        clr     temp1                   ;keep only the carry bit
        adc     ZH,temp1                ;add carry, if any, to ZH
        lpm     temp1,Z                 ;get character from table
```

```
       rjmp   am4                      ;done
am3:   ldi    temp1,$80                ;output stop-bit for invalid chars
am4:   ret

CQTEST2:
       ldi    temp1,'c'                ;send a 'c'
       rcall  AsciiToMorse
       rcall  MorseOut
       ldi    temp1,'q'                ;send a 'q'
       rcall  AsciiToMorse
       rcall  MorseOut
       ret

RESETLINE2:
;      used by MorseMsg to prep LCD line 2
       rcall  ClearLine2               ;erase line2
       ldi    temp1,$C0                ;set cursor to start of line
       rcall  LCDCMD
       clr    temp3                    ;clear char counter
       ret

MORSEMSG:
;      call with Z pointing to message
;      will output Morse and show it on LCD line 2
       rcall  ResetLine2               ;prep line2 for display
mm1:   lpm    temp1,Z+                 ;get next ASCII character
       tst    temp1                    ;look for 0=stop byte
       breq   mm2                      ;done
       push   temp1                    ;save char
       rcall  LCDCHR                   ;show char on LCD
       pop    temp1                    ;restore char
       rcall  AsciiToMorse             ;convert char to morse
       rcall  MorseOut                 ;and send it
       inc    temp3                    ;incr character counter
       cpi    temp3,16                 ;is line2 full=16 chars?
       breq   MorseMsg                 ;yes, clear it & continue
       rjmp   mm1                      ;no, keep going
mm2:   ret

SENDMORSE:
       ldi    temp1,10                 ;Show 'Sending Message'
       rcall  DisplayLine1             ;on LCD line 1
       rcall  MorseMsg                 ;send the message
       ldi    temp1,0                  ;restore LCD
       rcall  ChangeMode               ;to tuning mode
       ret

SENDMORSE1:
       ldi    ZH,high(2*cwmem1)        ;point to first CW msg
       ldi    ZL,low(2*cwmem1)
       rcall  SendMorse
       ret

SENDMORSE2:
       ldi    ZH,high(2*cwmem2)        ;point to second CW msg
       ldi    ZL,low(2*cwmem2)
       rcall  SendMorse
       ret


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

```
;The following table & keyer memories are placed at
;the end of your source code:


mtable:
;This table converts ASCII characters into their morse equivalent
;The ASCII character is listed as a comment above each code
;Read the code from left to right, with 1=dit and 0=dah
;An extra, silent '1' is added at the end as a stop-bit

;       * (SK)      + (AR)      ,           - (BT)
.db 0b11101010, 0b10101100, 0b00110010, 0b01110100
;       .           /           0           1
.db 0b01010110, 0b01101100, 0b00000100, 0b10000100
;       2           3           4           5
.db 0b11000100, 0b11100100, 0b11110100, 0b11111100
;       6           7           8           9
.db 0b01111000, 0b00111100, 0b00011100, 0b00001100
;       :           ;           <           =
.db 0b00000000, 0b00000000, 0b00000000, 0b00000000
;       >           ?           @           A
.db 0b00000000, 0b11001110, 0b10010110, 0b10100000
;       B           C           D           E
.db 0b01111000, 0b01011000, 0b01110000, 0b11000000
;       F           G           H           I
.db 0b11011000, 0b00110000, 0b11111000, 0b11100000
;       J           K           L           M
.db 0b10001000, 0b01010000, 0b10111000, 0b00100000
;       N           O           P           Q
.db 0b01100000, 0b00010000, 0b10011000, 0b00101000
;       R           S           T           U
.db 0b10110000, 0b11110000, 0b01000000, 0b11010000
;       V           W           X           Y
.db 0b11101000, 0b10010000, 0b01101000, 0b01001000
;       Z           [           \           ]
.db 0b00110000, 0b00000000, 0b00000000, 0b00000000


cwmem1:
.db "CQ CQ CQ de W8BH W8BH W8BH K",0,0
cwmem2:
.db "TNX FER GUD QSO - 73 73 * de W8BH K",0
```